Higher-Order Nested Data Parallelism:
Semantics and Implementation

vorgelegt von
Diplom-Informatiker
Roman Leshchinskiy

Von der Fakultät IV — Elektrotechnik und Informatik —
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
— Dr. rer. nat. —

genehmigte Dissertation

Promotionsausschuß:

Vorsitzender: Prof. Dr. Peter Pepper
Berichter: Prof. Dr. Stefan Jähnichen
Berichter: Prof. Dr. Sergei Gorlatch

Tag der mündlichen Prüfung: 16. Dezember 2005

Berlin 2006

D 83

Zusammenfassung

Das veschachtelt datenparallele Programmiermodell ist ein attraktiver Ansatz zur Entwicklung von Software für massiv parallele Systeme. Das Modell erlaubt es, komplexes paralleles Verhalten durch verschachtelte kollektive Operationen zu spezifizieren, ohne maschinennahe Details wie Synchronisation oder Kommunikation betrachten zu müssen. Dieses hohe Abstraktionsniveau wird erreicht, indem die Aufgabe leicht parallelisierbare Repräsentationen von Datenstrukturen abzuleiten und Berechnungen entsprechend zu transformieren, vom Programmierer auf den Compiler übertragen wird.

Erreicht wird dies durch die Flattening-Transformation, die verschachtelten Parallelismus eliminiert und flache datenparallele Programme erzeugt. Letztere können mit Hilfe einer Reihe bekannter Techniken optimiert und schließlich effizient auf modernen parallelen Rechnern ausgeführt werden. Die Korrektheit des Flattening hängt entscheidend von der referenziellen Transparenz der Quellsprache ab. Allerdings war die Benutzbarkeit der Transformation in einem rein funktionalen Ansatz bisher stark eingeschränkt, da sie Funktionen höherer Ordnung nicht behandeln konnte.

Die vorliegende Dissertation führt diese Einschränkung auf das in der Standardsemantik der Lambda-Kalküls inhärente Vermischen von Berechnungen und Daten zurück und zeigt auf, wie dieses Problem durch eine Kombination von Flattening mit *Closure Conversion* gelöst werden kann. *Closure Conversion* ist eine bekannte Programmtransformation, die genau die erforderlich Trennung zwischen Code und Daten vornimmt. Zur Validierung des Ansatzes wird eine repräsentative Menge von parallelen Operationen betrachtet und aufgezeigt, wird diese erweitern werden können, um Funktionen höherer Ordnung in durch *Closure Conversion* transformierten Programmen zu unterstützen.

Ferner enthält die Dissertation eine detaillierte Untersuchung der Interaktionen zwischen Flattening und nicht-strikter Auswertung. Diese sind sehr komplex, da die durch die Transformation erzeugten Datenstrukturen auf strikten Vektoren basieren. Durch eine im Vergleich zu früheren Arbeiten modifizierte Definition der Flattening-Transformation können nicht-strikte Programme trotzdem korrekt übersetzt werden.

Schließlich wird die Korrektheit von Flattening für eine nicht-strikte Sprache mit Funktionen höherer Ordnung beweisen. Für den Beweis wird eine neue Form des Lambda-Kalküls eingeführt, in der die Trennung zwischen Berechnungen und Daten syntaktisch sichergestellt wird.

Diese Ergebnisse erlauben es, Flattening und somit verschachtelten Datenparallelismus transparent in eine funktionale Programmiersprache wie Haskell zu intergrieren. Wir glauben, daß dies sowohl der Akzeptanz des Programmiermodells als auch der Entwicklung von paralleler Software zugute kommen wird.

Abstract

Nested data-parallel programming is an attractive approach to implementing applications for massively parallel systems. It allows complex parallel behaviour to be specified by combining and nesting operations on parallel collections and liberates the programmer from low-level concerns such as synchronisation and communication. The high degree of abstraction is achieved by transferring to the compiler the problem of finding easily parallelisable representations for data structures and transforming the computations accordingly.

This task is carried out by the flattening transformation which eliminates nested uses of parallelism, generating flat data-parallel code which is amenable to a wide range of optimisations and can be executed efficiently on modern parallel architectures. The correctness of flattening crucially depends on referential transparency of the source language; but until now, the usefulness of this transformation in a purely functional setting has been severely restricted as it could not handle higher-order functions.

The present dissertation shows that this shortcoming is due to the intertwining of computation and data inherent in the standard semantics of the lambda calculus and proposes to combine flattening with closure conversion as a solution to this problem. Closure conversion is a well-established program transformation which provides precisely the required degree of separation between code and data. The viability of this approach is validated by investigating a representative range of parallel operations and demonstrating how they can be naturally extended to support higher-order functions in closure-converted programs.

Moreover, the dissertation provides a detailed discussion of the interactions between flattening and lazy evaluation. These interactions are quite intricate, as the flat data structures derived by the transformation a based on strict, unboxed arrays. Nevertheless, it is shown that lazy programs can be correctly flattened by suitably modifying the transformation as compared to previous accounts.

Finally, the correctness of the flattening transformation in a higher-order, non-strict setting is proved. The proof relies on a novel form of the lambda calculus which syntactically enforces the separation of computation and data achieved by closure conversion.

These results allow flattening and, hence, nested data parallelism to be seamlessly integrated into a full-fledged functional language, such as Haskell. We believe that this will have a beneficial impact on the acceptance of this programming model and on the delevopment of parallel software applications.

Acknowledgements

This work would not have been possible without the many people who have contributed to it. I would like to thank my supervisor, Stefan Jähnichen, for his support, confidence and encouragement. The work environment of the Software Engineering Group was both informal and highly stimulating. I am grateful to the past and present members of the group for the good fun we had both during and after work. In particular, my heartfelt thanks go to Gabi Ambach for trying to keep me out of trouble and, amazingly, succeeding most of the time. Wolf Pfannenstiel introduced me to the world of parallel programming when I was a student and turned out to be the perfect office mate after I graduated.

I am infinitely indebted to Gabi Keller and Manuel Chakravarty. Large parts of this thesis are direct results of the countless discussions we had about nested data parallelism and functional programming. The opportunity to work at the University of New South Wales had a highly beneficial impact on this research and was very interesting from a personal point of view. More importantly, they are very good friends. I would also like to thank the other members of the Programming Languages and Systems group at the UNSW for the friendly atmosphere and the challenging discussions. Finally, Leon Chakravarty often provided the necessary distraction.

I am grateful to Sergei Gorlatch, Holger Bischof and Martin Alt for the good fun and the interesting discussions we had during their employment in Berlin.

Last but not least, I am grateful to my family who never fail to believe in me and to provide the necessary support and encouragement.

Sydney, 16.9.2005 Roman Leshchinskiy

Contents

List of Figures

Chapter 1

Introduction

In 1965, Gordon Moore formulated his famous law stating that the complexity of microprocessors doubles every twelve months. This prediction has proved to be astonishingly accurate ever since. However, Moore did not foresee or, at least, did not express with such precision that the complexity of computations performed by computers will increase at an even more dramatic rate. The performance of a single processor is inadequate for a large variety of problems solved with the help of computers; this discrepancy is perhaps even larger today than it was in 1965. Thus, the need for parallel processing is more urgent than ever. Massively parallel systems are used for searching the internet, forecasting the weather, simulating physical processes and chemical reactions and other tasks too numerous to mention here.

However, while our understanding of how to build such systems has increased significantly, their programming remains a difficult and highly error-prone task. Mainly, this is due to the lack, at least lack of acceptance, of programming models which provide the programmer with a sufficiently high degree of abstraction while still maintaining the desired performance. Parallel programs are usually implemented in Fortran or C, and rely on low-level communication libraries, such as MPI, which, while very efficient, require the user to deal with the minutiae of parallel programming best left to the compiler. In this situation, it is not surprising that even moderately complex parallel software systems are perceived to be hard to write, expensive to maintain and almost impossible to get right. As Gorlatch (2004) points out, point-to-point communication, the most popular parallel programming technique today, is comparable to the infamous *goto* statement and equally harmful from the software engineering point of view.

This is all the more unfortunate as a large number of structured and highly expressive approaches to the development of parallel software have been proposed. However, many of them have suffered from serious drawbacks. Frequently, only a limited number of language constructs and programming techniques are supported, leaving out those which, while useful, are difficult to handle in the presence of parallelism. Moreover, the available implementations are often just proofs of concept and unusable for real-world programming.

1.1 Nested data parallelism

A prime example of a promising technique which exhibits these deficiencies is the nested data-parallel programming model. Based on arrays with an inherently parallel semantics, it provides collective operations as the only means of expressing parallelism. This approach eliminates many stumbling blocks in the development of parallel applications. In particular,

nested data-parallel programs have only a single control flow, obliviating the need and, in fact, making it impossible to explicitly specify synchronisation and communication patterns. This allows the programmer to concentrate on the algorithms and data structures involved in solving the problem at hand rather than trying to avoid deadlocks, race conditions and other troublesome aspects of low-level communication.

Nested data parallelism (NDP) was originally conceived as a generalisation of data parallel constructs found in languages like High Performance Fortran and C*. The latter provide parallel collections and efficient parallel operations on them but do not allow either the former or the latter to be nested. This restriction is lifted in the NDP model, thus enabling it to transparently support a much wider range of algorithms, including irregular ones (Blelloch, 1996). In fact, the nesting of parallel computations allows for surprisingly concise and intuitive specifications of complex parallel behaviour.

Blelloch and Sabot (1990) have demonstrated that this highly expressive programming model can be translated into code which runs very efficiently on modern massively parallel systems. Their approach relies on the *flattening transformation* which maps nested data parallel computations to flat ones; the latter can be compiled using well-established techniques. Thus, the task of finding a suitable parallel representation for irregular data structures and transforming the computations accordingly is transferred form the programmer to the compiler.

Due to the high degree of reordering performed by flattening, this compilation technique induces numerous non-obvious restrictions in an imperative language, in particular severely limiting the use of side effects (Pfannenstiel et al., 1998). The picture changes dramatically when flattening is applied to purely functional languages, however. Here, computations may be reordered freely as side effects are provided in a way which does not violate referential transparency (Achten et al., 1993; Peyton Jones and Wadler, 1993). This suggests that the functional paradigm and nested data parallelism can be combined to provide an expressive and efficient framework for parallel programming.

Not surprisingly, the first full implementation of transparent support for nested data parallelism was provided in the functional language NESL (Blelloch, 1995), which demonstrated the feasibility of the approach. Unfortunately, the language suffered from a number of limitations. In particular, it only provided a severely restricted set of types — neither recursive types nor sum types were available — and did not support higher-order functions. This was due to the corresponding shortcomings of the flattening transformation which, as originally presented, could not handle any of these features. Keller and Chakravarty (1998) extended flattening to recursive structures and, later, to algebraic sums (Chakravarty and Keller, 2000). However, even their technique does not treat functions as first-class citizens.

1.2 Higher-order computations

In a purely functional setting, this deficiency is particularily unfortunate — it is precisely the flexible treatment of computations that makes this programming paradigm so expressive and powerful. Moreover, higher-order programming has been demonstrated to be beneficial in a parallel setting, where frequently recurring patterns of parallelism and communication can be captured by skeletons (Cole, 1999; Rabhi and Gorlatch, 2002). Thus, a seamless integration of nested data parallelism into a full-fledged functional language is highly desirable and can be expected to allow for concise, understandable and easily maintainable implementations of

parallel algorithms. This goal can only be achieved by extending the flattening transformation to support arbitrary uses of higher-order functions.

Why, then, has this support not been provided hitherto? Essentially, this shortcoming is due to the way flattening eliminates nested parallelism by changing the representation of data structures and modifying computations specified by the programmer such that they operate on the new representation. This is only possible if computations and data can be manipulated independently of each other. However, the two are usually highly intertwined in functional programs. A prime example are partial applications of curried functions, which implicitly bind a computation, i.e., the function itself, to data encoded in the supplied arguments. The standard lambda calculus does not allow such bindings to be undone, thereby precluding their use in programs which are to be compiled by means of flattening.

In this dissertation, we propose to solve this problem by combining flattening with *closure conversion*, a well-known program transformation which has already been demonstrated to be highly beneficial in a sequential setting. Closure conversion retains the aforementioned connection between computation and data, but makes this connection explicit, providing precisely the necessary degree of separation between the two. Flattening can be extended to make use of this property, allowing it to handle arbitrary programs, including higher-order ones. Thus, the goal of integrating nested data parallelism and functional programming can be achieved without compromising the efficiency of generated code.

1.3 Correctness

No specification of a program transformation is complete without a proof of its correctness. In the case of flattening, a correctness proof is particularily important as the transformation is highly intrusive and its implications not at all obvious. Consequently, we validate our approach by showing that flattening, as defined in this work, is semantics-preserving. This task is significantly complicated by the choice of a non-strict evaluation strategy for the languages we consider, a choice motivated by the intention to ultimately implement nested data parallelism as an extension to Haskell, a popular and mature language with excellent tool support.

One of the main reasons for the efficiency of the code produced by flattening is the heavy use of unboxed arrays. But unboxed arrays are strict and the interactions between the non-strict semantics of the host language and the strictness properties of the generated data structures are quite intricate. In fact, these interactions have not been investigated in any meaningful degree of detail until now, as previous accounts of the flattening transformation either have assumed a call-by-value semantics or have disregarded the termination behaviour of non-strict programs.

In the present dissertation, we provide a novel formalisation of flattening and include a detailed discussion of its semantics in a non-strict setting. In addition to serving as a sound base for establishing the correctness of the transformation, this allows us to formulate a set of requirements on the supporting run-time system which are essential if the generated code is to behave as expected.

1.4 Goals and contributions

To summarise, the main goal of the present dissertation is to provide a sound foundation for the integration of nested data parallelism into a non-strict, purely functional programming language such as Haskell. To this end, contributions are made in the following points.

- Based on closure conversion, the dissertation extends the flattening transformation with support for higher-order and curried functions and demonstrates the flexibility of this strategy by discussing its interactions with various parallel operations. This allows nested data parallelism to be seamlessly integrated into purely functional languages, resulting in a high-level approach to designing and implementing parallel applications.
- It introduces a new form of the lambda calculus which captures the crucial aspects of closure-converted programs. This calculus plays an essential role in the formal development; but it also exhibits a number of properties which lead us to believe that its usefulness is not restricted to the rather specialised area of the compilation of nested data-parallel programs.
- It investigates the intricate implications of compilation by flattening on the semantics of non-strict programs and formalises the results. This clarifies the impact of the transformation on a call-by-name language but also serves as a guideline for future implementations of the approach.
- It includes a proof of correctness for the flattening transformation, with respect to both static and dynamic semantics.

1.5 Overview of the dissertation

The dissertation is organised as follows.

Nested Data-Parallel Programming. Chapter 2 introduces the nested data-parallel programming model as a conservative extension to Haskell and demonstrates its expressiveness. It also describes the underlying abstract model of a parallel machine and shows how it correlates to modern massively parallel systems.

Compiling Nested Data Parallelism. Chapter 3 outlines the compilation strategy and introduces the flattening transformation for first-order programs. It is, for the most part, based on previous work and provides a basis for subsequent discussion.

Higher-Order Functions. Chapter 4 demonstrates how closure conversion and flattening can be combined to compile higher-order NDP programs. It investigates the interactions between higher-order functions and operations on parallel collections and explains their impact on the implementation of nested data parallelism.

Formalising the Approach. Chapter 5 formalises the results of the preceding discussion. It defines the syntax and static semantics of the intermediate languages used in the compilation process and specifies closure conversion and flattening as transformations between these languages. Moreover, it establishes the type correctness of the latter transformation.

Semantics of Flattening. Chapter 6 establishes the operational correctness of the flattening transformation. To this end, it investigates the interactions between flattening and call-by-name evaluation and specifies the operational semantics of the intermediate languages.

Conclusion. Chapter 7 concludes the dissertation by summarising the results and discussing related and future work.

Chapter 2

Nested data-parallel programming

In this chapter, we give an overview of the nested data-parallel programming model and describe the underlying assumptions about the execution platform. Following Chakravarty et al. (2001), we introduce nested data parallelism as an extension to Haskell (Peyton Jones et al., 1999). The extension is conservative in the sense that the semantics of existing Haskell constructs and programs remains unchanged. It seamlessly integrates new syntax for expressing parallelism into the language (Section 2.1) and adds a set of parallel operations to the prelude (Section 2.2).

The choice of Haskell as the host language is easily justified. In the recent years, it has become the de-facto standard purely functional language and is the subject of active research. Moreover, the Glasgow Haskell Compiler (Peyton Jones et al., 1993) is an industrial-strength implementation which we intend to use as a basis for implementing the approach presented in this work.

After describing the extension, we demonstrate the merits of the approach by examining three parallel algorithms and discussing their implementations in the NDP framework. Section 2.3 describes the multiplication of a sparse matrix with a vector as an introductory example. In Section 2.4, we provide an intuitive implementation of Quicksort which crucially relies on transparent support for irregular parallelism. Section 2.5 discusses Wang's algorithm for solving systems of linear equations and explains how a combination of parallel and sequential data structures can be used to specify the desired parallel behaviour. Finally, in Section 2.6 we introduce the abstract machine which underlies the NDP model.

2.1 Parallel arrays

Our extension is based on *parallel arrays*, which are ordered, homogeneous collections of values with strictly parallel semantics. Since parallel arrays are ubiquitous in NDP programs, they enjoy the same level of syntactic support as lists, arguably the most heavily used data structure in Haskell. In particular, the brackets [: and :] denote array expressions and types. Thus, a parallel array containing elements of type τ has the type $[:\tau:]$; an array with n elements can be constructed directly by $[:x_1, \ldots, x_n:]$. An empty array is denoted by [::].

Contrary to earlier accounts of nested data parallelism in non-strict languages (Chakravarty et al., 2001; Chakravarty and Keller, 2000), parallel arrays as used in this work are not head-strict. They do, however, impose strictness constraints on their elements. As these do not affect the following examples, we defer the discussion of strictness to Chapter 6.

It is important to keep in mind that parallel arrays are not lists and, in particular, that they do not have an inductive definition. Rather, we view them as a built-in type with a semantics specialised for the needs of NDP programming. Although it is possible to implement computations on parallel arrays recursively, e.g., by iterating over the indices, the resulting program would be executed sequentially. Parallel computations, instead, rely on primitive operations provided by the standard prelude as described in the next section.

2.2 Parallel operations

Most standard list functions, such as *map* and *replicate*, have parallel versions which are distinguished from their sequential counterparts by the suffix P. Thus, the standard prelude contains, among others, the following declarations:

$$\begin{aligned} &mapP &&:: (\alpha \to \beta) \to [:\alpha:] \to [:\beta:] \\ &repP &&:: Int \to \alpha \to [:\alpha:] \end{aligned}$$

Here, *mapP* maps a function over a parallel array while *repP* creates a new array which contains n copies of a given value. Note that here and in the rest of this work, we deviate slightly from the naming scheme described above and abbreviate the parallel versions of the standard Haskell functions *replicate* and *length* by, respectively, *repP* and *lenP*. Figure 2.1 lists the most important array primitives.

Crucially, these functions have a parallel semantics. Thus, $mapP\ (+1)\ [:1, 2, 3, 4, 5, 6:]$ increments all numbers in the array simulatneously. Analogously, $repP\ n\ 1$ is executed in a single parallel step regardless of the value of n. This already allows us to easily formulate simple parallel algorithms. For instance, $zipWithP\ (*)\ xs\ ys$ denotes the elementwise multiplication of two vectors xs and ys; again, the result is obtained in one parallel step.

It should be noted that some standard list operations have an inherently sequential semantics and do not naturally carry over to a parallel setting. A well-known example are *foldl* and *foldr* which prescribe a strictly left-to-right and right-to-left evaluation order, respectively. If possible, we provide a similar but easily parallelisable operation in such cases. Thus, the prelude includes the primitive

$$foldP\ ::\ (\alpha \to \alpha \to \alpha) \to \alpha \to [:\alpha:]$$

which assumes that the reduction operator is associative and has an obvious parallel implementation.

Parallel array comprehensions. Haskell includes list comprehensions as a convenient alternative to explicit function calls for expressing computations on lists. In the following examples, we make use of array comprehensions which are modelled on Haskell's list syntax but, again, are executed in parallel. For instance,

$$[:sqrt\ x \mid x \leftarrow xs:]$$

simultaneously computes the square root for each element of the parallel array xs. Array comprehensions allow for a more concise formulation of parallel algorithms. However, it is important to realise that they do not add any functionality to the language and can always be replaced by explicit function calls; e.g., the above expression is equivalent to $mapP\ sqrt\ xs$. Chakravarty et al. (2001) provide the exact rules for eliminating comprehensions.

mapP :: $(\alpha \to \beta) \to [:\alpha:] \to [:\beta:]$	Mapping: *mapP f xs* maps the function f over *xs*
repP, *replicateP* :: $Int \to \alpha \to [:\alpha:]$	Replication: *repP n x* yields an array of length n containing x in every position
lenP, *lengthP* :: $[:\alpha:] \to Int$	Length: *lenP xs* yields the length of *xs*
(!:) :: $[:\alpha:] \to Int \to \alpha$	Indexing: *xs* !: *i* yields the element of *xs* at position i (starting with 0) Precondition: $i < lenP\ i$
(+++) :: $[:\alpha:] \to [:\alpha:] \to [:\alpha:]$	Concatenation: *xs* +++ *ys* yields the array obtained by concatenating *xs* and *ys*
zipP :: $[:\alpha:] \to [:\beta:] \to [:(\alpha, \beta):]$	Zipping: *zipP xs ys* yields an array of pairs obtained by elementwise tupling of *xs* and *ys* Precondition: $lenP\ xs == lenP\ ys$
zipWithP :: $(\alpha \to \beta \to \gamma) \to [:\alpha:] \to [:\beta:] \to [:\gamma:]$	Generalised zipping: *zipWithP f xs ys* yields an array obtained by applying f to corresponding elements of *xs* and *ys* Precondition: $lenP\ xs == lenP\ ys$
filterP :: $(\alpha \to \texttt{Bool}) \to [:\alpha:] \to [:\alpha:]$	Filtering: *filterP f xs* yields an array containing those elements of *xs* which satisfy the predicate f
packP :: $[:Bool:] \to [:\alpha:] \to [:\alpha:]$	Packing: *packP bs xs* yields an array containing those elements of *xs* for which the corresponding element in *bs* is *True* Precondition: $lenP\ bs == lenP\ xs$
combineP :: $[:Bool:] \to [:\alpha:] \to [:\alpha:] \to [:\alpha:]$	Combining: *combineP bs xs ys* merges *xs* and *ys* according to the flag vector *bs* Precondition: $falsesP\ bs == lenP\ xs$ $truesP\ bs == lenP\ ys$
foldP :: $(\alpha \to \alpha \to \alpha) \to \alpha \to [:\alpha:] \to \alpha$	Folding: *foldP* $(\oplus)$ *e xs* reduces *xs* with the binary operator $\oplus$ and neutral element e in unspecified order Precondition: $\oplus$ is associative
sumP :: $[:Int:] \to Int$	Sum: *sumP xs* computes the sums of all elements of *xs*
falsesP, *truesP* :: $[:Bool:] \to Bool$	False/True count: *falsesP xs* and *truesP xs* yield the number of *False* and *True* elements, respectively, in *xs*

FIGURE 2.1 Standard array operations

$$\begin{pmatrix} 2.5 & 0 & 0 & 1.4 \\ 0 & 3.7 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 5.2 & 0 & 4.3 & 0 \end{pmatrix} \quad \begin{array}{ll} [: & [:(0, 2.5), (3, 1.4):] \\ & [:(1, 3.7):] \\ & [::] \\ & [:(0, 5.2), (2, 4.3):] \quad :] \end{array}$$

FIGURE 2.2 Representation of a sparse matrix

Nested parallelism. A key property of the NDP model is its transparent support for nested and irregular parallelism. In particular, the nesting level of parallel arrays does not affect the degree of parallelism available in a computation. Thus, the expression

$$mapP\ (mapP\ (+1))\ [:[:1, 2:], [:3, 4, 5:], [::], [:6:]:]$$

is still evaluated in a single step even though it operates on a nested array. This becomes evident if we consider the semantics of this term. When applied to an array, $mapP\ (+1)$ simultaneously increments its elements. Since this function is applied to all element arrays at once, as specified by the outer $mapP$, all incrementations are performed in parallel.

2.3 Sparse matrices

As a first example which demonstrates the expressiveness of our approach, we consider the multiplication of a sparse matrix with a dense vector. The elegant formulation of this algorithm presented below was first introduced by Blelloch (1996). It is based on the *compressed row format*, a well-known, optimised representation of sparse matrices. For each row, it stores only the non-zero elements associated with their indices; a matrix is an array of such rows. The following two data types capture this principle:

$$\begin{array}{l} \textbf{type}\ Row\ \ = [:(Int, Float):] \\ \textbf{type}\ Matrix\ = [:Row:] \end{array}$$

An example is given in Figure 2.2, which depicts a sparse matrix together with its representation.

The multiplication of such a matrix with a dense parallel vector is easily expressed with array comprehensions by nesting three levels of parallel operations.

$$\begin{array}{l} smvm\ \ :: Matrix \to [:Float:] \to [:Float:] \\ smvm\ m\ v\ = [:sumP\ \underbrace{[:x * (v\ !:\ i)\ |\ (i, x)\ \leftarrow\ row:]}_{\text{products of one row}}\ |\ row\ \leftarrow\ m:] \end{array}$$

For each row, the products are computed by the inner comprehension by multiplying the value of the row element with the corresponding element of the vector. The latter has the same index within the vector as the one associated with the row element. Then, $sumP$ computes the sum of the resulting array of products in a parallel reduction. The outer comprehension specifies that this computation is to be applied to all rows simultaneously.

The parallel complexity of the algorithm can be determined informally by making the following observations. For each row, the products are computed simultaneously; since all

rows are processed at once, *all* products are obtained in one parallel step. Each of the following reductions is logarithmic in the number of elements in the corresponding row. Since they, too, are executed simultaneously, the entire computation has the parallel complexity of $\mathcal{O}(log\ n)$ where n is the maximal number of non-zero elements in a row.

Note that the above implementation achieves optimal parallel complexity while still being clear and concise. In fact, it is nearly identical to the sequential, list-based formulation of the algorithm — the latter can be obtained by simply replacing parallel operations, including comprehensions, by their sequential counterparts.

2.4 Quicksort

A slightly more involved example is Quicksort, an algorithm which is frequently used to demonstrate the advantages of functional programming. The following implementation is nearly identical to the one found in many introductory textbooks.

$$
\begin{array}{lcl}
qsort & :: & Ord\ \alpha \Rightarrow [:\alpha:] \to [:\alpha:] \\
qsort\ [::] & = & [::] \\
qsort\ xs & = & \textbf{let} \\
 & & \quad m = xs\ !:\ (lenP\ xs\ `div`\ 2) \\
 & & \quad ls = [:x \mid x \leftarrow xs,\ x < m:] \\
 & & \quad ms = [:x \mid x \leftarrow xs,\ x == m:] \\
 & & \quad gs = [:x \mid x \leftarrow xs,\ x > m:] \\
 & & \quad ss = [:qsort\ ys \mid ys \leftarrow [:ls, gs:]:] \\
 & & \textbf{in} \\
 & & (ss\ !:\ 0)\ +\!\!+\ ms\ +\!\!+\ (ss\ !:\ 1)
\end{array}
$$

Again, we parallelise the algorithm by using parallel arrays instead of lists. We would severely restrict the available parallelism, however, if we embedded the recursion directly in the concatenation of the sorted subarrays, as in $qsort\ ls\ +\!\!+\ ms\ +\!\!+\ qsort\ gs$. In this case, the recursive calls would be executed one after another, which is clearly undesirable. Instead, the two arrays *ls* and *gs* are combined into a single, nested array structure, allowing us to perform the two recursive calls in parallel. Figure 2.3 depicts the evaluation strategy of this implementation. First, the initial array is recursively split into chunks of at most one element according to the quicksort strategy; these are then repeatedly concatenated, ultimately yielding the sorted array. In each recursion level, the operations are performed in parallel on all subarrays as indicated by the grey ovals. Obviously, the average parallel complexity of the implementation is, as expected, logarithmic in the length of the array.

Figure 2.3 illustrates the highly irregular parallel structure of the algorithm. Still, this is of no concern to the programmer — all she has to do is provide a high-level specification of the available parallelism which is then translated into efficient code by the compiler without further intervention. This transparent support for irregular parallelism is, in fact, one of the strongest merits of the NDP model.

2.5 Wang's algorithm

The previous two examples demonstrate that the nested data parallel programming model utilises all available parallelism as specified by the programmer. In particular, in the implementation of Quicksort we had to avoid unnecessary sequentialisation by using appropriate

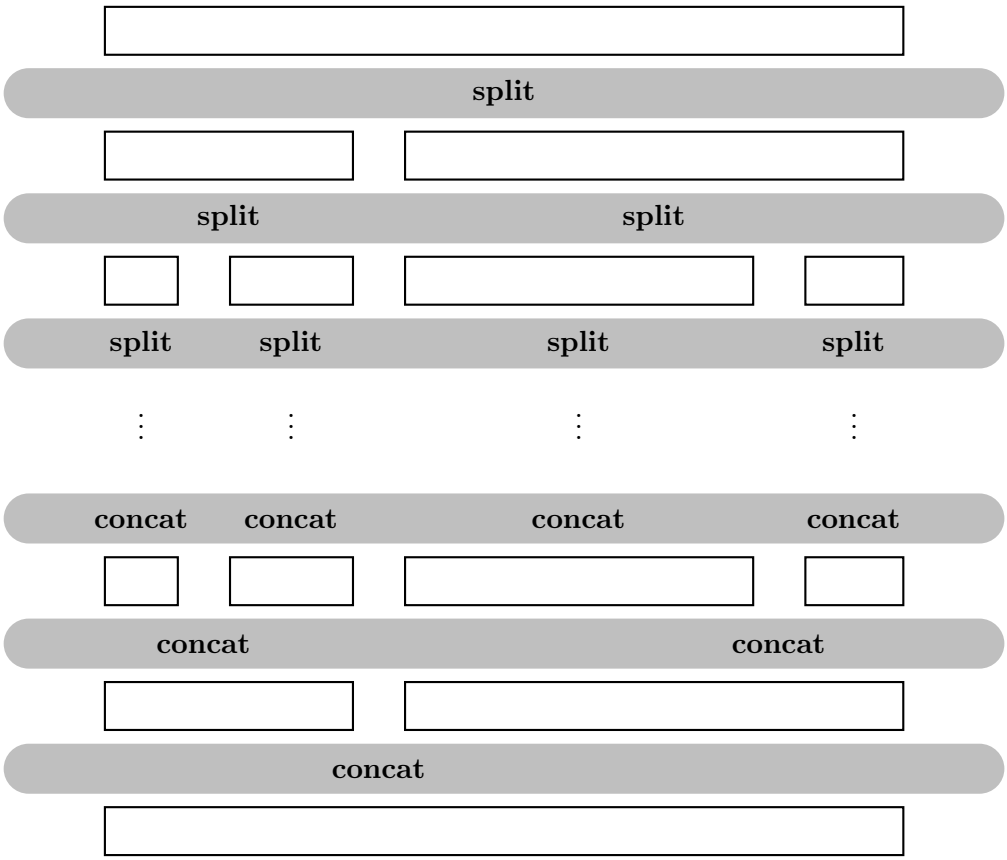


FIGURE 2.3 Evaluation strategy of Quicksort

data structures. Frequently, however, exactly the opposite is desirable — we want to *restrict* the degree of parallelism, either because doing so causes the compiler to generate more efficient code or due to interactions between parallelism and the complexity of an algorithm.

An interesting example of the latter class of algorithms is the one proposed by Wang (1981) for solving tridiagonal systems of linear equations. Here, the matrix representing the system of equations is subdivided into blocks of consecutive rows, as depicted on the left-hand side of Figure 2.4. In the first phase, each block is traversed first from top to bottom and then from bottom to top, eliminating the lower and upper diagonals, respectively, and replacing them by vertical chains of fill-in elements. The resulting matrix is depicted on the right-hand side of the figure. The next phase uses the last row of the first block to eliminate the left fill-in in the first and last row of the second one. The latter is, in turn, propagated to the third block and so on, until the last block is reached. This process is then repeated for the right fill-ins, starting with the last block and propagating the first rows of each block upwards. Finally, the last phase of the algorithm eliminates all fill-ins in each block using the information obtained from adjacent blocks.

In the context of this dissertation, we are only interested in the parallel behaviour exhibited by this algorithm. In the first phase, all blocks can be traversed indepedently — thus, assuming n rows evenly distributed over b blocks, this phase has the parallel complexity $\mathcal{O}(n/b)$. The second phase propagates values from block to block. This computation pattern

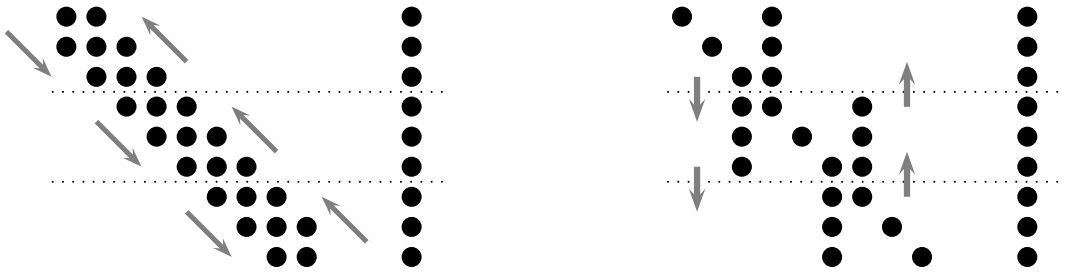


Figure 2.4 Wang's algorithm

is known as *pipelining* and is inherently sequential; even on a parallel machine, the complexity is $\mathcal{O}(b)$. Finally, all blocks can be processed simultaneously in the last phase. This again takes $\mathcal{O}(n/b)$ parallel steps, leading to an overall complexity of $\mathcal{O}(n/b + b)$.

Clearly, by allocating more blocks we can improve the performance of the first and last phases. Unfortunately, this also increases the length of the pipeline in the second phase. In an efficient parallel implementation, the number of blocks should be equal to the number of processors, obtaining the best possible performance in the parallel phases while preventing the pipelining from dominating the run time for sufficiently large matrices. This amounts to artificially restricting the parallelism in the specification of the two parallel phases.

This constraint can be naturally expressed by using a mixture of parallel arrays and sequential data structures, e.g. lists. Then, each block is represented as a list of rows; the blocks themselves, however, are stored in a parallel array.

$$\begin{aligned}&\textbf{type } \mathit{Row} &&= \ldots\\ &\textbf{type } \mathit{Block} &&= [\mathit{Row}]\\ &\textbf{type } \mathit{Matrix} &&= [:\mathit{Block}:]\end{aligned}$$

Note how these definitions precisely capture the structure of the first and last phases, where the blocks are processed in parallel while the rows within a block are traversed one after another. The top-level structure of the algorithm is easily implemented with this representation.

$$\begin{aligned}&\mathit{solve} :: \mathit{Matrix} \to [:[\mathit{Float}]:]\\ &\mathit{solve} = \mathit{mapP}\ \mathit{elimFillIns} \circ \mathit{pipeline} \circ \mathit{mapP}\ \mathit{elimDiagonals}\end{aligned}$$

In general, the ability to combine sequential and parallel data structures allows for very natural and arbitrarily fine-grained specifications of parallel behaviour. This is in stark contrast to low-level approaches, where such computation patterns have to be coded explicitly, leading to programs which are difficult to maintain and often contain a large number of undetected errors.

2.6 The execution model

We conclude the introductory chapter by describing the model of a parallel machine underlying the NDP approach and explaining how it correlates with modern massively parallel computers. Blelloch (1990) introduced the *Vector Random Access Machine* (VRAM) as an abstract

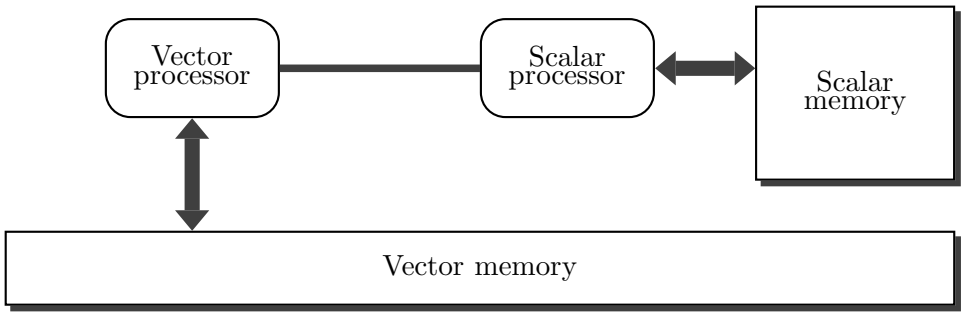


FIGURE 2.5 The VRAM model

architecture with explicit support for data-parallel computations. The VRAM, depicted in Figure 2.5, extends the usual model of a Random Access Machine with a conceptually infinite vector memory and a vector processor which can simultaneously apply an operation to all elements of a collection stored in the vector memory. The scalar components of the VRAM are responsible for computations which are executed outside of the parallel context.

This simple model is sufficient for executing arbitrary NDP programs. The flattening transformation replaces nested parallel computations and data structures by flat ones. In the next chapter, we will see that it also changes the representation of complex parallel arrays such that the resulting code operates only on flat arrays containing elements of primitive types. This eliminates the need for more involved memory and processor models.

The VRAM is largely based on vector computers which still dominated the supercomputing market in 1990 but have been mostly replaced by shared-memory multiprocessors and distributed memory machines, especially clusters, since then. Fortunately, the model is easy to implement on these as well. In the distributed memory case, the vector memory is simulated by distributing collections across all available processors. This is unproblematic as long as the collections do not contain pointer-based elements, a requirement ensured by the flattening transformation. The processors are then synchronised such that at each point, they perform exactly the same computation on the elements of some collection. If this principle is followed strictly, a multiprocessor is essentially reduced to operating as a SIMD machine which frequently leads to suboptimal performance. By selecting suitable distribution strategies and eliminating unnecessary synchronisation points, however, data-parallel programs can be executed very efficiently (Keller and Chakravarty, 1999).

Chapter 3

Compiling nested data parallelism

The compilation of first-order nested data-parallel programs is a fairly well understood process. The core idea is to transform nested data parallelism, used in the previous chapter as a convenient and abstract approach to implementing parallel algorithms, to programs which only contain flat parallel computations and data structures. Such programs can then be translated into efficient code for modern parallel architectures using standard techniques. In this chapter, we outline the relevant mechanisms and point out the aspects of our approach which differ from previous work. These differences are necessary to account for the non-strictness of the source language.

Keller and Chakravarty (1999) formulate the process of compiling nested data parallelism as a series of transformations, each making a program less abstract and bringing it closer to the target machine. In this work, we pursue the same approach, which we describe in Section 3.1. Our transformations are not source-to-source; instead, each transformation performs a translation from a more abstract intermediate language to a lower-level one. This allows us to enforce the constraints and invariants introduced in each phase by encoding them in the grammar of the respective language which significantly simplifies the formalisms in the following chapters. The intermediate languages are based on the lambda calculus and are described in Section 3.2.

The key technique for replacing nested data-parallel computations by flat ones is the flattening transformation, first suggested by Blelloch and Sabot (1990) and later refined and extended by Chakravarty and Keller (2000). It proceeds by selecting an efficient, flat representation for nested data structures and adjusting the computations accordingly. Section 3.3 introduces and justifies the flat representation for standard product-sum types. Section 3.4 describes how primitive operations on this representation are implemented using an approach to generic programming introduced by Hinze (2000). Finally, Section 3.5 explains how user-specified computations are modified to operate on the generated flat data structures.

Although we discuss several new insights into the compilation of NDP programs below, this chapter mainly reviews and explains already known techniques. As these cannot, in general, handle higher-order functions, we do not consider the latter in our exposition, with a few exceptions necessary for providing a context to the discussion. The next chapter expands on the material presented here and describes the approach to compiling higher-order parallelism which constitutes a key contribution of this work. Furthermore, we do not formally define the transformations and languages until Chapter 5; the account here only explains their important aspects to aid the understanding of the formal development.

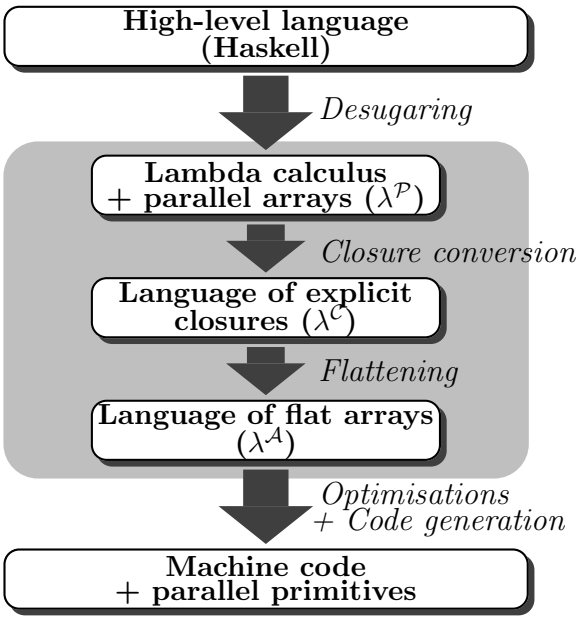


FIGURE 3.1 Compilation of NDP programs

3.1 Compilation by transformation

Compilation by transformation is a well-established approach to compiling high-level languages (Kelsey and Hudak, 1989; Peyton Jones, 1996). It increases the modularity of compilers and makes them easier to develop and maintain. Furthermore, the individual transformations are usually self-contained and, thus, can be specified and reasoned about independently from one another.

Keller and Chakravarty (1999) have demonstrated the merits of this strategy in the context of nested data parallelism. In fact, due to the complexity of the flattening transformation it is particularily important to cleanly separate it from subsequent optimisation passes indespensable for generating efficient machine code. Without this separation, the formalisation of flattening and the proof of its correctness, two major contributions of this dissertation, would not be possible.

Figure 3.1 illustrates the structure of an NDP compiler. A program written in a high-level language — Haskell, in our case, — is compiled to target code by the following steps.

Desugaring translates the program into a much smaller and more manageable core language by eliminating high-level constructs, such as comprehensions, and replacing them by equivalent lower-level code.

Closure conversion separates code from data and allows the two to be manipulated independently by using an explicit representation for function closures and rewriting functions such that they contain no free variables.

Flattening eliminates nested uses of data parallelism, derives an efficient representation for parallel arrays used in the program and transforms the computations such that they operate on the new representation.

Code generation, preceded by several optimisation passes, maps the flat data-parallel program to code directly executable on the target machine.

Of these steps, we only discuss closure conversion and flattening in the present dissertation. Desugaring, which includes various correctness checks such as type checking, is a well-understood process (Aho et al., 1986). Optimisation and generation of target code are the subjects of innumerable textbooks and research papers. In particular, Keller (1999) introduces a number of optimisation techniques specific to nested data parallelism.

3.2 Intermeditate languages

When considering a program transformation an important decision is whether to define it such that it simply rewrites a program without changing the underlying language or to specify it as a translation between two different languages. Neither of the two strategies is unconditionally better than the other. In the former case, only one language has to be defined; in the latter, the constraints and invariants introduced by the transformation can be enforced by the grammar of the target language, obliviating the need for additional validity checks.

For the transformations considered in this work, the second approach is clearly preferable. In particular, the proposed mechanisms for handling higher-order functions within the flattening transformation crucially rely on the separation of code and data achieved by closure conversion, so much so, in fact, that they cannot be specified in a manageable way if this separation is not enforced by the language. Thus, we formulate both closure conversion and flattening as translations between intermediate languages, each of which reflects the specific requirements of the corresponding compilation phase. As depicted in Figure 3.1, we use the following intermediate languages.

- $\lambda^{\mathcal{P}}$ is a variant of the monomorphic lambda calculus extended with parallel arrays. Enhanced lambda calculi are customarily used as a minimal but still sufficiently expressive intermediate representation in functional compilers.

- $\lambda^{\mathcal{C}}$, the language of explicit closures, is based on $\lambda^{\mathcal{P}}$, sharing its support for nested data parallelism. However, it captures the semantics of closure conversion by allowing only one variable to be in scope at a time, thus ruling out currying and partial applications of functions. The latter are replaced by explicit closures, i.e., objects which represent partial applications such that the functions and their arguments can be accessed indepedently.

- $\lambda^{\mathcal{A}}$, the language of flat arrays, shares many of its features with $\lambda^{\mathcal{C}}$, but restricts the support for data parallelism to flat arrays with elements of primitive types. Moreover, its type system is slightly more involved due to the way flattening maps algebraic data types to more efficient representations.

3.2.1 Monomorphism

The description of the intermediate languages immediately raises an important question. Haskell is, of course, a language with strong support for polymorphic and generic programming. Why, then, do we restrict ourselves to the monomorphic lambda calculus? This decision is has been influenced by two factors. First, Haskell's type system is not powerful enough to

transparently handle all aspects of the flattening transformation. We will see below that the type of parallel arrays is not a proper algebraic type constructor but, rather, a *type function*, which is resolved statically during compilation. The mechanisms provided by Haskell are insufficient for handling this kind of polymorphism. Chakravarty et al. (2005a,b) demonstrate how this shortcoming can be rectified and, in fact, use nested data parallelism as a motivating example. Their approach, however, relies on type classes and including this feature in the intermediate languages has been considered impractical.

More importantly, perhaps, the problems discussed in this dissertation are entirely orthogonal to the issue of polymorphism. In particular, we expect the proposed techniques to extend naturally to the second-order lambda calculus (Girard, 1971; Reynolds, 1974), a formalism which is used as the intermediate representation by, among others, the Glasgow Haskell Compiler. Therefore, the inclusion of polymorphism would complicate and obscure the formal development without leading to significant new insights.

3.2.2 Types

Unfortunately, the important aspects of the intermediate languages cannot be justified without explaining the principles underlying closure conversion and flattening. Moreover, while the pure lambda calculus is an excellent internal representation for a compiler, it hardly makes for readable examples. Thus, we do not formally introduce the intermediate languages until Chapter 5. In this and the next chapter, we use a rather informal and more intuitive notation which is based on Haskell's syntax. We are careful, however, not to use any features which cannot be easily rewritten to conform to the grammar of the intermediate languages. In particular, we abandon high-level type definitions in favor of a more basic set of types. These include

- the primitive types `Int`, `Bool` and `Float`,
- the unit type $\langle\rangle$, whose only element is, too, denoted by $\langle\rangle$,
- primitive unboxed arrays $\texttt{Arr}_{\texttt{Int}}$ and $\texttt{Arr}_{\texttt{Bool}}$, which have a parallel semantics and are described in Section 3.3.1,
- products types of the form $\tau_1 \times \tau_2$,
- sum types of the form $\tau_1 + \tau_2$,
- parallel array types $[:\tau:]$,
- μ-recursive types of the form $\mu\alpha.\tau$,
- function types of the form $\tau_1 \to \tau_2$ and
- closure types introduced in Section 4.2.

A crucial part of our approach to supporting higher-order functions in NDP programs is the elimination of currying by means of closure conversion. Consequently, $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$ allow functions only to have one parameter. In the following, we distinguish between curried primitives supported by $\lambda^{\mathcal{P}}$, and uncurried ones provided by the other two intermediate languages by typesetting them in a different font. For instance, *repP* refers to the curried form of replication in $\lambda^{\mathcal{P}}$:

$$repP \;::\; \texttt{Int} \to \alpha \to [:\alpha:]$$

whereas `repP` is the uncurried version:

$$\texttt{repP} \;::\; \texttt{Int} \times \alpha \to [:\alpha:]$$

In general, we assume that the curried versions are implemented in terms of the uncurried ones. Moreover, we will be careful to only use currying when presenting code which has not yet been transformed by closure conversion.

3.3 Representation of arrays

The overriding concern of a programmer implenting a parallel application is efficiency – if performance did not matter she would not be writing a parallel program in the first place. Thus, it is imperative that high-level NDP programs are translated into code which runs very efficiently on the underlying hardware. Unfortunately, functional languages, especially lazy ones, have not been famous for their efficiency. In this section, we investigate the reasons for their suboptimal performance and demonstrate how these can be eliminated in NDP programs.

3.3.1 Unboxed arrays

One of the main reasons for the suboptimal performance of functional languages is the necessity to use *boxed* representations even for primitive types like booleans or integers. A boxed value is represented by a pointer to a heap object storing the object's actual bit pattern. To access an element of such an array the corresponding pointer has to be followed; a traversal of the entire array implies following *all* pointers. Moreover, in a lazy language, an object pointed to from the array has not necessarily been evaluated. Instead, it might be represented by a *thunk*, i.e. a deferred computation which has to be executed before its value can be accessed.

Such a representation is very unfortunate with respect to performance. The additional operations necessary to dereference the pointers already might slow down the computations significantly. Much worse, however, is its impact on the locality of data and code. Modern computers rely on caches and prefetching to avoid expensive memory accesses. Both mechanisms perform best when the traversed data is stored in a contiguous block of memory. However, with a boxed representation only the *pointers* are stored in a contiguous block; the element objects themselves might lie anywhere on the heap. Moreover, evaluating a thunk might cause the execution of an arbitrary amount of code not related to the traversal itself. In the worst case, this will remove the traversal loop from the instruction cache and slow down the computation even further.

This is in stark contrast to an *unboxed* representation, where the array elements are stored directly in the array object. This is the representation used by all programming languages popular in the high-performance field, in particular by C and Fortran. In fact, even functional languages often provide unboxed arrays for this reason (e.g. *UArray* in Haskell). Typically, however, they can only store values of primitive types, but even in this case unboxing has important consequences, trading flexibility for performance. In particular, unboxed types cannot represent thunks, i.e. delayed computations, and are therefore always strict. Moreover, they are of limited use in a polymorphic context. Launchbury and Paterson (1996) and Peyton Jones and Launchbury (1991) discuss these trade-offs in detail.

The above implies that an implementation of NDP has to use unboxed arrays as much as possible if it is to be competitive with other solutions. Another reason for doing so becomes apparent if we take into consideration our target architecture. Data structures containing pointers (such as boxed arrays) are notoriously hard to represent and communicate efficiently on distributed memory machines. Typically, a notion of global addresses is required by not readily available. In contrast to this, distributing unboxed arrays across processors is very easy; we have already described the basic strategy in Section 2.6.

Consequently, parallel arrays in $\lambda^{\mathcal{P}}$, the target language of the flattening transformation, can only store elements of primitive types. This allows them to use an unboxed representation but is still sufficient to express the flat parallelism generated by the flattening transformation. The corresponding array types are denoted by $\texttt{Arr}_\tau$ where τ is a primitive type; terms of these types are constructed by $\{x_1, \ldots, x_n\}$. Note that in contrast to the type constructor $[:\cdot:]$, these types are not polymorphic. Thus, while they support the same set of operations as the more general parallel arrays used so far, these operations can be implemented very efficiently due to their closeness to the underlying hardware as well as to the absence of polymorphism.

3.3.2 Flattening parallel arrays

Blelloch and Sabot (1990) and Chakravarty and Keller (2000) have demonstrated how parallel arrays can be mapped to an efficient representation utilizing only unboxed arrays of primitive types. The key idea is to select a suitable representation based on the type of the array elements. Due to the requirements of a non-strict language, we slightly modify the translation algorithm described in this previous work. To understand the underlying concepts, let us first consider the flattening of non-recursive product-sum types.

Primitive types. For arrays containing elements of primitive types, the translation is straightforward — we simply use the corresponding unboxed arrays. Note, however, that an unboxed array diverges if any of its elements does. In Chapter 6, we will see that the flattening transformation is only correct correct if the length of an array is still available even if its elements diverge. Therefore, the length is stored a separate integer value such that, for example, $[:\texttt{Bool}:]$ is transformed to $\texttt{Int} \times \texttt{Arr}_{\texttt{Bool}}$. Although the intermediate languages only provide `Int` and `Bool` as primitive types, this scheme easily extends to other types, such as `Float`, as long as corresponding unboxed arrays are available. In the following discussion, we will sometimes use `Float` in examples.

Unit type. The type $[:\langle\rangle:]$ forms a special case. Since terms of unit type can only have one value, namely $\langle\rangle$, there is no need to store the individual elements separately. Instead, it is sufficient to know the overall number of the elements and whether any of them diverges. Thus, $[:\langle\rangle:]$ is transformed to $\texttt{Int} \times \langle\rangle$, the second component recording the termination behaviour of the elements.

Products. An array of pairs is represented by a pair of arrays such that one of them contains all left components and the other all right ones. Obviously, both will have the same length as the original array; nevertheless, the length is stored separately as for primitive types and for the same reasons. Thus, $[:\tau_1 \times \tau_2:]$ will be represented as $\texttt{Int} \times [:\tau_1:] \times [:\tau_2:]$. Note that the two arrays representing the first and second components of the pairs will undergo further transformation such that ultimately, only unboxed

$$
\begin{array}{ll}
[:\texttt{Int}:] \Longrightarrow \texttt{Int} \times \texttt{Arr}_{\texttt{Int}} & [:\tau_1 \times \tau_2:] \Longrightarrow \texttt{Int} \times [:\tau_1:] \times [:\tau_2:] \\
[:\texttt{Bool}:] \Longrightarrow \texttt{Int} \times \texttt{Arr}_{\texttt{Bool}} & [:\tau_1 + \tau_2:] \Longrightarrow [:\texttt{Bool}:] \times [:\tau_1:] \times [:\tau_2:] \\
[:\langle\rangle:] \Longrightarrow \texttt{Int} \times \langle\rangle & [:[:\tau:]:] \Longrightarrow [:\texttt{Int}:] \times [:\tau:]
\end{array}
$$

FIGURE 3.2 Transformation of product-sum types

arrays are used in the representation. For instance, the term $[:\langle 1, \mathit{True}\rangle, \langle 2, \mathit{False}\rangle:]$ is transformed to $\langle 2,\ [:1, 2:],\ [:\mathit{True},\ \mathit{False}:]\rangle$ and then to $\langle 2,\ \langle 2,\ \{1, 2\}\rangle,\ \langle 2,\ \{\mathit{True}, \mathit{False}\}\rangle\rangle$.

Sums. The type $[:\tau_1 + \tau_2:]$ is transformed in a similar manner. Again, two arrays of types $[:\tau_1:]$ and $[:\tau_2:]$ are used to store elements of the form `Left` x and `Right` y, respectively. In contrast to products, the two arrays do not in general have the same length since a different number of left and right elements can be present. An array of booleans, the *selector*, denotes for each element whether it has been constructed with `Left` or `Right`.[1] According to this strategy, the term [:`Left` 1, `Right` *False*, `Left` 2:] is represented as $\langle[:\mathit{False},\ \mathit{True},\ \mathit{False}:], [:1, 2:], [:\mathit{False}:]\rangle$. More generally, the type $[:\tau_1 + \tau_2:]$ is transformed to $[:\mathit{Bool}:] \times [:\tau_1:] \times [:\tau_2:]$. Again, parallel arrays are then eliminated completely by recursively applying the transformation.

Nested arrays. At the heart of the flattening transformation lies the elimination of nested arrays. This is achieved by storing all data elements of the subarrays in one flat *data array* and tupling it with a *segment descriptor*, an array which contains the lengths of the individual subarrays. For instance, the nested array $[:[:1, 2:], [::], [:3, 4, 5:]:]$ is transformed to $\langle[:2, 0, 3:], [:1, 2, 3, 4, 5:]\rangle$; the first array is the segment descriptor while the second one is the data array. In general, the type $[:[:\tau:]:]$ is transformed to $[:\texttt{Int}:] \times [:\tau:]$.

Figure 3.2 summarises the transformation rules. Let us return to the encoding of sparse matrices introduced in Section 2.3 to see how this approach works in practice. Figure 3.3 shows the flat representation of the sparse matrix depicted in Figure 2.2 on page 10. Recall that the compressed row format is captured by the type $[:[:\texttt{Int} \times \texttt{Float}:]:]$. In the first step, the nesting is resolved, resulting in the segment descriptor $\langle 4,\ \{2, 1, 0, 2\}\rangle$ (the length component is omitted in Figure 3.3 and in most of the following figures) and the as yet untransformed data array of type $[:\texttt{Int} \times \texttt{Float}:]$. The latter is then translated to $\texttt{Int} \times (\texttt{Int} \times \texttt{Arr}_{\texttt{Int}}) \times (\texttt{Int} \times \texttt{Arr}_{\texttt{Bool}})$, such that the first array contains all indices and the second one the data values.

3.3.3 Strictness

We have already mentioned that parallel arrays are not strict in the sense that their length is still available even if one of their elements diverges. This property is crucial for ensuring the correctness of the transformation. However, the flat representation is based on unboxed arrays which *are* strict in all elements. For instance, the elements of $[:1, \bot, 2:]$ are stored in an unboxed array in the flat representation. Since one of the elements diverges, the entire unboxed array diverges as well. This is precisely why the array's length is always stored separately

[1] This is precisely the reason why a built-in boolean type is needed instead of $\langle\rangle + \langle\rangle$. Of course, a language can still make just the algebraic boolean type available to the programmer and only use the built-in one internally.

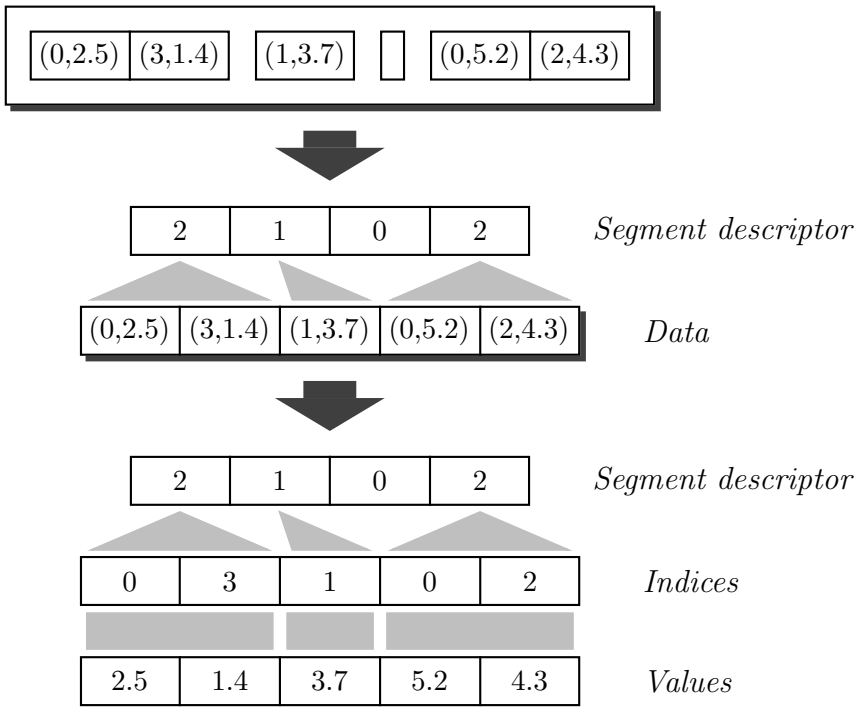


FIGURE 3.3 Flattening of a sparse matrix

from the data, even in cases where this might, at first sight, be unnecessary. Moreover, this obviously has far-reaching consequences for the semantics of nested data-parallel programs in a non-strict setting. A more thorough investigation of the interactions between flattening and strictness is provided in Chapter 6.

3.3.4 Recursive arrays

In Section 2.5, we have considered Wang's algorithm for solving tridiagonal equation systems, encoded by the Haskell type [:[*Row*]:]. The flat representation for such data structures is a natural consequence of the transformation rules introduced so far. In this section, we will consider the type [:[`Int`]:], i.e. parallel arrays of integer lists, to simplify presentation. However, the results are easily transferred to arbitrary sequential structures embedded in parallel arrays.

Integer lists are defined by the type $\mu\alpha.\langle\rangle + \texttt{Int} \times \alpha$. By the definition of μ-recursion, this is equivalent to the fixed point of the equation

$$\mathit{IntList} \;=\; \langle\rangle + \texttt{Int} \times \mathit{IntList}$$

How will an array of such lists be represented? By applying the array constructor to both sides of the equation, we obtain

$$[:\mathit{IntList}:] \;=\; [:\langle\rangle + \texttt{Int} \times \mathit{IntList}:]$$

Here, the right-hand side can be transformed according to the rules introduced in the previous section, yielding

$$[:\mathit{IntList}:] \;=\; [:\texttt{Bool}:] \times [:\langle\rangle:] \times \underbrace{(\texttt{Int} \times [:\texttt{Int}:] \times [:\mathit{IntList}:])}_{[:\texttt{Int} \times \mathit{IntList}:]}$$

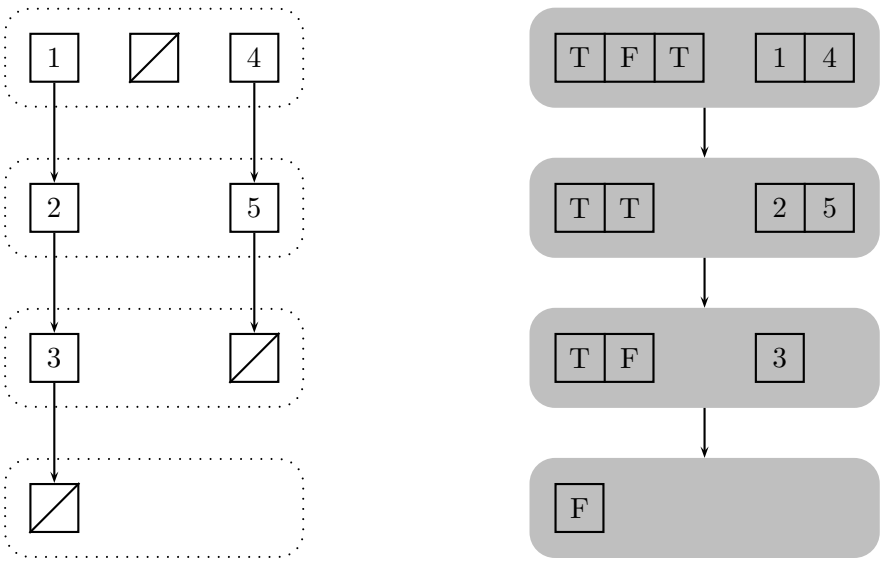


Figure 3.4 Flat representation of an array of integer lists

By substituting a type name, e.g. $\mathit{IntList}^{\uparrow}$, for $[:\mathit{IntList}:]$, we obtain a proper recursive type equation for the type of integer lists

$$\mathit{IntList}^{\uparrow} \;=\; [:\mathit{Bool}:] \times [:\langle\rangle:] \times (\texttt{Int} \times [:\texttt{Int}:] \times \mathit{IntList}^{\uparrow})$$

This is clearly equivalent to the type $\mu\beta.[:\mathit{Bool}:] \times [:\langle\rangle:] \times (\texttt{Int} \times [:\texttt{Int}:] \times \beta)$. Thus, the flat representation of a recursive data type can be obtained by simply applying the relevant transformation rules to its defining equation. The actual transformation introduced in Section 5.5.1 is slightly more involved; however, the underlying principle is the same and we will assume that the above mechanism is used for the rest of this chapter.

It is instructive to investigate the implications this technique has on a program's parallelism and the representation of recursive data structures. Consider, for instance, the term $[:[1,2,3],[],[4,5]:]$. Here, we use $[x_1,\ldots,x_n]$ to denote an *IntList* with n elements. With our definition of *IntList*, this is clearly equivalent to $[:\texttt{Left}\ \langle 1,[2,3]\rangle,\ \texttt{Right}\ \langle\rangle,\ \texttt{Left}\ \langle 4,[5]\rangle:]$, the first and third elements being represented by *cons* nodes and the second one by a *nil* node. By flattening this term using the rule for sum types, we obtain the triple of parallel arrays $\langle[:\mathit{True},\ \mathit{False},\ \mathit{True}:],\ [:\langle\rangle:],[:\langle 1,[2,3]\rangle,\langle 4,[5]\rangle:]\rangle$. Here, the booleans in the selector array indicate whether the corresponding list is non-empty. The second array contains as many $\langle\rangle$ as there are empty lists. Finally, the third one stores the topmost *cons* nodes of the non-empty lists. It is flattened further to $\langle 2,[:1,4:],[:[2,3],[5]:]\rangle$, such that the heads and tails of the non-empty lists are stored in separate arrays; the latter will again be flattened. Ultimately, the flattening transformation will traverse the element lists simultaneously, generating for each level a selector used to distingiushed empty and non-empty lists and an integer array which stores the head elements of the non-empty lists. Figure 3.4 illustrates this mechanism.

This approach has two immediate consequences for computations on recursive data structures in a data-parallel context. First, while all recursive elements of a parallel array will be processed simultaneously, their traversal will happen sequentially: the first elements of all lists will be processed in one step which, however, must be completed before the processing

of the second elements can begin. This gives the programmer a rather fine-grained control over the degree of parallelism exposed in a program. In particular, it is possible to enforce sequential processing of data by storing it in a sequential data structure — this is precisely the technique used in the encoding of Wang's algorithm.

A somewhat subtle limitation of this transformation becomes apparent if we contrast the definition of *IntList* with the type equation derived for $\mathit{IntList}^{\uparrow}$. The former uses a sum type as an anchor for the recursion. However, there is no such anchor in the latter — it defines a recursive type with only infinite elements. In fact, flattening as discussed in this work will generate infinite types whenever type recursion is used in a parallel context. This is not a problem in a non-strict language which can deal with corresponding infinite terms. In a strict language, however, sum types would have to be injected into the flat representation to provide termination points for the recursion.

3.4 Operations on the flat representation

The transformations described in the previous sections illustrate an important aspect of our approach: the type of parallel arrays is not algebraic and does not have a parametrically polymorphic definition. Instead, it is treated as a function over types which is resolved at compile time by the flattening transformation. It is precisely this property which enables the selection of the most efficient representation for each type of array elements.

Unfortunately, this translation scheme significantly complicates the implementation of primitive operations on parallel arrays. To see why, let us compare parallel arrays with standard algebraic types, such as lists.

In Haskell, the type of generic lists can be defined as follows:

$$\textbf{data}\ \mathit{List}\ \alpha\ =\ \mathit{Nil}\mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$$

The shape of the data structure is fixed by this definition and does not depend on the concrete type substituted for α. Consequently, a single, generic implementation of a function over lists, such as *length*, can be provided which is valid for all possible instantiations of *List*. Crucially, this is only possible because the definition of *List* is parametrically polymorphic, i.e. all lists share the same representation *regardless of their element type*. This is not true for parallel arrays, however, as their representation depends on the concrete instantiation. Thus, no single implementation of, e.g., `lenP` exists which works for arrays of products as well as for arrays of sums.

A closer look at the transformation rules reveals, however, that a finite number of implementations is sufficient to completely specify the semantics of `lenP`. Recall, for instance, that arrays of products are represented as $\texttt{Int} \times [:\tau_1:] \times [:\tau_2:]$, with the `Int` component storing the array's length. Thus, it is possible to provide a definition of `lenP` which, while specialised for arrays of products, is generic over all possible instantiations of such arrays.

$$\begin{aligned}&\texttt{lenP\_prod} :: [:\alpha \times \beta:] \to \texttt{Int}\\ &\texttt{lenP\_prod}\ \langle n, \langle xs, ys\rangle\rangle\ =\ n\end{aligned}$$

The validity of this definition depends crucially on the fact that it exhibits precisely the same degree of genericity as the corresponding transformation rule. We can describe the semantics of `lenP` by providing one such definition for each transformation rule. By doing so,

though, we end up with a family of functions which, while related on the semantical level, are completely independent entities in the language. This is clearly not desirable. The situation can be rectified by extending the language with a mechanism for passing *type arguments* to functions and for using them to select an appropriate computation path. Then, a generic version of `lenP` is easily obtained by combining the individual, type-specific implementations into a single definition.

$$
\begin{array}{ll}
\texttt{lenP}_{\langle\tau\rangle} & :: [:\tau:] \to \texttt{Int} \\
\texttt{lenP}_{\langle\langle\rangle\rangle}\ \langle n, u\rangle & = n \\
\texttt{lenP}_{\langle\texttt{Bool}\rangle}\ \langle n, bs\rangle & = n \\
\texttt{lenP}_{\langle\texttt{Int}\rangle}\ \langle n, is\rangle & = n \\
\texttt{lenP}_{\langle\tau_1\times\tau_2\rangle}\ \langle n, \langle xs, ys\rangle\rangle & = n \\
\texttt{lenP}_{\langle\tau_1+\tau_2\rangle}\ \langle bs, \langle xs, ys\rangle\rangle & = \texttt{lenP}_{\langle\texttt{Bool}\rangle}\ bs \\
\texttt{lenP}_{\langle[:\tau:]\rangle}\ \langle ss, xs\rangle & = \texttt{lenP}_{\langle\texttt{Int}\rangle}\ ss
\end{array}
$$

Here, we use angle brackets to denote type parameters; moreover, we use pattern matching on types in the definition. This notation follows Hinze (2000), who has demonstrated that such *type-indexed* functions can be supported in a functional language and are, in fact, useful in their own right. Note that under this definition, the name `lenP` does not refer to a function of polymorphic type $\forall\alpha.[:\alpha:] \to \texttt{Int}$. Rather, it denotes a *family* of functions such that each instantiation $\texttt{lenP}_{\langle\tau\rangle}$ is a monomorphic function of type $[:\tau:] \to \texttt{Int}$. This subtle distinction is crucial in the context of this work as our intermediate languages do not support polymorphism.

The above implementation relies on the transformation rules introduced previously, yielding the length component for arrays of primitive types and of products, the length of the selector for arrays of sums and the length of the segment descriptor for nested arrays. In general, type-indexed functions are defined by induction on the structure of their type parameters such that each case is handled by a specialised algorithm. As a more involved example, let us consider the following definition of replication:

$$
\begin{array}{ll}
\texttt{repP}_{\langle\tau\rangle} & :: \texttt{Int} \times \tau \to [:\tau:] \\
\texttt{repP}_{\langle\langle\rangle\rangle}\ \langle n, \langle\rangle\rangle & = \langle n, \langle\rangle\rangle \\
\texttt{repP}_{\langle\texttt{Bool}\rangle}\ \langle n, b\rangle & = \langle n, \texttt{rep}_{\texttt{Bool}}\ \langle n, b\rangle\rangle \\
\texttt{repP}_{\langle\texttt{Int}\rangle}\ \langle n, i\rangle & = \langle n, \texttt{rep}_{\texttt{Int}}\ \langle n, i\rangle\rangle \\
\texttt{repP}_{\langle\tau_1\times\tau_2\rangle}\ \langle n,\ \langle x, y\rangle\rangle & = \langle n,\ \langle \texttt{repP}_{\langle\tau_1\rangle}\ \langle n, x\rangle,\ \texttt{repP}_{\langle\tau_2\rangle}\ \langle n, y\rangle\rangle\rangle \\
\texttt{repP}_{\langle\tau_1+\tau_2\rangle}\ \langle n,\ \texttt{Left}\ x\rangle & = \langle \texttt{repP}_{\langle\texttt{Bool}\rangle}\ \langle n, \mathit{False}\rangle,\ \langle \texttt{repP}_{\langle\tau_1\rangle}\ \langle n, x\rangle,\ [::]\rangle\rangle \\
\texttt{repP}_{\langle\tau_1+\tau_2\rangle}\ \langle n,\ \texttt{Right}\ y\rangle & = \langle \texttt{repP}_{\langle\texttt{Bool}\rangle}\ \langle n, \mathit{True}\rangle,\ \langle [::],\ \texttt{repP}_{\langle\tau_2\rangle}\ \langle n, y\rangle\rangle\rangle \\
\texttt{repP}_{\langle[:\tau:]\rangle}\ \langle n,\ xs\rangle & = \langle \texttt{repP}_{\langle\texttt{Int}\rangle}\ \langle n,\ \texttt{lenP}_{\langle\tau\rangle}\ xs\rangle,\ \texttt{repeatP}_{\langle\tau\rangle}\ \langle n,\ xs\rangle\rangle
\end{array}
$$

Here, we assume that $\texttt{rep}_{\texttt{Bool}}$ and $\texttt{rep}_{\texttt{Int}}$ generate corresponding unboxed arrays. In contrast to `lenP`, the definition of `repP` make extensive use of recursion. For instance, replicating a tuple of type $\tau_1 \times \tau_2$ to an array of tuples of flat type $\texttt{Int} \times [:\tau_1:] \times [:\tau_2:]$ is performed by replicating the two tuple components to the required length. Binary sums are replicated in a similar way; in this case, however, the sum constructor determines the shape of the selector and the array which will hold the replicated element. Arrays are replicated to nested arrays of the form $\langle ss, ys\rangle$, where ss is the segment descriptor and ys the data array. The former is easily obtained by replicating the length of the argument array. The latter is computed by means of the primitive `repeatP` which repeats an array n times such that, e.g.,

$$\texttt{repeatP}\ \langle 3, [:4, 5:]\rangle\ =\ [:4, 5, 4, 5, 4, 5:]$$

Of course, `repeatP` can be implemented by repeatedly concatenating the argument array. This would be rather inefficient, however. A better solution is to use the type-indexed approach again, achieving constant parallel complexity.

3.5 Transforming computations

The primary method of performing parallel computations in the NDP model is the simultaneous application of a function to every element of a parallel array, a pattern captured by the *mapP* combinator. However, an efficient and conceptually clean implementation of *mapP* is far from trivial and requires significant support from the compiler. This becomes obvious if we consider that in the flat representation, the individual array elements are not stored separately; instead, the data is distributed over a potentially large number of primitive arrays. Thus, the traversal of the array implied by the conceptual semantics of *mapP* simply cannot be implemented directly.

An obvious solution would be to *reconstruct* each array element (perhaps by means of indexing) before applying the function to it and flatten the results of the application. However, this approach loses all benefits of the flat representation; in fact, we would be better off using boxed values in the first place.

Fortunately, *mapP* can be implemented much more efficiently by making use of the following property of flattening. Given a function f of type $\tau_1 \to \tau_2$, it is possible to automatically generate a *lifted* function $f^\uparrow$ of type $[{:}\tau_1{:}] \to [{:}\tau_2{:}]$ which, when applied to a flattened array, will compute precisely the result of applying f to each element of that array. This implies that the following equality holds:

$$mapP\ f\ =\ f^\uparrow$$

In fact, this is essentially the implementation of *mapP* used in this work. Crucially, $f^\uparrow$ works directly on the flat array representation thereby avoiding the inefficiencies described above.

3.5.1 Lifting

The lifted version $f^\uparrow$ of a function f is obtained by means of the *lifting transformation*. The transformation is purely syntactical, i.e. $f^\uparrow$ is derived from the implementation of f without any additional information. Crucially, lifting as defined in this dissertation can only handle uncurried functions.[2] The basic idea is surprisingly simple and is best demonstrated with an example.

Consider the following function which, for a floating-point value, computes the sinus of its square root.

$$\begin{aligned} f\ &::\ \texttt{Float} \to \texttt{Float} \\ f\ x\ &=\ sin\ (sqrt\ x) \end{aligned}$$

Our goal is to derive a lifted version which will perform the same computation on each element of an array. We will make the basic assumption that a lifted version is available or will be made available through lifting for every function used in the program, including standard

[2] While Chakravarty and Keller (2000) provide a specification of lifting which does not exhibit this restriction, their approach to flattening cannot handle some uses of partially applied functions.

primitives. This implies, in particular, the existance of the functions $sqrt^\uparrow$ and $sin^\uparrow$ which, given an array of numbers, compute the square root and the sinus, respectively, for each of them. The lifted version of $f^\uparrow$ can then be obtained simply by substituting these functions into the implementation of f:

$$\begin{aligned} f^\uparrow &:: [:\texttt{Float}:] \to [:\texttt{Float}:] \\ f^\uparrow\ x &= sin^\uparrow\ (sqrt^\uparrow\ x) \end{aligned}$$

The following function, which computes $3x+y$, represents a more complex example due to the inclusion of constants and binary function applications. Here, addition and multiplication have the uncurried type $\texttt{Int} \times \texttt{Int} \to \texttt{Int}$; thus, these operations must be applied to tuples which store both arguments.

$$\begin{aligned} g &:: \texttt{Int} \times \texttt{Int} \to \texttt{Int} \\ g\ x &= (+)\ \langle (*)\ \langle 3, fst\ x\rangle,\ snd\ x\rangle \end{aligned}$$

The simple lifting strategy employed in the previous example breaks down in this case. For instance, the lifted multiplication operator $*^\uparrow$ has the type $[:\texttt{Int} \times \texttt{Int}:] \to [:\texttt{Int}:]$; thus, if $*^\uparrow$ is to replace $*$ in the definition, it must be applied to an array of tuples derived from the term $\langle 3, fst\ x\rangle$. We do not yet know how to lift the latter, however, because the tuple constructor $\langle\cdot,\cdot\rangle$ cannot be a function — in contrast to Haskell, where it essentially has the type $\alpha \to \beta \to \alpha \times \beta$, it cannot be assigned a meaningful type if currying is disallowed. Instead, our intermediate languages include tuples as a built-in feature and, in particular, treat the tuple constructor as a syntactic form rather than a primitive function.

In lifting such terms, we make use of the following observation: a lifted function operates in a parallel context determined by the shape of its argument. In other words, the length of the argument array fixes the lengths of the arrays involved in the evaluation of the function. For the above example, this leads to the following lifting strategy:

- the constant 3 is replicated to the length of the parallel context,
- the term $fst\ x$ is transformed as before, by replacing fst with $fst^\uparrow$,
- the tuples are lifted by zipping their lifted components.

This results in the following code derived for $g^\uparrow$:

$$\begin{aligned} g^\uparrow &:: [:\texttt{Int} \times \texttt{Int}:] \to \texttt{Int} \\ g^\uparrow\ x &= (+^\uparrow)\ (\texttt{zipP}\ \langle (*^\uparrow)\ (\texttt{zipP}\ \langle \underbrace{\texttt{repP}\ \langle \texttt{lenP}\ x, 3\rangle}_{\text{lifted } 3},\ fst^\uparrow\ x\rangle),\ snd^\uparrow\ x\rangle) \end{aligned}$$

To see that this implementation has the desired behaviour, let us consider how the application of $g^\uparrow$ to an example array is evaluated.

$$\begin{aligned} & g^\uparrow\ [:\langle 4,5\rangle, \langle 6,7\rangle:] \\ =\ & (+^\uparrow)\ (\texttt{zipP}\ \langle (*^\uparrow)\ (\texttt{zipP}\ \langle \texttt{repP}\ \langle \texttt{lenP}[:\langle 4,5\rangle, \langle 6,7\rangle:], 3\rangle,\ fst^\uparrow\ [:\langle 4,5\rangle, \langle 6,7\rangle:]\rangle), \\ & \qquad\qquad snd^\uparrow [:\langle 4,5\rangle, \langle 6,7\rangle:]\rangle) \\ =\ & (+^\uparrow)\ (\texttt{zipP}\ \langle (*^\uparrow)\ (\texttt{zipP}\ \langle \texttt{repP}\ \langle 2,3\rangle,\ [:4,6:]\rangle), [:5,7:]\rangle) \\ =\ & (+^\uparrow)\ (\texttt{zipP}\ \langle (*^\uparrow)\ [:\langle 3,4\rangle, \langle 3,6\rangle:], [:5,7:]\rangle) \\ =\ & (+^\uparrow)\ (\texttt{zipP}\ \langle [:12,18:], [:5,7:]\rangle) \end{aligned}$$

$$= (+^{\uparrow})\ [:\langle 12, 5\rangle, \langle 18, 7\rangle:]$$
$$= [:17, 25:]$$
$$(= [:4 * 3 + 5,\ 6 * 3 + 7:])$$

In general, given a term of type τ, lifting generates a term of type $[:\tau:]$ which evaluates to an array whose length is determined by the parallel context. The only exception are function applications, where it simply replaces the function by its lifted version.

3.5.2 Vectorisation

The remaining question is how lifted functions are associated with their sequential counterparts. Clearly, if $mapP\ f$ is to evaluate to $f^{\uparrow}$, there must exist a mapping from f to $f^{\uparrow}$. NESL achieves this by providing only named, first-order functions and replacing the *mapP* combinator with a form of array comprehensions. The expression

$$\{f\ x\ :\ x\ \mathbf{in}\ xs\}$$

(apply f to each element of xs in parallel) is then transformed to $f^{\uparrow}\ xs$ based on purely textual manipulation of function names.

This method is unsatisfactory, however, if we want to be able to support both unnamed and higher-order functions. Chakravarty and Keller (2000) suggest a different approach. The entire program is *vectorised* by tupling each function with its lifted version. Ordinary function application are translated to applications of the tuple's first (sequential) component; applications in array contexts use the second (lifted) component. The essential steps of this *vectorising* transformation are as follows (here, τ' and e' denote the vectorised version of a type τ and a term e, respectively).

- Function types of the form $\tau_1 \to \tau_2$ are replaced by $(\tau_1' \to \tau_2') \times ([:\tau_1':] \to [:\tau_2':])$.
- A lambda term $\lambda x.e$ is replaced by the tuple $\langle \lambda x.e', \lambda x.e^{\uparrow} \rangle$, where $e^{\uparrow}$ is obtained by lifting e to the context x as described above.
- Applications of the form $e_1\ e_2$ are replaced by $(fst\ e_1')\ e_2'$, thus applying the sequential function to the argument.
- Lifted functions are used in parallel contexts as necessary.

This strategy maintains the required association between sequential and lifted versions of functions but does not impose any restrictions on how they can be used. While it does introduce some inefficiencies by effectively enlarging the representation of functions at run time, we expect these to be neglectable and, in many cases, easy to eliminate by subsequent optimisations.

3.5.3 Lifted primitives

The requirement that a lifted version must exist for every function in a program obviously applies to primitives, including type-indexed ones. Unfortunately, the lifted versions of the latter cannot be derived by lifting their definitions. This is because type-indexed primitives operate on the flat representation of parallel arrays, whereas lifting, as part of the flattening transformation, only applies to computations operating on the nested representation. Thus,

for every type-indexed primitive the standard prelude has to provide a lifted version with the correct semantics.

For `lenP`, this is easily done. Given a nested array, its lifted version $\texttt{lenP}^{\uparrow}$ computes the lengths of its subarrays. This information is already stored in the segment descriptor:

$$\begin{array}{ll} \texttt{lenP}^{\uparrow} & :: \overbrace{[:\texttt{Int}:] \times [:\alpha:]}^{[:[:\alpha:]:]} \to [:\texttt{Int}:] \\ \texttt{lenP}^{\uparrow}\ \langle ss, xs \rangle & = ss \end{array}$$

Other lifted primitives, however, can only be defined using the type-indexed approach. We do not discuss their definitions here and refer the interested reader to Keller's (1999) thesis.

Chapter 4

Higher-order functions

One of the key aspects of functional programming languages is the treatment of functions as first-class citizens. Functions can be passed as arguments to other functions, returned as results and stored in data structures. This flexibility is one of the features (if not the main one) which make functional languages so attractive (Hughes, 1989).

This freedom comes at a cost, though. In a typical functional program, computation and data are highly intertwined. When translating NDP programs, however, it is necessary to separate the two such that they can be manipulated independently. A prime example are partially applied functions which occur whenever a curried function is applied to fewer arguments than it expects. Partial application are unproblematic as long as they are not used in a parallel context. However, the ability to treat such functions as data implies that they can be stored in parallel arrays on which arbitrary array computations can be performed.

Unfortunately, the techniques used in the previous chapter for deriving flat representations of parallel arrays are not directly applicable to arrays of functions. This is due to the fact that ultimately, such arrays contain computations which, even in a functional language, cannot be broken up into primitive items in the same way as was done for product-sum types in the previous chapter. Closure conversion has already been introduced as a solution to this problem. While still not allowing us to deconstruct computations, it separates them from the embedded data for which a flat representation can be derived. This is achieved by representing partial applications as *closures*, which store the function and the argument separately and allow them to be accessed and manipulated independently from each other.

This turns out to be sufficient for the purposes of compiling nested data parallelism, even in the presence of higher-order functions. In Section 4.1 we provide a more detailed motivation for closure conversion by demonstrating how computations on arrays of functions can be rewritten to operate on the already bound arguments of these functions, provided the latter can be accessed at runtime. Section 4.2 describes closure conversion and introduces some aspects of the two intermediate languages $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{P}}$ not mentioned in the previous chapter.

A key aspect of this transformation is that it replaces all functions in a program, including those stored in arrays, by closures. In Section 4.3, we derive a flat representation for arrays of closures generated in this way. Importantly, we do so by relying on parametric properties of array operations rather than on ad-hoc mechanisms. Section 4.5 uses the same well-founded approach to introduce the elimination of nested parallelism based on closures. Finally, in Section 4.6 we describe a representation for a limited form of arrays of functions. These

do not occur in closure-converted programs and are only intended for internal use by the compiler. While not strictly necessary, they simplify the formalisation of flattening in the next chapter.

As in the previous chapter, we have tried to keep the account fairly informal. In Chapters 5 and 6, the techniques and concepts introduced here are formalised and their correctness proved.

4.1 Computation and data

Why are the techniques introduced in the previous section not sufficient to handle higher-order functions? In Section 3.5.1, we have described how sequential computation can be transformed into data-parallel ones such that they operate on entire arrays instead of individual elements. The key idea was to generate, for every function occuring in the program, a version which is lifted into array space. Thus, given a function f of type $\tau_1 \to \tau_2$, its lifted version $f^\uparrow$ would have the type $[:\tau_1:] \to [:\tau_2:]$.

This works well as long as functions are first-order. Consider, however, what happens if τ_1 or τ_2 contain function types, as in the following example:

$$\mathit{apply} \;::\; (\alpha \to \beta) \to \alpha \to \beta$$

According to the strategy described earlier, the lifted function $\mathit{apply}^\uparrow$ will have the following signature:

$$\mathit{apply}^\uparrow \;::\; [:\alpha \to \beta:] \to [:\alpha \to \beta:]$$

Here, lifting introduces arrays of functions and, obviously, it does so for every higher-order function, including curried ones. While innocuous at first sight, this property of lifting is the very reason why higher-order functions are not supported by the flattening transformation.

4.1.1 Mutable computations

To see why arrays of functions are so problematic, let us consider case distinction as an example, since this operation highlights the major difficulties. The language $\lambda^\mathcal{P}$ provides the primitive

$$\texttt{case} \;::\; ((\alpha \to \gamma) \times (\beta \to \gamma)) \times (\alpha + \beta) \to \gamma$$

as the only means of inspecting and deconstructing sum types. Consider the following function:

$$\begin{aligned} &f \;::\; \texttt{Int} \times (\texttt{Int} + \texttt{Int}) \to \texttt{Int} \\ &f\,\langle m, \texttt{Left}\; n\rangle \;=\; m + n \\ &f\,\langle m, \texttt{Right}\; n\rangle \;=\; n \end{aligned}$$

By eliminating pattern matching and using `case` for case distinction, it can be rewritten to conform to the grammar of $\lambda^\mathcal{P}$:

$$f \;=\; \lambda x.\, \texttt{case}\,\langle\langle \lambda n.\, \texttt{fst}\; x \;+\; n,\; \lambda n.\, n\rangle,\; \texttt{snd}\; x\rangle$$

From this implementation, the flattening transformation must derive the lifted function $f^\uparrow$. Using the strategy described in Section 3.5.1, we obtain:

$$\begin{aligned} f^\uparrow &:: \texttt{Int} \times [:\texttt{Int}:] \times (\overbrace{[:\texttt{Bool}:] \times [:\texttt{Int}:] \times [:\texttt{Int}:]}^{[:\texttt{Int}+\texttt{Int}:]}) \to [:\texttt{Int}:] \\ f^\uparrow &= \lambda x.\, \texttt{case}^\uparrow\, (\texttt{zipP}\, \langle \texttt{zipP}\, \langle (\lambda n.\, \texttt{fst}\, x\, +\, n)^\uparrow,\, (\lambda n.\, n)^\uparrow \rangle,\, \texttt{snd}^\uparrow\, x \rangle) \end{aligned}$$

The type of $f^\uparrow$ is obtained by flattening $[:\texttt{Int} \times \texttt{Int} + \texttt{Int}:] \to [:\texttt{Int}:]$. Note that $\texttt{case}^\uparrow$ is, essentially, applied to three arguments:

- the function array denoted by $(\lambda n.\, \texttt{fst}\, x\, +\, n)^\uparrow$,
- the function array denoted by $(\lambda n.\, n)^\uparrow$ and
- the array of sums computed by $snd^\uparrow\, x$.

This is consistent with the type of $\texttt{case}^\uparrow$, which is easily derived from the signature of `case`:

$$\texttt{case}^\uparrow :: \texttt{Int} \times (\underbrace{\texttt{Int} \times [:\alpha \to \gamma:] \times [:\beta \to \gamma:]}_{[:(\alpha\to\gamma)\times(\beta\to\gamma):]}) \times (\underbrace{[:\texttt{Bool}:] \times [:\alpha:] \times [:\beta:]}_{[:\alpha+\beta:]}) \to [:\gamma:]$$

How, then, can the lambda abstractions $\lambda n.\, \texttt{fst}\, x\, +\, n$ and $\lambda n.\, n$ be lifted? A first attempt might be to simply employ the same algorithm used for lifting f, such that

$$\begin{aligned} (\lambda n.\, \texttt{fst}\, x\, +\, n)^\uparrow &= \lambda n.\, \texttt{fst}^\uparrow\, x\, +^\uparrow\, n \\ (\lambda n.\, n)^\uparrow &= \lambda n.\, n \end{aligned}$$

This would imply that an array of functions of type $[:\tau \to \upsilon:]$ would be represented by a function over arrays of type $[:\tau:] \to [:\upsilon:]$ and, indeed, Chakravarty and Keller (2000) observe that there is a strong connection, though not quite an isomorphism, between the two types.

Observe, however, what happens if $f^\uparrow$ is applied to an argument such as, for instance, the array $[:\langle 1, \texttt{Left}\, 5\rangle,\, \langle 3,\, \texttt{Right}\, 4\rangle,\, \langle 7,\, \texttt{Left}\, 2\rangle:]$. By the usual β-reduction rules, this array is substituted for x, which implies that the following arguments are passed to $\texttt{case}^\uparrow$:

- the lifted function $\lambda n.\, [:1, 3, 7:]\, +^\uparrow\, n$,
- the lifted function $\lambda n.\, n$ and
- the array $[:\texttt{Left}\ 5,\ \texttt{Right}\ 4,\ \texttt{Left}\ 2:]$, represented as $\langle [:\mathit{False}, \mathit{True}, \mathit{False}:], \langle [:5, 2:], [:4:]\rangle\rangle$.

The result of this computation should be $[:6, 4, 9:]$ but, unfortunately, there exists no implementation of $\texttt{case}^\uparrow$ which could compute this result. This is because the two lifted functions passed to $\texttt{case}^\uparrow$ represent the computations which should be performed on the left and right components of the array of sums. In particular, $\lambda n.\, [:1, 3, 7:]\, +^\uparrow\, n$ encodes the computation which must be applied to the array $[:5, 2:]$; but this application leads to an error, as it ultimately invokes $[:1, 3, 7:] +^\uparrow [:5, 2:]$, thereby violating the precondition that lifted functions always operate on arrays of the same length.

This problem arises because $+^\uparrow$ should only be applied to those tuples stored in x for which the second component has been constructed with `Left`, but $[:1, 3, 7:]$ is computed by $\texttt{fst}^\uparrow\, x$ and, therefore, contains the first components of *all* tuples and has too many elements. This is, unfortunately, a necessary consequence of the way substitution works in the standard

lambda calculus. The abstraction $\lambda n.\, \texttt{fst}^{\uparrow}\ x\ +^{\uparrow}\ n$ contains references to the parameter x of the outer function $f^{\uparrow}$. As soon as x is bound to some value, this value also gets substituted into the abstraction, thereby fixing the first argument to $+^{\uparrow}$. The binding happens too early, however, as the array which $+^{\uparrow}$ should be applied to is not yet known at this point.

Conceptually, the function $\lambda n.\, \texttt{fst}^{\uparrow}\ x\ +^{\uparrow}\ n$ encodes a computation for every element of the array of sums; however, only some of these should be executed due to the semantics of case distinction. It turns out that this problem cannot be solved in the framework of the standard lambda calculus since here, substitution and, hence, β-reduction can *mutate* computations by replacing variables with data in lambda abstractions. The difficulties only arise when the computation is intertwined with data, i.e., when a concrete array is substituted for x. In the above example, the function $\lambda n.\, \texttt{fst}^{\uparrow}\ x\ +^{\uparrow}\ n$ encodes *almost* the correct computation. This observation can be made explicit by replacing it with the equivalent term $(\lambda x.\, \lambda n.\, \texttt{fst}^{\uparrow}\ x\ +^{\uparrow}\ n)\ x$. Now, the abstraction represents *precisely* the correct computation which, however, is partially applied to an incorrect argument.

4.1.2 Functions in arrays

The previous discussion demonstrates that arrays of functions cannot be represented by functions over arrays. This becomes even more obvious if we consider that the former must support all array operations, such as concatenation and packing, a flexibility which the latter cannot easily provide.

In fact, the semantics of $\texttt{case}^{\uparrow}$ crucially relies on this support. While Haskell's type system — and, consequently, the one adopted in this work, — is not powerful enough to express the requirements on the lengths of array arguments, dependent types (Augustsson, 1998; Xi and Pfenning, 1999) easily allow us to do so. For instance, using a rather informal notation, we could write $n \times [:\texttt{Int}:]_n$ to denote the type of tuples $\langle n, xs\rangle$ such that n is an integer and xs an integer array of length n. Then, $\texttt{case}^{\uparrow}$ could be assigned the following type:

$$\texttt{case}^{\uparrow}\ ::\ n \times (n \times [:\alpha \to \gamma:]_n \times [:\beta \to \gamma:]_n) \times ([:\texttt{Bool}:]_n \times [:\alpha:]_p \times [:\beta:]_q) \to [:\gamma:]_n$$

This type captures the requirement that $\texttt{case}^{\uparrow}$, as every lifted primitive, expects all its arguments to have the same length. This does not hold for all arrays passed to it, however, as exemplified by the types $[:\alpha:]_p$ and $[:\beta:]_q$ which, together with the selector $[:\texttt{Bool}:]_n$, constitute the flat representation of $[:\alpha + \beta:]_n$. Here, p and q depend on the number of left and right elements; the only requirement (which we have not tried to encode in the dependent type) is $p + q = n$.

Note, however, that the arrays of functions passed to $\texttt{case}^{\uparrow}$ must have the length n. Assume that $\texttt{case}^{\uparrow}$ is applied to the function arrays *fs*, *gs* and the array of sums $\langle sel, \langle xs, ys\rangle\rangle$. Then, the constraints encoded by the type imply that *fs* cannot be directly applied to *xs*, since the former contains n and the latter $p \leq n$ elements. This is, in fact, precisely the difficulty we have encountered in the previous example.

This problem is easily solved if arrays of functions support the usual array operations. Consider what happens in the previous example if, instead of lifting the abstraction $\lambda n.\, \texttt{fst}\ x\ +\ n$, we simply generate an array containing a separate function for each element of x, as encoded by $[:\lambda n.\, 1 + n,\ \lambda n.\, 3 + n,\ \lambda n.\, 7 + n:]$. The array can then be packed according to the selector, retaining only those elements for the corresponding element of x has the form $\langle m,\ \texttt{Left}\ n\rangle$. This results in $[:\lambda n.\, 1 + n,\ \lambda n.\, 7 + n:]$, which can be applied elementwise to $[:5, 2:]$ to perform

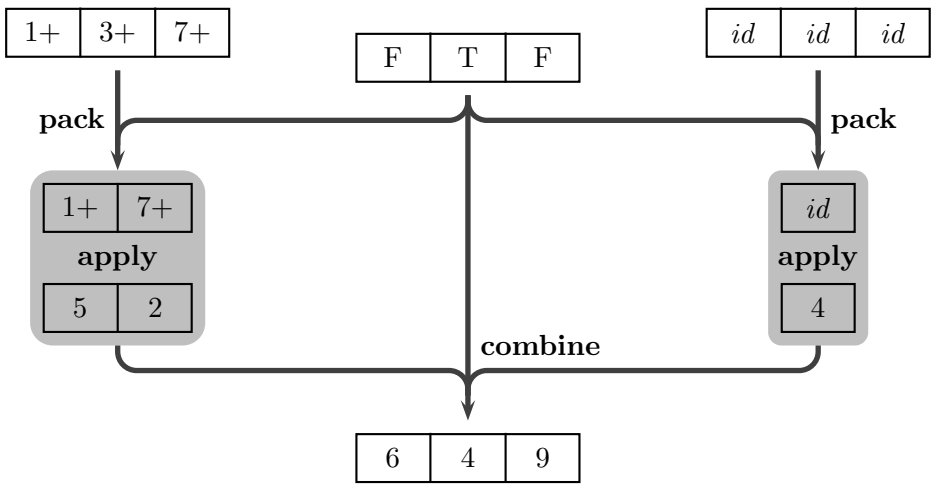


FIGURE 4.1 Lifted case distinction

the desired computation, yielding [:6, 9:]. By the same mechanism, [:$\lambda n.\ n, \lambda n.\ n, \lambda n.\ n$:] generated from the second abstraction can be packed and the applied to [:4:], which contains the only element constructed with `Right`. The results of the two computations can then be combined, again according to the selector, to ultimately obtain [:6, 4, 9:], the correct overall result, as illustrated by Figure 4.1.

An obviously correct implementation of $\texttt{case}^{\uparrow}$ is easily derived from the principle described above:

$$
\begin{array}{l}
\texttt{case}^{\uparrow}\ \langle n_1, \langle\langle n_2, \langle gs, hs\rangle\rangle, \langle sel, \langle xs, ys\rangle\rangle\rangle\rangle\ = \\
\quad \textbf{let} \\
\qquad gs' \ = \ \texttt{packP}\ \langle not^{\uparrow}\ sel,\ gs\rangle \\
\qquad hs' \ = \ \texttt{packP}\ \langle sel,\ hs\rangle \\
\qquad xs' \ = \ \texttt{zipWithP}\ \langle(\$),\ gs',\ xs\rangle \\
\qquad ys' \ = \ \texttt{zipWithP}\ \langle(\$),\ hs',\ ys\rangle \\
\quad \textbf{in} \\
\quad \texttt{combineP}\ \langle sel,\ xs',\ ys'\rangle
\end{array}
$$

Here, (\$) is Haskell's explicit application operator, i.e., $g\ \$\ x\ =\ g\ x$ and, thus, the expression $\texttt{zipWithP}\ \langle(\$), gs', xs\rangle$ denotes the elementwise application of the function array gs' to xs. This is essentially the implementation of $\texttt{case}^{\uparrow}$ used in this work. Note that it crucially depends on the ability to pack arrays of functions, which, as demonstrated earlier, is lost if lifted functions can be mutated by substitution.

To summarise, the preceding discussion outlines the problems in handling functions with embedded data or, equivalently, partially applied functions within the flattening framework, especially in parallel contexts. The rest of this chapter is devoted to finding a representation for arrays of functions which

- supports all array primitives,
- retains the data-parallel nature of the source program,
- has a clear and easy-to-understand semantics and
- does not introduce unacceptable inefficiencies.

4.1.3 Boxed representations

A simple solution is to forgo the flat representation of function arrays and use boxed arrays instead, for which operations such as packing and concatenation can be supported in the obvious way — this is precisely what we have done previously by considering arrays containing lambda abstractions. While attractive at first, this approach has a number of drawbacks. Obviously, in the above example, the individual elements of the array bound to x would have to be converted to boxed objects expected by `fst` and $+$. More generally, the unboxed representation would have to be given up and its performance benefits lost for arguments of partially applied functions. This, in turn, would imply that the following two functions, while semantically equivalent, would have two very different performance characteristics.

$$\begin{aligned} f\ xs &= \texttt{mapP}\ \langle sqrt, xs\rangle \\ g\ xs &= \texttt{zipWithP}\ \langle(\$),\ \underbrace{\texttt{repP}\ \langle \texttt{lenP}\ xs,\ sqrt\rangle}_{\text{array of functions}},\ xs\rangle \end{aligned}$$

Both functions compute the square root for each element of xs. However, while the implementation of f is straightforward, g first constructs an array containing *sqrt* in every position and then applied it elementwise to xs. Clearly, f will be much more efficient than g if a boxed representation is used in the implementation of the latter. Introducing such non-obvious performance traps into a general-purpose functional language is unacceptable from the programmers' point of view.

More importantly, perhaps, this still does not solve all problems caused by partially applied functions. In a parallel setting, array elements often have to be communicated between processors due to load balancing concerns or because the programmer has requested the communication explicitly. Consider, for instance, the expression $\texttt{foldP}\ (\circ)\ fs$ which reduces an array of functions by composing its elements. On a distributed memory machine, *fs* will be distributed across processors such that each node only stores some of the functions in its local memory. The result of the computation, however, must be available in the local memory of every node, i.e., the individual functions will have to be communicated in some way. If *fs* contains partially applied functions, their already bound arguments will have to be communicated as well, thus requiring a general mechanism for sending and receiving arbitrary boxed objects, including thunks. While such mechanisms exist, they are based on virtual shared memory and have a non-trivial impact both on the complexity of the run-time system and on the performance of parallel programs (Loidl, 2002).

Finally, the interactions with the data-parallel programming model must be considered. So far, we have assumed that at any given moment, all processors perform the same computation. For many parallel algorithms, this principle leads to a very natural implementation with little synchronisation and communication. Maintaining this property becomes difficult, though, if the above representation is used. Now, a parallel array may contain arbitrary functions. When such an array is to be applied elementwise to some argument array, different strategies can be chosen.

- All applications are performed in parallel, even if this causes two different functions to be applied at the same time. This does not lead to problems as long as no parallel computations are involved. Once that happens, however, different processors will in general have to execute *different* parallel operations at the same time, thereby invalidating several key assumptions underlying the usual strategy of implementing data

parallelism. This is not merely a theoretical issue — in fact, it is not at all clear how such computations can be supported.

- The runtime system somehow keeps track of equivalence between element functions and applies only equivalent ones in parallel. Still, this does not ensure that the transformed program is data parallel since applications of the same function to different arguments might follow different computation paths, ultimately causing different parallel operations to be executed simultaneously.
- The applications are performed sequentially, one after another. For the example given in Section 4.1.1, this implies that, e.g., the additions are serialised even though they could be easily executed in parallel. Thus, this strategy is overly pessimistic and unacceptable for performance reasons.

All in all, it becomes evident that in the long run, a boxed representation causes more problems than it solves and has to be rejected.

4.1.4 A calculational approach

Let us now take a step back and investigate how operations on arrays of partially applied functions can be described from a semantical point of view. Consider, for instance, the term $\texttt{packP}\,\langle bs,\, \texttt{mapP}\,\langle(+),\, xs\rangle\rangle$, i.e. the packing of an array containing partially applied functions (here, we assume that $(+)$ is curried). Crucially, this term can be evaluated by packing *the argument array* first and then applying the addition to the result. This strategy is obviously sound because `packP` and `mapP` satisfy the following equality:

$$\texttt{packP}\,\langle bs,\, \texttt{mapP}\,\langle(+),\, xs\rangle\rangle \;=\; \texttt{mapP}\,\langle(+),\, \texttt{packP}\,\langle bs,\, xs\rangle\rangle$$

This insight suggests that in the example from Section 4.1.1, x should be packed before being substituted into the abstraction. The example can be modified accordingly, but this principle cannot be used to implement $\texttt{case}^{\uparrow}$, since it would have to be able to pack data which has *already* been substituted into a function.

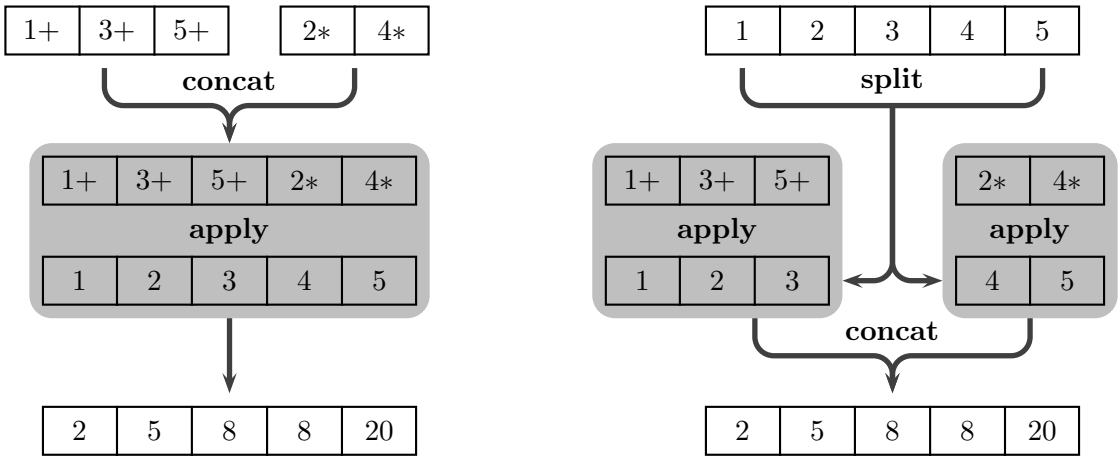
As a more involved example, let us consider the elementwise application of an array of functions obtained by concatenation, as in the term $\texttt{zipWithP}\,\langle(\$),\, fs +\!\!+ gs,\, xs\rangle$. The same result can be obtained by splitting xs into two arrays according to the lengths of fs and gs, applying the function arrays individually and concatenating the results:

$$\begin{aligned}\texttt{zipWithP}\,\langle(\$),\, fs +\!\!+ gs,\, xs\rangle \;=\; &\texttt{zipWithP}\,\langle(\$),\, fs,\, \texttt{takeP}\,\langle\texttt{lenP}\,fs,\, xs\rangle\rangle \\ &+\!\!+\; \texttt{zipWithP}\,\langle(\$),\, gs,\, \texttt{dropP}\,\langle\texttt{lenP}\,fs,\, xs\rangle\rangle\end{aligned}$$

The two evaluation strategies are shown in Figure 4.2.

Note how the two laws discussed so far eliminate operations on arrays of functions by performing suitable computations on arrays containing the arguments to which the former are applied. Interestingly, a number of such laws can be derived. In fact, this principle is easily recognisable as an application of *parametricity* (Wadler, 1989). In Section 4.3, we expand upon this idea and show that *all* problematic operations on arrays of functions can be eliminated using this strategy.

For our introductory example, this elimination can easily be performed at compile time by a suitable transformation. Unfortunately, this is not always possible — the necessary



a) Concatenate, then apply

b) Split, apply, then concatenate

FIGURE 4.2 Concatenation and elementwise application

bookkeeping is, in general, intractable and cannot be done statically. Thus, it must be possible to make use of these transformations dynamically during the execution of a program. For instance, when applied to an array of partially applied functions, `packP` must be able to extract and pack the argument array. In general, this requires that already bound arguments of partial applications can be manipulated by array operations at any time.

4.2 Closure conversion

Fortunately, closure conversion achieves exactly the required separation between code and data. It does so by expressing all partial applications as *closures* which store the function's code and its environment, i.e., the arguments it has already been applied to, as two separate components which can be accessed independently. Closure conversion has originally been developed for the untyped lamdba calculus (Reynolds, 1972; Cousineau et al., 1985; Steele, Jr., 1978); later, Minamide et al. (1996) reformulated it as a type-preserving transformation. In this dissertation, we largely follow the simplified version proposed by Morrisett et al. (1999). In contrast to this previous work, however, we introduce a novel, variableless target language which syntactically enforces the crucial properties of closure-converted programs.

A simple example of this transformation is the conversion of the following function.

$$f :: \texttt{Bool} \to \texttt{Int} \to \texttt{Bool}$$
$$f = \lambda b.\, \lambda n.\, \textbf{if}\ b\ \textbf{then}\ n\ \textbf{else}\ 0$$

Here, the inner abstraction contains references to the variable b, which is bound by the outer abstraction. Closure conversion transforms it such that it must be applied to a pair which stores both b, its *environment*, and n, its actual argument. Accesses to the two variables are replaced by accesses to the composite argument, as in the following code.

$$f_{inner} :: \texttt{Bool} \times \texttt{Int} \to \texttt{Int}$$
$$f_{inner} = \lambda y.\, \textbf{if}\ \underbrace{\mathit{fst}\ y}_{b}\ \textbf{then}\ \underbrace{\mathit{snd}\ y}_{n}\ \textbf{else}\ 0$$

Note that this is just a simple instance of uncurrying. The outer function is transformed in a similar manner. In this case, however, an environment is not needed as the function only makes use of its direct argument and does not access any variables bound outside of its scope. To keep the transformation consistent it still has to be given an environment of unit type. The parameter is now a tuple of type $\langle\rangle \times \texttt{Bool}$.

To understand how the outer function is implemented, let us consider what happens when it is applied to some tuple $\langle\langle\rangle, b\rangle$. The environment of f_{inner} must then be bound to b; its direct argument is not yet known, though. Such partial applications are represented by *closures* of the form $\langle\!\langle f, \tau, e \rangle\!\rangle$ where

- f is the (closure-converted) *closure function* of type $\tau \times \tau_1 \to \tau_2$,
- τ is the type of the environment and
- e is the function's environment (of type τ).

In the above example, the closure $\langle\!\langle f_{inner}, \texttt{Bool}, \mathit{True} \rangle\!\rangle$ would be created to represent the partial application of f_{inner} to the boolean argument *True*.

For a program to be well-typed after closure conversion, the type of the environment may not appear in the type of the closure. This becomes evident if we consider that given

$$\begin{aligned} g &:: \texttt{Bool} \to \texttt{Bool} \to \texttt{Int} \\ h &:: \texttt{Int} \to \texttt{Bool} \to \texttt{Int} \end{aligned}$$

the terms $g\ \mathit{True}$ and $h\ 1$ will have the same type, namely $\texttt{Bool} \to \texttt{Int}$. Thus, the two closures representing these partial applications must also have the same type, even though the first one will have a boolean-typed environment and the second an integer-typed one. Therefore, the type of a closure depends only on the type of the missing argument and the type of the result, but not on the environment. We denote the type of closures which, when applied to a value of type τ_1, yield a value of type τ_2 by $\tau_1 \Rightarrow \tau_2$. The closure created by binding the environment of f_{inner} would then have the type $\texttt{Int} \Rightarrow \texttt{Int}$. The outer function from our example is thus transformed to

$$\begin{aligned} f_{outer} &:: \langle\rangle \times \texttt{Bool} \to (\texttt{Int} \Rightarrow \texttt{Int}) \\ f_{outer} &= \lambda x.\, \langle\!\langle f_{inner}, \texttt{Bool}, \underbrace{\mathit{snd}\ x}_{b} \rangle\!\rangle \end{aligned}$$

In general, closure conversion transforms every function of type $\tau_1 \to \tau_2$ to a closure of type $\tau_1 \Rightarrow \tau_2$. For our example, this implies that the function f will be transformed to a closure of type $\texttt{Bool} \Rightarrow (\texttt{Int} \Rightarrow \texttt{Int})$. Indeed, the following code will be generated in this case (note that in analogy to $\to$, we assume $\Rightarrow$ to be right-associative).

$$\begin{aligned} f &:: \texttt{Bool} \Rightarrow \texttt{Int} \Rightarrow \texttt{Int} \\ f &= \langle\!\langle f_{outer}, \langle\rangle, \langle\rangle \rangle\!\rangle \end{aligned}$$

This is, in fact, necessary because the value of the closure's environment is only known at its definition site; the callers have no way of determining it. Thus, the environment of f_{outer} must be bound in the definition of f.

4.2.1 Applying closures

We have not yet described how a closure can be applied to an argument. The semantics of this operation is straightforward: it extracts the environment from the closure, tuples it with the argument and applies the closure's function to the resulting pair. We denote the application of a closure c to some value x by $c \dagger x$. Essentially, this operation can be implemented as follows:

$$\begin{array}{ll} (\dagger) & :: (\alpha \Rightarrow \beta) \times \alpha \to \beta \\ \langle\!\langle f, \tau, e \rangle\!\rangle \dagger x & = f\ \langle e, x \rangle \end{array}$$

However, the flattening transformation is considerably simplified if closure application is built into the language rather than provided as a library function. Thus, the above definition is only a specification of the operation's semantics rather than an actual implementation.

4.2.2 Eliminating variables

Let us consider what happens if a function contains more that one free variable, as in the following example.

$$\begin{array}{l} g :: \texttt{Bool} \to \texttt{Int} \to \texttt{Int} \to \texttt{Int} \\ g = \lambda b.\, \lambda m.\, \lambda n.\, \textbf{if}\ b\ \textbf{then}\ m + n\ \textbf{else}\ 0 \end{array}$$

Again, the inner function is transformed by eliminating references to parameters of enclosing abstractions and replacing them by references to the environment. In this case, the latter will contain the values of both b and x; thus, it has the type $\texttt{Bool} \times \texttt{Int}$.

$$\begin{array}{l} g_{inner} :: (\texttt{Bool} \times \texttt{Int}) \times \texttt{Int} \to \texttt{Int} \\ g_{inner} = \lambda z.\, \textbf{if}\ \underbrace{fst\,(fst\,z)}_{b}\ \textbf{then}\ \underbrace{snd\,(fst\,z)}_{m} + \underbrace{snd\,z}_{n}\ \textbf{else}\ 0 \end{array}$$

The outer two abstractions are transformed as in the previous example:

$$\begin{array}{ll} g_{middle} & :: \texttt{Bool} \times \texttt{Int} \to (\texttt{Int} \Rightarrow \texttt{Int}) \\ g_{middle} & = \lambda y.\, \langle\!\langle g_{inner}, \texttt{Bool} \times \texttt{Int}, \langle \underbrace{fst\,y}_{b}, \underbrace{snd\,y}_{m} \rangle \rangle\!\rangle \\ g_{outer} & :: \langle\rangle \times \texttt{Bool} \to (\texttt{Int} \Rightarrow \texttt{Int} \Rightarrow \texttt{Int}) \\ g_{outer} & = \lambda x.\, \langle\!\langle g_{middle}, \texttt{Bool}, \underbrace{snd\,x}_{b} \rangle\!\rangle \\ g & :: \texttt{Bool} \Rightarrow \texttt{Int} \Rightarrow \texttt{Int} \Rightarrow \texttt{Int} \\ g & = \langle\!\langle g_{outer}, \langle\rangle, \langle\rangle \rangle\!\rangle \end{array}$$

Note that after closure conversion, at most one variable, namely the parameter of the innermost lambda abstraction, is visible at any given point in the program. This becomes evident if we inline g_{inner}, g_{middle} and g_{outer} into the definition of g:

$$\begin{array}{l} g = \langle\!\langle \lambda x. \langle\!\langle \lambda y. \langle\!\langle \lambda z. \textbf{if}\ fst\ (fst\ z)\ \textbf{then}\ snd\ (fst\ z) + snd\ z\ \textbf{else}\ 0, \\ \qquad\qquad \texttt{Bool} \times \texttt{Int}, \langle fst\ y, snd\ y \rangle \rangle\!\rangle, \\ \qquad\quad \texttt{Bool}, snd\ x \rangle\!\rangle, \\ \qquad \langle\rangle, \langle\rangle \rangle\!\rangle \end{array}$$

Here, the inner abstraction does not refer either to x or to y. In fact, it is precisely the task of closure conversion to eliminate all references to free variables. This suggests that closure-converted programs do not need variable names at all. Consequently, the language of explicit closures, the target language of the transformation, abandons named value variables in favour of a more concise notation. A lambda abstraction has the form $\lambda\bullet.\ e$; within e, the symbol $\bullet$ is used to reference the parameter. Thus, the definition of g is rewritten as follows:

$$\begin{aligned} g \ = \ & \langle\!\langle \lambda\bullet.\langle\!\langle \lambda\bullet.\langle\!\langle \lambda\bullet.\mathbf{if}\ \mathit{fst}\ (\mathit{fst}\ \bullet)\ \mathbf{then}\ \mathit{snd}\ (\mathit{fst}\ \bullet) + \mathit{snd}\ \bullet\ \mathbf{else}\ 0, \\ & \qquad\qquad \mathtt{Bool} \times \mathtt{Int},\ \langle \mathit{fst}\ \bullet,\ \mathit{snd}\ \bullet\rangle\rangle\!\rangle, \\ & \qquad \mathtt{Bool},\ \mathit{snd}\ \bullet\rangle\!\rangle, \\ & \langle\rangle,\ \langle\rangle\rangle\!\rangle \end{aligned}$$

Crucially, by abandoning named variables we eliminate the problems associated with data embedded into computations as discussed in Section 4.1. Since conceptually, all variables now have the same name, substitution can no longer modify terms within a lambda abstraction. Instead, the data which would be substituted directly into the code in the standard lambda calculus is collected in environments. This implies that in the calculus described here, computations are *immutable* and, in contrast to previous work, their immutability is enforced syntactically. This property is formalised and discussed in more detail in Section 6.5.

4.2.3 Primitives

The elimination of curried functions has important consequences for the operations provided by the standard prelude. Consider, for instance, the signature of mapP:

$$\mathit{mapP}\ ::\ (\alpha \to \beta) \to [:\alpha:] \to [:\beta:]$$

After closure conversion, no less than three versions of this operation will be used:

$$\begin{aligned} &\mathtt{mapP} && ::\ (\alpha \Rightarrow \beta) \times [:\alpha:] \to [:\beta:] \\ &\quad --- \\ &\mathit{mapP}_0 && ::\ \langle\rangle \times (\alpha \Rightarrow \beta) \to ([:\alpha:] \Rightarrow [:\beta:]) \\ &\mathit{mapP}_0 && =\ \lambda\bullet.\langle\!\langle \mathtt{mapP},\ \alpha \Rightarrow \beta,\ \mathit{snd}\ \bullet\rangle\!\rangle \\ &\quad --- \\ &\mathit{mapP} && ::\ (\alpha \Rightarrow \beta) \Rightarrow ([:\alpha:] \Rightarrow [:\beta:]) \\ &\mathit{mapP} && =\ \langle\!\langle \mathit{mapP}_0,\ \langle\rangle,\ \langle\rangle\rangle\!\rangle \end{aligned}$$

Here, `mapP` is the primitive performing the actual computation, mapP is the curried closure and mapP_0 is an auxiliary function which binds the first argument. This is analogous to the mechanisms discussed in the preceding sections; however, it is important to realise that all three functions must be included into the standard prelude.

4.2.4 Notation

To avoid a proliferation of identifiers in the text, we adopt the following convention. As described before, identifiers set in `typescript` refer to the uncurried primitives, whereas those set in *italics* denote the corresponding closure in $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$, as in the above example, or the curried primitive in $\lambda^{\mathcal{P}}$. Auxiliary functions like mapP_0 play no role in the following exposition.

Moreover, in informal discussions, we will use the notation $\langle\!\langle f \rangle\!\rangle$ to refer to the closure representing the unapplied function f. Thus, $\langle\!\langle \texttt{mapP} \rangle\!\rangle$ would denote the closure $mapP$. While somewhat imprecise, this notation is sufficiently clear and unambiguous and avoids unnecessary verbosity.

4.2.5 Representation of closures

Commonly, existential types are used for encoding closures in a strongly-typed language (Mitchell and Plotkin, 1988). The semantics of the type constructor $\Rightarrow$ can be captured by the following pseudo-Haskell type definition:

$$\textbf{type}\ \alpha \Rightarrow \beta\ =\ \exists\tau.(\tau \times \alpha \to \beta) \times \tau$$

There exists a large body of work on selecting the best representation for the environment and on its effects on the run-time performance of functional languages (Shao and Appel, 1994; Appel and Jim, 1988; Wand and Steckler, 1994). However, for the purposes of this dissertation, neither the representation of closures nor that of environments is relevant, as we expect these issues to be resolved later in the compilation process. Consequently, we treat closures as an abstract data type with a well-defined interface.

4.3 Array closures

Although the inclusion of closure conversion in the compilation of NDP programs has been motivated by the desire to handle arrays of functions correctly and efficiently, we have not considered how closures interact with parallelism so far. We will now demonstrate that closure conversion fits very naturally with the flattening transformation and, in fact, is a complete and lightweight solution to the problems we encountered earlier.

In the previous section, we described how the transformation replaces functions by closures. Clearly, this implies that arrays of functions will be converted to arrays of closures. Thus, we mainly concentrate on deriving an efficient representation for the latter; while we briefly discuss arrays of functions in Section 4.6, they play only a minor role in the transformation since they are severly restricted and do not occur in closure-converted programs unless introduced by subsequent optimisations.

4.3.1 Vectorisation of closures

First, however, it is important to consider how closures are treated by vectorisation. Recall that the main purpose of the latter is to tuple each function with its lifted version, as described in Section 3.5.2. Every closure represents a partial application and thus contains a function which will be transformed accordingly. After vectorisation, a closure $\langle\!\langle f, \tau, e \rangle\!\rangle$ will then be represented by $\langle\!\langle \langle f', f^{\uparrow} \rangle, \tau', e' \rangle\!\rangle$ where τ' and e' are the results of vectorising τ and e, respectively, f' is the vectorised version of f and $f^{\uparrow}$ its lifted version. Thus, after flattening, closures store pairs of functions such that the second one is the lifted version of the first. We will see below that our compilation strategy crucially relies on this fact.

Obviously, the closure application operator has to be modified to account for this. After vectorisation, its has the following semantics.

$$\langle\!\langle \langle f, f^{\uparrow} \rangle, \tau, e \rangle\!\rangle \dagger x\ =\ f\ \langle e, x \rangle$$

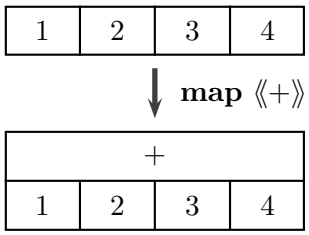


Figure 4.3 An array closure

Here, the sequential version of the closure's function is applied to the environment and the new argument. This corresponds directly to the semantics of closure application before flattening as described in the previous section.

4.3.2 Arrays of closures

How, then, can arrays of closures be represented? To answer this question, let us consider a simple form of such arrays, namely those generated by applications of `mapP`, as in, e.g., `mapP` $\langle\langle\!\langle + \rangle\!\rangle, xs\rangle$. Here, the resulting array will, in each position, contain a partial application of the same function (i.e., $(+)$ in this example) to the corresponding element of xs. This suggests that such arrays can be represented much like regular closures, except that they should store *arrays of environments*, such that each element defines the environment of the closure at the corresponding position.

Indeed, this is the representation we use in this work. An array of closures is transformed to an *array closure* of the form $\langle\!\langle\!:\langle f, f^{\uparrow}\rangle, \tau, es:\rangle\!\rangle$, where

- f and $f^{\uparrow}$ are functions of types $\tau \times \tau_1 \to \tau_2$ and $[:\tau \times \tau_1:] \to [:\tau_2:]$, respectively,
- τ is the type of the individual environments and
- es is the array of environments of type $[:\tau:]$.

We denote the type of such closures by $[:\tau_1:] \Rrightarrow [:\tau_2:]$; the flattening transformation replaces all occurences of $[:\tau_1 \Rightarrow \tau_2:]$ by this type. Note that array closures only appear in $\lambda^{\mathcal{A}}$ where the types $[:\tau_1:]$ and $[:\tau_2:]$ are flattened and the type constructor $[:\cdot:]$ eliminated. However, for the sake of readability of the informal presentation we will continue to use a mixed notation, considering array types and their flat representations to be equivalent.

While the above representation allows for differing environments, it requires the function to be same at every position of the array. Although this may seem like a rather severe restriction, we will see below that this approach is entirely sufficient for representing arbitrary closure arrays and, in fact, ensures that the semantics of generated code is consistent with the data-parallel model. In particular, it naturally supports operations such as concatenation, as well as the nesting of closure arrays, discussed in Sections 4.4 and 4.5, respectively.

With this representation, the partial application of a curried function to each element of an array simply constructs an array closure which, in addition to the function, stores the argument array as the environment. For instance, `mapP` $\langle\langle\!\langle + \rangle\!\rangle, xs\rangle$ ultimately evaluates to $\langle\!\langle\!:\langle (+), (+^{\uparrow})\rangle, \texttt{Int}, xs:\rangle\!\rangle$. Figure 4.3 illustrates this representation.

4.3.3 Elementwise application

Perhaps the most important operation on array closures is their elementwise application to arrays of arguments. Although we have previously used $\texttt{zipWithP}\,\langle(\$), fs, xs\rangle$ to express this operation, it must be provided as a built-in operator in $\lambda^{\mathcal{A}}$. In analogy to †, the application of sequential closures, we denote the elementwise application of array closures by ‡. The semantics of this operation is as follows:

$$(‡) \;::\; ([:\alpha:] \Rrightarrow [:\beta:]) \times [:\alpha:] \to [:\beta:]$$
$$\langle\!|:\langle f, f^{\uparrow}\rangle, \tau, es:|\!\rangle \;‡\; xs \;=\; f^{\uparrow}\;(\texttt{zipP}\,\langle es, xs\rangle)$$

Recall that the closure represents the partial application of f to each element of es. Thus, the result of the elementwise application contains, at each position i, the term $f\,\langle e_i, x_i\rangle$, where e_i and x_i are the elements of es and xs, respectively, at that position. This is precisely the array obtained by mapping f over or, equivalently, applying $f^{\uparrow}$ to $\texttt{zipP}\,\langle es, xs\rangle$.

4.3.4 Polymorphic operations

In Section 4.1.4, we introduced rewriting rules which replaced operations on arrays of partially applied functions by those on their arguments. Array closures, by providing direct access to the already bound arguments, allow these rules to be used for implementing primitive operations. Consider, for instance, the following rule (here, we assume that f is a binary function):

$$\texttt{packP}\,\langle bs,\, \texttt{mapP}\,\langle\langle\!|f|\!\rangle, xs\rangle\rangle \;=\; \texttt{mapP}\,\langle\langle\!|f|\!\rangle,\, \texttt{packP}\,\langle bs, xs\rangle\rangle$$

By replacing applications of `mapP` by array closures, the above can be rewritten as follows:

$$\texttt{packP}\,\langle bs,\, \langle\!|:\langle f, f^{\uparrow}\rangle, \tau, xs:|\!\rangle\rangle \;=\; \langle\!|:\langle f, f^{\uparrow}\rangle, \tau, \texttt{packP}\,\langle bs, xs\rangle:|\!\rangle$$

Clearly, this completely specifies the semantics of applying `packP` to array closures. In the following, we give the signature and implementation specialised for array closures for each operation we consider; the latter is added to the type-indexed implementation as defined in Appendix C, while the former is only included for exposition purposes. For `packP`, we can derive the following implementation:

$$\texttt{packP}_{\langle\tau_1 \Rightarrow \tau_2\rangle} \;::\; [:\texttt{Bool}:] \times ([:\tau_1:] \Rrightarrow [:\tau_2:]) \to ([:\tau_1:] \Rrightarrow [:\tau_2:])$$
$$\texttt{packP}_{\langle\tau_1 \Rightarrow \tau_2\rangle}\,\langle bs,\, \langle\!|:\langle f, f^{\uparrow}\rangle, \tau, xs:|\!\rangle\rangle \;=\; \langle\!|:\langle f, f^{\uparrow}\rangle, \tau, \texttt{packP}_{\langle\tau\rangle}\,\langle bs, xs\rangle:|\!\rangle$$

Figure 4.4 depicts this evaluation strategy.

Other array operations can be implemented in a similar way, by performing corresponding computations on the environments. Note, however, how the type argument for the recursive call to `packP` is obtained from the closure and does *not* depend on the closure's type. This highlights an important constraint: for this implementation strategy to be viable, the operation must be polymorphic in the type of the array's elements.

This approach is closely related to the concept of parametricity and free theorems, as discussed by Wadler (1989). Here, properties of polymorphic operations are derived from their types, without inspecting the actual implementations. For instance, the following famous theorem covers the semantics of *map* on lists. Given a polymorphic list function g of type $\forall\alpha.[\alpha] \to [\alpha]$, the following equality holds:

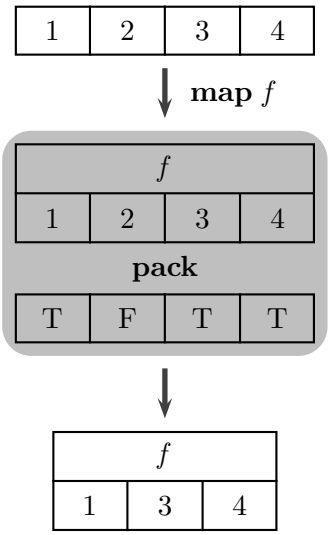


Figure 4.4 Packing array closures

$$g \circ map\, f \;=\; map\, f \circ g$$

A similarily general law cannot be easily derived for parallel arrays. Although type-indexed primitives, which form the bulk of the interesting operations on parallel arrays, are polymorphic, this polymorphism is not parametric, and, thus, they are not amenable to the same kind of free theorems as polymorphic functions in, e.g., Haskell (Hinze, 2000). Moreover, in Chaper 6, we will see that parallel arrays and operations on them exhibit some non-obvious strictness properties which make the usual formulation of parametricity break down in much same way as it does in the presence of `seq` (Johann and Voigtländer, 2004). However, by restricting ourselves to arrays of partially applied functions, we can formulate the following more specialised principle.

> *Let f be a function of type $\tau_1 \to (\tau_2 \Rightarrow \tau_3)$, g a type-indexed primitive and xs a parallel array of type $[:\tau_1:]$. Then, $g\,(\texttt{mapP}\,\langle\!\langle f \rangle\!\rangle, xs\rangle) \;=\; \texttt{mapP}\,\langle\!\langle f \rangle\!\rangle,\; g\; xs\rangle$.*

We do not claim this statement to be in any way precise or universally valid, nor do we intend to give it a more formal meaning later. However, we will use it as a guideline when deriving the implementations of type-indexed primitives.

The above can be rewritten in terms of array closures by suitably replacing applications of `mapP`:

$$g\,\langle\!\langle\!:\langle f, f^{\uparrow}\rangle, \tau, xs:\rangle\!\rangle \;=\; \langle\!\langle\!:\langle f, f^{\uparrow}\rangle, \tau, g\; xs:\rangle\!\rangle$$

This is precisely the principle employed in the implementation of `packP` discussed earlier. However, parametricity suggests that the equality holds for arbitrary array operations regardless of their actual semantics, as long as they are polymorphic in the type of the array elements. Thus, this technique is immediately applicable to a large number of primitives, such as `reverseP`:

$$\texttt{reverseP}_{\langle \tau_1 \Rightarrow \tau_2 \rangle}\,\langle\!\langle\!:\langle f, f^{\uparrow}\rangle, \tau, xs:\rangle\!\rangle \;=\; \langle\!\langle\!:\langle f, f^{\uparrow}\rangle, \tau, \texttt{reverseP}_{\langle \tau \rangle}\; xs:\rangle\!\rangle$$

It is easy to see that this implementation has the desired semantics. In fact, this property of array closures can also be used for implementing primitives other than those of type $[:\tau:] \to [:\tau:]$. In the following, we consider three such operations, deriving their implementations from the corresponding laws.

Length. The length of an array closure is obviously determined by the length of the environment array, as indicated by the following equation:

$$\texttt{lenP}\,(\texttt{mapP}\,\langle\langle\!\langle f \rangle\!\rangle, xs\rangle) \;=\; \texttt{lenP}\,xs$$

It is important to note that the above only holds if none of the terms involved in the computation diverge. For now, we will assume that this is the case. In Chapter 6, we will see that the strictness properties of array closures guarantee that the following is a valid implementation:

$$\begin{aligned}&\texttt{lenP}_{\langle\tau_1 \Rightarrow \tau_2\rangle} \;::\; ([:\tau_1:] \Rrightarrow [:\tau_2:]) \to \texttt{Int}\\ &\texttt{lenP}_{\langle\tau_1 \Rightarrow \tau_2\rangle}\,\langle\!\langle [:\langle f, f^{\uparrow}\rangle, \tau, xs:] \rangle\!\rangle \;=\; \texttt{lenP}_{\langle\tau\rangle}\,xs\end{aligned}$$

Replication. The following law covers the replication of partially applied functions:

$$\texttt{repP}\,\langle n, f\;x\rangle \;=\; \texttt{mapP}\,\langle\langle\!\langle f \rangle\!\rangle, \texttt{repP}\,\langle n, x\rangle\rangle$$

This suggests that a closure of the form $\langle\langle\!\langle \langle f, f^{\uparrow}\rangle, \tau, x \rangle\!\rangle\!\rangle$ is replicated to an array closure by replicating the environment x, while leaving the pair of functions $\langle f, f^{\uparrow}\rangle$ unchanged:

$$\begin{aligned}&\texttt{repP}_{\langle\tau_1 \Rightarrow \tau_2\rangle} \;::\; \texttt{Int} \times (\tau_1 \Rightarrow \tau_2) \to ([:\tau_1:] \Rrightarrow [:\tau_2:])\\ &\texttt{repP}_{\langle\tau_1 \Rightarrow \tau_2\rangle}\,\langle n, \langle\langle\!\langle \langle f, f^{\uparrow}\rangle,\; \tau,\; x \rangle\!\rangle\!\rangle\rangle \;=\; \langle\langle\!\langle [:\langle f, f^{\uparrow}\rangle,\; \tau,\; \texttt{repP}_{\langle\tau\rangle}\,\langle n, x\rangle :] \rangle\!\rangle\!\rangle\end{aligned}$$

Indexing. Indexing is, roughly speaking, the inverse of replication. For arrays of partially applied functions, its semantics is given by

$$\texttt{mapP}\,\langle\langle\!\langle f \rangle\!\rangle, xs\rangle \;!:\; i \;=\; f\,(xs \;!:\; i)$$

Similar to the previous case, this means that the operation can be implemented by leaving the closure functions unchanged and extracting the corresponding element of the environment array:

$$\begin{aligned}&(!:_{\langle\tau_1 \Rightarrow \tau_2\rangle}) \;::\; ([:\tau_1:] \Rrightarrow [:\tau_2:]) \times \texttt{Int} \to (\tau_1 \Rightarrow \tau_2)\\ &\langle\langle\!\langle [:\langle f, f^{\uparrow}\rangle,\; \tau,\; xs:] \rangle\!\rangle\!\rangle \;!:_{\langle\tau_1 \Rightarrow \tau_2\rangle}\; i \;=\; \langle\langle\!\langle \langle f, f^{\uparrow}\rangle,\; \tau,\; xs \;!:_{\langle\tau\rangle}\; i \rangle\!\rangle\!\rangle\end{aligned}$$

The last two operations rely crucially on the association between functions and their lifted versions introduced by vectorisation. For instance, in the array closure $\langle\langle\!\langle [:\langle f, f^{\uparrow}\rangle, \tau, es:] \rangle\!\rangle\!\rangle$, the unlifted function f is not used by either elementwise application or any other primitive operation except indexing. The latter, however, could not be implemented if f was not stored along with $f^{\uparrow}$. Likewise, the lifted functions in regular closures are unnecessary for most operations but essential for converting them to array closures, either by replication of by mapping.

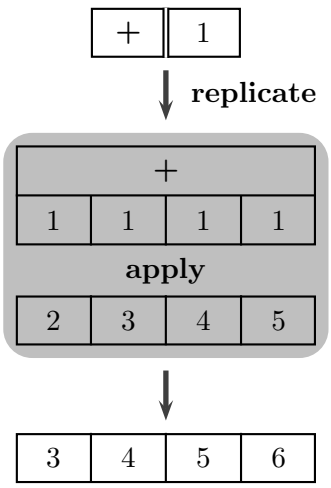


Figure 4.5 Evaluation of `mapP`

4.3.5 Mapping

Although `mapP` has been used extensively in the preceding discussion, we have not yet shown its implementation. The latter is easily derived by considering the semantics of this operation: Given a closure c of type $\tau_1 \Rightarrow \tau_2$ and an array xs of type $[:\tau_1:]$, `mapP` applies c to each element of xs. This is clearly equivalent to replicating c to the length of xs and applying the resulting array closure elementwise:

$$\begin{aligned}&\texttt{mapP} :: (\alpha \Rightarrow \beta) \times [:\alpha:] \to [:\beta:]\\ &\texttt{mapP}\,\langle c,\ xs\rangle\ =\ \texttt{repP}\,\langle \texttt{lenP}\,xs,\ c\rangle\ \ddagger\ xs\end{aligned}$$

The definitions of `repP` and ($\ddagger$) ensure that this implementation has the expected semantics. This is easily verified by considering how an application of `mapP` is evaluated:

$$\begin{aligned}&\texttt{mapP}\,\langle\langle\!\langle\langle f, f^{\uparrow}\rangle, \tau, e\rangle\!\rangle, xs\rangle\\ &\quad=\ \texttt{repP}\,\langle \texttt{lenP}\,xs,\ \langle\!\langle\langle f, f^{\uparrow}\rangle, \tau, e\rangle\!\rangle\rangle\ \ddagger\ xs\\ &\quad=\ \langle\!\langle\!:\langle f, f^{\uparrow}\rangle,\ \tau,\ \texttt{repP}\,\langle \texttt{lenP}\,xs, e\rangle :\!\rangle\!\rangle\ \ddagger\ xs\\ &\quad=\ f^{\uparrow}\,(\texttt{zipP}\,\langle \texttt{repP}\,\langle \texttt{lenP}\,xs, e\rangle, xs\rangle)\end{aligned}$$

Ultimately, the lifted version of the function is applied to the environment, which has been replicated to the required length, and the argument array. This evaluation strategy is depicted in Figure 4.5.

The above demonstrates that `mapP` is not a primitive operation in the sense that it does not have to be built into the language and does not even require a type-indexed implementation. This is a major advantage of using closure conversion in the compilation process. In Section 4.5, we will see that the elimination of nested parallel computations, which had to be specified explicitly in previous flattening-based approaches, arises as a natural consequence of this implementation of mapping.

4.3.6 Case distinction

Let us revisit the example from Section 4.3.6 which we used to demonstrate the difficulties arising in combining nested data parallelism with higher-order functions:

$$
\begin{array}{l}
f \;::\; \mathtt{Int} \times (\mathtt{Int} + \mathtt{Int}) \to \mathtt{Int} \\
f \;=\; \lambda x.\, \mathtt{case}\, \langle \langle \lambda n.\, \mathtt{fst}\; x \;+\; n,\; \lambda n.\, n \rangle,\; \mathtt{snd}\; x \rangle
\end{array}
$$

Closure conversion essentially transforms this function as follows (we omit unnecessary details for the sake of clarity):

$$
\begin{array}{l}
g \;::\; (\mathtt{Int} \times (\mathtt{Int} + \mathtt{Int})) \times \mathtt{Int} \to \mathtt{Int} \\
g \;=\; \lambda \bullet.\, \mathtt{fst}\, (\mathtt{fst}\, \bullet) \;+\; \mathtt{snd}\, \bullet \\
h \;::\; \langle\rangle \times \mathtt{Int} \to \mathtt{Int} \\
h \;=\; \lambda \bullet.\, \mathtt{snd}\, \bullet \\
f \;::\; \mathtt{Int} \times (\mathtt{Int} + \mathtt{Int}) \to \mathtt{Int} \\
f \;=\; \lambda \bullet.\, \mathtt{case}\, \langle \langle \langle\!\langle g, \mathtt{Int} \times (\mathtt{Int} + \mathtt{Int}), \bullet \rangle\!\rangle,\; \langle\!\langle h, \langle\rangle, \langle\rangle \rangle\!\rangle \rangle, \mathtt{snd}\, \bullet \rangle
\end{array}
$$

Here, g and h are the functions generated for the two lambda abstractions, which we have hoisted to the top level. Both are now binary functions, expecting $\bullet$ to be bound to an environment/argument pair. In the case of g, the environment, accessed by $\mathtt{fst}\, \bullet$, corresponds to the value of x in the original abstraction, while $\mathtt{snd}\, \bullet$ accesses the value of n. The environment of h is empty, since no free variables occured in the original abstraction. In the body of f, the two functions have been replaced by closures; the one generated for g stores, as expected, the argument to f in the environment. Note that after closure conversion, the function arguments to `case` are replaced by closures, i.e., it has the type

$$\mathtt{case} :: ((\alpha \Rightarrow \gamma) \times (\beta \Rightarrow \gamma)) \times (\alpha + \beta) \to \gamma$$

Let us investigate how these functions are transformed by flattening. Their lifted versions are generated according to the strategy described in Section 3.5.1:

$$
\begin{array}{l}
g^{\uparrow} \;::\; [:(\mathtt{Int} \times (\mathtt{Int} + \mathtt{Int})) \times \mathtt{Int}:] \to [:\mathtt{Int}:] \\
g^{\uparrow} \;=\; \lambda \bullet.\, \mathtt{fst}^{\uparrow}\, (\mathtt{fst}^{\uparrow}\, \bullet) \;+^{\uparrow}\; \mathtt{snd}^{\uparrow}\, \bullet \\
h^{\uparrow} \;::\; [:\langle\rangle \times \mathtt{Int}:] \to [:\mathtt{Int}:] \\
h^{\uparrow} \;=\; \lambda \bullet.\, \mathtt{snd}^{\uparrow}\, \bullet \\
f^{\uparrow} \;::\; [:\mathtt{Int} \times (\mathtt{Int} + \mathtt{Int}):] \to [:\mathtt{Int}:] \\
f^{\uparrow} \;=\; \lambda \bullet.\, \mathtt{case}^{\uparrow}\, (\mathtt{zipP}\, \langle \mathtt{zipP}\, \langle [\!:\langle g, g^{\uparrow} \rangle, \mathtt{Int} \times (\mathtt{Int} + \mathtt{Int}), \bullet :\!]\!], \\
\qquad\qquad\qquad\qquad\qquad\qquad [\!:\langle h, h^{\uparrow} \rangle, \langle\rangle, \mathtt{repP}\, \langle \mathtt{lenP}\, \bullet, \langle\rangle \rangle :\!]\!] \rangle, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathtt{snd}^{\uparrow}\, \bullet \rangle)
\end{array}
$$

Closures are lifted to array closures by leaving the function tuples unchanged, as in the case of replication, and lifting the environments.

Of course, the type of $\mathtt{case}^{\uparrow}$ undergoes the same modifications as that of `case`, i.e., the primitive now expects to be passed array closures instead of arrays of functions. Earlier, we have derived an implementation of $\mathtt{case}^{\uparrow}$ which relied on the ability to pack array of functions and therefore could not be supported in the standard lambda calculus. The combination of closure conversion and flattening, however, ensures that packing can be used by $\mathtt{case}^{\uparrow}$. The implementation given on 4.1.2 can now be reformulated in terms of array closures:

$$\texttt{case}^{\uparrow} :: \texttt{Int} \times (\overbrace{\texttt{Int} \times ([:\alpha:] \Rrightarrow [:\gamma:]) \times ([:\beta:] \Rrightarrow [:\gamma:])}^{[:(\alpha \Rightarrow \gamma) \times (\beta \Rightarrow \gamma):]}) \times (\overbrace{[:\texttt{Bool}:] \times [:\alpha:] \times [:\beta:]}^{[:\alpha+\beta:]}) \to [:\gamma:]$$
$$\begin{array}{l}
\texttt{case}^{\uparrow}\ \langle n_1, \langle \langle n_2, \langle fs, gs \rangle \rangle, \langle sel, \langle xs, ys \rangle \rangle \rangle \rangle\ = \\
\quad \textbf{let} \\
\quad\quad gs' = \texttt{packP}\ \langle \texttt{not}^{\uparrow}\ sel,\ fs \rangle \\
\quad\quad hs' = \texttt{packP}\ \langle sel,\ gs \rangle \\
\quad\quad xs' = gs' \ddagger xs \\
\quad\quad ys' = hs' \ddagger ys \\
\quad \textbf{in} \\
\quad \texttt{combineP}\ \langle sel, xs', ys' \rangle
\end{array}$$

The application of $f^{\uparrow}$ to $[:\langle 1, \texttt{Left}\ 5\rangle,\ \langle 3, \texttt{Right}\ 4\rangle,\ \langle 7, \texttt{Left}\ 2\rangle:]$ is evaluated by invoking $\texttt{case}^{\uparrow}$ with three suitably zipped array arguments:

- the array closure $\langle\!\langle :\langle g, g^{\uparrow}\rangle, \texttt{Int} \times (\texttt{Int} + \texttt{Int}), [:\langle 1, \texttt{Left}\ 5\rangle,\ \langle 3, \texttt{Right}4\rangle,\ \langle 7, \texttt{Left}\ 2\rangle:] :\rangle\!\rangle$, representing the partial elementwise application of $g^{\uparrow}$,
- the array closure $\langle\!\langle :\langle h, h^{\uparrow}\rangle, \langle\rangle, [:\langle\rangle, \langle\rangle, \langle\rangle:] :\rangle\!\rangle$ and
- the array of sums $[:\texttt{Left}\ 5, \texttt{Right}\ 4, \texttt{Left}\ 2:]$.

In $\texttt{case}^{\uparrow}$, the values of gs' and hs' are obtained by packing the two array closures according to the selector:

$$\begin{array}{rcl}
gs' & = & \texttt{packP}\ \langle [:True, False, True:], \\
 & & \quad \langle\!\langle :\langle g, g^{\uparrow}\rangle, \texttt{Int} \times (\texttt{Int} + \texttt{Int}), [:\langle 1, \texttt{Left}\ 5\rangle,\ \langle 3, \texttt{Right}\ 4\rangle,\ \langle 7, \texttt{Left}\ 2\rangle:] :\rangle\!\rangle \rangle \\
 & = & \langle\!\langle :\langle g, g^{\uparrow}\rangle, \texttt{Int} \times (\texttt{Int} + \texttt{Int}), \\
 & & \quad \texttt{packP}\ \langle [:True, False, True:],\ [:\langle 1, \texttt{Left}\ 5\rangle,\ \langle 3, \texttt{Right}\ 4\rangle,\ \langle 7, \texttt{Left}\ 2\rangle:] \rangle :\rangle\!\rangle \\
 & = & \langle\!\langle :\langle g, g^{\uparrow}\rangle, \texttt{Int} \times (\texttt{Int} + \texttt{Int}), [:\langle 1, \texttt{Left}\ 5\rangle,\ \langle 7, \texttt{Left}\ 2\rangle:] :\rangle\!\rangle \\
hs' & = & \texttt{packP}\ \langle [:False, True, False:], \langle\!\langle :\langle h, h^{\uparrow}\rangle, \langle\rangle, [:\langle\rangle, \langle\rangle, \langle\rangle:] :\rangle\!\rangle \rangle \\
 & = & \langle\!\langle :\langle h, h^{\uparrow}\rangle, \langle\rangle, \texttt{packP}\ \langle [:False, True, False:], [:\langle\rangle, \langle\rangle, \langle\rangle:] \rangle :\rangle\!\rangle \\
 & = & \langle\!\langle :\langle h, h^{\uparrow}\rangle, \langle\rangle, [:\langle\rangle:] :\rangle\!\rangle
\end{array}$$

Packing retains precisely those environments which are needed for the subsequent elementwise applications:

$$\begin{array}{rcl}
xs' & = & gs' \ddagger [:5, 2:] \\
 & = & g^{\uparrow}\ (\texttt{zipP}\ \langle [:\langle 1, \texttt{Left}\ 5\rangle,\ \langle 7, \texttt{Left}\ 2\rangle:],\ [:5,\ 2:] \rangle) \\
 & = & [:1, 7:]\ +^{\uparrow}\ [:5, 2:] \\
 & = & [:6, 9:] \\
ys' & = & hs' \ddagger [:4:] \\
 & = & h^{\uparrow}\ (\texttt{zipP}\ \langle [:\langle\rangle:],\ [:4:] \rangle) \\
 & = & [:4:]
\end{array}$$

The selector determined how the resulting arrays are combined to obtain the final result:

$$\texttt{combineP}\ \langle [:False, True, False:],\ [:6, 9:],\ [:4:] \rangle\ =\ [:6, 4, 9:]$$

It is easy to see that this is exactly the evaluation strategy depicted in Figure 4.1.

4.4 Composite closures

So far, we have only considered operations which obviously maintain the invariant that arrays of closures contain partial applications of the same function in every position and, even though we have claimed that array closures can represent arbitrary arrays, the relevant mechanisms have not been explained. We are now in the position to correct this omission, using the concatenation of array closures as a running example.

4.4.1 Concatenation

In the following, we consider the concatenation of two array closures c_f and c_g of type $[:\tau_1:] \Rrightarrow [:\tau_2:]$, where

$$c_f = \langle\!\langle \langle f, f^\uparrow \rangle, \tau_f, xs \rangle\!\rangle \quad \text{and}$$
$$c_g = \langle\!\langle \langle g, g^\uparrow \rangle, \tau_g, ys \rangle\!\rangle$$

The result of concatenating c_f and c_g will be some array closure d, again of type $[:\tau_1:] \Rrightarrow [:\tau_2:]$, such that

$$d = c_f \mathbin{+\!\!+\!\!+} c_g = \langle\!\langle \langle \texttt{cappP}, \texttt{cappP}^\uparrow \rangle, \upsilon, es \rangle\!\rangle$$

for suitable values of `cappP` and es. We will see below that `cappP` can be treated as a primitive function, i.e., a single implementation provided by the standard library is sufficient to cover all uses of concatenated array closures.

In Section 4.1.4, the elementwise application of an array of functions obtained by concatenation has been used as an example demonstrating the merits of the calculational approach to representing such arrays. In particular, the following rewriting rule (slightly adapted for the purposes of this discussion) has been derived for this operation:

$$\begin{aligned}&\texttt{zipWithP}\,\langle(\$), \texttt{mapP}\,\langle f, xs\rangle \mathbin{+\!\!+\!\!+} \texttt{mapP}\,\langle g, ys\rangle,\ zs\rangle = \\ &\quad \texttt{zipWithP}\,\langle(\$), \texttt{mapP}\,\langle f, xs\rangle, \texttt{takeP}\,\langle \texttt{lenP}\ xs, zs\rangle\rangle \\ &\quad \mathbin{+\!\!+\!\!+} \texttt{zipWithP}\,\langle(\$), \texttt{mapP}\,\langle g, ys\rangle, \texttt{dropP}\,\langle \texttt{lenP}\ xs, zs\rangle\rangle\end{aligned}$$

Here, instead of concatenating the two function arrays before the elementwise application, each of them is applied to the respective part of the argument array and the results are concatenated, as illustrated by Figure 4.2. Of course, this assumes that $\texttt{lenP}\ zs = \texttt{lenP}\ xs + \texttt{lenP}\ ys$. By taking into account the semantics of closure conversion, this equation can be rewritten as follows:

$$\begin{aligned}&(c_f \mathbin{+\!\!+\!\!+} c_g) \ddagger zs \\ &\quad = d \ddagger zs && (\textit{by definition of } d) \\ &\quad = \texttt{cappP}^\uparrow\,(\texttt{zipP}\,\langle es, zs\rangle) && (\textit{by definition of } \ddagger) \\ &\quad = (c_f \ddagger \texttt{takeP}\,\langle \texttt{lenP}\ c_f,\ zs\rangle) \mathbin{+\!\!+\!\!+} (c_g \ddagger \texttt{dropP}\,\langle \texttt{lenP}\ c_g,\ zs\rangle)\end{aligned}$$

Again, the argument array is split, allowing the concatenation to be executed after the elementwise application. But how can `cappP` and es be chosen to satisfy this condition?

The answer to this question is a direct consequence of the following observation. In $\lambda^{\mathcal{C}}$, i.e. before flattening but after closure conversion, c_f, c_g and d represent arrays of closures of type $[:\tau_1 \Rightarrow \tau_2:]$. Moreover, each element of d is equal either to some element of c_f or to some element of c_g, i.e.,

$$d = [:c_f\ !:\ 0, \ldots, c_f\ !:\ (m-1), c_g\ !:\ 0, \ldots, c_g\ !:\ (n-1):]$$

where m and n are the lengths of c_f and c_g, respectively. However, we have seen that the elementwise application of d can be reduced to applications of c_f and c_g, which suggests that splitting the two closures, either by indexing or by some other means, is unnecessary. This can, indeed, be avoided by choosing the environment es of d such that it has the $\lambda^{\mathcal{C}}$ type $[:(\tau_1 \Rightarrow \tau_2) + (\tau_1 \Rightarrow \tau_2):]$ and is computed as follows:

$$es \;=\; [:\texttt{Left}\,(c_f\,!:\,0),\;\ldots,\;\texttt{Left}\,(c_f\,!:\,(m-1)),\;\texttt{Right}\,(c_g\,!:\,0),\;\ldots,\;\texttt{Right}\,(c_g\,!:\,(n-1)):]$$

or, in other words,

$$es \;=\; \texttt{mapP}\,\langle\langle\!|\texttt{Left}|\!\rangle, c_f\rangle \;+\!\!+\; \texttt{mapP}\,\langle\langle\!|\texttt{Right}|\!\rangle, c_g\rangle$$

Here, es stores all elements of c_f and c_g; the sum constructors `Left` and `Right` indicate from which of the two arrays an element has been obtained. Note that we argue simultaneously on two levels of abstraction: the array closure d and hence, the environment es only exist after flattening but the value assigned to es is only valid before that transformation has taken place. Unfortunately, this is necessary in order to demonstrate both the semantics and the validity of the technique. Thus, before considering the impact of this representation on elementwise application and the implementation of `cappP`, let us investigate how it interacts with the flattening transformation.

Under the transformation rules given in Section 3.3.2, the type of es is translated to $[:\texttt{Bool}:] \times [:\tau_1 \Rightarrow \tau_2:] \times [:\tau_1 \Rightarrow \tau_2:]$ and then to $[:\texttt{Bool}:] \times ([:\tau_1:] \Rrightarrow [:\tau_2:]) \times ([:\tau_1:] \Rrightarrow [:\tau_2:])$ Recall that the latter two components store the left and the right elements of es, respectively, whereas the boolean values indicate whether the corresponding element has been constructed with `Left` or `Right`. Thus, es has the following flat representation:

$$es \;=\; \langle[:\underbrace{\mathit{False},\;\ldots,\;\mathit{False}}_{m\text{ times}},\;\underbrace{\mathit{True},\;\ldots,\;\mathit{True}}_{n\text{ times}}:],\;c_f,\;c_g\rangle$$

Since c_f and c_g store precisely the left and right elements of es, respectively, they can be used directly in the representation of the environment array; this is crucial synergy effect between flattening and the proposed handling of closures. Ultimately, the environment of d consists of the two array closures and an array of booleans, each indicating which closure contains the corresponding element, as shown in Figure 4.6.

Accordingly, the concatenation of two array closures is implemented as follows:

$$\begin{aligned}
&(+\!\!+\!\!+_{\langle\tau_1\Rightarrow\tau_2\rangle}) \;::\; ([:\tau_1:] \Rrightarrow [:\tau_2:]) \times ([:\tau_1:] \Rrightarrow [:\tau_2:]) \to ([:\tau_1:] \Rrightarrow [:\tau_2:])\\
&c_f \;+\!\!+\!\!+_{\langle\tau_1\Rightarrow\tau_2\rangle}\; c_g \;=\\
&\quad \langle\!|:\langle\texttt{cappP}, \texttt{cappP}^{\uparrow}\rangle,\\
&\qquad (\tau_1 \Rightarrow \tau_2) + (\tau_1 \Rightarrow \tau_2),\\
&\qquad \langle\texttt{repP}\,\langle\texttt{lenP}\;c_f, \mathit{False}\rangle \;+\!\!+\!\!+_{\langle\texttt{Bool}\rangle}\; \texttt{repP}\,\langle\texttt{lenP}\;c_g, \mathit{True}\rangle,\;\langle c_f, c_g\rangle\rangle:|\!\rangle
\end{aligned}$$

Here, the selector is computed directly by replicating *False* and *True* to the length of c_f and c_g, respectively. This implementation obviously generates precisely the desired representation.

4.4.2 Indexing

Having fixed the representation of the environment, we will now turn our attention to the implementations of the two functions `cappP` and $\texttt{cappP}^{\uparrow}$. Since the latter can be obtained by

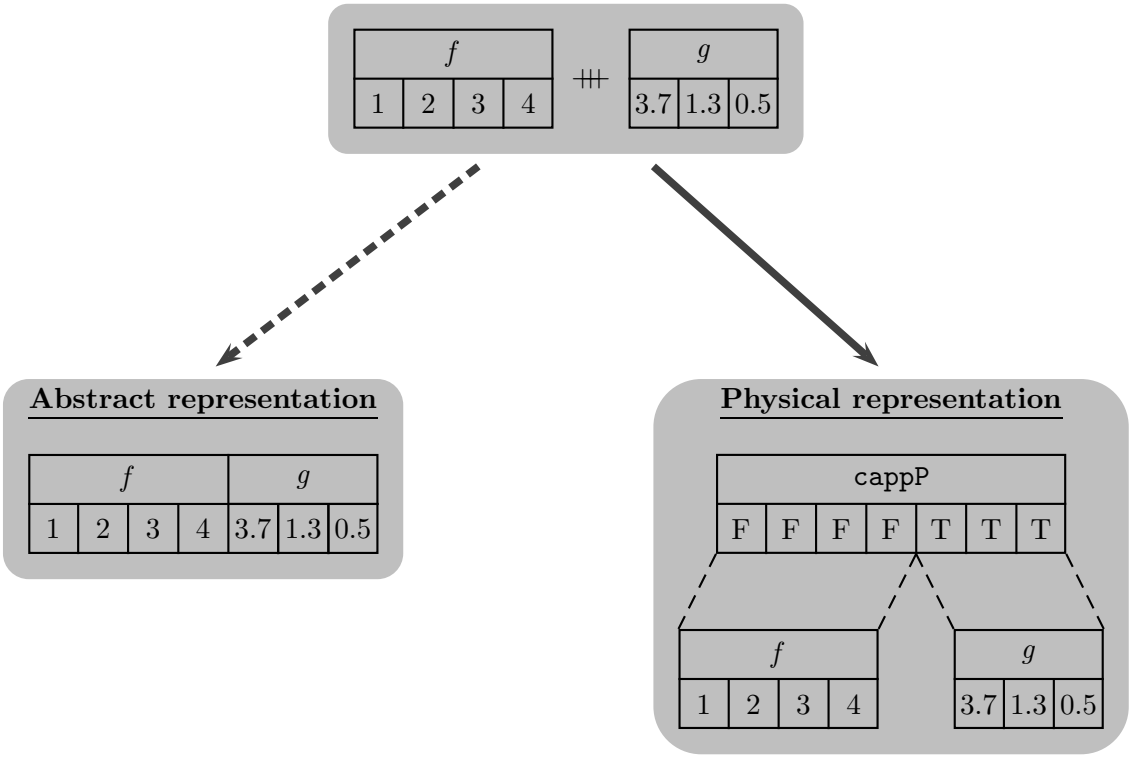


FIGURE 4.6 Concatenation of array closures

lifting the former, we consider `cappP` first. Recall that it is only used for applying a single element of d, i.e., a closure of type $\tau_1 \Rightarrow \tau_2$, to an argument. Thus, its semantics is best explained by considering such an application, as in $(d \mathbin{!:} i) \dagger x$. By the definitions of d, $(!:)$ and $(\dagger)$, this can be rewritten as

$$\begin{aligned}&(d \mathbin{!:} i) \dagger x\\ &\quad = \langle\!\langle \langle \texttt{cappP}, \texttt{cappP}^{\uparrow} \rangle,\ (\tau_1 \Rightarrow \tau_2) + (\tau_1 \Rightarrow \tau_2),\ es \mathbin{!:} i \rangle\!\rangle \dagger x\\ &\quad = \texttt{cappP}\ \langle es \mathbin{!:} i,\ x \rangle\end{aligned}$$

Depending on the value of i, the term $es \mathbin{!:} i$ evaluates to either `Left` $(c_f \mathbin{!:} i_1)$ or `Right` $(c_g \mathbin{!:} i_2)$, i.e., to a closure obtained, depending on the index, from either c_f or c_g and embedded in a sum type constructor which designates its origin. The latter information is important in a parallel context but useless when dealing with individual elements. Thus, `cappP` simply has to apply the embedded closure to x, as in the following implementation:

$$\begin{aligned}&\texttt{cappP} && :: ((\tau_1 \Rightarrow \tau_2) + (\tau_1 \Rightarrow \tau_2)) \times \tau_1 \to \tau_2\\ &\texttt{cappP}\ \langle \texttt{Left}\ c,\ x \rangle &&= c \dagger x\\ &\texttt{cappP}\ \langle \texttt{Right}\ c,\ x \rangle &&= c \dagger x\end{aligned}$$

Let us return to Figure 4.6. The second element of the concatenated closure has the value

$$\langle\!\langle \langle \texttt{cappP}, \texttt{cappP}^{\uparrow} \rangle,\ (\tau_1 \Rightarrow \tau_2) + (\tau_1 \Rightarrow \tau_2),\ \texttt{Left}\ \underbrace{\langle\!\langle \langle f, f^{\uparrow} \rangle, \texttt{Int}, 2 \rangle\!\rangle}_{c_f \mathbin{!:} 1} \rangle\!\rangle$$

The result of its application to some x is obtained by applying the closure $\langle\!\langle \langle f, f^{\uparrow} \rangle, \texttt{Int}, 2 \rangle\!\rangle$, i.e., the second element of c_f, to x. Ultimately, this performs the computation $f\ \langle 2, x \rangle$, which is exactly the desired result.

4.4.3 Elementwise application

The semantics of $\texttt{cappP}^{\uparrow}$ and, thus, of elementwise application of concatenated array closures is fixed by the above definition of `cappP`. Still, it is important to understand the underlying evaluation strategy. The implementation of $\texttt{cappP}^{\uparrow}$, as obtained by lifting `cappP`, while correct, is quite unreadable. For the sake of clarity, we present a semantically equivalent but simplified version:

$$
\begin{aligned}
&\texttt{cappP}^{\uparrow}\ ::\ ([:\texttt{Bool}:] \times (([:\tau_1:] \Rrightarrow [:\tau_2:]) \times ([:\tau_1:] \Rrightarrow [:\tau_2:]))) \times [:\tau_1:] \to [:\tau_2:] \\
&\texttt{cappP}^{\uparrow}\ \langle \langle sel, \langle c_f, c_g \rangle \rangle, xs \rangle\ =\ \textbf{let} \\
&\qquad\qquad\qquad\qquad\qquad\qquad xs_1\ =\ \texttt{packP}\ \langle \texttt{not}^{\uparrow}\ sel,\ xs \rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad xs_2\ =\ \texttt{packP}\ \langle sel,\ xs \rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad ys_1\ =\ c_f\ \ddagger\ xs_1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad ys_2\ =\ c_g\ \ddagger\ xs_2 \\
&\qquad\qquad\qquad\qquad\qquad\ \textbf{in} \\
&\qquad\qquad\qquad\qquad\qquad\ \texttt{combineP}\ \langle sel,\ \langle ys_1, ys_2 \rangle \rangle
\end{aligned}
$$

Here, $\texttt{cappP}^{\uparrow}$ is applied to the environment of an concatenated array and the argument array xs, the former consisting, as before, of a selector sel and two array closures c_f and c_g. The argument array is split into two arrays xs_1 and xs_2 as determined by the selector, such that they contain precisely those elements to which c_f and c_g, respectively, must be applied. After performing these applications, the results are combined into a single array, again according to the selector.

The implementation of $\texttt{cappP}^{\uparrow}$ is quite similar to the one suggested for $\texttt{case}^{\uparrow}$. There, however, the *argument* was an array of binary sums, which required the two closures to be packed and applied separately to its component arrays. In the case of $\texttt{cappP}^{\uparrow}$, on the other hand, the *closures* are represented by an array of sums, which is applied to an argument array by splitting the latter and then performing the actual applications.

Figure 4.7 illustrates this evaluation strategy. Note how it precisely corresponds to the one described in Section 4.1.4 and depicted in Figure 4.2. This should not come as a surprise since it has been derived by applying the calculation approach outlined in that discussion.

Crucially, these mechanisms fulfill the requirements formulated in Section 4.1. In particular, they obviously do not violate the constraints imposed by the data-parallel model. In the above example, f and g represent different computations which should not be executed simultaneously. The implementation of `cappP` accounts for this by splitting the argument array and applying the two functions one after another. How these applications will be performed depends on the structure of the respective closures. If they, too, have been obtained by concatenation or similar operations and, thus, embed multiple computations, the argument array will be split further, ensuring that the latter are not executed in parallel. However, if an array closure has been constructed such that each of its elements represents the same computation, e.g., by mapping or replication, then the application will be executed in one parallel step, retaining as much parallelism as possible.

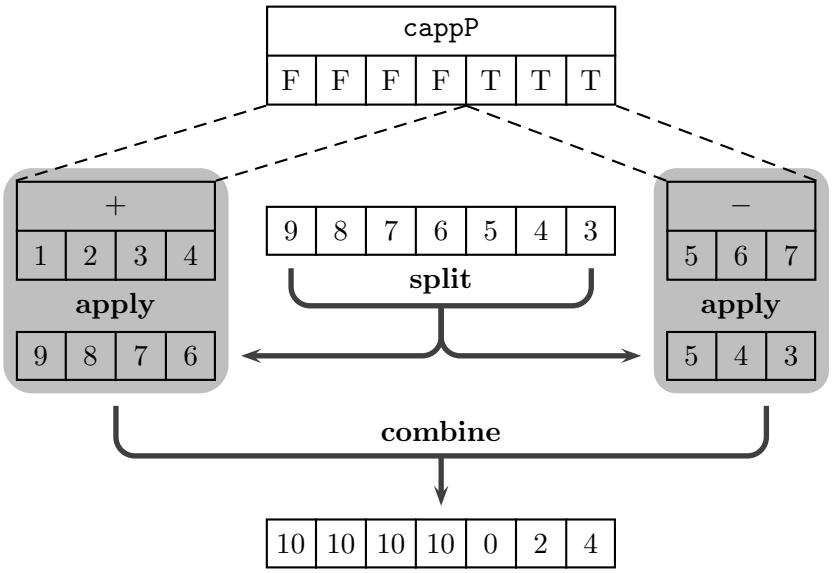


FIGURE 4.7 Application of concatenated array closures

4.4.4 Array operations

In Section 4.3.4, we have demonstrated how array primitives are implemented for array closures. Crucially, our implementation of concatenation ensures that these implementations handle concatenated closures correctly. For instance, `packP` packs an array closure by packing its environment array which, in the case of concatenated closures, stores two closures as an array of binary sums. The latter naturally supports all array operations, including packing, which will be performed by packing the selector and the two embedded array closures as required. In general, since array closures generated by concatenation do not violate any of the constraints formulated in Section 4.3.2, they are indistinguishable from those obtained by mapping or replication.

4.4.5 Combining array closures

While the approach to concatenating array closures described in the previous section is semantically sound, it is, in a sense, suboptimal. In particular, the selector always has the form $[:\mathit{False}, \ldots, \mathit{False}, \mathit{True}, \ldots, \mathit{True}:]$; clearly, it could be represented more compactly than by an array of booleans. However, the usefulness of this representation is not restricted to concatenation, as exemplified by `combineP`.

Recall that this operation, which can be seen as the inverse of packing, combines two parallel arrays into a single one according to an array of booleans. Thus, it suffers from the same problems as concatenation when applied to array closures. Although the two closures do not represent contiguous chunks of the resulting array in this case, the mechanisms used for implementing concatenation also apply to `combineP`. The two operations only differ in how the selector is obtained. Concatenation computes the selector according to the lengths of the two array closures. In the case of `combineP`, however, the array of booleans already *is* a valid selector. Thus, this operation can be implemented as follows for array closures:

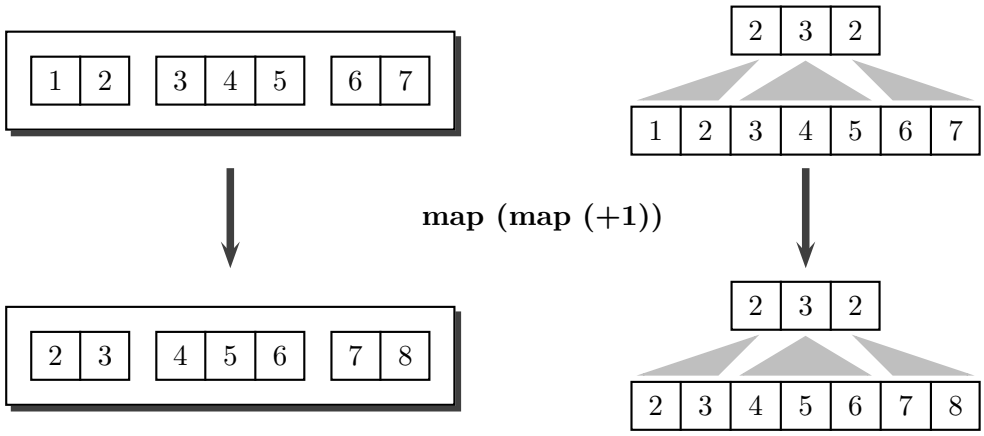


FIGURE 4.8 Nested mapping before and after flattening

$$\texttt{combineP}_{\langle\tau_1\Rightarrow\tau_2\rangle} \;::\; [:Bool:] \times ([:\tau_1:] \Rrightarrow [:\tau_2:]) \times ([:\tau_1:] \Rrightarrow [:\tau_2:]) \to ([:\tau_1:] \Rrightarrow [:\tau_2:])$$
$$\texttt{combineP}_{\langle\tau_1\Rightarrow\tau_2\rangle}\ \langle bs,\ \langle c, d\rangle\rangle \;=\; \langle\!\langle [:\langle \texttt{cappP}, \texttt{cappP}^{\uparrow}\rangle,\ (\tau_1 \Rightarrow \tau_2) + (\tau_1 \Rightarrow \tau_2),\ \langle bs,\ \langle c, d\rangle\rangle :] \rangle\!\rangle$$

It is easy to verify that the resulting closure exhibits exactly the desired semantics.

Thus, while it might be possible to implement concatenation more efficiently, the approach taken in this work has, in our view, the overwhelming advantage of being much more general and natural. Furthermore, a compiler might still provide optimised versions of specific operations; this, however, should remain an implementation detail.

4.5 Nested parallelism

The elimination of nested parallel computations is one of the crucial tasks of the flattening transformation. The archetypal example of nested parallelism is the nested application of *mapP*, as in *mapP* (*mapP* (1+)). Before demonstrating how this case is handled in our approach, let us consider the semantics of this computation.

Given a nested array of type [:[:Int:]:], it increments every integer element while preserving the array's nesting structure. This strategy is naturally supported by the flat representation of parallel arrays. Recall that after flattening, the nested array has the type [:Int:] × [:Int:], where the segment descriptor contains the lengths of the subarrays while the integers are stored in the flat data array. The desired result can be obtained by incrementing the elements of the data array — a flat data-parallel computation — and leaving the segment descriptor unchanged, as illustrated by Figure 4.8. In general, nested uses of mapping can be transformed to flat ones as follows:

$$\texttt{mapP}\ \langle\langle\!\langle \texttt{mapP}\ \langle\!\langle f \rangle\!\rangle \rangle\!\rangle,\ \langle ss, xs\rangle\rangle \;=\; \langle ss,\ \texttt{mapP}\ \langle\langle\!\langle f \rangle\!\rangle, xs\rangle\rangle$$

The application of mapP to a partially applied function f is represented by the following closure:

$$\langle\!\langle \langle \texttt{mapP}, \texttt{mapP}^{\uparrow}\rangle,\ \tau_1 \Rightarrow \tau_2,\ \underbrace{\langle\!\langle \langle f, f^{\uparrow}\rangle, \tau, e\rangle\!\rangle}_{f\ e} \rangle\!\rangle$$

Then, the definitions of `mapP` and `repP` can be unfolded in the left-hand side of the above equation:

$$\begin{aligned}&\texttt{mapP}\ \langle\!\langle\!\langle \texttt{mapP}, \texttt{mapP}^{\uparrow}\rangle,\ \tau_1 \Rightarrow \tau_2,\ \langle\!\langle\!\langle f, f^{\uparrow}\rangle, \tau, e\rangle\!\rangle\!\rangle,\ \langle ss, xs\rangle\rangle\\ &\quad = \texttt{mapP}^{\uparrow}\ (\texttt{zipP}\ \langle \texttt{repP}\ \langle \texttt{lenP}\ xss,\ \langle\!\langle\!\langle f, f^{\uparrow}\rangle, \tau, e\rangle\!\rangle\!\rangle,\ \langle ss, xs\rangle\rangle)\\ &\quad = \texttt{mapP}^{\uparrow}\ (\texttt{zipP}\ \langle [\!:\langle f, f^{\uparrow}\rangle,\ \tau,\ \texttt{repP}\ \langle \texttt{lenP}\ \langle ss, xs\rangle, e\rangle :\!]\!],\ \langle ss, xs\rangle\rangle)\end{aligned}$$

It should not come as a surprise that the two applications of `mapP` are evaluated to a single application of $\texttt{mapP}^{\uparrow}$. Thus, to see how the nested parallelism in this example is eliminated, the implementation of this lifted primitive must be investigated.

4.5.1 Lifted mapping

Given an array of closure/subarray pairs, $\texttt{mapP}^{\uparrow}$ maps each closure over the corresponding subarray, as indicated by its nested type $[\!:(\alpha \Rightarrow \beta) \times [\!:\alpha:]\!:] \to [\!:[\!:\beta:]\!:]$. Flattening transforms the latter to $\texttt{Int} \times (([\!:\alpha:] \Rrightarrow [\!:\beta:]) \times ([\!:\texttt{Int}:] \times [\!:\alpha:])) \to [\!:\texttt{Int}:] \times [\!:\beta:]$. Thus, arguments to $\texttt{mapP}^{\uparrow}$ are tuples of the form $\langle n, \langle c, \langle ss, xs\rangle\rangle\rangle$, where

- n is the length of the arrays involved in the computation,
- c is an array closure of the form $[\!\![:\langle f, f^{\uparrow}\rangle, \tau, es:]\!\!]$, where the environment array has the length n,
- ss is the segment descriptor, again of length n, of the nested argument array and
- xs is the data array.

Ultimately, our goal is to apply the array closure c to the flat data array. Note, however, that its environment has the same length as the segment descriptor; in other words, it contains as many elements as there are subarrays in the nested representation. Before the application can be performed, the environment must be expanded such that each of its elements is repeated as many times as there are elements in the corresponding subarray. The primitive `expandP` does just this: Given an array of integers and a data array, it repeats each data element as many times as specified by the corresponding integer, as in the following example:

$$\texttt{expandP}\ \langle [\!:2, 3, 2:]\!, [\!:1, 2, 3:]\!\rangle\ =\ [\!:1, 1, 2, 2, 2, 3, 3:]\!$$

This suggests that $\texttt{mapP}^{\uparrow}$ can be implemented as follows:

$$\begin{aligned}&\texttt{mapP}^{\uparrow}\ \langle n,\ \langle [\!\![:\langle f, f^{\uparrow}\rangle, \tau, es:]\!\!],\ \langle ss, xs\rangle\rangle\rangle\ =\\ &\quad \langle ss,\ [\!\![:\langle f, f^{\uparrow}\rangle,\ \tau,\ \texttt{expandP}\ \langle ss, es\rangle :]\!\!]\ \ddagger\ xs\rangle\end{aligned}$$

Thus, $\texttt{mapP}^{\uparrow}$ expands the environment of the array closure according to the segment descriptor and applies it elementwise to the data array; the segment descriptor determines the nesting structure of the result. This evaluation strategy is depicted in Figure 4.9.

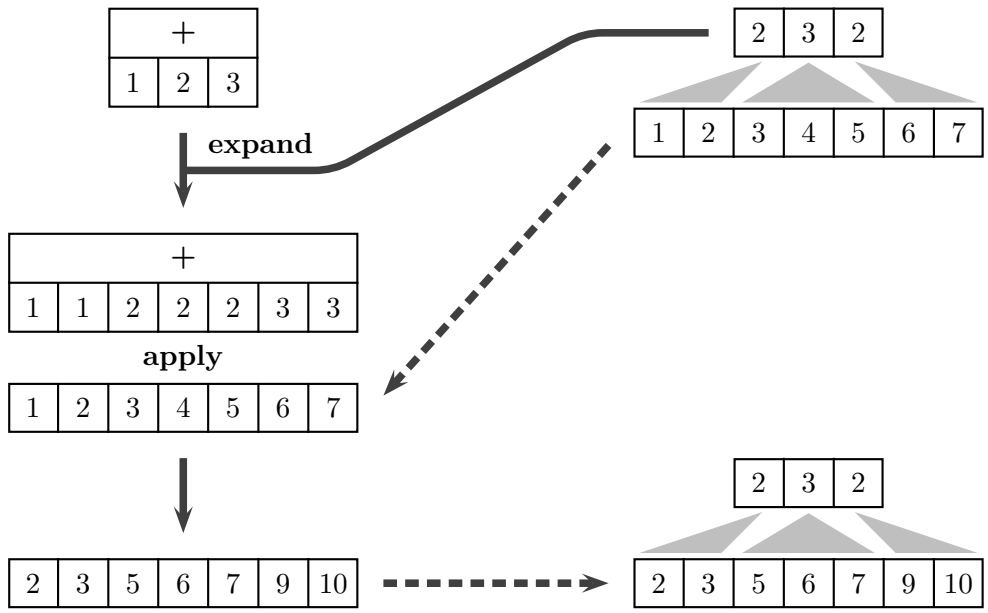


FIGURE 4.9 Evaluation of $\texttt{mapP}^{\uparrow}$

4.5.2 Nested mapping

We are now in the position to demonstrate how nested applications of `mapP` are transformed to flat computations. The definition of $\texttt{mapP}^{\uparrow}$ can be substituted into the equation derived on page 56:

$$\begin{aligned}
&\texttt{mapP}\ \langle\langle\langle \texttt{mapP}, \texttt{mapP}^{\uparrow}\rangle,\ \tau_1 \Rightarrow \tau_2,\ \langle\langle\langle f, f^{\uparrow}\rangle, \tau, e\rangle\rangle\rangle,\ \langle ss, xs\rangle\rangle \\
&\quad = \texttt{mapP}^{\uparrow}\ (\texttt{zipP}\ \langle\langle\langle :\langle f, f^{\uparrow}\rangle,\ \tau,\ \texttt{repP}\ \langle \texttt{lenP}\ \langle ss, xs\rangle, e\rangle :\rangle\rangle,\ \langle ss, xs\rangle\rangle) \\
&\quad = \langle ss,\ \langle\langle :\langle f, f^{\uparrow}\rangle,\ \tau,\ \underbrace{\texttt{expandP}\ \langle ss,\ \texttt{repP}\ \langle \texttt{lenP}\ ss, e\rangle\rangle}_{environment} :\rangle\rangle\ \ddagger\ xs\rangle
\end{aligned}$$

The last term makes it obvious that the nested application of `mapP` has been reduced to the elementwise application of a suitably derived array closure to the data array. The closure's environment is computed in two steps as follows:

- first, it is replicated to the length of the segment descriptor as specified by the definition of `mapP` such that it contains exactly one element for each subarray and
- then, $\texttt{mapP}^{\uparrow}$ expands it such that ultimately, it has the same number of elements as the data array.

It it easy to verify that this leads to the desired results by comparing this process with the evaluation strategy depicted in Figure 4.8.

As the environment will ultimately contain the same value in every position, it can be computed more efficiently by making use of the following equivalence:

$$\texttt{expandP}\ \langle ss,\ \texttt{repP}\ \langle \texttt{lenP}\ ss,\ e\rangle\rangle\ =\ \texttt{repP}\ \langle \texttt{sumP}\ ss,\ e\rangle$$

Unfortunately, it is not clear if and how this case can be recognised at runtime. We believe, however, that the impact of this inefficiency will be minimal on real-world programs. In

particular, in many cases it can be eliminated by judicious use of inlining and subsequent compiler optimisations.

This key technique for replacing nested computations by flat ones was first suggested by Blelloch (1996). In his approach, it was used for generating repeatedly lifted functions. This is not required and, in fact, not possible with the strategy described in this work. Instead, the elimination of nested computation is achieved by performing suitable operations on array closures as described above.

4.6 Arrays of functions

The combination of closure conversion and flattening used in this work ensures that functions do not occur outside of closures in generated programs. In particular, arrays of functions are replaced by arrays of closures and, theoretically, do not have to be handled by flattening. It is, however, beneficial, with respect to the uniformity of the flattening transformation as specified in Chapter 5 as well as for subsequent optimisations, if a restricted form of such arrays is supported.

Again, we require that an array of functions contain the same function in every position. This allows us to represent an array of functions of type $[:\tau_1 \to \tau_2:]$ by a triple of type $\texttt{Int} \times (\tau_1 \to \tau_2) \times ([:\tau_1:] \to [:\tau_2:])$ where

- the array's length is stored in the first component,
- the second component is some function f and
- the third component is the lifted function $f^{\uparrow}$ derived from f.

Note that here we again make use of the associations between unlifted and lifted functions introduced by vectorisation.

This representation is, in fact, very similar to the one used for closures. In contrast to closures, however, such arrays only support a very limited set of operations, mainly indexing, replication and elementwise application. In particular, concatenation and similar operations cannot be provided in any meaningful way with this representation. This does not lead to problems as such arrays are inaccessible to the programmer and can only be generated internally by the compiler. Thus, the author of the latter is responsible for only using the provided operations.

Those array primitives which can be supported by this representation are easily implemented. For instance, `lenP` simply returns the length component. Since vectorisation transforms type of the form $\tau_1 \to \tau_2$ to $(\tau_1 \to \tau_2) \times ([:\tau_1:] \to [:\tau_2:])$, replication is essentially the identity function:

$$\begin{aligned}
&\texttt{repP}_{\langle \tau_1 \to \tau_2 \rangle} && :: \texttt{Int} \times \overbrace{((\tau_1 \to \tau_2) \times ([:\tau_1:] \to [:\tau_2:]))}^{\tau_1 \to \tau_2} \\
& && \to \underbrace{\texttt{Int} \times ((\tau_1 \to \tau_2) \times ([:\tau_1:] \to [:\tau_2:]))}_{[:\tau_1 \to \tau_2:]} \\
&\texttt{repP}_{\langle \tau_1 \to \tau_2 \rangle}\ \langle n,\ \langle f, f^{\uparrow} \rangle \rangle && = \langle n,\ \langle f, f^{\uparrow} \rangle \rangle
\end{aligned}$$

Other primitives have equally simple-minded implementations.

Why, then, are arrays of functions necessary at all? The reason for this becomes clear if we consider that lifting, in general, transforms terms of type τ to those of type $[:\tau:]$.

Obviously, functions could not be handled in this way if the corresponding arrays could not be generated. While it is possible to formulate lifting such that this problem does not arise, a limited form of function arrays makes the transformation much simpler to specify and reason about. Crucially, the function arrays generated by lifting are only obtained by replication, thereby guaranteeing that this simple representation is entirely sufficient.

Chapter 5

Formalising the approach

In the previous chapters we have informally introduced our strategy to compiling nested data parallelism and described how flattening and closure conversion can be combined to support higher-order functions. It is now time to formalise the approach and show that the proposed techniques are, indeed, correct. In this, our main focus is on the flattening transformation as closure conversion is a fairly standard compilation technique and its correctness has been proved by Minamide et al. (1996), albeit for a strict language.

Nevertheless, a formal account of closure conversion is both helpful in understanding the concepts involved and necessary due to the novelty of the target language used in this work. Thus, after introducing the languages $\lambda^{\mathcal{P}}$ and $\lambda^{\mathcal{C}}$ in Sections 5.1 and 5.2, we include a detailed discussion of this transformation in Section 5.3. We do not, however, provide correctness proofs.

The rest of the chapter is devoted to the flattening transformation. Its target language, the language of flat arrays $\lambda^{\mathcal{A}}$, is defined in Section 5.4. Although it is quite similar to $\lambda^{\mathcal{C}}$, its type system is more powerful due to requirements imposed by flattening. Section 5.5 formalises the mechanisms discussed in Chapters 3 and 4. Building on the work of Blelloch and Sabot (1990), Keller (1999) and, in particular, of Chakravarty and Keller (2000), we define flattening in terms of three closely related transformations — flattening of types, vectorisation and lifting — which, when applied to a nested data-parallel program, generate the corresponding flat $\lambda^{\mathcal{A}}$ code. We also explain how they relate to the informal account of this process given earlier.

Based on this formalisation, we begin the validation of our approach by proving that flattening is correct with respect to static semantics. To this end, we establish some standard properties of the type systems of $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$ required for the subsequent discussion. Finally, in Section 5.7 we show that flattening preserves type correctness. The operational correctness of the transformation is investigated in the next chapter.

5.1 The simply-typed lambda calculus

The language $\lambda^{\mathcal{P}}$ is based on the simply typed lambda calculus with μ-recursive types and terms. It is intended as an intermediate representation of programs written in a high-level language and, therefore, does not provide any syntactic sugar. Its syntax and static semantics are, for the most part, entirely standard. The only exceptions are the support for nested data parallelism and minor technical issues required for a clean formalisation of subsequent transformations.

Constants

$$
\begin{aligned}
c^0 &::= + \mid - \mid \ldots \\
c^1 &::= \texttt{lenP} \mid \texttt{repP} \mid \ldots \\
c^2 &::= \texttt{Left} \mid \texttt{Right} \mid \texttt{mapP} \mid \ldots \\
&\ldots
\end{aligned}
$$

Literals

$$
\begin{aligned}
i &::= 0 \mid 1 \mid \ldots \\
b &::= \mathit{True} \mid \mathit{False}
\end{aligned}
$$

Types

$$\tau ::= \alpha \mid \texttt{Int} \mid \texttt{Bool} \mid \langle\rangle \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 \mid [:\tau:] \mid \mu\alpha.\tau$$

Terms

$$
\begin{aligned}
e &::= u^\tau && (\textit{annotated terms}) \\
u &::= i && (\textit{integer literals}) \\
&\mid\ b && (\textit{boolean literals}) \\
&\mid\ \langle\rangle && (\textit{unit}) \\
&\mid\ c^n_{\langle\tau_1,\ldots,\tau_n\rangle} && (\textit{constants}) \\
&\mid\ v^\tau && (\textit{variables}) \\
&\mid\ \langle e_1, e_2\rangle && (\textit{tuples}) \\
&\mid\ e.i\ (i \in \{1,2\}) && (\textit{tuple projections}) \\
&\mid\ e_1\ e_2 && (\textit{applications}) \\
&\mid\ \lambda v : \tau_1.\ e^{\tau_2} && (\textit{abstractions}) \\
&\mid\ \mu v : \tau.\ e && (\textit{recursion})
\end{aligned}
$$

FIGURE 5.1 The language $\lambda^{\mathcal{P}}$

5.1.1 Syntax

The syntax of $\lambda^{\mathcal{P}}$, given in Figure 5.1, includes the usual constructs of the simply-typed lambda calculus with recursive types. In the following, we explain the most important ones, but focus more on the non-standard extensions to this formalism. It should be noted that most of these properties carry over to $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$ as well; we will not explain them in as much detail when specifying these languages.

Types

$\lambda^{\mathcal{P}}$ has a rather large number of built-in types. In particular, it includes the unit type $\langle\rangle$, products, sums and μ-recursive types. This is both due to the requirements imposed by the flattening transformation and because we want to demonstrate how arbitrary product-sum types are handled in our approach. The built-in boolean type `Bool` is distinct from the type $\langle\rangle + \langle\rangle$. This is necessary because flattening relies on unboxed arrays of booleans for

representing sum types (cf. Section 3.3.2), which are hard to provide if `Bool` is not a primitive type. Still, the implementation can hide its existence from the programmer, exporting the standard Haskell definition of `Bool` instead, and only use it internally.

Type annotations

Following Morrisett et al. (1999), we assume that all terms are annotated with their types, as specified by the productions u (unannotated terms) and e (annotated terms) in the grammar. This simplifies the definition of transformations by making them independent of the typing derivations. The well-formedness of the annotations is checked by the typing rules, but we omit them in informal discussion to avoid clutter.

Variables

As usual, we assume all variables, both type and value ones, to be distinct. Moreover, we assume that α-equivalent types and terms (i.e. those equivalent up to renaming of bound variables) are equal. The function FVS yields value variables occuring free in a term. Unless specified otherwise, we let α, β, γ and δ range over type variables, v and w over value variables, other small latin letter over terms and τ and υ over types. We write $\upsilon[\tau/\alpha]$ for capture-avoiding substitution of types and $e_1[e_2/v]$ for capture-avoiding substitution of terms.

Constants

In Section 3.4, we have described a generic approach to implementing primitive array operations by using type-indexed definitions. Primitives defined in this way, such as `lenP`, are not functions in their own right; instead, `lenP` denotes a family of functions of the form $\texttt{lenP}_{\langle\tau\rangle}$, specialised depending on the value of τ. Each $\texttt{lenP}_{\langle\tau\rangle}$ is treated as a separate constant in the language. Thus, in the grammar of $\lambda^{\mathcal{P}}$ c^n designates an n-ary *family of constants* such that $c^n_{\langle\tau_1,\ldots,\tau_n\rangle}$ is a monomorphic constant. Note that the arity specified by the superscript refers to the number of type parameters.

Moreover, we assume that all constants are functions, i.e. have a type of the form $\tau_1 \to \tau_2$. Where necessary, a dummy parameter of unit type can be added. This does not pose a serious restriction but considerably simplifies the transformations.

Additionally, the language includes integer and boolean literals which are syntactically different from constants. While unnecessary from a purely formal point of view, this allows for a more concise presentation of examples.

Fixing a set of constants is unrealistic for a real-world language. Instead, we axiomise their properties and introduce constants required for the transformations or used for exposition purposes as necessary. This flexible approach allows us to support an arbitrary standard library, provided that the properties of the individual primitives are defined formally.

Recursion

The grammar of $\lambda^{\mathcal{P}}$ includes μ-recursive types and values which have an entirely standard semantics. In particular, we assume that for each type $\mu\alpha.\tau$ there exists a constructor $\texttt{in}_{\mu\alpha.\tau}$ and a destructor $\texttt{out}_{\mu\alpha.\tau}$.

Products and sums

Tuples are constructed by $\langle e_1, e_2 \rangle$ and deconstructed by $e.1$ and $e.2$, which extract the first and second component of a tuple, respectively. Since $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$ do not allow for curried functions, the construction of tuples must necessarily rely on built-in syntax in these languages; for the sake of uniformity, it is treated the same way in $\lambda^{\mathcal{P}}$.

The binary sums have the usual constructors `Left` and `Right`. The language does not include a pattern-matching facility. Instead, the primitive

$$\mathtt{case}_{\langle \tau_1, \tau_2, \upsilon \rangle} \;::\; (\tau_1 \to \upsilon) \times (\tau_2 \to \upsilon) \times (\tau_1 + \tau_2) \to \upsilon$$

is used for inspecting and deconstructing binary sums.

Parallel arrays

The attentive reader will have noticed that while $\lambda^{\mathcal{P}}$ includes type type constructor $[:\cdot:]$, terms of the form $[:e_1, \ldots, e_n:]$, which we have used extensively so far, are missing from its grammar. This is not an oversight, as this representation of parallel arrays would make reasoning about semantics unnecessarily hard. An inductive definition is much better suited for formalisations and proofs and, most importantly, can be used before flattening, although it will have to be resolved by the transformation.

Consequently, both $\lambda^{\mathcal{P}}$ and $\lambda^{\mathcal{C}}$ employ a definition of arrays which is closely modelled on the one customarily used for lists. To this end, we introduce two families of array constructors, `nilP` and `consP`, with the following types:[1]

$$\begin{aligned} \mathtt{nilP}_{\langle \tau \rangle} &\;::\; \langle\rangle \to [:\tau:] \\ \mathtt{consP}_{\langle \tau \rangle} &\;::\; (\mathtt{Int} \times \tau) \times [:\tau:] \to [:\tau:] \end{aligned}$$

Empty arrays are represented by $\mathtt{nilP}\,\langle\rangle$, whereas $\mathtt{consP}\langle x, xs \rangle$ denotes the array obtained by prepending x to the array xs. The syntax $[:e_1, \ldots, e_n:]$ is the easily translated according to the following scheme:

$$[:e_1, \ldots, e_n:] \;=\; \mathtt{consP}\,\langle e_1,\, \mathtt{consP}\,\langle \ldots,\, \mathtt{consP}\,\langle e_n,\, \mathtt{nilP}\,\langle\rangle\rangle\; \ldots \rangle\rangle$$

In addition to facilitating formal reasoning, this representation also allows use to cleanly formalise the strictness properties of the flattening transformation discussed in Section 6.1.

5.1.2 Static semantics

We use two kinds of contexts for the typing judgements: *type variable contexts*, which are simply sets of type variables in all three intermediate languages, and *type contexts* which in $\lambda^{\mathcal{P}}$ are sets of type assignments of the form $v : \tau$. We let Δ range over type variable contexts and Γ over type contexts. As usual, we assume that the assigments are unique, i.e. that a type context contains at most one assignment for each variable. The static semantics of $\lambda^{\mathcal{P}}$ is then defined by the following judgements.

$\Delta \vdash_{\mathcal{P}} \tau$	the type τ is valid under the context Δ
$\Gamma \vdash_{\mathcal{P}} e : \tau$	the term e has type τ under the context Γ

[1] The parameter of `nilP` is ignored; it is only necessary because primitives must have function types.

Types

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha}\ (\textsc{Var-Type}) \qquad \frac{C \in \{\texttt{Int}, \texttt{Bool}, \langle\rangle\}}{\Delta \vdash C}\ (\textsc{Prim-Type})$$

$$\frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2 \qquad \otimes \in \{\times, +, \to\}}{\Delta \vdash \tau_1 \otimes \tau_2}\ (\textsc{Bin-Type}) \qquad \frac{\Delta \vdash \tau}{\Delta \vdash [:\tau:]}\ (\textsc{Array-Type})$$

$$\frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \mu\alpha.\tau}\ (\textsc{Rec-Type})$$

Terms

$$\Gamma \vdash i \,:\, \texttt{Int}\ (\textsc{Int}) \qquad \Gamma \vdash b \,:\, \texttt{Bool}\ (\textsc{Bool}) \qquad \Gamma \vdash \langle\rangle \,:\, \langle\rangle\ (\textsc{Unit})$$

$$\frac{\vdash \tau_1 \qquad \ldots \qquad \vdash \tau_n}{\Gamma \vdash c^n_{\langle\tau_1,\ldots,\tau_n\rangle} \,:\, \mathcal{T}_{\mathcal{P}}(c^n_{\langle\tau_1,\ldots,\tau_n\rangle})}\ (\textsc{Const}) \qquad \frac{v : \tau \in \Gamma}{\Gamma \vdash v^\tau \,:\, \tau}\ (\textsc{Var})$$

$$\frac{\Gamma \vdash e_1 \,:\, \tau_1 \qquad \Gamma \vdash e_2 \,:\, \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle \,:\, \tau_1 \times \tau_2}\ (\textsc{Tup}) \qquad \frac{\Gamma \vdash e \,:\, \tau_1 \times \tau_2}{\Gamma \vdash e.i \,:\, \tau_i}\ (\textsc{Proj})$$

$$\frac{\Gamma \vdash e_1 \,:\, \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \,:\, \tau_1}{\Gamma \vdash e_1\, e_2 \,:\, \tau_2}\ (\textsc{App}) \qquad \frac{\vdash \tau_1 \qquad \Gamma, v : \tau_1 \vdash e \,:\, \tau_2}{\Gamma \vdash \lambda v : \tau_1.\, e \,:\, \tau_1 \to \tau_2}\ (\textsc{Lam})$$

$$\frac{\vdash \tau \qquad \Gamma, v : \tau \vdash e \,:\, \tau}{\Gamma \vdash \mu v : \tau.\, e \,:\, \tau}\ (\textsc{Rec}) \qquad \frac{\Gamma \vdash u : \tau}{\Gamma \vdash u^\tau : \tau}\ (\textsc{Ann})$$

FIGURE 5.2 Static semantics of $\lambda^{\mathcal{P}}$

The typing rules are given in Figure 5.2. We write $\vdash_{\mathcal{P}} \tau$ and $e : \tau$ for $\emptyset \vdash_{\mathcal{P}} \tau$ and $\emptyset \vdash_{\mathcal{P}} e : \tau$, respectively, and omit the subscript $\mathcal{P}$ whereever the omission does not lead to ambiguities.

For constants, the type is determined by the function $\mathcal{T}_{\mathcal{P}}(\cdot)$, which is introduced by the following definition.

5.1 DEFINITION (TYPES OF CONSTANTS IN $\lambda^{\mathcal{P}}$)
Let $c^n_{\langle\tau_1,\ldots,\tau_n\rangle}$ be a constant in $\lambda^{\mathcal{P}}$. Then, $\mathcal{T}_{\mathcal{P}}(c^n_{\langle\tau_1,\ldots,\tau_n\rangle})$ denotes its type which satisfies the following conditions.

1. It is a function type, i.e. there exist two $\lambda^{\mathcal{P}}$ types υ_1 and υ_2 such that $\mathcal{T}_{\mathcal{P}}(c^n_{\langle\tau_1,\ldots,\tau_n\rangle}) = \upsilon_1 \to \upsilon_2$.

2. It is a valid type if all type arguments are valid, i.e., $\vdash \mathcal{T}_{\mathcal{P}}(c^n_{\langle\tau_1,\ldots,\tau_n\rangle})$ if $\vdash \tau_i$ for each $1 \le i \le n$.

5.2 The language of explicit closures $\lambda^{\mathcal{C}}$

The language of explicit closures $\lambda^{\mathcal{C}}$ is the target language for closure conversion and the source language for the flattening transformation. It is closely modelled on $\lambda^{\mathcal{P}}$, sharing with the latter the important properties of the type system and most of the syntax. Therefore, it, too, can be considered a variant of the simply typed lambda calculus. There are two major differences, however: the absence of named value variables and the explicit support for closures, which have been described in detail in Section 4.2. Especially the treatment of value variables is novel and has not, to our knowledge, been hitherto proposed in this form. The flattening transformation relies crucially on this aspect of $\lambda^{\mathcal{C}}$ — it is not clear if and how nested data parallelism can be compiled in the presence of both higher-order functions and named value variables.

5.2.1 Syntax

The syntax of $\lambda^{\mathcal{C}}$ is given in Figure 5.3. In the following, we describe its major features and point out the differences as compared to $\lambda^{\mathcal{P}}$.

Value variables

As described in Section 4.2, closure conversion completely eliminates named value variables. Thus, at every point in a $\lambda^{\mathcal{C}}$ program, at most one value variable is in scope, namely the parameter of the innermost lambda abstraction, which we denote by $\bullet$. Lambda abstractions and recursion have the form $\lambda\bullet : \tau.\ e$ and $\mu\bullet : \tau.\ e$, respectively.

Closures

Closures have already been explained in detail in Section 4.2. Compared to the simplified form used there, in $\lambda^{\mathcal{C}}$ they additionally store their argument and result type. Even though this information is recorded by type annotations, this will allow us to omit the latter in the operational semantics defined in Chapter 6. Thus, a closure has the form $\langle\!\langle e_1, \tau_1, \tau_2, \tau_3, e_2 \rangle\!\rangle$ where e_1 is the closure function of type $\tau_1 \times \tau_2 \to \tau_3$ and e_2 the environment of type τ_1. Such a closure has the type $\tau_2 \Rightarrow \tau_3$ and represents the partial application of e_1 to e_2. The built-in operator $\dagger$ applies a closure to an argument, which is equivalent to applying the closure function to the environment tupled with the argument.

Parallel arrays

Parallel arrays in $\lambda^{\mathcal{C}}$, including arrays of closures, are encoded much like they are in $\lambda^{\mathcal{P}}$, based on the two constructors `nilP` and `consP`. In particular, array closures described in Section 4.3 are absent from $\lambda^{\mathcal{C}}$ — this mechanism only applies to the flat arrays in $\lambda^{\mathcal{A}}$. This implies that a specialised treatment of parallel arrays during closure conversion is unnecessary as the techniques used for translating constants handle the array constructors transparently.

5.2.2 Static semantics

In the type system of $\lambda^{\mathcal{C}}$, type contexts are restricted to reflect the lack of named variables in $\lambda^{\mathcal{C}}$. Recall that a type context assigns types to term variables. Since at most one such

Constants

$$
\begin{array}{rcl}
c^0 & ::= & + \mid - \mid \ldots \\
c^1 & ::= & \texttt{lenP} \mid \texttt{repP} \mid \ldots \\
\ldots & &
\end{array}
$$

Literals

$$
\begin{array}{rcl}
i & ::= & 0 \mid 1 \mid \ldots \\
b & ::= & \mathit{True} \mid \mathit{False}
\end{array}
$$

Types

$$\tau ::= \alpha \mid \texttt{Int} \mid \texttt{Bool} \mid \langle\rangle \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 \mid \tau_1 \Rightarrow \tau_2 \mid [\!:\tau:\!] \mid \mu\alpha.\tau$$

Terms

$$
\begin{array}{rcll}
e & ::= & u^{\tau} & (\textit{annotated terms}) \\
u & ::= & i & (\textit{integer literals}) \\
 & \mid & b & (\textit{boolean literals}) \\
 & \mid & \langle\rangle & (\textit{unit}) \\
 & \mid & c^n_{\langle\tau_1,\ldots,\tau_n\rangle} & (\textit{constants}) \\
 & \mid & \bullet & (\textit{innermost parameter}) \\
 & \mid & \langle e_1, e_2\rangle & (\textit{tuples}) \\
 & \mid & e.i \ (i \in \{1,2\}) & (\textit{tuple projections}) \\
 & \mid & e_1\ e_2 & (\textit{applications}) \\
 & \mid & \lambda\bullet : \tau.\ e & (\textit{abstractions}) \\
 & \mid & \langle\!\langle e_1, \tau_1, \tau_2, \tau_3, e_2\rangle\!\rangle & (\textit{closures}) \\
 & \mid & e_1 \dagger e_2 & (\textit{closure applications}) \\
 & \mid & \mu\bullet : \tau.\ e & (\textit{recursion})
\end{array}
$$

FIGURE 5.3 The language of explicit closures $\lambda^{\mathcal{C}}$

variable, namely $\bullet$, can occur free in a term, the type contexts contain at most one such assignment. Accordingly, type contexts in $\lambda^{\mathcal{C}}$ (and $\lambda^{\mathcal{A}}$) conform to the following grammar:

$$\Gamma\ =\ \cdot \mid \bullet : \tau$$

The static semantics of the language is defined by the following judgements:

$\Delta \vdash_{\mathcal{C}} \tau$	the type τ is valid under the context Δ
$\Gamma \vdash_{\mathcal{C}} e : \tau$	the term e has type τ under the context Γ

Figure 5.4 details the typing rules.

The rules LAM and REC capture the visibility of $\bullet$ by replacing the type context instead of adding to it. The rules for closures and closure applications correspond to their semantics as explained in Section 4.2. In particular, the type of the environment does not appear in the type of the closure but must be compatible with the type of the closure function.

Types

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha}\ (\textsc{Var-Type}) \qquad \frac{C \in \{\texttt{Int}, \texttt{Bool}, \langle\rangle\}}{\Delta \vdash C}\ (\textsc{Prim-Type})$$

$$\frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2 \qquad \otimes \in \{\times, +, \rightarrow, \Rightarrow\}}{\Delta \vdash \tau_1 \otimes \tau_2}\ (\textsc{Bin-Type}) \qquad \frac{\Delta \vdash \tau}{\Delta \vdash [:\tau:]}\ (\textsc{Array-Type})$$

$$\frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \mu\alpha.\tau}\ (\textsc{Rec-Type})$$

Terms

$$\Gamma \vdash i \ : \ \texttt{Int}\ (\textsc{Int}) \qquad \Gamma \vdash b \ : \ \texttt{Bool}\ (\textsc{Bool}) \qquad \Gamma \vdash \langle\rangle \ : \ \langle\rangle\ (\textsc{Unit})$$

$$\frac{\vdash \tau_1 \qquad \ldots \qquad \vdash \tau_n}{\Gamma \vdash c^n_{\langle \tau_1, \ldots, \tau_n \rangle} \ : \ \mathcal{T}_{\mathcal{C}}(c^n_{\langle \tau_1, \ldots, \tau_n \rangle})}\ (\textsc{Const}) \qquad \bullet : \tau \vdash \bullet : \tau\ (\textsc{Var})$$

$$\frac{\Gamma \vdash e_1 \ : \ \tau_1 \qquad \Gamma \vdash e_2 \ : \ \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle \ : \ \tau_1 \times \tau_2}\ (\textsc{Tup}) \qquad \frac{\Gamma \vdash e \ : \ \tau_1 \times \tau_2}{\Gamma \vdash e.i \ : \ \tau_i}\ (\textsc{Proj})$$

$$\frac{\Gamma \vdash e_1 \ : \ \tau_1 \rightarrow \tau_2 \qquad \Delta; \Gamma \vdash e_2 \ : \ \tau_1}{\Gamma \vdash e_1\, e_2 \ : \ \tau_2}\ (\textsc{App}) \qquad \frac{\vdash \tau_1 \qquad \bullet : \tau_1 \vdash e \ : \ \tau_2}{\lambda\bullet : \tau_1.\, e \ : \ \tau_1 \rightarrow \tau_2}\ (\textsc{Lam})$$

$$\frac{\vdash \tau \qquad \bullet : \tau \vdash e \ : \ \tau}{\mu\bullet : \tau.\, e \ : \ \tau}\ (\textsc{Rec}) \qquad \frac{\Gamma \vdash e_1 \ : \ \tau_1 \times \tau_2 \rightarrow \tau_3 \qquad \Gamma \vdash e_2 \ : \ \tau_1}{\Gamma \vdash \langle\!\langle e_1, \tau_1, \tau_2, \tau_3, e_2 \rangle\!\rangle \ : \ \tau_2 \Rightarrow \tau_3}\ (\textsc{Clo})$$

$$\frac{\Gamma \vdash e_1 \ : \ \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash e_2 \ : \ \tau_1}{\Gamma \vdash e_1 \dagger e_2 \ : \ \tau_2}\ (\textsc{CloApp}) \qquad \frac{\Gamma \vdash u : \tau}{\Gamma \vdash u^{\tau} : \tau}\ (\textsc{Ann})$$

FIGURE 5.4 Static semantics of $\lambda^{\mathcal{C}}$

Constants are handled much like in $\lambda^{\mathcal{P}}$: the function $\mathcal{T}_{\mathcal{C}}(\cdot)$ yields the constant's type which must satisfy the same validity requirements.

5.2 DEFINITION (TYPES OF CONSTANTS IN $\lambda^{\mathcal{C}}$)
Let $c^n_{\langle \tau_1, \ldots, \tau_n \rangle}$ be a constant in $\lambda^{\mathcal{C}}$. Then, $\mathcal{T}_{\mathcal{C}}(c^n_{\langle \tau_1, \ldots, \tau_n \rangle})$ denotes its type which satisfies the following requirements.

1. It is a function type, i.e. there exist two $\lambda^{\mathcal{C}}$ types υ_1 and υ_2 such that $\mathcal{T}_{\mathcal{C}}(c^n_{\langle \tau_1, \ldots, \tau_n \rangle}) \ = \ \upsilon_1 \rightarrow \upsilon_2$.

2. It is valid if all type arguments are valid, i.e. $\vdash \mathcal{T}_{\mathcal{C}}(c^n_{\langle \tau_1, \ldots, \tau_n \rangle})$ if $\vdash \tau_i$ for each $1 \leq i \leq n$.

It is important to point out that in $\lambda^{\mathcal{C}}$, too, constants must be functions, even though one might expect them to be replaced by closures. This task is instead performed by closure conversion, as discussed in the next section.

5.3 Closure conversion

Closure conversion eliminates all free variables in functions and replaces the latter by closures, which store the function code and the embedded data separately. This is achieved by making every variable appearing free in a function part of the argument. These variables are then collected in the environment which, together with the now closed function, makes up the function's closure. Such closures are applied by extracting the environment, tupling it with the argument and passing the result to the function which obtains all necessary data from this tuple.

This transformation has been studied extensively. In particular, Minamide et al. (1996) provide a formulation of closure conversion for the simply typed lambda calculus and show its correctness. The The formalisation of this transformation presented below is largely based on this work, but, crucially, uses the novel target language $\lambda^{\mathcal{C}}$ which syntactically enforces the important constraints on closure-converted programs.

Since closure conversion is a well-researched compilation technique, we do not discuss it in as much detail as flattening in this dissertation. In particular, we do not prove its correctness, neither with respect to static nor to dynamic semantics. The formalisation is only provided to highlight the underlying concepts, demonstrate their feasibility and, most importantly, justify the design of $\lambda^{\mathcal{C}}$.

The closure conversion algorithm is given in Figure 5.5. It is specified as two transformations: $\mathcal{C}_\tau[\![\cdot]\!]$ translates types, whereas $\mathcal{C}_E[\![\cdot,\cdot]\!]$ converts type-annotated terms. Since this specification is not used for formal reasoning, we extend the syntax of $\lambda^{\mathcal{C}}$ slightly by making use of n-ary tuples $\langle e_1, \ldots, e_n \rangle$ even though $\lambda^{\mathcal{C}}$ only includes binary tuples. The translation from the former to the latter is trivial but tedious and would significantly clutter the presentation without leading to any new insights. Moreover, we omit type annotations except when they are essential for the correctness of a rule.

Closure conversion must perform the following tasks:

- replace all function types by corresponding closure types,
- eliminate named variables, replacing them by references to the innermost parameter $\bullet$,
- replace lambda abstractions by closures storing a closed function and an environment,
- replace constants, which are necessarily functions, by suitable closures,
- transform μ-recursion, which also relies on named variables and
- convert function applications to closure applications.

The first and last goals are easily achieved by the corresponding transformation rules. The other four, however, are considerably more involved. In the following, we provide a detailed explaination for each of them.

5.3.1 Variables

In addition to a $\lambda^{\mathcal{P}}$ term, $\mathcal{C}_\tau[\![\cdot,\cdot]\!]$ is parametrised by the *conversion context* $\mathcal{E}$ which is populated by the rule translating lambda abstractions. The conversion context is a set of mappings of the form $v \mapsto e$, where v is a $\lambda^{\mathcal{P}}$ variable visible in the current scope and e the $\lambda^{\mathcal{C}}$ term by which v should be replaced. When the transformation encounters v, it simply looks up the mapping in the context and performs this replacement.

Types

$$
\begin{array}{lcl}
\mathcal{C}_\tau[\![\alpha]\!] & = & \alpha \\
\mathcal{C}_\tau[\![\texttt{Bool}]\!] & = & \texttt{Bool} \\
\mathcal{C}_\tau[\![\texttt{Int}]\!] & = & \texttt{Int} \\
\mathcal{C}_\tau[\![\langle\rangle]\!] & = & \langle\rangle \\
\mathcal{C}_\tau[\![\tau_1 \times \tau_2]\!] & = & \mathcal{C}_\tau[\![\tau_1]\!] \times \mathcal{C}_\tau[\![\tau_2]\!] \\
\mathcal{C}_\tau[\![\tau_1 + \tau_2]\!] & = & \mathcal{C}_\tau[\![\tau_1]\!] + \mathcal{C}_\tau[\![\tau_2]\!] \\
\mathcal{C}_\tau[\![\tau_1 \to \tau_2]\!] & = & \mathcal{C}_\tau[\![\tau_1]\!] \Rightarrow \mathcal{C}_\tau[\![\tau_2]\!] \\
\mathcal{C}_\tau[\![[:\tau:]]\!] & = & [:\mathcal{C}_\tau[\![\tau]\!]:] \\
\mathcal{C}_\tau[\![\mu\alpha.\tau]\!] & = & \mu\alpha.\mathcal{C}_\tau[\![\tau]\!]
\end{array}
$$

Terms

$$
\begin{array}{lcl}
\mathcal{C}_E[\![\mathcal{E},\ i]\!] & = & i \\
\mathcal{C}_E[\![\mathcal{E},\ b]\!] & = & b \\
\mathcal{C}_E[\![\mathcal{E},\ \langle\rangle]\!] & = & \langle\rangle \\
\mathcal{C}_E[\![\mathcal{E},\ (c^n_{\langle\tau_1,\ldots,\tau_n\rangle})^{\upsilon_1\to\upsilon_2}]\!] & = & \langle\!\langle f,\ \tau_{env},\ \mathcal{C}_\tau[\![\upsilon_1]\!],\ \mathcal{C}_\tau[\![\upsilon_2]\!],\ env \rangle\!\rangle \\
\quad \textbf{where} & & \\
\qquad env & = & \langle\rangle \\
\qquad \tau_{env} & = & \langle\rangle \\
\qquad p & = & \mathcal{C}_C(c^n)_{\langle\mathcal{C}_\tau[\![\tau_1]\!],\ldots,\mathcal{C}_\tau[\![\tau_n]\!]\rangle} \\
\qquad f & = & \lambda\bullet : \tau_{env} \times \mathcal{C}_\tau[\![\upsilon_1]\!].\ (p\ \bullet.2) \\
\mathcal{C}_E[\![\mathcal{E},\ v^\tau]\!] & = & e \textbf{ where } v \mapsto e \in \mathcal{E} \\
\mathcal{C}_E[\![\mathcal{E},\ \langle e_1, e_2\rangle]\!] & = & \langle \mathcal{C}_E[\![\mathcal{E},\ e_1]\!],\ \mathcal{C}_E[\![\mathcal{E},\ e_2]\!]\rangle \\
\mathcal{C}_E[\![\mathcal{E},\ e.i]\!] & = & \mathcal{C}_E[\![\mathcal{E},\ e]\!].i \\
\mathcal{C}_E[\![\mathcal{E},\ e_1\ e_2]\!] & = & \mathcal{C}_E[\![\mathcal{E},\ e_1]\!]\ \dagger\ \mathcal{C}_E[\![\mathcal{E},\ e_2]\!] \\
\mathcal{C}_E[\![\mathcal{E},\ (\lambda v : \tau_1.\ e)^{\tau_1\to\tau_2}]\!] & = & \langle\!\langle f,\ \tau_{env},\ \mathcal{C}_\tau[\![\tau_1]\!],\ \mathcal{C}_\tau[\![\tau_2]\!],\ env \rangle\!\rangle \\
\quad \textbf{where} & & \\
\qquad \{w_1^{\upsilon_1},\ \ldots,\ w_n^{\upsilon_n}\} & = & FVS(\lambda v : \tau_1.\ e^{\tau_2}) \\
\qquad env & = & \langle e_1, \ldots, e_n\rangle \textbf{ where } w_i \mapsto e_i \in \mathcal{E} \textbf{ for each } 1 \le i \le n \\
\qquad \tau_{env} & = & \mathcal{C}_\tau[\![\upsilon_1]\!] \times \cdots \times \mathcal{C}_\tau[\![\upsilon_n]\!] \\
\qquad \mathcal{E}' & = & \{w_1 \mapsto \bullet.1.1,\ \ldots,\ w_n \mapsto \bullet.1.n\} \cup \{v \mapsto \bullet.2\} \\
\qquad f & = & \lambda\bullet : \tau_{env} \times \mathcal{C}_\tau[\![\tau_1]\!].\ \mathcal{C}_E[\![\mathcal{E}',\ e]\!] \\
\mathcal{C}_E[\![\mathcal{E},\ \mu v : \tau.\ e]\!] & = & (\mu\bullet : \tau_{env}.\ \langle\!\langle f,\ \tau_{env},\ \tau_{arg},\ \mathcal{C}_\tau[\![\tau]\!],\ \bullet \rangle\!\rangle)\ \dagger\ arg \\
\quad \textbf{where} & & \\
\qquad \{w_1^{\upsilon_1},\ \ldots,\ w_n^{\upsilon_n}\} & = & FVS(\mu v : \tau.\ e) \\
\qquad arg & = & \langle e_1, \ldots, e_n\rangle \textbf{ where } w_i \mapsto e_i \in \mathcal{E} \\
\qquad \tau_{arg} & = & \mathcal{C}_\tau[\![\upsilon_1]\!] \times \cdots \times \mathcal{C}_\tau[\![\upsilon_n]\!] \\
\qquad \mathcal{E}' & = & \{w_1 \mapsto \bullet.2.1,\ \ldots,\ w_n \mapsto \bullet.2.n\} \cup \{v \mapsto \bullet.1\ \dagger\ \bullet.2\} \\
\qquad \tau_{env} & = & \tau_{arg} \Rightarrow \mathcal{C}_\tau[\![\tau]\!] \\
\qquad f & = & \lambda\bullet : \tau_{env} \times \tau_{arg}.\ \mathcal{C}_E[\![\mathcal{E}', e]\!]
\end{array}
$$

FIGURE 5.5 Closure conversion

5.3.2 Functions

The most interesting rule is the one converting lambda abstractions of the form $(\lambda v : \tau_1.\ e)^{\tau_1 \to \tau_2}$ to suitable closures. First, the variables $w_1, \ldots, w_n$ occuring free in the abstraction as well as their types $\upsilon_1, \ldots, \upsilon_n$ are determined. The corresponding $\lambda^{\mathcal{C}}$ terms $e_1, \ldots, e_n$, obtained from the conversion context, form the closure environment *env* which has the type τ_{env}. The transformed function will be applied to environment/argument pairs of type $\tau_{env} \times \mathcal{C}_\tau[\![\tau_1]\!]$. Thus, within the lambda abstraction each occurence of a free variable w_i must be replaced by $\bullet.1.i$, which extracts the corresponding component from the environment, while the parameter v is replaced by $\bullet.2$. This information is recorded in the new context $\mathcal{E}'$ and used for converting the abstraction term e. Finally, the closed function f is constructed which encodes precisely the same computation as the original lambda abstraction.

5.3.3 Constants

Constants are transformed analogously to lambda abstractions; however, due to the absence of free variables the corresponding rule is considerably simpler. We assume that for every family of constants c^n in $\lambda^{\mathcal{P}}$, $\mathcal{C}_C(c^n)$ yields the corresponding family in $\lambda^{\mathcal{C}}$. Since constants are closed functions, the environment of the generated closure must be empty. The function f simply discards the environment and applies the $\lambda^{\mathcal{C}}$ constant to the new argument.

5.3.4 Recursion

The transformation of recursive terms also relies on closures to factor out free variables. The mechanism described here is similar to the one suggested by Morrisett and Harper (1998). The key idea is to transform a recursive value to a partial application of a recursive function which, in turn, can be represented by a *recursive closure*.

This transformation can be justified by considering the semantics of fixpoint recursion in $\lambda^{\mathcal{P}}$. Given a function *fix* with $\mathit{fix}\, f \;=\; f\ (\mathit{fix}\, f)$, we have:

$$\begin{aligned} \mu v.e \;&=\; \mathit{fix}\ (\lambda v.\ e) \\ &=\; \mathit{fix}\ (\lambda f.\ \lambda \mathit{arg}.\ e[f\ \mathit{arg}/v])\ x \\ &=\; (\mu f.(\lambda f.\ \lambda \mathit{arg}.\ e[f\ \mathit{arg}/v])\ f)\ x \end{aligned}$$

The first step is just the definition of μ-recursion. The second step adds a dummy parameter arg to the function and replaces all occurences of the original parameter v by $f\ w$. Finally, the last step replaces *fix* by mu-recursion.

The important step is the introduction of the dummy parameter arg. Since it is never inspected, it can be bound to any value. In particular, it can be used to store the values of variables that occur free in e:

$$\begin{aligned} &(\mu f.(\lambda f.\ \lambda \mathit{arg}.\ e[f\ \mathit{arg}/v])\ f)\ x \\ =\ &(\mu f.(\lambda f.\ \lambda \mathit{arg}.\ e[f\ \mathit{arg}/v, \mathit{arg}.1/w_1, \ldots, \mathit{arg}.n/w_n])\ f)\ \langle w_1, \ldots, w_n \rangle \\ &\textbf{where}\ \{w_1, \ldots, w_n\} \;=\; \mathit{FVS}(\lambda f.\ \lambda \mathit{arg}.\ e[f\ \mathit{arg}/v]) \end{aligned}$$

Crucially, the partially applied function under μ is now closed. Thus, it can be uncurried and replaced by a $\lambda^{\mathcal{C}}$ closure. The environment of the latter is f, which is recursively bound to the closure itself. The closure is then applied to the free variables of the original term, yielding a term with the desired semantics. Note that in contrast to lambda abstractions,

the environment does not store the free variables in this case but is used for encoding the recursion.

5.3 EXAMPLE
Consider the term $\mu xs.(x + y) \;:\; xs$, where (:) is Haskell's lazy list constructor and x and y are free variables. It is easy to see that it is equivalent to

$$(\mu f.(\lambda f.\, \lambda arg.\, (arg.1 + arg.2) \;:\; f\ arg)\, f)\, \langle x, y\rangle$$

This recursive term can be represented by the following $\lambda^{\mathcal{C}}$ closure:

$$\mu\bullet.\, \langle\!\langle \lambda\bullet.\, (\bullet.2.1 + \bullet.2.2) \;:\; \bullet.1 \dagger \bullet.2,\, (\texttt{Int} \times \texttt{Int}) \Rightarrow [\texttt{Int}],\, \texttt{Int} \times \texttt{Int},\, [\texttt{Int}],\, \bullet \rangle\!\rangle$$

Inside the lambda abstraction, $\bullet.1$ refers the closure itself and $\bullet.2.1$ and $\bullet.2.2$ to the values of x and y, respectively. When applied to $\langle x, y\rangle$, the closure yields an infinite list of $x + y$, just like the original term.

5.4 The language of flat arrays $\lambda^{\mathcal{A}}$

The language of flat arrays $\lambda^{\mathcal{A}}$ shares most of its structure with $\lambda^{\mathcal{C}}$. As the target language for the flattening transformation, it removes support for nested arrays and includes unboxed arrays and array closures instead. Moreover, it features a slightly more complex type system necessary for the formalisation of flattening for types.

5.4.1 Syntax

Figure 5.6 defines the syntax of $\lambda^{\mathcal{A}}$. Note that type-annotated terms are missing from the language, since they are no longer necessary. Compared to $\lambda^{\mathcal{C}}$, the type constructor $[:\cdot:]$ is replaced by the unboxed array types $\texttt{Arr}_{\texttt{Int}}$ and $\texttt{Arr}_{\texttt{Bool}}$ and the type of array closures $\tau_1 \Rrightarrow \tau_2$. The term grammar includes array closures $\langle\!\langle :e_1, \tau_1, \tau_2, \tau_3, e_2: \rangle\!\rangle$ and applications of such closures of the form $e_1 \ddagger e_2$. These concepts have been discussed in detail in Section 4.3.

The flat representation of parallel arrays discussed in Section 3.3 relies on unboxed arrays of integers and booleans. These have the types $\texttt{Arr}_{\texttt{Int}}$ and $\texttt{Arr}_{\texttt{Bool}}$ and are represented by literals of the form $\{i_1, \ldots, i_n\}_{\texttt{Int}}$ and $\{b_1, \ldots, b_n\}_{\texttt{Bool}}$. Note that the grammar ensures that these are indeed literals by only allowing constant values to be used for their construction.

In addition to these changes, flattening requires some support from the type system. In $\lambda^{\mathcal{C}}$, the two types τ and $[:\tau:]$ are clearly related. Flattening, however, transforms them to two $\lambda^{\mathcal{A}}$ types which are syntactically unrelated. In particular, there is no way (or, at least, we do not require that one exist) to obtain one from the other. In general, the association between an array type and the type of its elements is lost after flattening.

This association must be preserved in some cases discussed below, however. We solve this problem by introducing *type tuples* in $\lambda^{\mathcal{A}}$. A type tuple is a type of the form $\langle\tau_1, \tau_2\rangle$ where τ_1 and τ_2 are again types. The semantics is similar to that of term tuples; in fact, we reuse the syntax $\tau.i$ $(i \in \{1, 2\})$ for denoting the extraction of the ith component from a type tuple. In contrast to terms, however, the construction and destruction of type tuples are resolved statically.

In Section 5.5.1 we will see that a $\lambda^{\mathcal{C}}$ type υ is transformed to a tuple $\langle\tau_1, \tau_2\rangle$, where τ_1 and τ_2 are the flat representations of, respectively, υ and $[:\upsilon:]$. These *type associations* (cf. Definition 5.5) are used whenever a mapping between the two is necessary for ensuring the correctness of the transformation.

Kinds

$$\kappa \ ::= \ \star \mid \star \times \star$$

Types

$$\begin{aligned}\tau \ ::= \ & \alpha \mid \texttt{Int} \mid \texttt{Bool} \mid \texttt{Arr}_{\texttt{Int}} \mid \texttt{Arr}_{\texttt{Bool}} \mid \langle\rangle \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 \mid \tau_1 \Rightarrow \tau_2 \mid \tau_1 \Rrightarrow \tau_2 \\ & \mid \mu\alpha.\tau \mid \langle \tau_1, \tau_2 \rangle \mid \tau.i \ (i \in \{1,2\})\end{aligned}$$

Constants

$$\begin{aligned}c^0 \ & ::= \ + \mid - \mid \ldots \\ c^1 \ & ::= \ \texttt{lenP} \mid \texttt{repP} \mid \ldots \\ c^2 \ & ::= \ \texttt{Left} \mid \texttt{Right} \mid \texttt{mapP} \mid \ldots \\ & \ldots\end{aligned}$$

Literals

$$\begin{aligned}i \ & ::= \ 0 \mid 1 \mid \ldots \\ b \ & ::= \ \mathit{True} \mid \mathit{False}\end{aligned}$$

Terms

$e ::=$	i	(*integer literals*)
$\mid$	b	(*boolean literals*)
$\mid$	$\langle\rangle$	(*unit*)
$\mid$	$\{i_1, \ldots, i_n\}_{\texttt{Int}}$	(*integer arrays*)
$\mid$	$\{b_1, \ldots, b_n\}_{\texttt{Bool}}$	(*boolean arrays*)
$\mid$	$c^n_{\langle \tau_1, \ldots, \tau_n \rangle}$	(*constants*)
$\mid$	$\bullet$	(*innermost parameter*)
$\mid$	$e_1 \ e_2$	(*applications*)
$\mid$	$\lambda\bullet : \tau.\ e$	(*abstractions*)
$\mid$	$\langle e_1, e_2 \rangle$	(*tuples*)
$\mid$	$e.i \ (i \in \{1,2\})$	(*tuple projections*)
$\mid$	$\langle\!\langle e_1, \tau_1, \tau_2, \tau_3, e_2 \rangle\!\rangle$	(*closures*)
$\mid$	$e_1 \dagger e_2$	(*closure applications*)
$\mid$	$\langle\!\langle\!: e_1, \tau_1, \tau_2, \tau_3, e_2 :\!\rangle\!\rangle$	(*array closures*)
$\mid$	$e_1 \ddagger e_2$	(*array closure applications*)

FIGURE 5.6 Syntax of $\lambda^{\mathcal{A}}$

5.4.2 Static semantics

The inclusion of type tuples has two important consequences for the type system of $\lambda^{\mathcal{A}}$. First, it requires the introduction of a kind system to distinguish proper types such as `Int` from type tuples. Second, as a type can now be denoted by different type expressions (e.g. `Int` and $\langle$`Int`, `Bool`$\rangle$.1), a notion of type equivalence must be added. Thus, the type system of $\lambda^{\mathcal{P}}$ is based on the following judgements.

$\Delta \vdash_{\mathcal{A}} \tau : \kappa$	the type τ is valid and has the kind κ under the context Δ
$\Delta \vdash_{\mathcal{A}} \tau_1 \equiv \tau_2 : \kappa$	τ_1 and τ_2 are equivalent types of kind κ under the context Δ
$\Gamma \vdash_{\mathcal{A}} e : \tau$	the term e has type τ under the context Γ

The typing rules are given in Figure 5.7. For type equivalence, we only give the primary rules and omit the usual definitions of equivalence and congruence. In the following, we write $\Delta \vdash \tau$ and $\Delta \vdash \tau_1 \equiv \tau_2$ for $\exists\kappa.\, \Delta \vdash \tau : \kappa$ and $\exists\kappa.\, \Delta \vdash \tau_1 \equiv \tau_2 : \kappa$.

Since flattening only generates types of kind $\star$ and $\star \times \star$, using the latter to represent associations between array and element types, we do not permit tuples to be arbitrarily nested. Thus, types in $\lambda^{\mathcal{C}}$ are either proper types or tuples of proper types. This is reflected in the syntax of kinds and in the corresponding typing rules. Moreover, we assume that type variables always have the kind $\star \times \star$. This invariant is maintained by flattening which binds type variables only to type associations.

The type annotations in closures have a similar semantics as in $\lambda^{\mathcal{C}}$ but rely on type associations. In the rule CLO, for instance, τ_1 is a type association such that $\tau_1.1$ is the flat type of the environment and $\tau_1.2$ the corresponding flat array type. Recall that e_1 must evaluate to a tuple $f, f^{\uparrow}$ where $f^{\uparrow}$ is the lifted version of f. The components of the associations τ_1, τ_2 and τ_3 are used to ensure that e_1 indeed has the correct type. The rule ARRCLO is similar, except that in this case, the environment must have an array type.

Constants are handled analogously to $\lambda^{\mathcal{C}}$, but are also parametrised with type associations.

5.4 DEFINITION (TYPES OF CONSTANTS IN $\lambda^{\mathcal{A}}$)
Let $c^n_{\langle\tau_1,\ldots,\tau_n\rangle}$ be a constant in $\lambda^{\mathcal{A}}$. Then, $\mathcal{T}_{\mathcal{A}}(c^n_{\langle\tau_1,\ldots,\tau_n\rangle})$ denotes its type which satisfies the following requirements.

1. It is a function type, i.e. there exist two $\lambda^{\mathcal{A}}$ types υ_1 and υ_2 such that $\mathcal{T}_{\mathcal{A}}(c^n_{\langle\tau_1,\ldots,\tau_n\rangle}) = \upsilon_1 \to \upsilon_2$.

2. It is valid if all type association arguments are valid, i.e., $\vdash \mathcal{T}_{\mathcal{A}}(c^n_{\langle\tau_1,\ldots,\tau_n\rangle}) : \star$ if $\vdash \tau_i : \star \times \star$ for each $1 \leq i \leq n$.

5.5 The flattening transformation

The flattening transformation is the final step in translating a higher-order, nested data parallel program into a closure-based, flat one. It has two main tasks: (a) replace array types of the form $[:\tau:]$ by suitable flat types, thereby eliminating all occurences of $[:\cdot:]$, and (b) transform the computations accordingly by means of vectorisation and lifting. In this section, we formalise this process which has already been outlined in Chapters 3 and 4.

Types

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha : \star \times \star}\ (\textsc{Var-Type}) \qquad \frac{C \in \{\mathtt{Int}, \mathtt{Bool}, \mathtt{Arr}_{\mathtt{Int}}, \mathtt{Arr}_{\mathtt{Bool}}, \langle\rangle\}}{\Delta \vdash C : \star}\ (\textsc{Prim-Type})$$

$$\frac{\Delta \vdash \tau_1 : \star \qquad \Delta \vdash \tau_2 : \star \qquad \otimes \in \{\times, +, \rightarrow, \Rightarrow, \Rrightarrow\}}{\Delta \vdash \tau_1 \otimes \tau_2 : \star}\ (\textsc{Bin-Type})$$

$$\frac{\Delta, \alpha \vdash \tau : \star \times \star}{\Delta \vdash \mu\alpha.\tau : \star \times \star}\ (\textsc{Rec-Type}) \qquad \frac{\Delta \vdash \tau_1 : \star \qquad \Delta \vdash \tau_2 : \star}{\Delta \vdash \langle \tau_1, \tau_2 \rangle : \star \times \star}\ (\textsc{Tup-Type})$$

$$\frac{\Delta \vdash \tau : \star \times \star}{\Delta \vdash \tau.i : \star}\ (\textsc{Proj-Type})$$

Type equivalence (primary rules)

$$\frac{\Delta \vdash \langle \tau_1, \tau_2 \rangle : \star \times \star}{\Delta \vdash \langle \tau_1, \tau_2 \rangle.i \equiv \tau_i : \star} \qquad \frac{\Delta \vdash \tau : \star \times \star}{\Delta \vdash \tau \equiv \langle \tau.1, \tau.2 \rangle : \star \times \star}$$

Terms

$$\Gamma \vdash i \ : \ \mathtt{Int}\ (\textsc{Int}) \qquad \Gamma \vdash b \ : \ \mathtt{Bool}\ (\textsc{Bool}) \qquad \Gamma \vdash \langle\rangle \ : \ \langle\rangle\ (\textsc{Unit})$$

$$\Gamma \vdash \{i_1, \ldots, i_n\}_{\mathtt{Int}} \ : \ \mathtt{Arr}_{\mathtt{Int}}\ (\textsc{ArrInt}) \qquad \Gamma \vdash \{b_1, \ldots, b_n\}_{\mathtt{Bool}} \ : \ \mathtt{Arr}_{\mathtt{Bool}}\ (\textsc{ArrBool})$$

$$\frac{\vdash \tau_1 : \star \times \star \qquad \ldots \qquad \vdash \tau_n : \star \times \star}{\Gamma \vdash c^n_{\langle \tau_1, \ldots, \tau_n \rangle} \ : \ \mathcal{T}_{\mathcal{A}}(c^n_{\langle \tau_1, \ldots, \tau_n \rangle})}\ (\textsc{Const}) \qquad \{\bullet : \tau\} \vdash \bullet \ : \ \tau\ (\textsc{Var})$$

$$\frac{\Gamma \vdash e_1 \ : \ \tau_1 \qquad \Gamma \vdash e_2 \ : \ \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle \ : \ \tau_1 \times \tau_2}\ (\textsc{Tup}) \qquad \frac{\Gamma \vdash e \ : \ \tau_1 \times \tau_2}{\Gamma \vdash e.i \ : \ \tau_i}\ (\textsc{Proj})$$

$$\frac{\Gamma \vdash e_1 \ : \ \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 \ : \ \tau_1}{\Gamma \vdash e_1\, e_2 \ : \ \tau_2}\ (\textsc{App}) \qquad \frac{\vdash \tau_1 : \star \qquad \{\bullet : \tau_1\} \vdash e \ : \ \tau_2}{\lambda\bullet : \tau_1.\, e \ : \ \tau_1 \rightarrow \tau_2}\ (\textsc{Lam})$$

$$\frac{\vdash \tau : \star \qquad \{\bullet : \tau\} \vdash e \ : \ \tau}{\mu\bullet : \tau.\, e \ : \ \tau}\ (\textsc{Rec})$$

$$\frac{\Gamma \vdash e_1 \ : \ (\tau_1.1 \times \tau_2.1 \rightarrow \tau_3.1) \ \times (\mathtt{Int} \times (\tau_1.2 \times \tau_2.2) \rightarrow \tau_3.2) \qquad \Gamma \vdash e_2 \ : \ \tau_1.1}{\langle\!\langle e_1, \tau_1, \tau_2, \tau_3, e_2 \rangle\!\rangle \ : \ \tau_2.1 \Rightarrow \tau_3.1}\ (\textsc{Clo})$$

$$\frac{\Gamma \vdash e_1 \ : \ (\tau_1.1 \times \tau_2.1 \rightarrow \tau_3.1) \ \times (\mathtt{Int} \times (\tau_1.2 \times \tau_2.2) \rightarrow \tau_3.2) \qquad \Gamma \vdash e_2 \ : \ \tau_1.2}{\langle\!\langle\!\langle e_1, \tau_1, \tau_2, \tau_3, e_2 \rangle\!\rangle\!\rangle \ : \ \tau_2.2 \Rrightarrow \tau_3.2}\ (\textsc{ArrClo})$$

$$\frac{\Gamma \vdash e_1 \ : \ \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash e_2 \ : \ \tau_1}{\Gamma \vdash e_1 \dagger e_2 \ : \ \tau_2}\ (\textsc{CloApp})$$

$$\frac{\Gamma \vdash e_1 \ : \ \tau_1 \Rrightarrow \tau_2 \qquad \Gamma \vdash e_2 \ : \ \tau_1}{\Gamma \vdash e_1 \ddagger e_2 \ : \ \tau_2}\ (\textsc{ArrCloApp}) \qquad \frac{\Gamma \vdash e \ : \ \tau_1 \qquad \vdash \tau_1 \equiv \tau_2 : \kappa}{\Gamma \vdash e \ : \ \tau_2}\ (\textsc{Conv})$$

FIGURE 5.7 Static semantics of $\lambda^{\mathcal{A}}$

5.5.1 Flattening types

The flattening of types is defined in terms of a transformation $\mathcal{A}[\![\cdot]\!]$ which, given a $\lambda^{\mathcal{C}}$ type, generates the corresponding $\lambda^{\mathcal{A}}$ type association. Before defining this transformation, let us specify the concept of type associations and the associated notation more precisely.

5.5 DEFINITION (TYPE ASSOCIATIONS)
Let υ be a type in $\lambda^{\mathcal{C}}$ and τ a type in $\lambda^{\mathcal{A}}$ of kind $\star \times \star$ such that $\tau.1$ is the flat representation of υ and $\tau.2$ the flat representation of $[:\upsilon:]$. We call τ a *type association*, $\tau.2$ the *associated array type* of $\tau.1$ and υ the *original type* of τ.

The flattening transformation only generates type associations which are correct in the sense that they have been derived from a $\lambda^{\mathcal{C}}$ type. Subsequent transformations, e.g. optimisations, have to be specified carefully such as to maintain this principle.

For notational convenience, we will now fix the flat representations for each $\lambda^{\mathcal{C}}$ type constructor and for nested arrays. These rules correspond precisely to the flat, unboxed representations introduced in Sections 3.3 and 4.3.

5.6 DEFINITION (FLAT REPRESENTATIONS)
Let C be an n-ary type constructor in $\lambda^{\mathcal{C}}$, $\upsilon_1, \ldots, \upsilon_n$ types in $\lambda^{\mathcal{C}}$ and $\tau_1, \ldots, \tau_n$ the corresponding type associations in $\lambda^{\mathcal{A}}$. Then, $\underline{C}\,\tau_1\ \ldots\ \tau_n$ denotes the type association generated from $C\,\upsilon_1\ \ldots\ \upsilon_n$. The following equations define the appropriate associations.

$$
\begin{array}{lcl}
\underline{\langle\rangle} & = & \langle\langle\rangle, \mathtt{Int}\rangle \\
\underline{\mathtt{Int}} & = & \langle \mathtt{Int}, \mathtt{Arr}_{\mathtt{Int}}\rangle \\
\underline{\mathtt{Bool}} & = & \langle \mathtt{Bool}, \mathtt{Arr}_{\mathtt{Bool}}\rangle \\
\tau_1 \mathbin{\underline{\times}} \tau_2 & = & \langle \tau_1.1 \times \tau_2.1,\ \mathtt{Int} \times \tau_1.2 \times \tau_2.2\rangle \\
\tau_1 \mathbin{\underline{+}} \tau_2 & = & \langle \tau_1.1 + \tau_2.1,\ \mathtt{Arr}_{\mathtt{Bool}} \times \tau_1.2 \times \tau_2.2\rangle \\
\tau_1 \mathbin{\underline{\rightarrow}} \tau_2 & = & \langle (\tau_1.1 \rightarrow \tau_2.1) \times (\tau_1.2 \rightarrow \tau_2.2),\ \mathtt{Int} \times ((\tau_1.1 \rightarrow \tau_2.1) \times (\tau_1.2 \rightarrow \tau_2.2))\rangle \\
\tau_1 \mathbin{\underline{\Rightarrow}} \tau_2 & = & \langle \tau_1.1 \Rightarrow \tau_2.1,\ \tau_1.2 \Rightarrow \tau_2.2\rangle
\end{array}
$$

Note that the effects of vectorisation, i.e. the tupling of functions with their lifted versions, on the types of generated terms have been folded into the definition of $\underline{\rightarrow}$.

5.7 DEFINITION (LIFTED ASSOCIATIONS)
Let τ be a type association in $\lambda^{\mathcal{A}}$. Then, $\tau^{\uparrow}$ is the association obtained by the following rule:

$$\tau^{\uparrow} = \langle \tau.2,\ \mathtt{Arr}_{\mathtt{Int}} \times \tau.2\rangle$$

We will see below that if τ is a type association generated for a $\lambda^{\mathcal{C}}$ type υ, then $\tau^{\uparrow}$ is the association obtained for $[:\upsilon:]$. In the following, we will make extensive use of the notation introduced by the above two definitions. In particular, the next definition employs it to specify the flattening transformation for types.

5.8 DEFINITION (FLATTENING OF TYPES)
Given a $\lambda^{\mathcal{C}}$ type τ, $\mathcal{A}[\![\tau]\!]$ denotes the corresponding type association in $\lambda^{\mathcal{A}}$. The transformation $\mathcal{A}[\![\cdot]\!]$ is defined by the following set of recursive equations.

$$
\begin{array}{llll}
\mathcal{A}[\![\alpha]\!] & = \alpha & \mathcal{A}[\![\tau_1 + \tau_2]\!] & = \mathcal{A}[\![\tau_1]\!] \mathbin{\underline{+}} \mathcal{A}[\![\tau_2]\!] \\
\mathcal{A}[\![\langle\rangle]\!] & = \underline{\langle\rangle} & \mathcal{A}[\![\tau_1 \to \tau_2]\!] & = \mathcal{A}[\![\tau_1]\!] \mathbin{\underline{\to}} \mathcal{A}[\![\tau_2]\!] \\
\mathcal{A}[\![\mathtt{Int}]\!] & = \underline{\mathtt{Int}} & \mathcal{A}[\![\tau_1 \Rightarrow \tau_2]\!] & = \mathcal{A}[\![\tau_1]\!] \mathbin{\underline{\Rightarrow}} \mathcal{A}[\![\tau_2]\!] \\
\mathcal{A}[\![\mathtt{Bool}]\!] & = \underline{\mathtt{Bool}} & \mathcal{A}[\![[:\tau:]]\!] & = \mathcal{A}[\![\tau]\!]^{\uparrow} \\
\mathcal{A}[\![\tau_1 \times \tau_2]\!] & = \mathcal{A}[\![\tau_1]\!] \mathbin{\underline{\times}} \mathcal{A}[\![\tau_2]\!] & \mathcal{A}[\![\mu\alpha.\tau]\!] & = \mu\alpha.\mathcal{A}[\![\tau]\!]
\end{array}
$$

Furthermore, $\mathcal{F}[\![\tau]\!]$ denotes the flat representation of τ in $\lambda^{\mathcal{A}}$, i.e. $\mathcal{F}[\![\tau]\!] = \mathcal{A}[\![\tau]\!].1$.

The above definition reflects an important property already discussed in Chapter 3: the transformation rule for selecting the flat representation of a type only depends on the outermost type constructor. This is, in fact, crucial for the validity of the type-indexed approach to implementing primitive array operations.

5.9 Example
The $\lambda^{\mathcal{C}}$ type $\mathtt{Int} + [:\mathtt{Bool}:]$ is transformed as follows:

$$
\begin{aligned}
& \mathcal{A}[\![\mathtt{Int} + [:\mathtt{Bool}:]]\!] \\
= {} & \underline{\mathtt{Int}} \mathbin{\underline{+}} \underline{\mathtt{Bool}}^{\uparrow} \\
= {} & \langle \underline{\mathtt{Int}}.1 + (\underline{\mathtt{Bool}}^{\uparrow}).1,\ \mathtt{Arr}_{\mathtt{Bool}} \times (\underline{\mathtt{Int}}.2 \times (\underline{\mathtt{Bool}}^{\uparrow}).2) \rangle \\
= {} & \ldots \\
\equiv {} & \langle \mathtt{Int} + \mathtt{Arr}_{\mathtt{Bool}}, \mathtt{Arr}_{\mathtt{Bool}} \times (\mathtt{Arr}_{\mathtt{Int}} \times (\mathtt{Arr}_{\mathtt{Int}} \times \mathtt{Arr}_{\mathtt{Bool}})) \rangle
\end{aligned}
$$

An important aspect of type flattening is the translation of recursive types. Given a $\lambda^{\mathcal{C}}$ type $\mu\alpha.v$, the flattening generates the type association $\mu\alpha.\mathcal{A}[\![v]\!]$ which contains the flat representations of $\mu\alpha.v$ and $[:\mu\alpha.v:]$. As the latter can be mutually recursive, the recursion is captured by the type association itself. This ensures that at every occurence of the recursion variable α in v, the appropriate representation can be selected depending on whether α is used within an array context. Since type variables are only used for recursion, this strategy trivially ensures that the former are always bound to types of kind $\star \times \star$, as required by the static semantics of $\lambda^{\mathcal{A}}$.

5.10 Example
The $\lambda^{\mathcal{C}}$ type $\mu\alpha.\alpha \to \mathtt{Int}$ is flattened as follows:

$$
\begin{aligned}
& \mathcal{A}[\![\mu\alpha.\alpha \to \mathtt{Int}]\!] \\
= {} & \mu\alpha.\mathcal{A}[\![\alpha \to \mathtt{Int}]\!] \\
\equiv {} & \mu\alpha.\langle (\alpha.1 \to \mathtt{Int}) \times (\alpha.2 \to \mathtt{Arr}_{\mathtt{Int}}),\ \mathtt{Int} \times (\alpha.1 \to \mathtt{Int}) \times (\alpha.2 \to \mathtt{Arr}_{\mathtt{Int}}) \rangle
\end{aligned}
$$

Note that the two components of the generated association are indeed mutually recursive.

5.11 Example
Consider again the type of integer lists $\mu l.\langle\rangle + \mathtt{Int} \times l$ introduced in Section 3.3.4. Flattening transforms it to

$$
\mu\alpha.\langle\langle\rangle + (\mathtt{Int} \times \alpha.1),\ \mathtt{Arr}_{\mathtt{Bool}} \times \underbrace{\mathtt{Int}}_{[:\langle\rangle:]} \times (\underbrace{\mathtt{Int} \times \mathtt{Arr}_{\mathtt{Int}} \times \alpha.2}_{[:\mathtt{Int} \times \alpha:]}) \rangle
$$

It is easy to see that the first component of the recursive type is equivalent to the original list type, whereas the second corresponds precisely to the representation of arrays of lists discussed previously.

The last example demonstrates that the two components of a recursive type association are not necessarily mutually recursive. In fact, mutual recursion can only be introduced by function types, as in Example 5.10, since the corresponding rule is the only one to "leave" an array context.

It is important to point out that the use of type tuples is, strictly speaking, not necessary for translating recursive types. The flat representation derived in Example 5.10 can be rewritten as follows:

$$\begin{aligned} t &= \mu\alpha.(\alpha \to \mathtt{Int}) \times (t^{\uparrow} \to \mathtt{Arr}_{\mathtt{Int}}) \\ t^{\uparrow} &= \mu\beta.\mathtt{Int} \times (t \to \mathtt{Int}) \times (\beta \to \mathtt{Arr}_{\mathtt{Int}}) \end{aligned}$$

By substituting the definitions of t and $t^{\uparrow}$ and replacing references to these types by appropriate type variables, we obtain

$$\begin{aligned} t &= \mu\alpha.(\alpha \to \mathtt{Int}) \times ((\mu\beta.\mathtt{Int} \times (\alpha \to \mathtt{Int}) \times (\beta \to \mathtt{Arr}_{\mathtt{Int}})) \to \mathtt{Arr}_{\mathtt{Int}}) \\ t^{\uparrow} &= \mu\beta.\mathtt{Int} \times ((\mu\alpha.(\alpha \to \mathtt{Int}) \times (\beta \to \mathtt{Arr}_{\mathtt{Int}})) \to \mathtt{Int}) \times (\beta \to \mathtt{Arr}_{\mathtt{Int}}) \end{aligned}$$

It is not hard to see that the types $t^{\uparrow}$ and $\mathtt{Int} \times t$ and, in particular, corresponding function types in t and $t^{\uparrow}$ are *semantically equivalent*. However, the introduction of semantical equivalence of recursive types into the type system has non-trivial consequences, significantly complicating type checking and type inference (Cardone and Coppo, 1991; Amadio and Cardelli, 1993). Our approach, while, in a sense, less pure, allows us to only consider *syntactical equivalence* modulo projections from type tuples, which is easily decided.

As before, we assume that for each recursive type $\mu\alpha.\tau$, there exists a constructor $\mathtt{in}_{\mu\alpha.\tau}$ and a destructor $\mathtt{out}_{\mu\alpha.\tau}$. For $\lambda^{\mathcal{A}}$, we also assume the existence of lifted versions of these constants, $\mathtt{in}^{\uparrow}_{\mu\alpha.\tau}$ and $\mathtt{out}^{\uparrow}_{\mu\alpha.\tau}$ which construct and destruct flat arrays with elements of type $\mu\alpha.\tau$.

5.5.2 Vectorisation and lifting

Having defined the flattening transformation for types in the previous section, we will now turn our attention to transforming computations. Recall that this process

- generates a lifted version for every function in a program,
- tuples every function with its lifted version and
- vectorises the code such that the correct version of a function is selected for each application in the program.

The flattening of terms is defined as two mutually recursive transformations, vectorisation ($\mathcal{V}[\![\cdot]\!]$) and lifting ($\mathcal{L}[\![\cdot,\cdot]\!]$), which we have already described in Section 3.5. Vectorisation is defined for all terms in $\lambda^{\mathcal{C}}$; lifting, however, only applies to code within a lambda abstraction. Given a $\lambda^{\mathcal{C}}$ term e of type τ and a $\lambda^{\mathcal{A}}$ term l, $\mathcal{L}[\![l, e]\!]$ generates a term of type $[:\tau:]$ (suitably flattened), i.e. the resulting computation is performed in array space. The length of the arrays is determined by the *lifting context* l, which is always obtained from the parameter of the lambda abstraction.

5.12 DEFINITION (VECTORISED AND LIFTED CONSTANTS)
Let c be an n-ary family of constants in $\lambda^{\mathcal{C}}$. Then, $\mathcal{V}_C[\![c]\!]$ and $\mathcal{L}_C[\![c]\!]$ are n-ary families of constants in $\lambda^{\mathcal{A}}$.

Vectorisation

$$
\begin{array}{lcl}
\mathcal{V}[\![i]\!] & = & i \\
\mathcal{V}[\![b]\!] & = & b \\
\mathcal{V}[\![\langle\rangle]\!] & = & \langle\rangle \\
\mathcal{V}[\![c_{\langle\tau_1,\ldots,\tau_n\rangle}]\!] & = & \langle \mathcal{V}_C[\![c]\!]_{\langle\mathcal{A}[\![\tau_1]\!],\ldots,\mathcal{A}[\![\tau_n]\!]\rangle},\ \mathcal{L}_C[\![c]\!]_{\langle\mathcal{A}[\![\tau_1]\!],\ldots,\mathcal{A}[\![\tau_n]\!]\rangle}\rangle \\
\mathcal{V}[\![\bullet]\!] & = & \bullet \\
\mathcal{V}[\![e_1\ e_2]\!] & = & \mathcal{V}[\![e_1]\!].1\ \mathcal{V}[\![e_2]\!] \\
\mathcal{V}[\![\lambda\bullet:\tau.\ e]\!] & = & \langle \lambda\bullet:\mathcal{F}[\![\tau]\!].\ \mathcal{V}[\![e]\!],\ \lambda\bullet:\mathcal{F}[\![[:\tau:]]\!].\ \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\tau]\!]\rangle}\ \bullet, e]\!]\rangle \\
\mathcal{V}[\![\mu\bullet:\tau.\ e]\!] & = & \mu\bullet:\mathcal{F}[\![\tau]\!].\ \mathcal{V}[\![e]\!] \\
\mathcal{V}[\![\langle e_1, e_2\rangle]\!] & = & \langle \mathcal{V}[\![e_1]\!], \mathcal{V}[\![e_2]\!]\rangle \\
\mathcal{V}[\![e.i]\!] & = & \mathcal{V}[\![e]\!].i \\
\mathcal{V}[\![\langle\!\langle e_1, \tau_1, \tau_2, \tau_3, e_2\rangle\!\rangle]\!] & = & \langle\!\langle \mathcal{V}[\![e_1]\!], \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \mathcal{V}[\![e_2]\!]\rangle\!\rangle \\
\mathcal{V}[\![e_1 \dagger e_2]\!] & = & \mathcal{V}[\![e_1]\!] \dagger \mathcal{V}[\![e_2]\!]
\end{array}
$$

Lifting

$$
\begin{array}{lcl}
\mathcal{L}[\![l, i]\!] & = & \mathtt{repP}_{\langle\underline{\mathtt{Int}}\rangle}\ \langle l, i\rangle \\
\mathcal{L}[\![l, b]\!] & = & \mathtt{repP}_{\langle\underline{\mathtt{Bool}}\rangle}\ \langle l, b\rangle \\
\mathcal{L}[\![l, \langle\rangle]\!] & = & \mathtt{repP}_{\langle\underline{\langle\rangle}\rangle}\ \langle l, \langle\rangle\rangle \\
\mathcal{L}[\![l, (c_{\langle\tau_1,\ldots,\tau_n\rangle})^{\upsilon_1\to\upsilon_2}]\!] & = & \mathtt{repP}_{\langle\mathcal{A}[\![\upsilon_1]\!]\underline{\to}\mathcal{A}[\![\upsilon_2]\!]\rangle}\ \langle l, \mathcal{V}[\![c_{\langle\tau_1,\ldots,\tau_n\rangle}]\!]\rangle \\
\mathcal{L}[\![l, \bullet]\!] & = & \bullet \\
\mathcal{L}[\![l, (e_1\ e_2)^\tau]\!] & = & \mathtt{attachP}_{\langle\mathcal{A}[\![\tau]\!]\rangle}\ \langle l,\ \mathcal{L}[\![l, e_1]\!].2.2\ \mathcal{L}[\![l, e_2]\!]\rangle \\
\mathcal{L}[\![l, (\lambda\bullet:\tau_1.\ e)^{\tau_1\to\tau_2}]\!] & = & \mathtt{repP}_{\langle\mathcal{A}[\![\tau_1]\!]\underline{\to}\mathcal{A}[\![\tau_2]\!]\rangle}\ \langle l,\ \mathcal{V}[\![\lambda\bullet:\tau_1.\ e]\!]\rangle \\
\mathcal{L}[\![l, \mu\bullet:\tau.\ e]\!] & = & \mathtt{repP}_{\langle\mathcal{A}[\![\tau]\!]\rangle}\ \langle l, \mathcal{V}[\![\mu\bullet:\tau.\ e]\!]\rangle \\
\mathcal{L}[\![l, \langle e_1, e_2\rangle^{\tau_1\times\tau_2}]\!] & = & \mathtt{zipP}_{\langle\mathcal{A}[\![\tau_1]\!],\mathcal{A}[\![\tau_2]\!]\rangle}\ \langle\mathcal{L}[\![l, e_1]\!], \mathcal{L}[\![l, e_2]\!]\rangle \\
\mathcal{L}[\![l, (e.i)^\tau]\!] & = & \mathtt{attachP}_{\langle\mathcal{A}[\![\tau]\!]\rangle}\ \langle l,\ \mathcal{L}[\![l, e]\!].2.i\rangle \\
\mathcal{L}[\![l, \langle\!\langle e_1, \tau_1, \tau_2, \tau_3, e_2\rangle\!\rangle]\!] & = & \langle\!\langle\mathcal{L}[\![l, e_1]\!].2, \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \mathcal{L}[\![l, e_2]\!]\rangle\!\rangle \\
\mathcal{L}[\![l, (e_1 \dagger e_2)^\tau]\!] & = & \mathtt{attachP}_{\langle\mathcal{A}[\![\tau]\!]\rangle}\ \langle l,\ \mathcal{L}[\![l, e_1]\!] \ddagger \mathcal{L}[\![l, e_2]\!]\rangle
\end{array}
$$

FIGURE 5.8 Vectorisation and lifting

The intention is that $\mathcal{V}_C[\![c]\!]$ is the vectorised version of c and $\mathcal{L}_C[\![c]\!]$ is its lifted version. We will formalise these properties later, when discussing the static and dynamic semantics of flattening.

5.13 DEFINITION (VECTORISATION AND LIFTING)
Let e be a term in $\lambda^{\mathcal{C}}$ and l a term in $\lambda^{\mathcal{A}}$. Then, $\mathcal{V}[\![e]\!]$ and $\mathcal{L}[\![l, e]\!]$ are terms in $\lambda^{\mathcal{A}}$ such that the former is the vectorised version of e and the latter its lifted version under the lifting context l. The two transformations are defined by the set of mutually recursive equations given in Figure 5.8.

The basic algorithm is quite straightforward. Vectorisation traverses a term recursively, tupling functions with their lifted versions, selecting unlifted functions in applications and transforming type annotations. Lifting pursues a similar approach but, additionally, ensures that the generated terms operate on arrays. This essentially amounts to replicating the static

components of the term. Thus, terms which are guaranteed to be closed (e.g., μ-recursions) are simply vectorised and replicated. For terms potentially containing free occurences of $\bullet$, such as tuples, the replication algorithm is folded into the transformation since $\bullet$ is already an array and, thus, should not be replicated. Below, we describe the most interesting transformation rules in more detail.

Functions

A function in $\lambda^{\mathcal{C}}$ is either a constant or a lambda abstraction. The vectorisation of constants relies on the transformations introduced in Definition 5.12. For lambda abstractions, the vectorised and lifted versions are obtained by vectorising and lifting, respectively, the inner term. In the latter case, `lenP` $\bullet$ is used as the lifting context, thus ensuring that the generated arrays always have the same length as the argument the abstraction is applied to. Since functions are always closed, lifting simply replicates the tuple obtained by vectorisation, which is equivalent to attaching the required length as described in Section 4.6.

Applications

The transformation of applications $e_1\ e_2$ accounts for the tupling performed by vectorisation by selecting the appropriate function from the transformed version of e_1. In the case of vectorisation, this is the first component. Lifting, however, transforms e_1 to an array of functions which then must be applied elementwise to the array of arguments obtained by lifting e_2. Since the former will have the form $\langle n, \langle f, f^{\uparrow} \rangle \rangle$, this amounts to applying $f^{\uparrow}$, or, equivalently, $\mathcal{L}[\![l, e_1]\!].2.2$ to the argument array.

Care has to be taken, however, such as not to introduce nontermination not present in the original program. Consider that the semantics of the lifted application should be equivalent to mapping the vectorised version of e_1 over the array generated by lifting e_2. Clearly, if $\mathcal{L}[\![l, e_1]\!]$ diverges, then so does $\mathcal{L}[\![l, e_1]\!].2.2\ \mathcal{L}[\![l, e_2]\!]$. But in Chapter 6, we will see that for flattening to be correct, mapping over parallel arrays may not be strict in the function in the sense that even if the function diverges, the length of the resulting array must still be available.

This is ensured by explicitly attaching the length to the result of the lifted application. The primitive `attachP` does not change the elements of an array but sets its length component, such that `lenP` (`attachP` $\langle n, xs \rangle$) $= n$. Note that this strategy crucially relies on the implicit assumption that all arrays generated by lifting either diverge or have the length specified by the lifting context.

Closures

The vectorised version of a closure is obtained by vectorising the closure function and environment and flattening the type annotations. Lifting generates closure arrays by lifting the environment and selecting the function tuple from the function array obtained by lifting the closure function. The application of closure arrays is performed by $\ddagger$. The length must be attached explicitly for the same reasons as before.

Literals and tuples

Literals are lifted by replicating them to the length determined by the lifting context. Tuples are lifted by lifting and then zipping the components. For projections, lifting relies on the

principle that arrays of tuples are represented by tuples of arrays and simply selectes the appropriate component of the flat representation. Again, the length has to be attached explicitly to the result to ensure convergence.

5.6 Properties of typing

In the rest of this chapter, we show that the flattening transformation as defined in the previous section is type correct. In this section, we establish a number of important properties of the $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$ type systems. These are necessary both for showing the type correctness of flattening and for the proof of semantical correctness given in the next chapter.

Validity

A critical property of every type system is *validity*. For $\lambda^{\mathcal{C}}$, it is sufficient to show that only valid type are assigned to terms.

5.14 Lemma (Validity of typing in $\lambda^{\mathcal{C}}$)
If $\Gamma \vdash_{\mathcal{C}} e : \tau$, *then* $\vdash_{\mathcal{C}} \tau$.

Proof. By induction on the given derivation. □

The type system of $\lambda^{\mathcal{A}}$ is more complex. Here, we must also show that equivalence only applies to valid types and that the types have appropriate kinds.

5.15 Lemma (Validity of typing in $\lambda^{\mathcal{A}}$)
1. *If* $\Gamma \vdash_{\mathcal{A}} e : \tau$, *then* $\vdash_{\mathcal{A}} \tau : \star$.
2. *If* $\Delta \vdash_{\mathcal{A}} \tau_1 \equiv \tau_2 : \kappa$, *then* $\Delta \vdash_{\mathcal{A}} \tau_1 : \kappa$ *and* $\Delta \vdash \tau_2 : \kappa$.

Proof. By straightforward induction on the given derivations. □

Conversion of contexts

Next, we will show that weakening and strengthening of type variable and type contexts are admissible in $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$. Given a valid derivation $\Delta \vdash \tau$, the context Δ can be weakened by adding new type variables to it and strengthened by removing type variables not used in τ.

5.16 Lemma (Conversion of type variable contexts in $\lambda^{\mathcal{C}}$)
1. *If* $\Delta \vdash_{\mathcal{C}} \tau$, *then* $\Delta, \alpha \vdash_{\mathcal{C}} \tau$.
2. *If* $\Delta, \alpha \vdash_{\mathcal{C}} \tau$ *and* α *does not occur free in* τ, *then* $\Delta \vdash_{\mathcal{C}} \tau$.

Proof. By straightforward induction on the given derivations. □

5.17 Lemma (Conversion of type variable contexts in $\lambda^{\mathcal{A}}$)
1. *If* $\Delta \vdash_{\mathcal{A}} \tau : \kappa$, *then* $\Delta, \alpha \vdash_{\mathcal{A}} \tau : \kappa$.
2. *If* $\Delta \vdash \tau_1 \equiv \tau_2 : \kappa$, *then* $\Delta, \alpha \vdash \tau_1 \equiv \tau_2 : \kappa$.
3. *If* $\Delta, \alpha \vdash_{\mathcal{A}} \tau : \kappa$ *and* α *does not occur free in* τ, *then* $\Delta \vdash_{\mathcal{A}} \tau : \kappa$.
4. *If* $\Delta, \alpha \vdash \tau_1 \equiv \tau_2 : \kappa$ *and* α *does not occur free in* τ_1 *and* τ_2, *then* $\Delta \vdash \tau_1 \equiv \tau_2 : \kappa$.

PROOF. By straightforward induction on the given derivations. □

A similar principle can be derived for type contexts. Here, it only applies to terms in which $\bullet$ does not occur outside of lambda abstraction, i.e. which are closed with respect to $\bullet$. The following definition formalises closedness for both $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$.

5.18 DEFINITION (CLOSED TERMS)
A term e in $\lambda^{\mathcal{C}}$ or $\lambda^{\mathcal{A}}$ is *closed* if $\bullet$ does not occur outside of lambda abstractions in e.

5.19 LEMMA (CONVERSION OF TYPE CONTEXTS)
1. *Let e be a closed term in $\lambda^{\mathcal{C}}$ such that $\Gamma \vdash_{\mathcal{C}} e : \tau$ for some type context Γ. Then, for every valid type context Γ', $\Gamma' \vdash_{\mathcal{C}} e : \tau$.*
2. *Let e be a closed term in $\lambda^{\mathcal{A}}$ such that $\Gamma \vdash_{\mathcal{A}} e : \tau$ for some type context Γ. Then, for every valid type context Γ', $\Gamma' \vdash_{\mathcal{A}} e : \tau$.*

PROOF. By straightforward induction on the given derivation. □

5.7 Type correctness of flattening

In this section, we show the correctness of flattening with respect to the static semantics of $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$. Intuitively, we want to show, for every term e of type τ in $\lambda^{\mathcal{C}}$, the following properties of vectorisation and lifting:

- $\mathcal{V}[\![e]\!]$ has the type $\mathcal{A}[\![\tau]\!].1$ and
- $\mathcal{L}[\![l, e]\!]$ has the type $\mathcal{A}[\![\tau]\!].1$.

As the proof relies on a number of key properties of flattened types, we will establish these first.

5.7.1 Properties of flattened types

We begin by showing that the type flattening transformation $\mathcal{F}[\![\cdot]\!]$ preserves the validity of types.

5.20 LEMMA (VALIDITY OF FLATTENED TYPES)
1. *Let τ be a type in $\lambda^{\mathcal{C}}$ such that $\Delta \vdash_{\mathcal{C}} \tau$ for some Δ. Then, $\Delta \vdash_{\mathcal{A}} \mathcal{A}[\![\tau]\!] : \star \times \star$.*
2. *Let τ be a type in $\lambda^{\mathcal{C}}$ such that $\Delta \vdash_{\mathcal{C}} \tau$ for some Δ. Then, $\Delta \vdash_{\mathcal{A}} \mathcal{F}[\![\tau]\!] : \star$.*

PROOF. Part 1 is by straightforward induction on τ and inversion on the derivation $\Delta \vdash_{\mathcal{C}} \tau$. Part 2 is a direct consequence of Part 1. □

Another crucial property is substituitivity: substituting a type υ into a type τ and flattening the result is equivalent to flattening τ first and then substituting the type association generated for τ.

5.21 LEMMA (SUBSTITUTIVITY OF TYPE FLATTENING)
Let τ and υ be types $\lambda^{\mathcal{C}}$. Then, $\mathcal{A}[\![\tau[\upsilon/\alpha]]\!] = \mathcal{A}[\![\tau]\!][\mathcal{A}[\![\upsilon]\!]/\alpha]$.

PROOF. By straightforward induction on τ. □

5.7.2 Type correctness of vectorisation and lifting

We will see below that the type correctness of vectorisation and lifting is established simultaneously. In particular, this implies that given a $\lambda^{\mathcal{C}}$ term $\lambda\bullet : \tau.\ e$ of type $\tau \to \upsilon$, we must show $\{\bullet : \mathcal{F}[\![\tau]\!]\} \vdash \mathcal{V}[\![e]\!] : \mathcal{F}[\![\upsilon]\!]$ and $\{\bullet : \mathcal{F}[\![[:\tau:]]\!]\} \vdash \mathcal{L}[\![l, e]\!] : \mathcal{F}[\![[:\upsilon:]]\!]$. The change in the type context is captured by the following definition.

5.22 Definition (Lifted type contexts)
Let Γ be a type context in $\lambda^{\mathcal{C}}$. Then, $\Gamma^{\uparrow}$ is the type context derived by the following rules.

$$\begin{array}{lcl} \emptyset^{\uparrow} & = & \emptyset \\ \{\bullet : \tau\}^{\uparrow} & = & \{\bullet : [:\tau:]\} \end{array}$$

5.23 Lemma
If $\vdash_{\mathcal{C}} \Gamma$, *then* $\vdash_{\mathcal{C}} \Gamma^{\uparrow}$.

Proof. Immediate from the definition of $\Gamma^{\uparrow}$. □

The following definition introduces a translation of type contexts from $\lambda^{\mathcal{C}}$ to $\lambda^{\mathcal{A}}$. Note that a similar translation for type variable contexts is not required as they are simple sets of type variables in both languages.

5.24 Definition (Flattening of type contexts)
Let Γ be a type context in $\lambda^{\mathcal{C}}$. Then, $\mathcal{G}[\![\Gamma]\!]$ is a type context in $\lambda^{\mathcal{A}}$ derived by the following rules.

$$\begin{array}{lcl} \mathcal{G}[\![\emptyset]\!] & = & \emptyset \\ \mathcal{G}[\![\{\bullet : \tau\}]\!] & = & \{\bullet : \mathcal{F}[\![\tau]\!]\} \end{array}$$

5.25 Lemma (Validity of flattened type contexts)
If $\vdash_{\mathcal{C}} \Gamma$, *then* $\vdash_{\mathcal{A}} \mathcal{G}[\![\Gamma]\!]$.

Proof. Immediate by validity of flattened types (Lemma 5.20). □

For flattening to be type correct, the constants in $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$ related by the functions $\mathcal{V}_C[\![\cdot]\!]$ and $\mathcal{L}_C[\![\cdot]\!]$ (cf. Definition 5.12) must have compatible types in the following sense.

5.26 Property (Types of vectorised and lifted constants)
Let c be an n-ary family of constants and $\tau_1, \ldots, \tau_n$ types in $\lambda^{\mathcal{C}}$ such that $\mathcal{T}_{\mathcal{C}}(c_{\langle\tau_1,\ldots,\tau_n\rangle}) = \upsilon_1 \to \upsilon_2$ for some types υ_1 and υ_2. Then the types of the corresponding constants in $\lambda^{\mathcal{A}}$ satisfy the following conditions:

$$\begin{array}{lcl} \mathcal{T}_{\mathcal{A}}(\mathcal{V}_C[\![c]\!]_{\langle\mathcal{A}[\![\tau_1]\!],\ldots,\mathcal{A}[\![\tau_n]\!]\rangle}) & = & \mathcal{F}[\![\upsilon_1]\!] \to \mathcal{F}[\![\upsilon_2]\!] \\ \mathcal{T}_{\mathcal{A}}(\mathcal{L}_C[\![c]\!]_{\langle\mathcal{A}[\![\tau_1]\!],\ldots,\mathcal{A}[\![\tau_n]\!]\rangle}) & = & \mathcal{F}[\![[:\upsilon_1:]]\!] \to \mathcal{F}[\![[:\upsilon_2:]]\!] \end{array}$$

It is easy to see that the primitives used in this work meet these constraints. They must be verified should new constants be added to the language. In this sense, the above requirement can be interpreted as an axiom.

We are now in the position to establish the type correctness of the flattening transformation as stated at the beginning of this section. The following theorem shows that the terms generated by vectorisation and lifting are (a) valid and (b) have the expected types.

5.27 THEOREM (TYPE CORRECTNESS OF FLATTENING)
Let e be a term and τ a type in $\lambda^{\mathcal{C}}$ such that $\Gamma \vdash_{\mathcal{C}} e : \tau$ for some Γ.

1. *Then, $\mathcal{G}[\![\Gamma]\!] \vdash_{\mathcal{A}} \mathcal{V}[\![e]\!] : \mathcal{F}[\![\tau]\!]$.*
2. *Moreover, if l is a term in $\lambda^{\mathcal{A}}$ such that $\mathcal{G}[\![\Gamma^{\uparrow}]\!] \vdash_{\mathcal{A}} l : \texttt{Int}$, then $\mathcal{G}[\![\Gamma^{\uparrow}]\!] \vdash_{\mathcal{A}} \mathcal{L}[\![l, e]\!] : \mathcal{F}[\![[:\tau:]]\!]$.*

PROOF. By simultaneous induction on the derivation $\Gamma \vdash_{\mathcal{C}} e : \tau$. The complete proof is given in Appendix A.1. □

Chapter 6

Semantics of flattening

The development in the previous chapter provides a solid foundation for establishing the operational correctness of the flattening transformation. Ultimately, our goal is to show that flattening preserves the meaning of programs, which, of course, implies that we must define the operational semantics of the source and target languages of the transformation. This task is significantly complicated by the impact of unboxed arrays used in the flat representation of data structures on the termination behaviour of computations, an issue which we have largely side-stepped until now. Although unboxed arrays only appear in $\lambda^{\mathcal{A}}$, the strictness induced by their use must be reflected in the semantics of $\lambda^{\mathcal{C}}$ if we are to obtain any meaningful correctness results. Section 6.1 discusses and formalises this aspect of the flattening transformation.

We have already seen that flattened programs crucially rely on a number of primitives operations. Section 6.2 introduces the algebraic laws which must be satisfied by the primitives. These laws are used in the subsequent development instead of the implementations given in Appendices B and C, thereby separating the correctness results from a particular implementation of the standard library.

The operational semantics of the languages $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$ is defined in Section 6.3 as a small-step reduction system. As usual, the operational semantics induces a notion of semantic equivalence which is introduced in Section 6.4. In Chapter 4, we have informally justified the elimination of named variables from the two languages. Section 6.5 formalises the properties of substitution and β-reduction in these mononomic calculi, which are of crucial importance for subsequent correctness proofs.

After introducing some auxiliary results about flattened terms in Section 6.6, we finally tackle the problem of establishing the correctness of vectorisation and lifting in the last two sections. In Section 6.7, we show that lifted functions satisfy the usual *map* laws, which is a sufficient correctness criterion for this transformation. Finding such a criterion for vectorisation and, thus, flattening is more problematic, as the usual techniques are not directly applicable here. In particular, flattening does not general semantic equivalence for arbitrary terms. This is because semantic equivalence implies that two terms can be used interchangeably without altering the meaning of a program, but vectorised terms rely on the invariant that vectorised functions are always tupled with their lifted versions and can behave differently in contexts which violate this invariant. We circumvent this problem by considering only flattened *programs*, i.e., closed terms. In Section 6.8, we prove that such programs yield the same results as their nested $\lambda^{\mathcal{C}}$ versions. We also show that flattening does not introduce nontermination. While the converse conclusion — that flattening *preserves* nontermination

— would also be necessary for establishing the total correctness of the transformation, it is unclear how its proof could be constructed. We do not consider this to be a pressing problem, however, as long as flattening preserves the meaning of terminating programs.

Until now, we have not made a distinction between syntactic equality and semantic equivalence. This distinction is highly important when discussing and reasoning about operational behaviour. Consequently, in the rest of this chapter we use $\cong$ to denote semantic equivalence (which, however, is not defined formally until Section 6.4) and $=$ for syntactic equality.

6.1 Strictness of flattening

Before establishing the correctness of the flattening transformation, we must investigate the impact of using unboxed arrays on the strictness properties of a program. In Section 3.3.1, we have cited the efficiency of such arrays as one of the main benefits of an unboxed representation. This efficiency is due to the fact that unboxed arrays are strict in all elements, thereby avoiding indirections necessary for representing possibly suspended computations. Thus, an unboxed array diverges if any of its elements does, i.e., $\{\ldots, \bot, \ldots\} \cong \bot$.

This has far-reaching consequences for the semantics of parallel arrays. Consider, for instance, the translation of the following three $\lambda^{\mathcal{C}}$ terms (recall that the $\lambda^{\mathcal{C}}$ type [:`Int`:] is transformed to $\texttt{Int} \times \texttt{Arr}_{\texttt{Int}}$):

$$\begin{array}{lcl} [:1, \bot:] & \Longrightarrow & \langle 2, \{1, \bot\}\rangle \;\cong\; \langle 2, \bot\rangle \\ [:\bot, 2:] & \Longrightarrow & \langle 2, \{\bot, 2\}\rangle \;\cong\; \langle 2, \bot\rangle \\ [:\bot, \bot:] & \Longrightarrow & \langle 2, \{\bot, \bot\}\rangle \;\cong\; \langle 2, \bot\rangle \end{array}$$

Note that due to the strictness of unboxed arrays, the flat representations of these terms are equivalent. This highlights two aspects of the semantics of parallel arrays under the flattening transformation.

- If one element of a parallel array diverges, then all of its elements do.
- The length of a parallel array is available even if its elements diverge.

We will discuss the importance of the latter property for the correctness of flattening before turning our attention to the first observation.

6.1.1 Shape and data

In Section 2.1, we have already mentioned that, contrary to our earlier work (Chakravarty et al., 2001), we require parallel arrays not to be head-strict. This requirement is due to both technical and theoretical reasons. The operational correctness of flattening relies on a number of algebraic laws satisfied by primitive array operations, of which the following is perhaps the most important one:

$$\texttt{lenP}\,(\texttt{repP}\,\langle n, x\rangle) \;\cong\; n$$

It is easy to see that the above would not hold for $x = \bot$ if arrays were head-strict. This problem is avoided in our translation by completely separating the length of a parallel array from its elements in the flat representation, such that the former can still be accessed even if the latter diverge. For instance, in the case of integer arrays we have:

$$\texttt{lenP}_{\langle\texttt{Int}\rangle}\,(\texttt{repP}_{\langle\texttt{Int}\rangle}\,\langle n, \bot\rangle) \;\cong\; \texttt{lenP}_{\langle\texttt{Int}\rangle}\,\langle n, \bot\rangle \;\cong\; n$$

Note, however, that parallel arrays are strict in the length. Unfortunately, this constraint cannot be directly encoded in the representation types since in the intermediate languages, like in Haskell, all types are lifted, i.e., contain a bottom element. Instead, it must be enforced by suitably implementing array primitives, such that, for instance, $\texttt{repP}\,\langle\bot, x\rangle \;\cong\; \bot$. The definition of $\mathcal{L}[\![\cdot,\cdot]\!]$ uses `seq` to explicitly ensure this invariant in the case of tuples.

This separation can also be justified from a more abstract point of view. Jay (1995b) introduces the concept of *shapes* which capture the structure of collections but abstract from the data. For inductive data types, shapes largely correspond to the rather informal notion of spine; in the case of arrays, the shape is just their length. Thus, we can say that parallel arrays are strict in the shape but lazy in the data. In this, they are not different from proper algebraic collection types such as lists, at least if we only consider complete traversals. This is certainly beneficial as it leads to a less surprising semantics of parallel arrays and makes the language more uniform. In the context of flattening, this property is also essential with respect to the correctness of the transformation.

This principle also implies that `mapP` and other mapping-related primitives may not be strict in the mapped function or closure, such that, e.g., $\texttt{lenP}\,(\texttt{mapP}\,\langle f, xs\rangle) \cong \texttt{lenP}\,xs$. This, again, corresponds to the semantics of list operations.

6.1.2 Induced undefinedness

Despite the obvious similarities, parallel arrays are not merely an efficient representation of finite lists since, as described earlier, they are strict in all elements. This property is easily explained and formalised for arrays of primitive types, such as integers. The situation becomes more complex, however, if we take products and sums into account. The following example illustrates this:

$$\begin{array}{c}
[:\langle \mathit{True}, 1\rangle, \langle \mathit{False}, \bot\rangle:] \\
\Downarrow \\
\langle 2,\ \langle [:\mathit{True}, \mathit{False}:],\ [:1, \bot:]\rangle\rangle \\
\Downarrow \\
\langle 2,\ \langle 2,\ \{\mathit{True}, \mathit{False}\}\rangle,\ \langle 2, \{1, \bot\}\rangle\rangle \\
\Downarrow \\
\langle 2,\ \langle 2,\ \{\mathit{True}, \mathit{False}\}\rangle,\ \langle 2,\ \bot\rangle\rangle
\end{array}
\qquad
\begin{array}{c}
[:\langle \mathit{True}, \bot\rangle, \langle \mathit{False}, \bot\rangle:] \\
\Downarrow \\
\langle 2,\ \langle [:\mathit{True}, \mathit{False}:],\ [:\bot, \bot:]\rangle\rangle \\
\Downarrow \\
\langle 2,\ \langle 2,\ \{\mathit{True}, \mathit{False}\}\rangle,\ \langle 2, \{\bot, \bot\}\rangle\rangle \\
\Downarrow \\
\langle 2,\ \langle 2,\ \{\mathit{True}, \mathit{False}\}\rangle,\ \langle 2,\ \bot\rangle\rangle
\end{array}$$

Here, the same flat representation is derived for two arrays which are, at first glance, different but contain no diverging elements. This is because the two integers 1 and $\bot$ in the first array are eventually mapped to the same unboxed array which diverges since one of its elements does. In the second array, both integers diverge, and, again, so does the corresponding unboxed array in the flat representation.

This implies that the two arrays must be semantically equivalent in $\lambda^{\mathcal{C}}$ because the flattening transformation *induces undefinedness* in the first one. In particular, the result of extracting the element at position 0 from either array must be $\langle \mathit{True}, \bot\rangle$. This is a somewhat unfortunate consequence of flattening, as these effects are by no means obvious to the programmer. Still, in our view the overall benefits of the NDP approach far outweigh this quite obscure deficiency.

Unfortunately, induced undefinedness also significantly complicates reasoning about the semantics of flattening. Assuming that f is defined as

$$\begin{array}{l} f \quad :: \texttt{Bool} \times \texttt{Int} \to \texttt{Int} \\ f\ \langle b, n\rangle \ = \ \textbf{if}\ b\ \textbf{then}\ n\ \textbf{else}\ 0 \end{array}$$

we have

$$\texttt{mapP}\ \langle \langle\!\langle f \rangle\!\rangle, [:\langle \mathit{True}, 1\rangle:]\rangle +\!\!+ \texttt{mapP}\ \langle \langle\!\langle f \rangle\!\rangle, [:\langle \mathit{False}, \bot\rangle:]\rangle \ \cong\ [:1:] +\!\!+ [:0:] \ \cong\ [:1, 0:]$$

but

$$\begin{array}{l} \texttt{mapP}\ \langle \langle\!\langle f \rangle\!\rangle, [:\langle \mathit{True}, 1\rangle, \langle \mathit{False}, \bot\rangle:]\rangle \cong\ \texttt{mapP}\ \langle \langle\!\langle f \rangle\!\rangle, [:\langle \mathit{True}, \bot\rangle, \langle \mathit{False}, \bot\rangle:]\rangle \cong\ [:\bot, 0:] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cong\ [:\bot, \bot:] \end{array}$$

This becomes even more obvious if we consider how these computations are performed in $\lambda^{\mathcal{A}}$. By vectorising and flattening the above example, we obtain:

$$\begin{array}{l} f^{\uparrow}\ \langle 1, \langle \overbrace{\langle 1, \{\mathit{True}\}\rangle}^{[:\mathit{True}:]}, \overbrace{\langle 1, \{1\}\rangle}^{[:1:]}\rangle\rangle +\!\!+ f^{\uparrow}\ \langle 1, \langle \overbrace{\langle 1, \{\mathit{False}\}\rangle}^{[:\mathit{False}:]}, \overbrace{\langle 1, \bot\rangle}^{[:\bot:]}\rangle\rangle \\ \cong \underbrace{\langle 1, \{1\}\rangle}_{[:1:]} +\!\!+ \underbrace{\langle 1, \{0\}\rangle}_{[:0:]} \cong \underbrace{\langle 1, \{1, 0\}\rangle}_{[:1,0:]} \end{array}$$

even though

$$f^{\uparrow}\ \langle 2, \langle \underbrace{\langle 2, \{\mathit{True}, \mathit{False}\}\rangle}_{[:\mathit{True},\mathit{False}:]}, \underbrace{\langle 2, \bot\rangle}_{[:\bot,\bot:]}\rangle\rangle \cong \underbrace{\langle 2, \bot\rangle}_{[:\bot,\bot:]}$$

In general, this implies that the following equivalence does not necessarily hold in the presence of nontermination:

$$\texttt{mapP}\ \langle f,\ xs +\!\!+ ys\rangle \ \cong\ \texttt{mapP}\ \langle f,\ xs\rangle +\!\!+ \texttt{mapP}\ \langle f,\ ys\rangle$$

To summarise, `mapP` in $\lambda^{\mathcal{C}}$ and lifted functions in $\lambda^{\mathcal{A}}$ are not immediately distributive over concatenated arrays. Still, a limited form of distributivity can be established which is of crucial importance for the correctness proofs given in this chapter.

6.1.3 Formalising strictness

Undefinedness induced by flattening in $\lambda^{\mathcal{C}}$ can be characterised more precisely as follows. Let x and y be two terms of same type. The result of $[:x:] +\!\!+ [:y:]$ is then equivalent to $[:x', y':]$ such that

- x' is equivalent to x except that it diverges in all positions in which y diverges and
- the reverse holds for y'.

In order to specify the semantics of concatenation and related operations, we must be able to compute x' and y' without using $+\!\!+$. In the following, we introduce a new primitive which allows us to do so.

We have already encountered the primitive `seq` which returns its second argument unless the first argument diverges.

$$\begin{array}{lll}
\texttt{seq} & :: \alpha \times \beta \to \beta & \\
\texttt{seq}\,\langle \bot,\, y\rangle & = \bot & \\
\texttt{seq}\,\langle x,\, y\rangle & = y & \textbf{if } x \neq \bot
\end{array}$$

This operator already has the desired semantics for primitive types: if x and y are integers, then $x' = \texttt{seq}\,\langle y, x\rangle$. However, it only inspects the topmost nodes of the two terms and thus cannot be used directly for products and sums. By combining `seq` with type-indexed functions, we can define the operator $\rhd$ (*impose*) which imposes the definedness structure of its first argument on the second one by recursively traversing the two terms in lockstep and, essentially, applying `seq` at each constructor node.

$$\begin{array}{ll}
\rhd_{\langle\tau\rangle} & :: \tau \times \tau \to \tau \\
x \rhd_{\langle\texttt{Int}\rangle} y & = \texttt{seq}\,\langle x,\, y\rangle \\
\langle x_1, x_2\rangle \rhd_{\langle\tau_1\times\tau_2\rangle} \langle y_1, y_2\rangle & = \langle x_1 \rhd_{\langle\tau_1\rangle} y_1,\; x_2 \rhd_{\langle\tau_2\rangle} y_2\rangle \\
\texttt{Left}\; x \rhd_{\langle\tau_1+\tau_2\rangle} \texttt{Left}\; y & = \texttt{Left}\,(x \rhd_{\langle\tau_1\rangle} y) \\
\texttt{repP}\,\langle m, x\rangle \rhd_{\langle[:\tau:]\rangle} \texttt{repP}\,\langle n, y\rangle & = \texttt{repP}\,\langle \texttt{seq}\,\langle m, n\rangle,\; x \rhd_{\langle\tau\rangle} y\rangle \\
\texttt{repP}\,\langle m, x\rangle \rhd_{\langle[:\tau:]\rangle} (ys +\!\!+\!\!+ zs) & = (\texttt{repP}\,\langle m, x\rangle \rhd_{\langle[:\tau:]\rangle} ys) +\!\!+\!\!+ (\texttt{repP}\,\langle m, x\rangle \rhd_{\langle[:\tau:]\rangle} zs) \\
\ldots &
\end{array}$$

The above is an excerpt from the definition of $\rhd$ in $\lambda^{\mathcal{C}}$ (the complete implementation is given in Appendix B). For types other than products, sums and parallel arrays, it is equivalent to `seq`. In the other cases, the semantics of `seq` are implicit in the pattern matching used at each constructor node. Additionally, corresponding components are combined recursively. This algorithm is obvious for tuples and sums; in the case of parallel arrays, $\rhd$ eventually imposes each element of the first array on each element of the second one by descending into replicated nodes and distributing over concatenated ones.

How can $\rhd$ be used to specify the strictness properties of array concatenation? Returning to the example from the previous section, it is easy to verify that $x' \cong y \rhd x$:

$$\langle \mathit{False}, \bot\rangle \rhd \langle \mathit{True}, 1\rangle \cong \langle \mathit{False} \rhd \mathit{True},\; \bot \rhd 1\rangle \cong \langle \mathit{True},\; \bot\rangle$$

Conversely, y' can be obtained by $y' \cong x \rhd y$. More importantly, since $[:x:] = \texttt{repP}\,\langle 1, x\rangle$, we also have

$$\begin{array}{ll}
& [:\langle \mathit{True}, 1\rangle:] +\!\!+\!\!+ [:\langle \mathit{False}, \bot\rangle:] \\
\cong & ([:\langle \mathit{False}, \bot\rangle:] \rhd [:\langle \mathit{True}, 1\rangle:]) +\!\!+\!\!+ ([:\langle \mathit{True}, 1\rangle:] \rhd [:\langle \mathit{False}, \bot\rangle:]) \\
\cong & [:\langle \mathit{False}, \bot\rangle \rhd \langle \mathit{True}, 1\rangle:] +\!\!+\!\!+ [:\langle \mathit{True}, 1\rangle \rhd \langle \mathit{False}, \bot\rangle:] \\
\cong & [:\langle \mathit{True}, \bot\rangle:] +\!\!+\!\!+ [:\langle \mathit{False}, \bot\rangle:]
\end{array}$$

In general, array concatenation satisfies the following equivalence:

$$xs +\!\!+\!\!+ ys \cong (ys \rhd xs) +\!\!+\!\!+ (xs \rhd ys)$$

From the above, the desired relation between mapping and concatenation is easily obtained for $\lambda^{\mathcal{C}}$:

$$\texttt{mapP}\,\langle f,\; xs +\!\!+\!\!+ ys\rangle \cong \texttt{mapP}\,\langle f,\; ys \rhd xs\rangle +\!\!+\!\!+ \texttt{mapP}\,\langle f,\; xs \rhd ys\rangle$$

In fact, every array primitive in $\lambda^{\mathcal{C}}$ whose semantics depends on the definedness of its arguments must be implemented using a similar strategy. For instance, indexing on arrays constructed with $+\!\!+$ is defined as follows:

$$(xs +\!\!+ ys) \mathbin{!:} i \;=\; (ys \rhd xs) \mathbin{!:} i \qquad \textbf{if } i < \texttt{lenP}\; xs$$
$$(xs +\!\!+ ys) \mathbin{!:} i \;=\; (xs \rhd ys) \mathbin{!:} (i - \texttt{lenP}\; xs) \textbf{ otherwise}$$

Of course, $\lambda^{\mathcal{A}}$ also defines the primitive $\rhd$ with analogous semantics. It has no separate lifted version since $\rhd^{\uparrow} = \rhd$. This becomes evident if we consider that $y \rhd^{\uparrow} x$ simply ensures that x diverges everywhere y does, which is precisely the computation performed by $y \rhd x$. Thus, the distributivity of lifted functions over concatenation in $\lambda^{\mathcal{A}}$ is formalised similarily to distributivity of mapping in $\lambda^{\mathcal{C}}$:

$$f^{\uparrow}\,(xs +\!\!+ ys) \;\cong\; f^{\uparrow}\,(ys \rhd xs) \;+\!\!+\; f^{\uparrow}\,(xs \rhd ys)$$

In Section 6.7, we prove this equivalence, which is fundamental to the correctness of the flattening transformation.

It is important to understand that $\rhd$ has only been introduced to for the purposes of reasoning about the semantics of parallel arrays and does not appear in the generated $\lambda^{\mathcal{A}}$ code. However, subsequent optimisations can utilise this primitive to achieve better performance while preserving the termination structure of the program. For instance, the following rewrite rule eliminates an unnecessary concatenation:

$$n \leq \texttt{lenP}\; xs \implies \texttt{takeP}\,\langle n,\; xs +\!\!+ ys\rangle \;\cong\; \texttt{takeP}\,\langle n, ys \rhd xs\rangle$$

6.2 Properties of primitives

In establishing the correctness of the flattening transformation, we will rely on a number of laws which are satisfied by the primitive functions used in this work. A formal derivation of these laws essentially amounts to specifying the semantics of each primitive and proving that the implementation conforms to it. Due to the large number of primitives and, especially, to the type-indexed nature of their implementations, it would be impractical to attempt this without the support of automated tools. Consequently, we do *not* provide such proofs in this work.

Nevertheless, the assumptions on the primitives must be made explicit. We summarise them in Figure 6.1 as a set of algebraic laws and a set of preconditions on the arguments of primitives. The former must be satisfied by the implementation of the standard library while the latter must be obeyed by the programmer. In the following, we assume that a primitive diverges if its preconditions are violated.

Note that some laws have non-trivial consequences. Consider, for instance, the instantiation of law *rep/nil* for terms of function type. The obvious implementation of `repP`:

$$\texttt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \rightrightarrows \mathcal{A}[\![\tau_2]\!]\rangle}\,\langle n, f\rangle \;=\; \langle n, f\rangle$$

clearly violates it, particularily so if f diverges. Therefore, replication must be specialised for $n = 0$ as follows:

$$\texttt{repP}_{\langle\tau\rangle}\,\langle 0, x\rangle \;=\; \texttt{nilP}_{\langle\tau\rangle}\,\langle\rangle$$

Note that Figure 6.1 also fixes the semantics of lifted primitives, since for every primitive p, $p^{\uparrow}\; xs \;\cong\; \texttt{mapP}\,\langle\langle\!\langle p\rangle\!\rangle, xs\rangle$. We discuss this in more detail in Section 6.7.

Algebraic laws

len/nil	$\texttt{lenP}\,(\texttt{nilP}\,\langle\rangle)$	$\cong$	0
len/rep	$\texttt{lenP}\,(\texttt{repP}\,\langle n, x\rangle)$	$\cong$	n
len/concat	$\texttt{lenP}\,(xs \mathbin{+\!\!+} ys)$	$\cong$	$\texttt{lenP}\,xs + \texttt{lenP}\,ys$
len/attach	$\texttt{lenP}\,(\texttt{attachP}\,\langle n, xs\rangle)$	$\cong$	n
len/zip	$\texttt{lenP}\,(\texttt{zipP}\,\langle xs, ys\rangle)$	$\cong$	$\texttt{seq}\,\langle\texttt{lenP}\,xs, \texttt{lenP}\,ys\rangle$
rep/nil	$\texttt{repP}\,\langle 0, x\rangle$	$\cong$	$\texttt{nilP}\,\langle\rangle$
rep/concat	$\texttt{repP}\,\langle m, x\rangle \mathbin{+\!\!+} \texttt{repP}\,\langle n, x\rangle$	$\cong$	$\texttt{repP}\,\langle m+n, x\rangle$
rep/zip	$\texttt{zipP}\,\langle\texttt{repP}\,\langle n, x\rangle, \texttt{repP}\,\langle n, y\rangle\rangle$	$\cong$	$\texttt{repP}\,\langle n, \langle x, y\rangle\rangle$
attach/nil	$\texttt{attachP}\,\langle 0, x\rangle$	$\cong$	$\texttt{nilP}\,\langle\rangle$
attach/len	$\texttt{attachP}\,\langle\texttt{lenP}\,xs, xs\rangle$	$\cong$	xs
attach/rep$_1$	$\texttt{attachP}\,\langle n, \texttt{repP}\,\langle n, x\rangle\rangle$	$\cong$	$\texttt{repP}\,\langle n, x\rangle$
attach/rep$_2$	$\texttt{attachP}\,\langle n, \bot\rangle$	$\cong$	$\texttt{repP}\,\langle n, \bot\rangle$
attach/concat	$\texttt{attachP}\,\langle m, xs\rangle \mathbin{+\!\!+} \texttt{attachP}\,\langle n, ys\rangle$	$\cong$	$\texttt{attachP}\,\langle m+n, xs \mathbin{+\!\!+} ys\rangle$
attach/zip	$\texttt{attachP}\,\langle n, \texttt{zipP}\,\langle xs, ys\rangle\rangle$	$\cong$	$\texttt{zipP}\,\langle\texttt{attachP}\,\langle n, xs\rangle, \texttt{attachP}\,\langle n, ys\rangle\rangle$
zip/nil	$\texttt{zipP}\,\langle\texttt{nilP}\,\langle\rangle, \texttt{nilP}\,\langle\rangle\rangle$	$\cong$	$\texttt{nilP}\,\langle\rangle$
concat/zip	$\texttt{zipP}\,\langle xs_1, ys_1\rangle \mathbin{+\!\!+} \texttt{zipP}\,\langle xs_2, ys_2\rangle$	$\cong$	$\texttt{zipP}\,\langle xs_1 \mathbin{+\!\!+} xs_2, ys_1 \mathbin{+\!\!+} ys_2\rangle$
combine/concat	$\texttt{combineP}\langle\texttt{repP}\langle m, \mathit{False}\rangle \mathbin{+\!\!+} \texttt{repP}\langle n, \mathit{True}\rangle, \langle xs, ys\rangle\rangle \cong xs \mathbin{+\!\!+} ys$		
pack/take	$\texttt{packP}\langle\texttt{repP}\langle m, \mathit{True}\rangle \mathbin{+\!\!+} \texttt{repP}\langle n, \mathit{False}\rangle, xs\rangle \cong \texttt{takeP}\langle m, xs\rangle$		
pack/drop	$\texttt{packP}\langle\texttt{repP}\langle m, \mathit{False}\rangle \mathbin{+\!\!+} \texttt{repP}\langle n, \mathit{True}\rangle, xs\rangle \cong \texttt{dropP}\langle m, xs\rangle$		
take/drop	$\texttt{takeP}\,\langle m, xs\rangle \mathbin{+\!\!+} \texttt{dropP}\,\langle m, xs\rangle$	$\cong$	xs
nil/apply	$(\texttt{nilP}\,\langle\rangle)\,(\texttt{nilP}\,\langle\rangle)$	$\cong$	$\texttt{nilP}\,\langle\rangle$
nil/capply	$\texttt{nilP}\,\langle\rangle \ddagger \texttt{nilP}\,\langle\rangle$	$\cong$	$\texttt{nilP}\,\langle\rangle$
map/rep	$\texttt{mapP}\,\langle c, x\rangle$	$\cong$	$\texttt{repP}\,\langle\texttt{lenP}\,x, c\rangle \ddagger x$
rep$_\bot$	$\texttt{repP}\,\langle\bot, x\rangle$	$\cong$	$\bot$
attach$_\bot$	$\texttt{attachP}\,\langle\bot, x\rangle$	$\cong$	$\bot$
zip$_\bot$	$\texttt{zipP}\,\langle\bot, x\rangle \cong \texttt{zipP}\,\langle x, \bot\rangle$	$\cong$	$\bot$

Preconditions

Operation	Precondition
$\texttt{attachP}\,\langle n, xs\rangle$	$\texttt{lenP}\,xs \cong n \vee \texttt{lenP}\,xs \cong \bot$
$\texttt{zipP}\,\langle xs, ys\rangle$	$\texttt{lenP}\,xs \cong \texttt{lenP}\,ys$
$\texttt{takeP}\,\langle n, xs\rangle$	$\texttt{lenP}\,xs \geq n$
$\texttt{dropP}\,\langle n, xs\rangle$	$\texttt{lenP}\,xs \geq n$
$\texttt{combineP}\,\langle bs, \langle xs, ys\rangle\rangle$	$\texttt{lenP}\,(\texttt{falsesP}\,bs) \cong \texttt{lenP}\,xs \wedge \texttt{lenP}\,(\texttt{truesP}\,bs) \cong \texttt{lenP}\,ys$

FIGURE 6.1 Properties of primitives

Values in $\lambda^{\mathcal{C}}$

$$
\begin{array}{lll}
v & ::= i \mid b \mid \langle\rangle & (\textit{literals}) \\
& \mid \quad \langle e_1, e_2 \rangle & (\textit{tuples}) \\
& \mid \quad \lambda\bullet : \tau.\ e \mid c^n_{\langle \tau_1,\ldots,\tau_n \rangle} & (\textit{functions}) \\
& \mid \quad \langle\!\langle v, \tau_1, \tau_2, \tau_3, e \rangle\!\rangle & (\textit{closures}) \\
& \mid \quad \mathtt{in}_{\mu\alpha.\tau}\ v & (\mu - \textit{constructors}) \\
& \mid \quad \mathtt{Left}_{\langle \tau_1,\tau_2 \rangle}\ e \mid \mathtt{Right}_{\langle \tau_1,\tau_2 \rangle}\ e & (\textit{sum constructors}) \\
& \mid \quad \mathtt{nilP}_{\langle \tau \rangle}\ \langle\rangle \mid \mathtt{repP}_{\langle \tau \rangle}\ \langle v, e \rangle \mid v_1 +\!\!+\!\!+_{\langle \tau \rangle}\ v_2 & (\textit{array constructors})
\end{array}
$$

Values in $\lambda^{\mathcal{A}}$

$$
\begin{array}{lll}
v & ::= i \mid b \mid \langle\rangle \mid \{i_1, \ldots, i_n\}_{\mathtt{Int}} \mid \{b_1, \ldots, b_n\}_{\mathtt{Bool}} & (\textit{literals}) \\
& \mid \quad \langle e_1, e_2 \rangle & (\textit{tuples}) \\
& \mid \quad \lambda\bullet : \tau.\ e \mid c^n_{\langle \tau_1,\ldots,\tau_n \rangle} & (\textit{functions}) \\
& \mid \quad \langle\!\langle \langle v_1, v_2 \rangle, \tau_1, \tau_2, \tau_3, e_2 \rangle\!\rangle \mid \langle\!\langle\!: e, \tau_1, \tau_2, \tau_3, v :\!\rangle\!\rangle & (\textit{closures}) \\
& \mid \quad \mathtt{in}_{\mu\alpha.\tau}\ v \mid \mathtt{in}^{\uparrow}_{\mu\alpha.\tau}\ v & (\mu - \textit{constructors}) \\
& \mid \quad \mathtt{Left}_{\langle \tau_1,\tau_2 \rangle}\ e \mid \mathtt{Right}_{\langle \tau_1,\tau_2 \rangle}\ e & (\textit{sum constructors})
\end{array}
$$

FIGURE 6.2 Values in $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$

6.3 Operational semantics

Before attempting to prove the correctness of the flattening transformation, we must, of course, define the semantics of its source and target languages. The languages $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$ are, like Haskell, non-strict and, consequently, we assume a call-by-name reduction strategy (Church, 1941). It should be noted that Haskell, as currently implemented, is a call-by-need language, such that terms are never evaluated more than once and the results of the evaluation are shared across all occurences of the term (Wadsworth, 1971). While an operational semantics can be defined such as to reflect sharing (Maraist et al., 1998; Launchbury, 1993), they, in general, only affect number of reductions and the size of the terms involved in a computation and not their observational equivalence. Intuitively, the former corresponds to the execution time and memory usage of a program, while the latter capture the correctness of the obtained results. Since it is precisely this notion of correctness which we investigate in this work, the effects of sharing are ignored in the operational semantics of the two languages and in the subsequent discussion.

6.3.1 Values

We begin the discussion of operational semantics by fixing the sets of values in $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$. A value is a term which cannot be reduced any further and is a valid result of a computation. As the two languages are non-strict, values define the weak head-normal forms of terms.

6.1 DEFINITION (VALUES)
A term in $\lambda^{\mathcal{C}}$ or $\lambda^{\mathcal{A}}$ is a value if it conforms to the respective grammar in Figure 6.2.

The definition of values highlights the strictness properties of parallel arrays and closures in the two languages. In $\lambda^{\mathcal{C}}$, a closure $\langle\!\langle f, \tau_1, \tau_2, \tau_3, e \rangle\!\rangle$ is strict in the closure function f, as reflected by the constraint that for the closure to be a value, f must a be value as well. That this is necessary should be obvious from the fact that such a closure represents the partial application of f to e and that $\bot\ e = \bot$. Note that closures are *not* strict in the environment, since in a non-strict language, $f\ \bot$ does not necessarily diverge.

The situation is similar in $\lambda^{\mathcal{A}}$. Here, closures are strict in both the vectorised and the lifted component of the function tuple. In contrast to this, array closures *lazy* in the function tuple and *strict* in the environment. This is due to the fact that array closures represent the result of mapping a curried function over an array. Therefore, they cannot be strict in the function, as discussed in Section 6.1, but, of course, the result of mapping over $\bot$ must diverge.

The most significant difference between the two languages lies, unsurprisingly, in the representation of arrays. Recall that the primitives `nilP`, `repP` and $(+\!\!+)$ are constructors in $\lambda^{\mathcal{C}}$ but functions in $\lambda^{\mathcal{A}}$. Consequently, while arrays constructed by these primitives are values in the former language, they are replaced by array literals and array closures in the latter. Note that `repP` is strict in the term determining the length of the array but lazy in the element which is to be replicated and $(+\!\!+)$ is strict in both arguments. This corresponds to the properties of these primitives discussed in the previous section.

6.3.2 Reduction

The operational semantics of $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$ is defined as a set of small-step reduction rules in the style of Plotkin (1981). For the core language constructs, these are given in Figure 6.3. The rules covering applications, lambda abstractions, tuples and recursive terms are shared between the two languages while those covering closures and arrays are different, capturing the differences in the strictness properties and semantics of the respective constructs. For the sake of clarity, we abbreviate $\langle\!\langle x, \tau_1, \tau_2, \tau_3, y \rangle\!\rangle$ to $\langle\!\langle x, \overline{\tau}, y \rangle\!\rangle$ and use a similar notation for array closures. In Figure 6.3, v ranges over values, requiring the corresponding term to be in weak head-normal form for the rule to apply.

In addition to the rules given in Figure 6.3, the operational semantics of the two languages also includes reduction rules for applications of primitives which are determined by the implementations of the latter given in Appendices B and C. For instance, the implementation of `repP` in $\lambda^{\mathcal{A}}$ includes the following clause:

$$\texttt{repP}_{\langle \underline{\texttt{Int}} \rangle}\ \langle !n,\ x \rangle\ =\ \langle n,\ \texttt{rep}_{\texttt{Int}}\ \langle n,\ x \rangle \rangle$$

Here, we write $!n$ to indicate that `repP` is strict in n. This definition has an obvious operational interpretation. In particular, it induces the following reduction rules:

$$\frac{\tau \equiv \underline{\texttt{Int}} \qquad e \longrightarrow e'}{\texttt{repP}_{\langle \tau \rangle}\ e \longrightarrow \texttt{repP}_{\langle \tau \rangle}\ e'} \qquad \frac{\tau \equiv \underline{\texttt{Int}} \qquad e_1 \longrightarrow e_1'}{\texttt{repP}_{\langle \tau \rangle}\ \langle e_1, e_2 \rangle \longrightarrow \texttt{repP}_{\langle \tau \rangle}\ \langle e_1', e_2 \rangle}$$

$$\frac{\tau \equiv \underline{\texttt{Int}}}{\texttt{repP}_{\langle \tau \rangle}\ \langle v, e \rangle \longrightarrow \langle v,\ \texttt{rep}_{\texttt{Int}}\ \langle v,\ x \rangle \rangle}$$

Note that in $\lambda^{\mathcal{A}}$, the evaluation of type-indexed primitives must take type equivalence into

Common rules

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad (\lambda\bullet : \tau.\ e_1)\ e_2 \longrightarrow e_1[e_2/\bullet] \qquad \frac{e_1 \longrightarrow e_1'}{e_1 \dagger e_2 \longrightarrow e_1' \dagger e_2} \qquad \frac{e \longrightarrow e'}{e.i \longrightarrow e'.i}$$

$$\langle e_1, e_2\rangle.i \longrightarrow e_i \qquad\qquad \mu\bullet : \tau.\ e \longrightarrow e[\mu\bullet : \tau.\ e/\bullet]$$

Additional rules for $\lambda^{\mathcal{C}}$

$$\frac{e_1 \longrightarrow e_1'}{\langle\!\langle e_1, \overline{\tau}, e_2\rangle\!\rangle \longrightarrow \langle\!\langle e_1', \overline{\tau}, e_2\rangle\!\rangle} \qquad \langle\!\langle v, \overline{\tau}, e_1\rangle\!\rangle \dagger e_2 \longrightarrow v\ \langle e_1, e_2\rangle$$

$$\frac{e_1 \longrightarrow e_1'}{\mathtt{repP}_{\langle\tau\rangle}\ \langle e_1, e_2\rangle \longrightarrow \mathtt{repP}_{\langle\tau\rangle}\langle e_1', e_2\rangle}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1 \mathbin{+\!\!+\!\!+}_{\langle\tau\rangle} e_2 \longrightarrow e_1' \mathbin{+\!\!+\!\!+}_{\langle\tau\rangle} e_2} \qquad \frac{e \longrightarrow e'}{v \mathbin{+\!\!+\!\!+}_{\langle\tau\rangle} e \longrightarrow v \mathbin{+\!\!+\!\!+}_{\langle\tau\rangle} e'}$$

Additional rules for $\lambda^{\mathcal{A}}$

$$\frac{e_1 \longrightarrow e_1'}{\langle\!\langle e_1, \overline{\tau}, e_2\rangle\!\rangle \longrightarrow \langle\!\langle e_1', \overline{\tau}, e_2\rangle\!\rangle} \qquad \frac{e_1 \longrightarrow e_1'}{\langle\!\langle \langle e_1, e_2\rangle, \overline{\tau}, e_3\rangle\!\rangle \longrightarrow \langle\!\langle \langle e_1', e_2\rangle, \overline{\tau}, e_3\rangle\!\rangle}$$

$$\frac{e_1 \longrightarrow e_1'}{\langle\!\langle \langle v, e_1\rangle, \overline{\tau}, e_2\rangle\!\rangle \longrightarrow \langle\!\langle \langle v, e_1'\rangle, \overline{\tau}, e_2\rangle\!\rangle} \qquad \langle\!\langle \langle v_1, v_2\rangle, \overline{\tau}, e_1\rangle\!\rangle \dagger e_2 \longrightarrow v_1\langle e_1, e_2\rangle$$

$$\frac{e_1 \longrightarrow e_1'}{e_1 \ddagger e_2 \longrightarrow e_1' \ddagger e_2} \qquad \frac{e_2 \longrightarrow e_2'}{\langle\!\langle\!: e_1, \overline{\tau}, e_2 :\!\rangle\!\rangle \longrightarrow \langle\!\langle\!: e_1, \overline{\tau}, e_2' :\!\rangle\!\rangle}$$

$$\langle\!\langle\!: e_1, \tau_1, \tau_2, \tau_3, v :\!\rangle\!\rangle \ddagger e_2 \longrightarrow e_1.2\ (\mathtt{zipP}_{\langle\tau_1,\tau_2\rangle}\ \langle v, e_2\rangle)$$

FIGURE 6.3 Operational semantics of $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$

account. The derivation of such rules is straightforward; it would be impractical, however, to include a complete list in this work. Thus, for each primitive defined in the Appendix, we tacitly assume that the corresponding rules are reflected by the reduction relation.

6.2 Definition (Operational semantics of $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$)
The reduction relation $\longrightarrow_{\mathcal{C}}$ is the least binary relation on terms in $\lambda^{\mathcal{C}}$ comforming to the rules given in Figure 6.3 and to those induced by the definitions of primitives in that language. The relation $\longrightarrow_{\mathcal{A}}$ is defined analogously for terms in $\lambda^{\mathcal{A}}$. The relations $\xrightarrow{*}_{\mathcal{C}}$ and $\xrightarrow{*}_{\mathcal{A}}$ are the reflexive transitive closures of the respective reduction relations. As usual, we omit the subscript when possible. We say that a term e evaluates to a value v and write $e \Downarrow v$ if $e \xrightarrow{*} v$. The term e converges, written as $e{\Downarrow}$, if $\exists v.e \Downarrow v$; it diverges, written as $e{\Uparrow}$, if $\neg(e{\Downarrow})$.

We conclude the discussion of operational semantics of the two languages by establishing subject reduction, a standard property stating that reduction does not change the type of a term.

6.3 Lemma (Subject reduction)
For all terms e and e' in $\lambda^{\mathcal{C}}$ or $\lambda^{\mathcal{A}}$ such that $e \longrightarrow e'$, if $e : \tau$, then $e' : \tau$.

Proof. By induction on the derivation $e \longrightarrow e'$. □

6.4 Semantic equivalence

We have already used the relation $\cong$ to denote semantic equivalence, but have so far not defined it formally. Semantic, or operational, equivalence relates terms which are syntactically different, but behave the same, i.e., encode the same computation. Commonly, it is defined as a *Morris-style contextual congruence* or *observational equivalence* (Morris, 1968; Barendregt, 1984). Observational equivalence is specified based on contexts, i.e., terms with a hole into which other terms can be "plugged in".

6.4 Definition (Contexts)
A context C in $\lambda^{\mathcal{C}}$ or $\lambda^{\mathcal{A}}$ is a term containing zero or more occurences of the hole $-$. $C[e]$ denotes the term obtained by replacing all occurences of $-$ by e. We say that C is a *closing context* for e if $C[e]$ is a closed term.

Contexts allow us to formalise the informal concept of "behaving the same": we consider two terms to be observationally equivalent if they are indistinguishable in all contexts. The observation we make is, as usual, simply the fact of convergence (Milner, 1977; Abramsky, 1990).

6.5 Definition (Semantic equivalence)
We say that two terms e and e' in $\lambda^{\mathcal{C}}$ or $\lambda^{\mathcal{A}}$ are semantically equivalent, written as $e \cong_{\mathcal{C}} e'$ and $e \cong_{\mathcal{A}} e'$, respectively, if for all closing contexts C in the language, $C[e]{\Downarrow} \iff C[e']{\Downarrow}$.

This definition is sufficient to capture the intuitive notion of equivalence since we can always construct contexts whose termination behaviour depends on the values e and e' evaluate to.

6.5 Mononomic lambda calculi

The languages $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$ differ from the standard lambda calculus in that they do not allows for multiple value variables. In the setting of this work, this principle is crucial as it provides precisely the separation between computation and data required to support higher-order functions. However, we expect such *mononomic* calculi to be more generally useful. To see why, we will now establish some properties which are necessary for the following correctness proofs but are also interesting in other contexts.

6.6 NOTATION
Given two terms e_1 and e_2 in $\lambda^{\mathcal{C}}$ or $\lambda^{\mathcal{A}}$, we write $e_1 \preceq e_2$ if e_1 is a subterm of e_2 and $e_1 \prec e_2$ if e_1 is a proper subterm of e_2.

As usual, the definition of operational semantics given in the previous section makes use of substitution to capture β-reduction. Substitution in a mononomic calculus, however, is much more restricted than in a lambda calculus with multiple value variables. This becomes obvious if we compare the respective definitions of substitution in lambda abstractions.

Mononomic substitution	Standard substitution
$(\lambda\bullet.\ e_1)[e_2/\bullet] \ = \ \lambda\bullet.\ e_1$	$(\lambda x.\ e_1)[e_2/y] \ = \ \begin{cases} \lambda x.\ e_1 & \text{if } x = y \\ \lambda x.\ e_1[e_2/y] & \text{otherwise} \end{cases}$

Standard substitution can descend into the term under λ if the substituted variable and the one bound by the abstraction are different. This, however, is never the cases in a mononomic calculus and, hence, substitution stops unconditionally. The following lemma captures this principle.

6.7 LEMMA
Let e_1 and e_2 be terms in $\lambda^{\mathcal{C}}$ or $\lambda^{\mathcal{A}}$. For every term $\lambda\bullet : \tau.\ e$ such that $\lambda\bullet : \tau.\ e \preceq e_1[e_2/\bullet]$, either $\lambda\bullet : \tau.\ e \preceq e_1$ or $\lambda\bullet : \tau.\ e \preceq e_2$.

PROOF. By induction on e_1. □

This property has far-reaching consequences for evaluation in mononomic calculi. Evaluation is based on β-reduction which, in turn, is based on substitution. Since substitution cannot mutate lambda abstraction, evaluation cannot, either. Thus, if we consider two terms e and e' such that $e \xrightarrow{*} e'$, then every lambda abstraction in e' must be present in e as well.

6.8 THEOREM (IMMUTABILITY OF COMPUTATION)
Let e be a term of function type in $\lambda^{\mathcal{C}}$ or $\lambda^{\mathcal{A}}$. If $e \xrightarrow{*} e'$ and $\lambda\bullet : \tau.\ x \preceq e'$, then $\lambda\bullet : \tau.\ x \preceq e$.

PROOF. Obvious from Lemma 6.7 and the operational semantics of the two languages. □

This insight is of crucial importance for the development in the rest of this chapter. In particular, it implies that if a statement holds for all abstractions in e, then it also holds for all abstractions in e' regardless of the reduction steps that lie between the two terms. We will frequently use this principle in the correctness proofs.

The same guarantee is provided by λ_σ and other lambda calculi with explicit substitutions (Abadi et al., 1991; Hardin et al., 1998). Here, the notion of substitution is internalised in the reduction rules, such that substitutions are only performed on a term when the latter

is reduced which, of course, never happens for lambda abstractions with outermost reduction. Operationally, lambda abstractions with deferred substitutions correspond precisely to closures. However, while computations are immutable in λ_σ, no theory exists which permits deferred substitutions to be manipulated by the program. The latter property is crucial for the implementation of array closures and is provided by the mononomic calculi.

6.6 Properties of flattened terms

In this section, we introduce a number of auxiliary results about terms generated by flattening, which are of crucial importance for the rest of the development. We begin by establishing the substitutivity of vectorisation.

6.9 Lemma (Substitutivity of vectorisation)
For all $\lambda^{\mathcal{C}}$ terms e and e', $\mathcal{V}[\![e]\!][\mathcal{V}[\![e']\!]/\bullet] = \mathcal{V}[\![e[e'/\bullet]]\!]$.

Proof. By straightforward induction on e. □

From the definition of reduction in $\lambda^{\mathcal{A}}$, it is easy to see that if, for some e_1, $\mathcal{V}[\![e_1]\!]$ reduces to some e_1', then there does not necessarily exist a e_2 such that $e_1' = \mathcal{V}[\![e_2]\!]$. In general, we only want to consider terms obtained by flattening; in the context of operational semantics, however, this implies that we also must be able to reason about their reducts. The following definition summarises the important invariants with respect to the tupling of functions which flattening relies upon and which are preserved by reduction.

6.10 Definition ($\mathcal{V}$-compatibility)
A $\lambda^{\mathcal{A}}$ term e is *$\mathcal{V}$-compatible* if it satisfies the following conditions.

1. For every $x \preceq e$ such that $x : \mathcal{F}[\![\tau_1 \to \tau_2]\!]$ and x converges, either $x \xrightarrow{*} \mathcal{V}[\![\lambda\bullet : \tau_1.\ y]\!]$ or $x \xrightarrow{*} \mathcal{V}[\![c^n_{\langle v_1,\ldots,v_n\rangle}]\!]$.
2. If $e \xrightarrow{*} e'$ and $\mathcal{V}[\![\lambda\bullet : \tau.\ x]\!] \preceq e'$, then $\mathcal{V}[\![\lambda\bullet : \tau.\ x]\!] \preceq e$.
3. If $e \xrightarrow{*} e'$, then e' is $\mathcal{V}$-compatible.

6.11 Fact
If e is a $\lambda^{\mathcal{C}}$ term, then $\mathcal{V}[\![e]\!]$ and $\mathcal{L}[\![n, e]\!]$ are $\mathcal{V}$-compatible.

Fact 6.11 follows from the obsevation that the reduction rules in $\lambda^{\mathcal{A}}$ (including those for primitives) never break up function tuples except for applying one of the components and never construct new ones.

6.7 Correctness of lifting

We are now in the position to establish the first major result about the correctness of flattening by identifying the relationship between vectorisation and lifting. The main idea underlying the flattening transformation is, for each function f, to generate a lifted function $f^{\uparrow}$, thereby combining the effects of mapping f over an array into a single algorithm. Clearly, f and $f^{\uparrow}$

are semantically related; this relation must be formalised, and we must show that the lifted functions derived by flattening are adequate with respect to it.

Fortunately, the algebraic properties of mapping are well-known. In particular, the Bird-Meertens formalism identifies three key properties of mapping over finite cons lists (Bird, 1989; Malcolm, 1990):

$$\begin{array}{lcl} \mathit{map}\ f\ [] & = & [] \\ \mathit{map}\ f\ [x] & = & [f\ x] \\ \mathit{map}\ f\ (xs +\!\!+ ys) & = & \mathit{map}\ f\ xs\ +\!\!+\ \mathit{map}\ f\ ys \end{array}$$

These laws are easily transferred to arrays, resulting in the following equivalences:

$$\begin{array}{lll} (1) & \texttt{mapP}\ \langle \mathcal{V}[\![c]\!],\ \bot \rangle & \cong\ \bot \\ (2) & \texttt{mapP}\ \langle \mathcal{V}[\![c]\!],\ \texttt{nilP}\ \langle\rangle \rangle & \cong\ \texttt{nilP}\ \langle\rangle \\ (3) & \texttt{mapP}\ \langle \mathcal{V}[\![c]\!],\ \texttt{repP}\ \langle n, x \rangle \rangle & \cong\ \texttt{repP}\ \langle n,\ \mathcal{V}[\![c]\!]\ \dagger\ x \rangle \\ (4) & \texttt{mapP}\ \langle \mathcal{V}[\![c]\!],\ xs +\!\!+\!\!+ ys \rangle & \cong\ \texttt{mapP}\ \langle \mathcal{V}[\![c]\!],\ ys \rhd xs \rangle\ +\!\!+\!\!+\ \texttt{mapP}\ \langle \mathcal{V}[\![c]\!],\ xs \rhd ys \rangle \end{array}$$

Equivalence (1) formalises the strictness of `mapP` in the array argument, which is usually left implicit in the Bird-Meertens formalism. Equivalence (4) is based on $\rhd$ as described in Section 6.1. Note that we only consider $\lambda^{\mathcal{A}}$ closures which have been obtained by vectorisation. In Section 6.8, we will see that this requirement is necessary, because these equivalence do not hold for all $\lambda^{\mathcal{A}}$ programs.

The above laws only describe properties of closures. These cannot be shown directly, however, as they ultimately rely on the correctness of lifting. Thus, we must demonstrate that lifted functions exhibit similar properties.

$$\begin{array}{lllll} (1a) & (\mathcal{V}[\![e]\!].2\ e')\Uparrow & & & \text{if } e'\Uparrow \\ (2a) & \mathcal{V}[\![e]\!].2\ (\texttt{nilP}\ \langle\rangle) & \cong\ \texttt{nilP}\ \langle\rangle & & \text{if } \mathcal{V}[\![e]\!]\Downarrow \\ (3a) & \mathcal{V}[\![e]\!].2\ (\texttt{repP}\ \langle 1, x \rangle) & \cong\ \texttt{repP}\ \langle 1,\ \mathcal{V}[\![e]\!].1\ x \rangle & & \text{if } \mathcal{V}[\![e]\!]\Downarrow \\ (4a) & \mathcal{V}[\![e]\!].2\ (xs +\!\!+\!\!+ ys) & \cong\ \mathcal{V}[\![e]\!].2\ (ys \rhd xs)\ +\!\!+\!\!+\ \mathcal{V}[\![e]\!].2\ (xs \rhd ys) & & \end{array}$$

Here, we assume that e has a function type and, hence, $\mathcal{V}[\![e]\!].2$ selects the appropriate lifted function. The correspondence to the laws formulated earlier is obvious, but, clearly, equivalences (2a) and (3a) can only hold if the function itself converges. Note that these laws, together with the ones given in Figure 6.1, also fix the semantics of lifted primitives. The rest of this section is devoted to demonstrating that these requirements are met by flattened programs.

6.7.1 Strictness

To establish equivalence (1a), we will first show a slightly modified version of the conclusion.

6.12 LEMMA
Let e be a $\lambda^{\mathcal{C}}$ term such that $\bullet : \upsilon \vdash e : \tau$. For all terms $e' : \mathcal{F}[\![[:\upsilon:]]\!]$ in $\lambda^{\mathcal{A}}$, if e' diverges, then so does $\mathcal{L}[\![\texttt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ e', e]\!][e'/\bullet]$.

PROOF. The proof is by straightforward induction on e. The conclusion follows from the fact that `repP` and `attachP` are strict in the length. □

With Lemma 6.12 in hand, we can now show that lifted functions are indeed strict in their argument.

6.13 Lemma (Strictness of lifting)
Let e be a term in $\lambda^{\mathcal{C}}$ and $e' : \mathcal{F}[\![[:\tau_1:]]\!]$ an $\mathcal{V}$-compatible term in $\lambda^{\mathcal{A}}$ such that $e'{\Uparrow}$.

1. *If $e : \tau_1 \to \tau_2$, then $(\mathcal{V}[\![e]\!].2\ e'){\Uparrow}$.*
2. *If $e : \tau_1 \Rightarrow \tau_2$, then $(\mathtt{mapP}\ \langle \mathcal{V}[\![e]\!], e' \rangle){\Uparrow}$.*

Proof. We show the two parts separately.

Part 1. The conclusion follows directly if $\mathcal{V}[\![e]\!]$ diverges. Assume that $\mathcal{V}[\![e]\!]$ converges; by Definition 6.10 and Fact 6.11, it either evaluates to some $\mathcal{V}[\![\lambda\bullet : \tau_1.\ e_1]\!]$ or to some $\mathcal{V}[\![c^n_{\langle v_1,\ldots,v_2\rangle}]\!]$.

Case $\mathcal{V}[\![e]\!] \Downarrow \mathcal{V}[\![\lambda\bullet : \tau_1.\ e_1]\!]$ Then, $\mathcal{V}[\![e]\!].2\ e' \cong \mathcal{V}[\![\lambda\bullet : \tau_1.\ e_1]\!].2\ e' \cong \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\tau_1]\!]\rangle}\ \bullet, e_1]\!][e'/\bullet]$. The last term diverges by Lemma 6.12.

Case $\mathcal{V}[\![e]\!] \Downarrow \mathcal{V}[\![c^n_{\langle v_1,\ldots,v_2\rangle}]\!]$ The conclusion holds if all lifted primitives in $\lambda^{\mathcal{A}}$ are strict in their argument, as required earlier.

Part 2. We have

$$
\begin{array}{ll}
& \mathtt{mapP}\ \langle \mathcal{V}[\![e]\!],\ \bot \rangle \\
\cong & \mathtt{repP}\ \langle \mathtt{lenP}\ \bot,\ e \rangle\ \ddagger\ \bot & \text{by Law } \textit{map/rep} \\
\cong & \bot\ \ddagger\ \bot & \text{by Law } \textit{rep}_\bot \\
\cong & \bot &
\end{array}
$$

□

6.7.2 Preservation of empty arrays

The approach to establishing equivalence (2) is similar to the one used above. Instead of proving the equivalence directly, we will first consider the substitution of `nilP` into a lifted term.

6.14 Lemma
For every $\lambda^{\mathcal{C}}$ term e such that $\bullet : v \vdash e : \tau$, $\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![v]\!]\rangle}\ \bullet, e]\!][\mathtt{nilP}_{\langle \mathcal{A}[\![v]\!]\rangle}\ \langle\rangle/\bullet] \cong \mathtt{nilP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle\rangle$.

Proof. The proof is by straightforward structural induction on e. In most cases, the conclusion follows directly from Laws *len/nil* and *rep/nil* or *attach/nil*. In the case $e = \bullet$, the conclusion is immediate by definition of $\mathcal{L}[\![\cdot,\cdot]\!]$ and substitution. In the case $e = e_1\ e_2$, the conclusion is by induction hypothesis on e_1 and e_2 and Law *zip/nil*. □

From Lemma 6.14, equivalence (2) is easily established.

6.15 Lemma (Preservation of empty arrays)
Let e be a term in $\lambda^{\mathcal{C}}$.

1. *If $e : \tau_1 \to \tau_2$ and $\mathcal{V}[\![e]\!]$ converges, then $\mathcal{V}[\![e]\!].2\ (\mathtt{nilP}_{\langle \mathcal{A}[\![\tau_1]\!]\rangle}\ \langle\rangle) \cong \mathtt{nilP}_{\langle \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle\rangle$.*
2. *If $e : \tau_1 \Rightarrow \tau_2$, then $\mathtt{mapP}\ \langle \mathcal{V}[\![e]\!], \mathtt{nilP}_{\langle \mathcal{A}[\![\tau_1]\!]\rangle}\ \langle\rangle\rangle \cong \mathtt{nilP}_{\langle \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle\rangle$.*

Proof. We show the two parts separately.

Part 1. By Definition 6.10 and Fact 6.11, $\mathcal{V}[\![e]\!]$ either evaluates to some $\mathcal{V}[\![\lambda\bullet : \tau_1.\ e']\!]$ or to some $\mathcal{V}[\![c^n_{\langle v_1,\ldots,v_2\rangle}]\!]$.

Case $\mathcal{V}[\![e]\!] \Downarrow \mathcal{V}[\![\lambda\bullet : \tau_1.\ e']\!]$ Then,

$$
\begin{array}{ll}
& \mathcal{V}[\![e]\!].2\ (\mathtt{nilP}_{\langle\mathcal{A}[\![\tau_1]\!]\rangle}\ \langle\rangle) \\
\cong & \mathcal{V}[\![\lambda\bullet : \tau_1.\ e']\!].2\ (\mathtt{nilP}_{\langle\mathcal{A}[\![\tau_1]\!]\rangle}\ \langle\rangle) \\
\cong & \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\tau_1]\!]\rangle}\ \bullet, e_1]\!][\mathtt{nilP}_{\langle\mathcal{A}[\![\tau_1]\!]\rangle}\ \langle\rangle/\bullet] & \text{by definition of } \mathcal{V}[\![\cdot]\!] \\
\cong & \mathtt{nilP}_{\langle\mathcal{A}[\![\tau_2]\!]\rangle}\ \langle\rangle & \text{by Lemma 6.14}
\end{array}
$$

Case $\mathcal{V}[\![e]\!] \Downarrow \mathcal{V}[\![c^n_{\langle v_1,\ldots,v_2\rangle}]\!]$ This case again relies on the correctness of lifted primitives.

Part 2. We have

$$
\begin{array}{ll}
& \mathtt{mapP}\ \langle\mathcal{V}[\![e]\!],\ \mathtt{nilP}_{\langle\mathcal{A}[\![\tau_1]\!]\rangle}\ \langle\rangle\rangle \\
\cong & \mathtt{repP}_{\langle\mathcal{A}[\![\tau_1]\!]\underline{\Rightarrow}\mathcal{A}[\![\tau_2]\!]\rangle}\ \langle 0,\ \mathcal{V}[\![e]\!]\rangle\ \ddagger\ \mathtt{nilP}_{\langle\mathcal{A}[\![\tau_1]\!]\rangle}\ \langle\rangle & \text{by Law } \textit{map/rep} \\
\cong & \mathtt{nilP}_{\langle\mathcal{A}[\![\tau_1]\!]\underline{\Rightarrow}\mathcal{A}[\![\tau_2]\!]\rangle}\ \langle\rangle\ \ddagger\ \mathtt{nilP}_{\langle\mathcal{A}[\![\tau_1]\!]\rangle}\ \langle\rangle & \text{by Law } \textit{rep/nil} \\
\cong & \mathtt{nilP}_{\langle\mathcal{A}[\![\tau_2]\!]\rangle}\ \langle\rangle & \text{by Law } \textit{nil/capply}
\end{array}
$$

□

6.7.3 Replicativity

The strategy for proving equivalence (3) is fairly similar to that used previously. Again, we proceed by investigating the corresponding property of substitution and derive the desired conclusion from that. Our intention is to show that $\mathtt{repP}\ \langle 1,\ \mathcal{V}[\![e]\!][e'/\bullet]\rangle \cong \mathcal{L}[\![1, e]\!][\mathtt{repP}\ \langle 1, e'\rangle/\bullet]$ by structural induction on e. Unfortunately, this strategy fails in the crucial case $e = e_1\ e_2$, where we derive

$$\mathtt{repP}\ \langle 1,\ \mathcal{V}[\![e_1]\!][e'/\bullet].1\ \mathcal{V}[\![e_2]\!][e'/\bullet]\rangle$$

for the left-hand side of the equivalence. Here, the induction hypothesis cannot be used directly. We know, however, that in a well-typed program $\mathcal{V}[\![e_1]\!][e'/\bullet].1$ either diverges, evaluates to a primitive or evaluates to a lambda abstraction. The first two cases are trivial. In the latter case, we have $\mathcal{V}[\![e_1]\!][e'/\bullet] \cong \lambda\bullet.\ e_3$ for some e_3 and, hence,

$$\mathtt{repP}\ \langle 1,\ \mathcal{V}[\![e_1]\!][e'/\bullet].1\ \mathcal{V}[\![e_2]\!][e'/\bullet]\rangle\ \cong\ \mathtt{repP}\ \langle 1,\ e_3[\mathcal{V}[\![e_1]\!][e'/\bullet]/\bullet]\rangle$$

Since lambda abstractions are never substituted into, $\lambda\bullet.\ e_3$ must be a subterm of either $\mathcal{V}[\![e_1]\!]$ or e'. In the former case, the induction hypothesis can be used on e_3 if the proof proceeds by *complete* structural induction. However, this still does not allows us to derive the conclusion if $\lambda\bullet.\ e_3$ is a subterm of e'. Thus, we must require that the vectorised functions in e' *already* satisfy the condition we are about to show, as formalised by the following definition.

6.16 DEFINITION (REPLICATIVITY)
A term e in $\lambda^{\mathcal{A}}$ is *replicative* if it is $\mathcal{V}$-compatible and for all $\mathcal{V}[\![\lambda\bullet : \tau_1.\ e']\!] \preceq e$ such that $\bullet : \tau_1 \vdash e' : \tau_2$ and all $x : \mathcal{F}[\![\tau_1]\!]$,

$$\mathcal{V}[\![\lambda\bullet : \tau_1.\ e']\!].2\ (\mathtt{repP}_{\langle\mathcal{A}[\![\tau_1]\!]\rangle}\ \langle 1,\ x\rangle)\ \cong\ \mathtt{repP}_{\langle\mathcal{A}[\![\tau_2]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![\lambda\bullet : \tau_1.\ e']\!].1\ x\rangle$$

By imposing the requirement that e' is replicative, we circumvent the problems described above. We will see below that this requirement is redundant; but the redundancy cannot be shown directly. Note that the proof strategy outlined above crucially relies on the immutability of computation in $\lambda^{\mathcal{A}}$. This is not surprising, as it is precisely this property which has allowed us to extend the flattening transformation to higher-order functions.

Before tackling the proof, let us identify two important properties of replicative terms.

6.17 Fact
All subterms of a replicative term are replicative.

Proof. Obvious from Definition 6.16. □

6.18 Proposition
Reduction and substitution preserve replicativity.

Proof. Immediate from the definition of $\mathcal{V}$-compatibility and Fact 6.17. □

The following lemma is similar to Lemma 6.14; however, in addition to identifying an equivalence with respect to substitution, it also includes replicativity in the conclusion. This is, strictly speaking, unnecessary, but slightly simplifies the proof.

6.19 Lemma
Let e be a $\lambda^{\mathcal{C}}$ term such that $\bullet : v \vdash e : \tau$. For all replicative terms $x : \mathcal{F}[\![v]\!]$,

1. *$\mathcal{V}[\![e]\!][x/\bullet]$ is replicative and*
2. *$\mathcal{L}[\![1, e]\!][\mathtt{repP}_{\langle \mathcal{A}[\![v]\!] \rangle}\ \langle 1,\ x\rangle/\bullet] \cong \mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e]\!][x/\bullet]\rangle$*

Proof. The two parts are shown simultaneously by complete induction on e. The complete proof is given in Appendix A.2. □

6.20 Corollary
All $\mathcal{V}$-compatible terms are replicative.

Proof. Evident from the observation that every $\mathcal{V}[\![\lambda\bullet : \tau.\ e]\!]$ is replicative by Part 1 of Lemma 6.19 with, e.g., $x = \langle\rangle$. □

Even though the proof of Lemma 6.19 crucially depends on the replicativity of x, Corollary 6.20 demonstrates that this requirement is redundant. Consequently, it can be omitted, allowing us to establish equivalence (3) for functions and a similar property for closures.

6.21 Lemma
Let e be a term in $\lambda^{\mathcal{C}}$ such that $\mathcal{V}[\![e]\!]\Downarrow$ and $x : \mathcal{F}[\![\tau_1]\!]$ a $\mathcal{V}$-compatible term in $\lambda^{\mathcal{A}}$.

1. *If $e : \tau_1 \to \tau_2$, then*
   $\mathcal{V}[\![e]\!].2\ (\mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \rangle}\ \langle 1, x\rangle) \cong \mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!] \rangle}\ \langle 1, \mathcal{V}[\![e]\!].1\ x\rangle$.
2. *If $e : \tau_1 \Rightarrow \tau_2$, then*
   $\mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \underline{\Rightarrow} \mathcal{A}[\![\tau_2]\!] \rangle} \langle 1, \mathcal{V}[\![e]\!]\rangle \ddagger \mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \rangle}\ \langle 1, x\rangle \cong \mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!] \rangle} \langle 1, \mathcal{V}[\![e]\!] \dagger x\rangle$.

Proof. We first establish Part 1 and then show that Part 2 is implied by it.

Part 1. $\mathcal{V}[\![e]\!]$ evaluates either to some $\mathcal{V}[\![\lambda\bullet : \tau_1.\ e']\!]$ or to some $\mathcal{V}[\![c^n_{\langle v_1,\ldots,v_n\rangle}]\!]$. The first case is immediate from Corollary 6.20. The second case follows from the requirements on lifted primitives.

Part 2. For Part 2, we assume that $\mathcal{V}[\![e]\!]$ evaluates to some $\langle\!\langle p, \mathcal{A}[\![v]\!], \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], z\rangle\!\rangle$. As before, either $p \cong \mathcal{V}[\![\lambda\bullet : v \times \tau_1.\ e']\!]$ or $p \cong \mathcal{V}[\![c^n_{\langle v_1,\ldots,v_n\rangle}]\!]$. In any case, we have

$$
\begin{aligned}
& \mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e]\!] \dagger x\rangle \\
\cong\ & \mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle 1,\ \langle\!\langle p, \mathcal{A}[\![v]\!], \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], z\rangle\!\rangle \dagger x\rangle \\
\cong\ & \mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle 1,\ p.1\ \langle z, x\rangle\rangle \\
\cong\ & p.2\ (\mathtt{repP}_{\langle \mathcal{A}[\![v]\!] \underline{\times} \mathcal{A}[\![\tau_1]\!]\rangle}\ \langle 1,\ \langle z, x\rangle\rangle) && \text{by Part 1} \\
\cong\ & p.2\ \langle 1,\ \langle \mathtt{repP}_{\langle \mathcal{A}[\![v]\!]\rangle}\ \langle 1, z\rangle,\ \mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!]\rangle}\ \langle 1, x\rangle\rangle\rangle && \text{by definition of } \mathtt{repP} \\
\cong\ & p.2\ (\mathtt{zipP}_{\langle \mathcal{A}[\![v]\!], \mathcal{A}[\![\tau_1]\!]\rangle} \langle \mathtt{repP}_{\langle \mathcal{A}[\![v]\!]\rangle}\ \langle 1, z\rangle,\ \mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!]\rangle}\ \langle 1, x\rangle\rangle) && \text{by definition of } \mathtt{zipP} \\
\cong\ & \langle\!\langle\!\langle p, \mathcal{A}[\![v]\!], \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathtt{repP}_{\langle \mathcal{A}[\![v]\!]\rangle}\ \langle 1, z\rangle\rangle\!\rangle\!\rangle \ddagger \mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!]\rangle}\ \langle 1, x\rangle \\
\cong\ & \mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \underline{\Rightarrow} \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle 1, \langle\!\langle\!\langle p, \mathcal{A}[\![v]\!], \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], z\rangle\!\rangle\!\rangle\rangle \ddagger \mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!]\rangle}\ \langle 1, x\rangle \\
& && \text{by definition of } \mathtt{repP} \\
\cong\ & \mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \underline{\Rightarrow} \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle 1, \mathcal{V}[\![e]\!]\rangle \ddagger \mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!]\rangle}\ \langle 1, x\rangle
\end{aligned}
$$

□

6.7.4 Concatenativity

The remaining requirement on lifting is equivalence (4), which captures the distributivity of mapping and concatenation. As before, we will first establish a corresponding property for substitution:

$$\mathcal{L}[\![\mathtt{lenP}\ \bullet, e]\!][y \rhd x/\bullet] +\!\!+\!\!+ \mathcal{L}[\![\mathtt{lenP}\ \bullet, e]\!][x \rhd y/\bullet] \ \cong\ \mathcal{L}[\![\mathtt{lenP}\ \bullet, e]\!][x +\!\!+\!\!+ y/\bullet]$$

Recall, however, that concatenation is not supported by arrays of functions, as described in Section 4.6. This implies that such arrays may not occur in the terms being concatenated. Unfortunately, this restriction also significantly complicates the proof of the equivalence.

The proof is, again, by structural induction on e. In the case $e = e_1\ e_2$, however, the induction hypothesis cannot be used on e_1 because the term $\mathcal{L}[\![\mathtt{lenP}\ \bullet, e_1]\!][y \rhd x/\bullet]$ represents an array of functions which cannot be concatenated. This makes a direct proof impossible.

We can circumvent this problem by making the following observation. The execution model underlying the semantics of the intermediate languages is data-parallel, which implies that the *structure* of computations performed in a parallel context does not depend on the *data*. Assume that f is a higher-order function of type $\tau_1 \to (\tau_2 \to \tau_3)$. By mapping it over an array of type $[:\tau_1:]$, we can obtain an array of functions of type $[:\tau_2 \to \tau_3:]$; but the computation encoded by this array will be the same regardless of the arguments supplied to f.

More specifically, the lifted version f will yield function arrays represented by the type $\mathtt{Int} \times ((\tau_2 \to \tau_3) \times ([:\tau_2:] \to [:\tau_3:]))$ and, satisfy the equality $(f^\uparrow\ x).2 = (f^\uparrow\ y).2$ for all values of x and y, unless either of the two results diverge. This principle also extends to more complex data structures which embed arrays of functions.

The technical problem described earlier can now be solved by establishing that the terms $\mathcal{L}[\![\mathtt{lenP}\ \bullet, e_1]\!][x/\bullet].2$ and $\mathcal{L}[\![\mathtt{lenP}\ \bullet, e_2]\!][y/\bullet].2$ evaluate to the same function tuple regardless of the values of x and y, thereby eliminating the need to rely on the induction hypothesis in this case. First, of course, we must formalise this principle. We do so by introducing a notion

of equivalence which only depends on termination and equality of functions but disregards the data stored in data structures.

6.22 Definition (Data-erasing equivalence)
Let $\sim_\tau$ be the largest reflexive, symmetric, transitive and type equivalence respecting family of relations obeying the rules given below. We call $\sim_\tau$ *data-erasing equivalence* and say that e and e' are *equivalent under data-erasure* if $e \sim_\tau e'$.

$$
\begin{array}{ll}
e \sim_\tau e' & \Longleftrightarrow (e{\Downarrow} \iff e'{\Downarrow}) \\
 & \quad \text{if } \tau \in \{\mathtt{Int}, \mathtt{Arr}_{\mathtt{Int}}, \mathtt{Arr}_{\mathtt{Bool}}\} \text{ or } \tau = \tau_1 \Rrightarrow \tau_2 \\
e \sim_{\tau_1 \times \tau_2} e' & \Longleftrightarrow (\forall e_1\ e_2.\ e \Downarrow \langle e_1, e_2\rangle \implies \exists e_1'\ e_2'.\ e' \Downarrow \langle e_1', e_2'\rangle \wedge\ e_1 \sim_{\tau_1} e_1' \wedge\ e_2 \sim_{\tau_2} e_2') \\
e \sim_{(\mu\alpha.\tau).2} e' & \Longleftrightarrow (\forall e_1.\ e \Downarrow \mathtt{in}^{\uparrow}_{\mu\alpha.\tau}\ e_1 \implies \exists e_1'.\ e' \Downarrow \mathtt{in}^{\uparrow}_{\mu\alpha.\tau}\ e_1' \wedge\ e_1 \sim_{\tau[\mu\alpha.\tau/\alpha].2} e_1') \\
e \sim_{\tau_1 \to \tau_2} e' & \Longleftrightarrow (\forall\, x.\ e \Downarrow \boldsymbol{\lambda}\bullet : \tau_1.\ x \implies e' \Downarrow \boldsymbol{\lambda}\bullet : \tau_1.\ x) \\
 & \quad \wedge\ (\forall\, c\ v_1\ \ldots\ v_n.\ e \Downarrow c^n_{\langle v_1,\ldots,v_n\rangle} \implies e' \Downarrow c^n_{\langle v_1,\ldots,v_n\rangle})
\end{array}
$$

The definition of data-erasing equivalence is, essentially, coinductive, which is not surprising given that we considering a non-strict language. Moreover, data-erasing equivalence is only defined for those types which are used for representing parallel arrays. It is easily extended to cover all types in $\lambda^{\mathcal{A}}$, but doing so does not seem worthwhile as we will only use it to compare the results of parallel computations.

The erasure of data is captured by the rules for integers and unboxed arrays which only require the terms to have the same termination behaviour in all positions, but abstract from their values. Crucially, only syntactically equal functions are equivalent under data erasure — this is a direct encoding of the principle described above.

Since data-erasing equivalence is defined in terms of reduction, we can immediately identify the following useful property.

6.23 Proposition
Reduction preserves data-erasing equivalence.

Proof. Obvious from Definition 6.22. □

Subsequent proofs rely in the primitives `lenP`, `attachP` and `zipP` being extensional with respect to data-erasing equivalence, as formalised by the following lemma.

6.24 Lemma
Let x and y be terms in $\lambda^{\mathcal{A}}$ of type τ such that $x \sim y$.

1. *If $\tau = \mathcal{F}[\![[:\tau':]]\!]$, then $\mathtt{lenP}_{\langle \mathcal{A}[\![\tau']\!]\rangle}\ x \sim \mathtt{lenP}_{\langle \mathcal{A}[\![\tau']\!]\rangle}\ y$.*
2. *If $\tau = \mathcal{F}[\![[:\tau_1:]]\!] \times \mathcal{F}[\![[:\tau_2:]]\!]$, then $\mathtt{zipP}_{\langle \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!]\rangle}\ x \sim \mathtt{zipP}_{\langle \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!]\rangle}\ y$.*
3. *If $\tau = \mathtt{Int} \times \mathcal{F}[\![[:\tau':]]\!]$, then $\mathtt{attachP}_{\langle \mathcal{A}[\![\tau']\!]\rangle}\ x \sim \mathtt{attachP}_{\langle \mathcal{A}[\![\tau']\!]\rangle}\ y$.*

Proof. Immediate from the definitions of these primitives given in Appendix C. □

We are now in the position to formulate the insight which led to the introduction of data-erasing equivalence: the results of applying a lifted function to two equivalent terms, are, again, equivalent under data-erasure. As before, the actual formalisation is based on substitution rather than application.

6.25 LEMMA
Let e be a $\lambda^{\mathcal{C}}$ term such that $\bullet : \upsilon \vdash e : \tau$. For all $x : \mathcal{F}[\![[:\upsilon:]]\!]$ and $y : \mathcal{F}[\![[:\upsilon:]]\!]$, if $x \sim_{\mathcal{F}[\![[:\upsilon:]]\!]} y$, then $\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\, \bullet, e]\!][x/\bullet] \sim_{\mathcal{F}[\![[:\tau:]]\!]} \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\, \bullet, e]\!][y/\bullet]$.

PROOF. By complete structural induction on e. The complete proof is given in Appendix A.3. □

Since we are ultimately interested in distributivity over concatenation, the relationship between this operation and data-erasing equivalence must be investigated. We begin this discussion by identifying the class of terms which do not evaluate to functions and, thus, can be concatenated.

6.26 DEFINITION (CONCATENABILITY)
A type τ in $\lambda^{\mathcal{C}}$ is *concatenable* if, for every type υ such that $[:\upsilon:]$ occurs in τ, υ does not contain function types. A type in $\lambda^{\mathcal{A}}$ is *concatenable* if it contains no function types. We say that two terms are concatenable if they are of the same concatenable type.

We have already described the impact of concatenation on definedness. If we restrict ourselves to concatenable terms, which cannot evaluate to arrays of functions, then definedness is precisely what is captured by data-erasing equivalence. Thus, we can formulate the following crucial principle.

6.27 LEMMA
Let x and y be concatenable terms of type $\mathcal{F}[\![[:\tau:]]\!]$ such that $x \sim y$. Then, $x \sim x +\!\!+_{\langle \mathcal{A}[\![\tau]\!]\rangle} y$.

PROOF. By induction on τ. □

For concatenable terms, there also exists an obvious relationship between data-erasing equivalence and operator $\rhd$, since the latter ensures that two terms diverge in exactly the same positions.

6.28 LEMMA
Let x and y be concatenable terms of type $\mathcal{F}[\![[:\tau:]]\!]$. Then, $y \rhd x \sim x \rhd y$.

PROOF. By induction on τ. □

6.29 LEMMA
Let e be a $\lambda^{\mathcal{C}}$ term such that $\bullet : \upsilon \vdash e : \tau$. For all concatenable x and y of type $\mathcal{F}[\![[:\upsilon:]]\!]$ such that $x \sim y$,

$$\begin{aligned}&\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\, \bullet, e]\!][x/\bullet] \;+\!\!+_{\langle \mathcal{A}[\![\tau]\!]\rangle}\; \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\, \bullet, e]\!][y/\bullet]\\ &\quad\cong\; \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\, \bullet, e]\!][x +\!\!+_{\langle \mathcal{A}[\![\upsilon]\!]\rangle} y/\bullet]\end{aligned}$$

PROOF. By complete structural induction on e. The complete proof is given in Appendix A.4. □

6.8 Correctness of vectorisation

Based on the properties of lifting formalised in the previous section, we are now in the position to show the operational correctness of vectorisation and, thus, the correctness of the flattening transformation. For this, we need to find a suitable definition of correctness first. A standard approach in such cases is to show that the transformation is equivalence-preserving, such that $e_1 \cong e_2$ in $\lambda^{\mathcal{C}}$ implies $\mathcal{V}[\![e_1]\!] \cong \mathcal{V}[\![e_2]\!]$ in $\lambda^{\mathcal{A}}$. Unfortunately, this implication does not hold in general. This is because semantically equivalent terms can be used interchangeably in all contexts, but in $\lambda^{\mathcal{A}}$, a context can violate the crucial assumption underlying vectorisation — the invariant that functions are always tupled with their lifted versions.

6.30 Example
Consider the two $\lambda^{\mathcal{C}}$ functions:

$$\begin{aligned} f &= \lambda\bullet.\, \texttt{mapP}\, \langle \bullet,\, \texttt{repP}\, \langle 1, 2\rangle\rangle \\ g &= \lambda\bullet.\, \texttt{repP}\, \langle 1,\, \bullet \dagger 2\rangle \end{aligned}$$

Both take a closure as the argument; f maps over the array [:2:], while g first applies it to 2 and then constructs the array. These two operations are equivalent in $\lambda^{\mathcal{C}}$ and, hence, $f \cong g$.

The vectorised versions of these functions are, however, not equivalent in $\lambda^{\mathcal{A}}$. For instance, if both are applied to the closure $\langle\!\langle \langle \lambda\bullet.\, 1, \lambda\bullet.\, \bullet.2\rangle, \overline{\tau}, \langle\rangle \rangle\!\rangle$, we have:

$$\begin{aligned} \mathcal{V}[\![f]\!].1\, \langle\!\langle \langle \lambda\bullet.\, 1, \lambda\bullet.\, \bullet.2\rangle, \overline{\tau}, \langle\rangle \rangle\!\rangle &\cong (\lambda\bullet.\, \bullet.2)\, (\texttt{zipP}\, \langle \texttt{repP}\, \langle 1, \langle\rangle\rangle, \texttt{repP}\, \langle 1, 2\rangle\rangle) \cong \texttt{repP}\, \langle 1, 2\rangle \\ \mathcal{V}[\![g]\!].1\, \langle\!\langle \langle \lambda\bullet.\, 1, \lambda\bullet.\, \bullet.2\rangle, \overline{\tau}, \langle\rangle \rangle\!\rangle &\cong \texttt{repP}\, \langle 1,\, (\lambda\bullet.\, 1)\, \langle\langle\rangle, 2\rangle\rangle \cong \texttt{repP}\, \langle 1, 1\rangle \end{aligned}$$

The results of the two computations are different since due to the definition of `mapP`, the former is ultimately obtained by applying $\lambda\bullet.\, \bullet.2$ to the array created by replication, whereas the latter is computed by first applying $\lambda\bullet.\, 1$ and then replicating the result. We will see below that these operations would be equivalent if $\lambda\bullet.\, \bullet.2$ were the lifted version of $\lambda\bullet.\, 1$. This, however, is obviously not the case. Therefore, $\mathcal{V}[\![f]\!] \ncong \mathcal{V}[\![g]\!]$; in particular, based on the above observation it is trivial to construct a context C such that $C[\mathcal{V}[\![f]\!]]$ converges while $C[\mathcal{V}[\![g]\!]]$ diverges.

The above example demonstrates that observational equivalence is too strong a requirement for our purposes. Moreover, as mentioned before, we do not intend to show that flattening preserves nontermination, which renders observational equivalence even less usable. Note, however, that the closure used to distinguish the two functions in Example 6.30 cannot be obtained by flattening. Our intention, however, is to apply flattening to entire programs, i.e., we do not consider the interactions between flattening and code written directly in $\lambda^{\mathcal{A}}$ (`?????`). Thus, we should only take into account those terms and, by extension, contexts which have been generated by the flattening transformation.

6.31 Definition (Flattening of contexts)
The transformations $\mathcal{V}[\![\cdot]\!]$ and $\mathcal{L}[\![\cdot, \cdot]\!]$ are naturally extended to contexts in $\lambda^{\mathcal{C}}$ by adding the following rules to the definitions in Figure 5.8:

$$\begin{aligned} \mathcal{V}[\![-]\!] &= - \\ \mathcal{L}[\![l, -]\!] &= \texttt{repP}\, \langle l, -\rangle \end{aligned}$$

Based on this definition, we specify a notion of *simulation* of programs in $\lambda^{\mathcal{C}}$ by those in $\lambda^{\mathcal{A}}$ which corresponds to our intentions much better than observational equivalence.

6.32 DEFINITION (SIMULATION)
A $\lambda^{\mathcal{C}}$ term e is simulated by a $\lambda^{\mathcal{A}}$ term e', written as $e \rightsquigarrow e'$, if for all $\lambda^{\mathcal{C}}$ contexts C, $C[e]\Downarrow \implies \mathcal{V}[\![C]\!][e']\Downarrow$.

Simulation is similar to observational equivalence in that it relates terms depending on their behaviour in all contexts. But it is also different, relating terms in different languages, only considering $\lambda^{\mathcal{A}}$ contexts obtainable by vectorisation and allowing the $\lambda^{\mathcal{A}}$ term to be more defined than its $\lambda^{\mathcal{C}}$ counterpart. This corresponds precisely to the notion of correctness which we informally described earlier: we will consider vectorisation to be correct if every $\lambda^{\mathcal{C}}$ term is simulated by its vectorised version.

To obtain any meaningful correctness results, we must establish a formal link between the operational semantics of $\lambda^{\mathcal{C}}$ and $\lambda^{\mathcal{A}}$. The semantics of the two languages is based on reduction which, therefore, will be our starting point. Ideally, we would like to show that if e and e' are terms in $\lambda^{\mathcal{C}}$ such that $e \longrightarrow e'$, then $\mathcal{V}[\![e]\!] \xrightarrow{*} \mathcal{V}[\![e']\!]$. Unfortunately, this is not the case in general.

This is due to the fact that while mapping and similar operations rely on the inductive definition of parallel arrays in $\lambda^{\mathcal{C}}$, they are performed in an entirely different manner in $\lambda^{\mathcal{A}}$. For instance, according to the reduction rules induced by the implementation of `mapP` in $\lambda^{\mathcal{C}}$ we have

$$\texttt{mapP}\,\langle c,\, \texttt{repP}\,\langle n, x\rangle\rangle \;\longrightarrow\; \texttt{repP}\,\langle n,\; c \dagger x\rangle$$

Let us consider how the vectorised versions of the two terms are evaluated. The second term remains essentially unchanged and is computed by applying the closure to x and replicating the result. The first term, however, encodes an entirely different evaluation strategy in $\lambda^{\mathcal{A}}$. Here, the closure is replicated and then applied elementwise to the argument array, eventually leading to a single application of the lifted closure function.

The reductions performed in the two cases are quite different, so much so that the desired direct correspondence cannot be established. However, while there exists no one-to-one or one-to-many correspondence between the individual reductions, the two computations are equivalent as investigated in the previous section. This suggests that the desired relationship between reduction in $\lambda^{\mathcal{C}}$ and the semantics of $\lambda^{\mathcal{A}}$ can be expressed in terms of semantic equivalence. The following lemma is a direct consequence of this insight.

6.33 LEMMA
Let e and e' be terms in $\lambda^{\mathcal{C}}$ such that $e \xrightarrow{*} e'$. Then, $\mathcal{V}[\![e]\!] \cong \mathcal{V}[\![e']\!]$.

PROOF. It is sufficient to show that if $e \longrightarrow e'$, then $\mathcal{V}[\![e]\!] \cong \mathcal{V}[\![e']\!]$, as the conclusion then follows by transitivity of semantic equivalence. The proof is by induction on the derivation $e \longrightarrow e'$.

Case $\mathcal{D} = (\lambda\bullet : \tau.\ e_1)\ e_2 \longrightarrow e_1[e_2/\bullet]$

$$\begin{aligned}
&\mathcal{V}[\![(\lambda\bullet : \tau.\ e_1)\ e_2]\!] \\
&= (\lambda\bullet : \mathcal{F}[\![\tau]\!].\ \mathcal{V}[\![e_1]\!])\ \mathcal{V}[\![e_2]\!] && \text{by definition of } \mathcal{V}[\![\cdot]\!] \\
&\cong \mathcal{V}[\![e_1]\!][\mathcal{V}[\![e_2]\!]/\bullet] && \text{by semantics of } \lambda^{\mathcal{A}} \\
&\cong \mathcal{V}[\![e_1[e_2/\bullet]]\!] && \text{by Lemma 6.9}
\end{aligned}$$

Case $\mathcal{D} = \mu\bullet : \tau.\ e \longrightarrow e[\mu\bullet : \tau.\ e/\bullet]$

$$
\begin{array}{lr}
\quad \mathcal{V}[\![\mu\bullet : \tau.\ e]\!] & \\
= \mu\bullet : \mathcal{F}[\![\tau]\!].\ \mathcal{V}[\![e]\!] & \text{by definition of } \mathcal{V}[\![\cdot]\!] \\
\cong \mathcal{V}[\![e]\!][\mu\bullet : \mathcal{F}[\![\tau]\!].\ \mathcal{V}[\![e]\!]/\bullet] & \text{by semantics of } \lambda^{\mathcal{A}} \\
= \mathcal{V}[\![e]\!][\mathcal{V}[\![\mu\bullet : \tau.\ e]\!]/\bullet] & \text{by definition of } \mathcal{V}[\![\cdot]\!] \\
\cong \mathcal{V}[\![e[\mu\bullet : \tau.\ e/\bullet]]\!] & \text{by Lemma 6.9}
\end{array}
$$

For the other core language derivations, the conclusion follows immediately from the induction hypothesis on the premise and corresponding properties of semantic equivalence. For primitives, the conclusion follows from their implementations and the properties of lifting established in the previous section. As before, we will not provide proofs for the primitives. □

Note that Lemma 6.33 does *not* contradict to Example 6.30, which assumes that e and e' are semantically equivalent and relies on the extensionality of this relation. Extensionality implies that we can change terms under lambdas; this, however, is never done by call-by-name reduction. Consequently, no such examples can be constructed if, as in Lemma 6.33, e and e' are connected by a sequence of reductions.

From the above, it is already obvious that flattening is correct for programs which produce scalar values. Assume, for instance, that e : `Int` is a converging $\lambda^{\mathcal{C}}$ term. Then, e evaluates to some integer literal i and by Lemma 6.33, $\mathcal{V}[\![e]\!] \cong \mathcal{V}[\![i]\!]$. Since $\mathcal{V}[\![i]\!] = i$, we have $\mathcal{V}[\![e]\!] \cong i$, i.e., the vectorised version of e yields the desired result.

Since we are mainly interested in parallel programs, however, ruling out programs which produce parallel arrays is not practical. Unfortunately, the above reasoning is not directly applicable to array values, as these are represented differently before and after flattening. In contrast to integers, vectorising a $\lambda^{\mathcal{C}}$ array value does not yield a value in $\lambda^{\mathcal{A}}$, since array constructors are mapped to applications of corresponding primitives which can be evaluated further. Fortunately, it is easy to show that such computations always converge.

6.34 Lemma
Let v be a value in $\lambda^{\mathcal{C}}$. Then, $\mathcal{V}[\![v]\!]\Downarrow$.

Proof. By straightforward induction on v. □

With this in mind, it is straightforward to prove that vectorisation never introduces non-termination.

6.35 Lemma (Preservation of termination)
Let e be a converging term in $\lambda^{\mathcal{C}}$. Then, $\mathcal{V}[\![e]\!]\Downarrow$.

Proof. Since e converges, there must exists a value v such that $e \xrightarrow{*} v$. By Lemma 6.33, $\mathcal{V}[\![e]\!] \cong \mathcal{V}[\![v]\!]$. By Lemma 6.34, $\mathcal{V}[\![v]\!]$ converges and by definition of observational equivalence, so does $\mathcal{V}[\![e]\!]$. □

The above lemma is central to establishing the correctness of vectorisation. The following observation enables us to use it in the proof.

6.36 Lemma
If C is a context and e a closed term in $\lambda^{\mathcal{C}}$, then $\mathcal{V}[\![C]\!][\mathcal{V}[\![e]\!]] \cong \mathcal{V}[\![C[e]]\!]$.

PROOF. By straightforward induction on C. □

Note that this is very similar to the substitutivity of vectorisation as formalised by Lemma 6.9. Here, however, we can only show semantic equivalence instead of syntactic equality because in lifted contexts, holes are replicated (cf. Definition 6.31), but the replication algorithm has been folded into the definition of flattening for terms in some cases.

From Lemmas 6.35 and 6.36, the correctness of vectorisation is easily established.

6.37 THEOREM (CORRECTNESS OF VECTORISATION)
Let e be a term in $\lambda^{\mathcal{C}}$. Then, $e \rightsquigarrow \mathcal{V}[\![e]\!]$.

PROOF. We must show that for every $\lambda^{\mathcal{C}}$ context C, $\mathcal{V}[\![C]\!][\mathcal{V}[\![e]\!]]$ converges if $C[e]$ does. Assume that $C[e]$ converges. Then, $\mathcal{V}[\![C]\!][\mathcal{V}[\![e]\!]] \cong \mathcal{V}[\![C[e]]\!]$ by Lemma 6.36 and $\mathcal{V}[\![C[e]]\!]$ converges by Lemma 6.35. □

Chapter 7

Conclusion

The combination of nested data parallelism and functional programming provides a highly expressive framework for writing parallel applications. It allows the programmer to focus on the overall structure of the algorithm and *specify* its parallel behaviour rather than having to implement it by hand. The examples in Chapter 2 demonstrate the conciseness and clarity of the resulting code.

This expressiveness comes at a cost, however, in the form of a significantly increased complexity of the compiler. Indeed, the nested data-parallel model would not be a viable approach if the high-level specifications could not be automatically translated into code which runs efficiently on modern massively parallel systems. The flattening transformation has long been recognised as a sensible and well-founded solution to this problem. But it was also an incomplete solution, since it did not cover higher-order functions which, arguably, are the defining feature of the functional paradigm. This had a highly negative impact on the utility and acceptance of flattening and, by extension, of nested data parallelism in general. First, this deficiency severely limited the expressiveness of the approach, thereby negating its main advantage to a certain extent. Secondly, and perhaps more importantly, it precluded a seamless integration of nested data parallelism into a complete and wide-spread functional language, thus making the technique unusable for real-world programming.

In Chapter 4, we have seen that the reasons for this shortcoming are deeply rooted in the interactions between flattening and the standard lambda calculus which serves as the foundation of functional programming languages. The former must be able to manipulate computations and data independently; the latter interweaves the two such that they cannot be easily separated. Identifying and resolving this conflict has been the main challenge in the research presented in this dissertation.

By combining flattening with closure conversion, we have been able to provide a clean and lightweight solution to this problem. Closure conversion achieves the required separation of computation and data by making partially applied functions explicit entities in the calculus which allows already bound arguments to be accessed and modified. A closure represents a single partial application; but this principle is easily extended to arrays of such functions. By suitably extending the flattening transformation, closures can be utilised to support arbitrary uses of higher-order functions in nested data-parallel programs. We provided a detailed discussion of the implications and the parallel behaviour of this compilation strategy and validated it by considering the semantics and implementations of fundamental parallel operations. An interesting aspect of this development is that closure conversion allows us to *derive*

many of the implementations by adopting a well-founded approach based on parametricity.

A program transformation, especially one as complex and intrusive as flattening, cannot be considered a viable compilation technique without a correctness proof. Consequently, after the detailed but informal discussion of higher-order nested data parallelism, we embarked on a programme to show that flattening is, indeed, correct with respect to both static and dynamic semantics. The proof of type correctness, together with the formal specifications of the flattening transformation and its source and target languages, is the main result of Chapter 5.

Not surprisingly, the operational correctness has been much harder to establish, particularly so since, motivated by the desire to integrate nested data parallelism into Haskell, we have adopted a call-by-name evaluation strategy for our intermediate languages. The data structures generated by flattening necessarily rely on unboxed arrays since the latter are easily parallelisable, in contrast to structures based on pointers and thunks. But unboxed arrays are strict and their use in a lazy setting has highly non-obvious implications on the operational behaviour of programs. A detailed analysis of the interactions between flattening in laziness has been the main topic of the first part of Chapter 6. In addition to providing a solid foundation for the subsequent formal development, this has also clarified the requirements on the implementation of nested data parallelism and the supporting library in lazy languages. We consider this to be one of the key contributions of this work, as this aspect has not been investigated before.

In the second part of Chapter 6, we have demonstrated that the flattening transformation preserves the semantics of terminating programs. The proof of this property crucially relies on the immutability of lambda abstractions which is syntactically enforced by the underlying calculi. This has justified the slightly unsual choice of our intermediate languages and highlighted the importance of closure conversion for higher-order nested data parallelism.

7.1 Related work

We have already covered specifically relevant related work in the respective chapters. Still, the embedding of support for parallel programming into functional languages is a very active research topic and it is, therefore, important to provide a context for the main results of this dissertation by reviewing similar approaches.

Trinder et al. (2002) provide a comprehensive overview of various approaches to extending Haskell with facilities for parallel programming. Of these, *Data Parallel Haskell* (Hill, 1994) is probably closest to the nested data parallel model as presented in this work. Data Parallel Haskell also relies on a special form of parallel collections (called PODs) and collective operations for expressing parallelism, and on a vectorising transformation for compiling it. However, in contrast to parallel arrays, PODs are *lazy*, both in their elements and in their shape. This brings their semantics closer to that of lists but leads to entirely different trade-offs and runtime behaviour. In particular, while PODs are more flexible, operations on them are harder to optimise as less static information is available to the compiler.

Data Field Haskell (Holmerin and Lisper, 2000) introduces a more general model of data parallelism based on the theory of data fields (Lisper and Hammarlund, 2001). Here, parallel collections are seen as partial functions from index domains to value domains. By selecting suitable index domains, a wide range of complex data structures can be specified. However, the efficiency of this approach remains an open question, as no parallel implementation has

been provided so far.

A large number of implicitly and explicitly control-parallel extensions to Haskell have been proposed. An example of the former is Parallel Haskell (Aditya et al., 1995) which allows very fine-grained parallelism to be utilised but changes the semantics of the language by requiring *lenient* evaluation and, ultimately, compromising referential transparency. Herrmann and Lengauer (2000) describe HDC, a strict subset of Haskell in which parallelism is automatically extracted from a program and compiled based on a library of predefined skeletons. Glasgow Parallel Haskell (Trinder et al., 1996; Trindler et al., 1998) is an explicitly parallel approach which allows the programmer to designate those expressions that may be evaluated in parallel and provides *evaluation strategies* as a means of specifying the required degree of evaluation. Eden (Loogen et al., 2005) uses stream processors and process abstractions to separate coordination from computation; unfortunately, it, too, breaks referential transparency. Although nested data parallelism can be seen as explicit in the sense that it requires the programmer to employ specialised data structures and operations, it is closer to the implicit approaches from the implementor's point of view, as the programs have to undergo extensive transformations. Importantly, the integration of NDP does not necessitate any changes to Haskell's semantics and has no impact on programs which do not use it.

Of course, functional approaches to parallel programming are not restricted to Haskell extensions. The supported forms of parallelism range from purely regular computations in Sisal (Feo et al., 1990) and SAC (Scholz, 1998) to arbitrary control-parallel structures, as in Concurrent Clean (Nocker et al., 1991) and Concurrent ML (Reppy, 1991). Hammond and Michaelson (1999) provide a comprehensive overview of several such approaches.

In the context of this dissertation, the language FISh (Jay and Steckler, 1998). Although not immediately parallel, it uses shape analysis (Jay and Sekanina, 1996; Jay, 1995a) to derive an efficient, unboxed representation for nested, polymorphic arrays. This approach is quite similar to the flattening transformation and, indeed, we have discussed the connection to shape theory in Section 6.1.1.

7.2 Future work

The research presented in this dissertation provides a foundation for a number of directions for future work. This section describes some of the most interesting ones.

7.2.1 An implementation

A final evaluation of the mechanisms proposed in this work is, of course, not possible until they are implemented in a compiler and applied to parallel applications of meaningful size. Our main focus has been on Haskell and we envisage the Glasgow Haskell Compiler to be the primary vehicle for integrating nested data parallelism into the language. Such an integration, however, is far from being straightforward. First and foremost, GHC does not perform closure conversion, which must be included in the compilation process before attempting to implement flattening. We expect this to have far-reaching consequences on the structure of the compiler, especially in the presence of separate compilation, but believe these consequences to be, for the most part, beneficial with respect to both the performance of the generated code and the architecture of GHC's backend and runtime system.

Another problem which, although orthogonal to the ones investigated in this dissertation, must be resolved for an implementation of NDP to be viable is the handling of polymorphism.

Chakravarty et al. (2005b,a) propose a sound and clean solution based on an extension of type classes with a mechanism for defining functions over types. Unfortunately, this approach has not yet been implemented in GHC or any other Haskell compiler. We expect this to change in the near future, however.

Finally, the code generated by the flattening transformation must undergo extensive optimisations if it is to exhibit competitive performance on distributed-memory machines. Such optimisations have been investigated in detail by (Keller, 1999) but these results have been obtained for a strict, first-order language. We believe that they are directly applicable to the higher-order techniques developed in this work, but this, again, cannot be concluded without a working implementation.

7.2.2 Cost model

The implementation of nested data parallelism provided by NESL is accompanied by a language-based cost model (Blelloch and Greiner, 1996). Such a model is highly desirable as it can be used to assess and compare different formulations of a parallel algorithm and to identify cases in which flattening increases the parallel step count of a program as compared to its source-level metrics (Riely and Prins, 2000). Moreover, a cost model can also guide the compiler in selecting and applying optimisations. In earlier work, we have investigated the complexity of flattened data-parallel *fold*-programs (Lechtchinsky et al., 2002). The results suggest that a standard set of nested data parallel skeletons can help the programmer in avoiding performance traps and allow programs to be costed automatically. In fact, the amenability of skeletons to well-founded cost models has long been recognised (Rangaswami, 1996; Hamdan, 1999; Hayashi, 2001). We expect that the general form of flattening developed in this dissertation will make these techniques applicable to nested data-parallel programs, ultimately facilitating a concise and uniform specification of a cost model.

7.2.3 Skeletons

We have already mentioned the relationship between nested data parallelism and skeleton-based parallel programming. The latter has received a lot of attention (Cole, 1999; Rabhi and Gorlatch, 2002) and, indeed, the idea of implementing parallel applications by instantiating and assembling predefined computation patterns with well-defined semantics and complexity is highly attractive. Many important skeletons are data-parallel — the prime example is, of course, *map*. Moreover, it is desirable for skeletons to be arbitrarily composable and nestable (Darlington et al., 1995).

Higher-order nested data parallelism can be seen as a well-founded and efficient platform for implementing a wide range of skeletons. This would amount to introducing another layer of abstraction on top of the bulk operations on parallel arrays. For instance, the implementation of Quicksort in Section 2.4 can easily be refactored into a generic divide-and-conquer skeleton. Similarily, Wang's algorithm discussed in Section 2.5 uses an instance of a pipelining skeleton, which is easily realised by combining parallel and sequential data structures. Thus, while our extension to Haskell already provides the programmer with a high-level framework for developing parallel applications, its expressiveness can be increased even further by furnishing it with a library of standard skeletons. Crucially, such a library does not have to be realised in a lower-level language since higher-order nested data parallelism already provides the tools necessary for its implementation.

Appendix A

Long proofs

A.1 Proof of Theorem 5.27 (Type correctness of flattening)

The two parts of the theorem are proved simultaneously by induction on the derivation $\Gamma \vdash_{\mathcal{C}} e : \sigma$.

Case $\mathcal{D} = \Gamma \vdash_{\mathcal{C}} i : \mathtt{Int}$

Vectorisation

$\mathcal{G}[\![\Gamma]\!] \vdash i : \mathtt{Int}$ by typing rules
$\mathcal{G}[\![\Gamma]\!] \vdash i : \underline{\mathtt{Int}}.1$ by rule (type conversion)

Lifting

$\mathcal{G}[\![\Gamma^{\uparrow}]\!] \vdash i : \mathtt{Int}$ by typing rules
$\mathcal{G}[\![\Gamma^{\uparrow}]\!] \vdash l : \mathtt{Int}$ assumption
$\mathcal{G}[\![\Gamma^{\uparrow}]\!] \vdash \langle l, i\rangle : \mathtt{Int} \times \mathtt{Int}$ by typing rules
$\mathcal{G}[\![\Gamma^{\uparrow}]\!] \vdash \langle l, i\rangle : (\underline{\mathtt{Int}} \times \underline{\mathtt{Int}}).1$ by rule (type conversion)
$\mathcal{G}[\![\Gamma^{\uparrow}]\!] \vdash \mathtt{repP}_{\langle \underline{\mathtt{Int}} \rangle}\ \langle i, l\rangle : \underline{\mathtt{Int}}.2$ by type of `repP`

Case $\mathcal{D} = \Gamma \vdash_{\mathcal{C}} b : \mathtt{Bool}$ Analogous to the previous case.

Case $\mathcal{D} = \Gamma \vdash_{\mathcal{C}} \langle\rangle : \langle\rangle$ Analogous to the previous case.

Case $\mathcal{D} = \dfrac{\vdash_{\mathcal{C}} \tau_1 \qquad \ldots \qquad \vdash_{\mathcal{C}} \tau_n}{\Gamma \vdash_{\mathcal{C}} c_{\langle \tau_1, \ldots, \tau_n\rangle} : \mathcal{T}_{\mathcal{C}}(c_{\langle \tau_1, \ldots, \tau_n\rangle})}$

Vectorisation

$\mathcal{P}_1$: $\mathcal{T}_{\mathcal{C}}(c_{\langle \tau_1, \ldots, \tau_n\rangle}) = \upsilon_1 \to \upsilon_2$ for some υ_1, υ_2 by Definition 5.2
$\mathcal{T}_{\mathcal{A}}(\mathcal{V}_C[\![c]\!]_{\langle \mathcal{A}[\![\tau_1]\!], \ldots, \mathcal{A}[\![\tau_n]\!]\rangle}) = \mathcal{A}[\![\upsilon_1]\!].1 \to \mathcal{A}[\![\upsilon_2]\!].1$ by Property 5.26
$\mathcal{P}_2$: $\mathcal{A}[\![\tau_i]\!] : \star \times \star$ for all $1 \le i \le n$ by Lemma 5.20
$\mathcal{P}_3$: $\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}_C[\![c]\!]_{\langle \mathcal{A}[\![\tau_1]\!], \ldots, \mathcal{A}[\![\tau_n]\!]\rangle} : \mathcal{A}[\![\upsilon_1]\!].1 \to \mathcal{A}[\![\upsilon_2]\!].1$ by typing rules

with $\mathcal{P}_1$
$\mathcal{T}_{\mathcal{A}}(\mathcal{L}_C[\![c]\!]_{\langle \mathcal{A}[\![\tau_1]\!],\ldots,\mathcal{A}[\![\tau_n]\!]\rangle}) \;=\; \mathcal{A}[\![\upsilon_1]\!].2 \to \mathcal{A}[\![\upsilon_2]\!].2$ by Property 5.26
with $\mathcal{P}_2$
$\mathcal{G}[\![\Gamma]\!] \vdash_{\mathcal{A}} \mathcal{L}_C[\![c]\!]_{\langle \mathcal{A}[\![\tau_1]\!],\ldots,\mathcal{A}[\![\tau_n]\!]\rangle} \;:\; \mathcal{A}[\![\upsilon_1]\!].2 \to \mathcal{A}[\![\upsilon_2]\!].2$ by typing rules
with $\mathcal{P}_3$
$\mathcal{G}[\![\Gamma]\!] \vdash \langle \mathcal{V}_C[\![c]\!]_{\langle \mathcal{A}[\![\tau_1]\!],\ldots,\mathcal{A}[\![\tau_n]\!]\rangle}, \mathcal{L}_C[\![c]\!]_{\langle \mathcal{A}[\![\tau_1]\!],\ldots,\mathcal{A}[\![\tau_n]\!]\rangle}\rangle$
$\quad : (\mathcal{A}[\![\upsilon_1]\!].1 \to \mathcal{A}[\![\upsilon_2]\!].1) \times (\mathcal{A}[\![\upsilon_1]\!].2 \to \mathcal{A}[\![\upsilon_2]\!].2)$
by typing rules
$\mathcal{G}[\![\Gamma]\!] \vdash \langle \mathcal{V}_C[\![c]\!]_{\langle \mathcal{A}[\![\tau_1]\!],\ldots,\mathcal{A}[\![\tau_n]\!]\rangle}, \mathcal{L}_C[\![c]\!]_{\langle \mathcal{A}[\![\tau_1]\!],\ldots,\mathcal{A}[\![\tau_n]\!]\rangle}\rangle \;:\; (\mathcal{A}[\![\upsilon_1]\!] \underline{\to} \mathcal{A}[\![\upsilon_2]\!]).1$
by rule (type conversion)
$\mathcal{P}_4$: $\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![c_{\langle \tau_1,\ldots,\tau_n\rangle}]\!] \;:\; (\mathcal{A}[\![\upsilon_1]\!] \underline{\to} \mathcal{A}[\![\upsilon_2]\!]).1$ by definition of $\mathcal{V}[\![\cdot]\!]$

Lifting

from $\mathcal{P}_4$
$\mathcal{G}[\![\Gamma^{\uparrow}]\!] \vdash \mathcal{V}[\![c_{\langle \tau_1,\ldots,\tau_n\rangle}]\!] \;:\; (\mathcal{A}[\![\upsilon_1]\!] \underline{\to} \mathcal{A}[\![\upsilon_2]\!]).1$ by conversion of type contexts
$\mathcal{G}[\![\Gamma^{\uparrow}]\!] \vdash l : \texttt{Int}$ assumption
$\mathcal{G}[\![\Gamma^{\uparrow}]\!] \vdash \langle l, \mathcal{V}[\![c_{\langle \tau_1,\ldots,\tau_n\rangle}]\!]\rangle \;:\; \texttt{Int} \times (\mathcal{A}[\![\upsilon_1]\!] \underline{\to} \mathcal{A}[\![\upsilon_2]\!]).1$ by typing rules
$\mathcal{G}[\![\Gamma^{\uparrow}]\!] \vdash \texttt{repP}_{\langle \mathcal{A}[\![\upsilon_1]\!] \underline{\to} \mathcal{A}[\![\upsilon_2]\!]\rangle}\, \langle l, \mathcal{V}[\![c_{\langle \tau_1,\ldots,\tau_n\rangle}]\!]\rangle \;:\; (\mathcal{A}[\![\upsilon_1]\!] \underline{\to} \mathcal{A}[\![\upsilon_2]\!]).2$ by type of `repP`

Case $\mathcal{D} = \bullet : \tau \vdash_{\mathcal{C}} \bullet : \tau$

Vectorisation

$\bullet : \mathcal{A}[\![\tau]\!].1 \vdash \bullet : \mathcal{A}[\![\tau]\!].1$ by typing rules
$\mathcal{G}[\![\bullet : \tau]\!] \vdash \bullet : \mathcal{A}[\![\tau]\!].1$ by definition of $\mathcal{G}[\![\cdot]\!]$

Lifting

$\bullet : \mathcal{A}[\![\tau]\!].2 \vdash \bullet : \mathcal{A}[\![\tau]\!].2$ by typing rules
$\mathcal{G}[\![\bullet : [\!:\tau:\!]]\!] \vdash \bullet : \mathcal{A}[\![\tau]\!].2$ by definition of $\mathcal{G}[\![\cdot]\!]$

$$\textbf{Case } \mathcal{D} \;=\; \frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \Gamma \vdash_{\mathcal{C}} e_1 : \tau_1 \to \tau_2 & \Gamma \vdash_{\mathcal{C}} e_2 : \tau_1\end{array}}{\Gamma \vdash_{\mathcal{C}} e_1\; e_2 : \tau_2}$$

Vectorisation

$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e_1]\!] : (\mathcal{A}[\![\tau_1]\!] \underline{\to} \mathcal{A}[\![\tau_2]\!]).1$ by ind. hyp. (1) on $\mathcal{D}_1$
$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e_1]\!] \;:\; (\mathcal{A}[\![\tau_1]\!].1 \to \mathcal{A}[\![\tau_2]\!].1) \times (\mathcal{A}[\![\tau_1]\!].2 \to \mathcal{A}[\![\tau_2]\!].2)$ by rule (type conversion)
$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e_1]\!].1 \;:\; \mathcal{A}[\![\tau_1]\!].1 \to \mathcal{A}[\![\tau_2]\!].1$ by typing rules
$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e_2]\!] \;:\; \mathcal{A}[\![\tau_1]\!].1$ by ind. hyp. (1) on $\mathcal{D}_2$
$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e_1]\!].1\, \mathcal{V}[\![e_2]\!] \;:\; \mathcal{A}[\![\tau_2]\!].1$ by typing rules
$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e_1\; e_2]\!] \;:\; \mathcal{A}[\![\tau_2]\!].1$ by definition of $\mathcal{V}[\![\cdot]\!]$

Lifting

$\mathcal{G}[\![\Gamma^{\uparrow}]\!] \vdash l : \texttt{Int}$ assumption

$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_1]\!] : (\mathcal{A}[\![\tau_1]\!] \underline{\to} \mathcal{A}[\![\tau_2]\!]).2$ by ind. hyp. (2) on $\mathcal{D}_1$
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_1]\!] : \mathtt{Int} \times ((\mathcal{A}[\![\tau_1]\!].1 \to \mathcal{A}[\![\tau_2]\!].1) \times (\mathcal{A}[\![\tau_1]\!].2 \to \mathcal{A}[\![\tau_2]\!].2))$
by rule (type conversion)
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_1.2.2]\!] : \mathcal{A}[\![\tau_1]\!].2 \to \mathcal{A}[\![\tau_2]\!].2$ by typing rules
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_2]\!] : \mathcal{A}[\![\tau_1]\!].2$ by ind. hyp. (2) on $\mathcal{D}_2$
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_1.2.2]\!]\, \mathcal{L}[\![l, e_2]\!] : \mathcal{A}[\![\tau_2]\!].2$ by typing rules
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_1\ e_2]\!] : \mathcal{A}[\![\tau_2]\!].2$ by definition of $\mathcal{L}[\![\cdot,\cdot]\!]$

Case $\mathcal{D} = \dfrac{\tau_1 \vdash_c e : \tau_2 \qquad \vdash_c \tau_1}{\Gamma \vdash \lambda\bullet : \tau_1.\ e : \tau_1 \to \tau_2}$

Vectorisation

$\mathcal{P}_1$: $\vdash \mathcal{A}[\![\tau_1]\!] : \star \times \star$ by Lemma 5.20
$\mathcal{A}[\![\tau_1]\!].1 \vdash \mathcal{V}[\![e]\!] : \mathcal{A}[\![\tau_2]\!].1$ by ind. hyp. (1)
$\mathcal{P}_2$: $\mathcal{G}[\![\Gamma]\!] \vdash \lambda\bullet : \mathcal{A}[\![\tau_1]\!].1.\ \mathcal{V}[\![e]\!] : \mathcal{A}[\![\tau_1]\!].1 \to \mathcal{A}[\![\tau_2]\!].1$ by typing rules

$\mathcal{A}[\![\tau_1]\!].2 \vdash \mathtt{lengthP}_{\langle \mathcal{A}[\![\tau_1]\!] \rangle}\ \bullet : \mathtt{Int}$ by type of `lengthP`
$\mathcal{A}[\![\tau_1]\!].2 \vdash \mathcal{L}[\![\mathtt{lengthP}_{\langle \mathcal{A}[\![\tau_1]\!] \rangle}\ \bullet, e]\!] : \mathcal{A}[\![\tau_2]\!].2$ by ind. hyp. (2) with $\mathcal{P}_1$
$\mathcal{G}[\![\Gamma]\!] \vdash \lambda\bullet : \mathcal{A}[\![\tau_1]\!].2.\ \mathcal{L}[\![\mathtt{lengthP}_{\langle \mathcal{A}[\![\tau_1]\!] \rangle}\ \bullet, e]\!] : \mathcal{A}[\![\tau_1]\!].2 \to \mathcal{A}[\![\tau_2]\!].2$ by typing rules with $\mathcal{P}_2$
$\mathcal{P}_3$: $\mathcal{G}[\![\Gamma]\!] \vdash \langle \lambda\bullet : \mathcal{A}[\![\tau_1]\!].1.\ \mathcal{V}[\![e]\!],\ \lambda\bullet : \mathcal{A}[\![\tau_1]\!].2.\ \mathcal{L}[\![\mathtt{lengthP}_{\langle \mathcal{A}[\![\tau_1]\!] \rangle}\ \bullet, e]\!] \rangle$
$: (\mathcal{A}[\![\tau_1]\!].1 \to \mathcal{A}[\![\tau_2]\!].1) \times (\mathcal{A}[\![\tau_1]\!].2 \to \mathcal{A}[\![\tau_2]\!].2)$ by typing rules
$\mathcal{G}[\![\Gamma]\!] \vdash \langle \lambda\bullet : \mathcal{A}[\![\tau_1]\!].1.\ \mathcal{V}[\![e]\!],\ \lambda\bullet : \mathcal{A}[\![\tau_1]\!].2.\ \mathcal{L}[\![\mathtt{lengthP}_{\langle \mathcal{A}[\![\tau_1]\!] \rangle}\ \bullet, e]\!] \rangle$
$: (\mathcal{A}[\![\tau_1]\!] \underline{\to} \mathcal{A}[\![\tau_2]\!]).1$ by rule (type conversion)
$\mathcal{P}_4$: $\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![\lambda\bullet : \tau_1.\ e]\!] : \mathcal{A}[\![\tau_1 \to \tau_2]\!].1$ by definitions of $\mathcal{V}[\![\cdot]\!]$ and $\mathcal{A}[\![\cdot]\!]$

Lifting

from $\mathcal{P}_4$
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{V}[\![\lambda\bullet : \tau_1.\ e]\!] : \mathcal{A}[\![\tau_1 \to \tau_2]\!].1$ by conversion of type contexts
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash l : \mathtt{Int}$ assumption
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \langle l, \mathcal{V}[\![\lambda\bullet : \tau_1.\ e]\!] \rangle : \mathtt{Int} \times (\mathcal{A}[\![v_1]\!] \underline{\to} \mathcal{A}[\![v_2]\!]).1$ by typing rules
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathtt{repP}_{\langle \mathcal{A}[\![v_1]\!] \underline{\to} \mathcal{A}[\![v_2]\!] \rangle}\ \langle l, \mathcal{V}[\![\lambda\bullet : \tau_1.\ e]\!] \rangle : (\mathcal{A}[\![v_1]\!] \underline{\to} \mathcal{A}[\![v_2]\!]).2$ by type of `repP`

Case $\mathcal{D} = \dfrac{\tau \vdash_c e : \tau \qquad \vdash_c \tau}{\Gamma \vdash \mu\bullet : \tau.\ e : \tau}$

Vectorisation

$\mathcal{A}[\![\tau]\!] : \star \times \star$ by Lemma 5.20
$\bullet : \mathcal{A}[\![\tau]\!].1 \vdash \mathcal{V}[\![e]\!] : \mathcal{A}[\![\tau]\!].1$ by ind. hyp. (1)
$\mathcal{G}[\![\Gamma]\!] \vdash \mu\bullet : \mathcal{A}[\![\tau]\!].1.\ \mathcal{V}[\![e]\!] : \mathcal{A}[\![\tau]\!].1$ by typing rules
$\mathcal{P}_1$: $\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![\mu\bullet : \tau.\ e]\!] : \mathcal{A}[\![\tau]\!].1$ by definition of $\mathcal{V}[\![\cdot]\!]$

Lifting

from $\mathcal{P}_1$
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{V}[\![\mu\bullet : \tau.\ e]\!] : \mathcal{A}[\![\tau]\!].1$ by conversion of type contexts
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash l : \texttt{Int}$ assumption
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \langle l, \mathcal{V}[\![\mu\bullet : \tau.\ e]\!]\rangle\ :\ \texttt{Int} \times \mathcal{A}[\![\tau]\!].1$ by typing rules
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \texttt{repP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle l, \mathcal{V}[\![\mu\bullet : \tau.\ e]\!]\rangle\ :\ \mathcal{A}[\![\tau]\!].2$ by type of `repP`

$$\textbf{Case}\ \mathcal{D}\ =\ \frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \Gamma \vdash_c e_1 : \tau_1 & \Gamma \vdash_c e_2 : \tau_2\end{array}}{\Gamma \vdash_c \langle e_1, e_2\rangle : \tau_1 \times \tau_2}$$

Vectorisation

$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e_1]\!] : \mathcal{A}[\![\tau_1]\!].1$ by ind. hyp. (1) on $\mathcal{D}_1$
$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e_2]\!] : \mathcal{A}[\![\tau_2]\!].1$ by ind. hyp. (1) on $\mathcal{D}_2$
$\mathcal{G}[\![\Gamma]\!] \vdash \langle \mathcal{V}[\![e_1]\!], \mathcal{V}[\![e_2]\!]\rangle :\ \mathcal{A}[\![\tau_1]\!].1 \times \mathcal{A}[\![\tau_2]\!].1$ by typing rules
$\mathcal{G}[\![\Gamma]\!] \vdash \langle \mathcal{V}[\![e_1]\!], \mathcal{V}[\![e_2]\!]\rangle :\ (\mathcal{A}[\![\tau_1]\!] \mathbin{\underline{\times}} \mathcal{A}[\![\tau_2]\!]).1$ by type conversion

Lifting

$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_1]\!] : \mathcal{A}[\![\tau_1]\!].2$ by ind. hyp. (2) on $\mathcal{D}_1$
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_2]\!] : \mathcal{A}[\![\tau_2]\!].2$ by ind. hyp. (2) on $\mathcal{D}_2$
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \langle \mathcal{L}[\![l, e_1]\!], \mathcal{L}[\![l, e_2]\!]\rangle :\ \mathcal{A}[\![\tau_1]\!].2 \times \mathcal{A}[\![\tau_2]\!].2$ by typing rules
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \texttt{zipP}_{\langle \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle \mathcal{L}[\![l, e_1]\!], \mathcal{L}[\![l, e_2]\!]\rangle\ :\ (\mathcal{A}[\![\tau_1]\!] \mathbin{\underline{\times}} \mathcal{A}[\![\tau_2]\!]).2$ by type of `zipP`

$$\textbf{Case}\ \mathcal{D}\ =\ \frac{\Gamma \vdash_c e : \tau_1 \times \tau_2}{\Gamma \vdash_c e.i : \tau_i}$$

Vectorisation

$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e]\!] : (\mathcal{A}[\![\tau_1]\!] \mathbin{\underline{\times}} \mathcal{A}[\![\tau_2]\!]).1$ by ind. hyp. (1)
$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e]\!] : \mathcal{A}[\![\tau_1]\!].1 \times \mathcal{A}[\![\tau_2]\!].1$ by type conversion
$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e]\!].i : \mathcal{A}[\![\tau_i]\!].1$ by typing rules

Lifting

$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e]\!]\ : (\mathcal{A}[\![\tau_1]\!] \mathbin{\underline{\times}} \mathcal{A}[\![\tau_2]\!]).2$ by ind. hyp. (2)
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e]\!]\ : \texttt{Int} \times (\mathcal{A}[\![\tau_1]\!].2 \times \mathcal{A}[\![\tau_2]\!].2)$ by type conversion
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e]\!].2.i\ :\ \mathcal{A}[\![\tau_i]\!].2$ by typing rules
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash l : \texttt{Int}$ assumption
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \texttt{attachP}_{\langle \mathcal{A}[\![\tau_i]\!]\rangle}\ \langle l,\ \mathcal{L}[\![l, e]\!].2.i\rangle\ :\ \mathcal{A}[\![\tau_i]\!].2$ by type of `attachP`

$$\textbf{Case}\ \mathcal{D}\ =\ \frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \Gamma \vdash_c e_1 : \tau_1 \times \tau_2 \to \tau_3 & \Gamma \vdash_c e_2 : \tau_1\end{array}}{\Gamma \vdash_c \langle\!\langle e_1, \tau_1, \tau_2, \tau_3, e_2\rangle\!\rangle\ :\ \tau_2 \Rightarrow \tau_3}$$

Vectorisation

$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e_1]\!] \ : (\mathcal{A}[\![\tau_1]\!] \mathbin{\underline{\times}} \mathcal{A}[\![\tau_2]\!] \mathbin{\underline{\rightarrow}} \mathcal{A}[\![\tau_3]\!]).1$ by ind. hyp. (1) on $\mathcal{D}_1$
$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e_1]\!] \ : (\mathcal{A}[\![\tau_1]\!].1 \times \mathcal{A}[\![\tau_2]\!].1 \to \mathcal{A}[\![\tau_3]\!].1) \times \ ((\mathcal{A}[\![\tau_1]\!] \mathbin{\underline{\times}} \mathcal{A}[\![\tau_2]\!]).2 \to \mathcal{A}[\![\tau_3]\!].2)$ by type conversion
$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e_2]\!] \ : \ \mathcal{A}[\![\tau_1]\!].1$ by ind. hyp. (1) on $\mathcal{D}_2$
$\mathcal{G}[\![\Gamma]\!] \vdash \langle\!\langle \mathcal{V}[\![e_1]\!], \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \mathcal{V}[\![e_2]\!] \rangle\!\rangle \ : \mathcal{A}[\![\tau_2]\!].1 \Rightarrow \mathcal{A}[\![\tau_3]\!].1$ by typing rules
$\mathcal{G}[\![\Gamma]\!] \vdash \langle\!\langle \mathcal{V}[\![e_1]\!], \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \mathcal{V}[\![e_2]\!] \rangle\!\rangle \ : (\mathcal{A}[\![\tau_2]\!] \mathbin{\underline{\Rightarrow}} \mathcal{A}[\![\tau_3]\!]).1$ by type conversion

Lifting

$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_1]\!] \ : (\mathcal{A}[\![\tau_1]\!] \mathbin{\underline{\times}} \mathcal{A}[\![\tau_2]\!] \mathbin{\underline{\rightarrow}} \mathcal{A}[\![\tau_3]\!]).2$ by ind. hyp. (2) on $\mathcal{D}_1$
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_1]\!] \ : \mathtt{Int} \times ((\mathcal{A}[\![\tau_1]\!].1 \times \mathcal{A}[\![\tau_2]\!].1 \to \mathcal{A}[\![\tau_3]\!].1) \times \ ((\mathcal{A}[\![\tau_1]\!] \mathbin{\underline{\times}} \mathcal{A}[\![\tau_2]\!]).2 \to \mathcal{A}[\![\tau_3]\!].2))$ by type conversion
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_1]\!].2 \ : (\mathcal{A}[\![\tau_1]\!].1 \times \mathcal{A}[\![\tau_2]\!].1 \to \mathcal{A}[\![\tau_3]\!].1) \times \ ((\mathcal{A}[\![\tau_1]\!] \mathbin{\underline{\times}} \mathcal{A}[\![\tau_2]\!]).2 \to \mathcal{A}[\![\tau_3]\!].2)$ by typing rules
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_2]\!] \ : \ \mathcal{A}[\![\tau_1]\!].2$ by ind. hyp. (2) on $\mathcal{D}_2$
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \langle\!\langle \mathcal{L}[\![l, e_1]\!].2, \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \mathcal{L}[\![l, e_2]\!] \rangle\!\rangle \ : \mathcal{A}[\![\tau_2]\!].2 \Rrightarrow \mathcal{A}[\![\tau_3]\!].2$ by typing rules
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \langle\!\langle \mathcal{L}[\![l, e_1]\!].2, \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \mathcal{L}[\![l, e_2]\!] \rangle\!\rangle \ : (\mathcal{A}[\![\tau_2]\!] \mathbin{\underline{\Rrightarrow}} \mathcal{A}[\![\tau_3]\!]).2$ by type conversion

$$\textbf{Case}\ \mathcal{D} \ = \ \frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \Gamma \vdash_c e_1 : \tau_1 \Rightarrow \tau_2 & \Gamma \vdash_c e_2 : \tau_1\end{array}}{\Gamma \vdash_c e_1 \dagger e_2 : \tau_2}$$

Vectorisation

$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e_1]\!] : (\mathcal{A}[\![\tau_1]\!] \mathbin{\underline{\Rightarrow}} \mathcal{A}[\![\tau_2]\!]).1$ by ind. hyp. (1) on $\mathcal{D}_1$
$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e_1]\!] : \mathcal{A}[\![\tau_1]\!].1 \Rightarrow \mathcal{A}[\![\tau_2]\!].1$ by type conversion
$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e_2]\!] : \mathcal{A}[\![\tau_1]\!].1$ by ind. hyp. (1) on $\mathcal{D}_2$
$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{V}[\![e_1]\!] \dagger \mathcal{V}[\![e_2]\!] : \mathcal{A}[\![\tau_2]\!].1$ by typing rules

Lifting

$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_1]\!] \ : (\mathcal{A}[\![\tau_1]\!] \mathbin{\underline{\Rightarrow}} \mathcal{A}[\![\tau_2]\!]).2$ by ind. hyp. (2) on $\mathcal{D}_1$
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_1]\!] \ : \mathcal{A}[\![\tau_1]\!].2 \Rightarrow \mathcal{A}[\![\tau_2]\!].2$ by type conversion
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_2]\!] \ : \ \mathcal{A}[\![\tau_1]\!].2$ by ind. hyp. (2) on $\mathcal{D}_2$
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathcal{L}[\![l, e_1]\!] \ddagger \mathcal{L}[\![l, e_2]\!] \ : \mathcal{A}[\![\tau_2]\!].2$ by typing rules
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash l : \mathtt{Int}$ assumption
$\mathcal{G}[\![\Gamma^\uparrow]\!] \vdash \mathtt{attachP}_{\langle \mathcal{A}[\![\tau_2]\!] \rangle} \langle l, \ \mathcal{L}[\![l, e_1]\!] \ddagger \mathcal{L}[\![l, e_2]\!] \rangle \ : \ \mathcal{A}[\![\tau_2]\!].2$ by type of `attachP`

A.2 Proof of Lemma 6.19

The proof is by complete structural induction on e.

Part 1.

Case $e = \bullet$ By assumption on x.

Case $e = \lambda\bullet : \tau_1.\ e'$ Then, $\tau = \tau_1 \to \tau_2$ for some τ_2. Let $y : \mathcal{F}[\![\tau_1]\!]$ be a replicative term. Then,

$$\begin{aligned}
&\mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![\lambda\bullet : \tau_1.\ e']\!].1\ y \rangle \\
\cong\ &\mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!] \rangle}\ \langle 1,\ (\lambda\bullet : \mathcal{F}[\![\tau_1]\!].\ \mathcal{V}[\![e']\!])\ y \rangle && \text{by definition of } \mathcal{V}[\![\cdot]\!] \\
\cong\ &\mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e']\!][y/\bullet] \rangle \\
\cong\ &\mathcal{L}[\![1, e']\!][\mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \rangle}\ \langle 1,\ y \rangle/\bullet] && \text{by ind. hyp. 2 on } e' \\
\cong\ &(\lambda\bullet : \mathcal{F}[\![[:\tau_1:]]\!].\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\tau_1]\!] \rangle}\ \bullet, e']\!])\ (\mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \rangle}\ \langle 1,\ y \rangle) \\
\cong\ &\mathcal{V}[\![\lambda\bullet : \tau_1.\ e']\!].2\ (\mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \rangle}\ \langle 1,\ y \rangle) && \text{by definition of } \mathcal{V}[\![\cdot]\!]
\end{aligned}$$

Moreover, e' is replicative by induction hypothesis 1. Hence, $\mathcal{V}[\![e]\!][x/\bullet]$ is replicative.

In all other cases, $\mathcal{V}[\![e]\!][x/\bullet]$ is replicative by induction hypothesis 1 on its immediate subterms.

Part 2.

Case $e = \bullet$ Then, $\tau = \upsilon$.

$$\begin{aligned}
&\mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![\bullet]\!][x/\bullet] \rangle \\
=\ &\mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ x \rangle && \text{by definition of } \mathcal{V}[\![\cdot]\!] \\
=\ &\mathcal{L}[\![1, \bullet]\!][\mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ x \rangle/\bullet] && \text{by definition of } \mathcal{L}[\![\cdot, \cdot]\!]
\end{aligned}$$

Case $e = e_1\ e_2$ Then, $\bullet : \upsilon \vdash e_1 : \tau' \to \tau$ and $\bullet : \upsilon \vdash e_2 : \tau'$ for some τ'. We have

$$\begin{aligned}
&\mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e_1\ e_2]\!][x/\bullet] \rangle \\
=\ &\mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e_1]\!][x/\bullet].1\ \mathcal{V}[\![e_2]\!][x/\bullet] \rangle && \text{by definition of } \mathcal{V}[\![\cdot]\!]
\end{aligned}$$

Moreover,

$$\begin{aligned}
&\mathcal{L}[\![1, e_1\ e_2]\!][\mathtt{repP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\langle 1, x \rangle/\bullet] \\
=\ &\mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ \mathcal{L}[\![1, e_1]\!][\mathtt{repP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\langle 1, x \rangle/\bullet].2.2\ \mathcal{L}[\![1, e_2]\!][\mathtt{repP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\langle 1, x \rangle/\bullet] \rangle \\
& && \text{by definition of } \mathcal{L}[\![\cdot, \cdot]\!] \\
\cong\ &\mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ (\mathtt{repP}_{\langle \mathcal{A}[\![\tau']\!] \underline{\to} \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e_1]\!][x/\bullet] \rangle).2.2\ (\mathtt{repP}_{\langle \mathcal{A}[\![\tau']\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e_2]\!][x/\bullet] \rangle) \rangle \\
& && \text{by ind. hyp. 2 on } e_1 \text{ and } e_2 \\
\cong\ &\mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e_1]\!][x/\bullet].2\ (\mathtt{repP}_{\langle \mathcal{A}[\![\tau']\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e_2]\!][x/\bullet] \rangle) \rangle && \text{by definition of } \mathtt{repP}
\end{aligned}$$

Since $\mathcal{V}[\![e_1]\!][x/\bullet]$ is $\mathcal{V}$-compatible, it either diverges or evaluates to $\mathcal{V}[\![\lambda\bullet : \tau'.\ e']\!] \preceq \mathcal{V}[\![e_1]\!][x/\bullet]$ for some e' or to $\mathcal{V}[\![c^n_{\langle \upsilon_1, \ldots, \upsilon_1 \rangle}]\!]$ for some $c^n_{\langle \upsilon_1, \ldots, \upsilon_1 \rangle}$. We consider these cases separately.

Case $\mathcal{V}[\![e_1]\!][x/\bullet]\Uparrow$ Then,

$$\begin{aligned}
&\mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e_1]\!][x/\bullet].1\ \mathcal{V}[\![e_2]\!][x/\bullet]\rangle \\
\cong\ &\mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ \bot\rangle \\
\cong\ &\mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ \bot\rangle && \text{by Law } \mathit{attach/rep}_2 \\
\cong\ &\mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e_1]\!][x/\bullet].2\ (\mathtt{repP}_{\langle \mathcal{A}[\![\tau']\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e_2]\!][x/\bullet]\rangle)\rangle
\end{aligned}$$

Case $\mathcal{V}[\![e_1]\!][x/\bullet] \Downarrow \mathcal{V}[\![\lambda\bullet : \tau'.\ e']\!]$ $\mathcal{V}[\![e_1]\!]$ is replicative by induction hypothesis 1 on e_1 and, hence, $\mathcal{V}[\![\lambda\bullet : \tau'.\ e']\!]$ is replicative by Proposition 6.18. Therefore,

$$\begin{aligned}
&\mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e_1]\!][x/\bullet].1\ \mathcal{V}[\![e_2]\!][x/\bullet]\rangle \\
\cong\ &\mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![\lambda\bullet : \tau'.\ e']\!].1\ \mathcal{V}[\![e_2]\!][x/\bullet]\rangle \\
\cong\ &\mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ \mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![\lambda\bullet : \tau'.\ e']\!].1\ \mathcal{V}[\![e_2]\!][x/\bullet]\rangle\rangle && \text{by Law } \mathit{attach/rep}_1 \\
\cong\ &\mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![\lambda\bullet : \tau'.\ e']\!].2\ (\mathtt{repP}_{\langle \mathcal{A}[\![\tau']\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e_2]\!][x/\bullet]\rangle)\rangle \\
& && \text{by replicativity of } \mathcal{V}[\![\lambda\bullet : \tau'.\ e']\!] \\
\cong\ &\mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e_1]\!][x/\bullet].2\ (\mathtt{repP}_{\langle \mathcal{A}[\![\tau']\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e_2]\!][x/\bullet]\rangle)\rangle
\end{aligned}$$

Case $\mathcal{V}[\![e_1]\!][x/\bullet] \Downarrow \mathcal{V}[\![c_{\langle v_1,\ldots,v_1\rangle}]\!]$ Follows by the requirements on lifted primitives

Case $e = \langle e_1, e_2\rangle$ Then, $\tau = \tau_1 \times \tau_2$ such that $\bullet : \upsilon \vdash e_1 : \tau_1$ and $\bullet : \upsilon \vdash e_2 : \tau_2$.

$$\begin{aligned}
&\mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \underline{\times} \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![\langle e_1, e_2\rangle]\!][x/\bullet]\rangle \\
=\ &\mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \underline{\times} \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle 1,\ \langle \mathcal{V}[\![e_1]\!][x/\bullet],\ \mathcal{V}[\![e_2]\!][x/\bullet]\rangle\rangle && \text{by definition of } \mathcal{V}[\![\cdot]\!] \\
\cong\ &\mathtt{zipP}_{\langle \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle \mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e_1]\!][x/\bullet]\rangle,\ \mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e_2]\!][x/\bullet]\rangle\rangle \\
& && \text{by Law } \mathit{rep/zip} \\
\cong\ &\mathtt{zipP}_{\langle \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle \mathcal{L}[\![1, e_1]\!][\mathtt{repP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\langle 1, x\rangle/\bullet],\ \mathcal{L}[\![1, e_2]\!][\mathtt{repP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\langle 1, x\rangle/\bullet]\rangle \\
& && \text{by ind. hyp. 2 on } e_1 \text{ and } e_2 \\
=\ &\mathcal{L}[\![1, \langle e_1, e_2\rangle]\!][\mathtt{repP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\langle 1, x\rangle/\bullet] && \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!]
\end{aligned}$$

Case $e = e'.i$ Assume without loss of generality $i = 1$. Then, $\bullet : \upsilon \vdash e' : \tau \times \tau'$ for some τ'.

$$\begin{aligned}
&\mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e'.1]\!][x/\bullet]\rangle \\
=\ &\mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e']\!][x/\bullet].1\rangle && \text{by definition of } \mathcal{V}[\![\cdot]\!] \\
\cong\ &\mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ \mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e']\!][x/\bullet].1\rangle\rangle && \text{by Law } \mathit{attach/rep}_1 \\
\cong\ &\mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ (\mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!] \underline{\times} \mathcal{A}[\![\tau']\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e']\!][x/\bullet]\rangle).2.1\rangle && \text{by definition of } \mathtt{repP} \\
\cong\ &\mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle 1,\ \mathcal{L}[\![1, e']\!][\mathtt{repP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\langle 1, x\rangle/\bullet].2.1\rangle && \text{by ind. hyp. 2 on } e' \\
=\ &\mathcal{L}[\![1, e'.1]\!][\mathtt{repP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\langle 1, x\rangle/\bullet] && \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!]
\end{aligned}$$

The proof is analogous for $i = 2$.

Case $e = \langle\!\langle e_1, \tau_1, \tau_2, \tau_3, e_2\rangle\!\rangle$ Then, $\tau = \tau_2 \Rightarrow \tau_3$.

$$\begin{aligned}
&\mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!] \underline{\Rightarrow} \mathcal{A}[\![\tau_3]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![\langle\!\langle e_1, \tau_1, \tau_2, \tau_2, e_2\rangle\!\rangle]\!][x/\bullet]\rangle \\
=\ &\mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!] \underline{\Rightarrow} \mathcal{A}[\![\tau_3]\!]\rangle}\ \langle 1,\ \langle\!\langle \mathcal{V}[\![e_1]\!][x/\bullet], \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \mathcal{V}[\![e_2]\!][x/\bullet]\rangle\!\rangle\rangle \\
\cong\ &\langle\!\langle :\mathcal{V}[\![e_1]\!][x/\bullet], \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e_2]\!][x/\bullet]\rangle :\rangle\!\rangle && \text{by definition of } \mathtt{repP} \\
\cong\ &\langle\!\langle :\mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \underline{\rightarrow} \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e_1]\!][x/\bullet].2, \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!]\rangle}\ \langle 1,\ \mathcal{V}[\![e_2]\!][x/\bullet]\rangle :\rangle\!\rangle \\
& && \text{by definition of } \mathtt{repP} \\
\cong\ &\langle\!\langle :\mathcal{L}[\![1, e_1]\!][\mathtt{repP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\langle 1, x\rangle/\bullet].2, \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \mathcal{L}[\![1, e_2]\!][\mathtt{repP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\langle 1, x\rangle/\bullet] :\rangle\!\rangle \\
& && \text{by ind. hyp. 2 on } e_1 \text{ and } e_2 \\
=\ &\mathcal{L}[\![1, \langle\!\langle e_1, \tau_1, \tau_2, \tau_3, e_2\rangle\!\rangle]\!][\mathtt{repP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\langle 1, x\rangle/\bullet] && \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!]
\end{aligned}$$

Case $e = e_1 \dagger e_2$ Then, $\bullet : v \vdash e_1 : \tau_2 \Rightarrow \tau$ and $\bullet : v \vdash e_2 : \tau_2$ for some τ_2. We have

$$\begin{aligned}
&\mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e_1 \dagger e_2]\!][x/\bullet] \rangle \\
&= \mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e_1]\!][x/\bullet] \dagger \mathcal{V}[\![e_2]\!][x/\bullet] \rangle && \text{by definition of } \mathcal{V}[\![\cdot]\!]
\end{aligned}$$

$\mathcal{V}[\![e_1]\!][x/\bullet]$ either diverges or evaluates to $\langle\!\langle p, \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau]\!], z \rangle\!\rangle$ for some p and z.

Case $\mathcal{V}[\![e_1]\!][x/\bullet]\Uparrow$ Then,

$$\begin{aligned}
&\mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e_1]\!][x/\bullet] \dagger \mathcal{V}[\![e_2]\!][x/\bullet] \rangle \\
&\cong \mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1, \bot \rangle \\
&\cong \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1, \bot \rangle && \text{by Law } \textit{attach/rep}_2 \\
&\cong \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1, \mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \underline{\Rightarrow} \mathcal{A}[\![\tau_2]\!] \rangle} \langle 1, \mathcal{V}[\![e_1]\!][x/\bullet] \rangle \ddagger \mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \rangle} \langle 1, \mathcal{V}[\![e_2]\!][x/\bullet] \rangle \rangle \\
& && \text{by definition of repP and semantics of } \ddagger \\
&\cong \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1, \mathcal{L}[\![1, e_1]\!][x/\bullet] \ddagger \mathcal{L}[\![1, e_2]\!][x/\bullet] \rangle && \text{by ind. hyp. 2 on } e_1 \text{ and } e_2 \\
&= \mathcal{L}[\![1, e_1 \dagger e_2]\!][x/\bullet] && \text{by definition of } \mathcal{L}[\![\cdot, \cdot]\!]
\end{aligned}$$

Case $\mathcal{V}[\![e_1]\!][x/\bullet] \Downarrow \langle\!\langle p, \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau]\!], z \rangle\!\rangle$ Then,

$$\begin{aligned}
&\mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e_1]\!][x/\bullet] \dagger \mathcal{V}[\![e_2]\!][x/\bullet] \rangle \\
&\cong \mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ \langle\!\langle p, \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], z \rangle\!\rangle \dagger \mathcal{V}[\![e_2]\!][x/\bullet] \rangle \\
&\cong \mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ p.1\ \langle z, \mathcal{V}[\![e_2]\!][x/\bullet] \rangle \rangle
\end{aligned}$$

Since p is $\mathcal{V}$-compatible, it either diverges or evaluates to some $\mathcal{V}[\![\lambda\bullet : \tau_1 \times \tau_2.\ e']\!] \preceq \mathcal{V}[\![e_1]\!][x/\bullet]$ or to some $\mathcal{V}[\![c_{\langle v_1, \ldots, v_n \rangle}]\!]$. By the same reasoning as in the case $e = e_1\ e_2$, we arrive at

$$\begin{aligned}
&\cong \mathtt{repP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ p.1\ \langle z, \mathcal{V}[\![e_2]\!][x/\bullet] \rangle \rangle \\
&\cong \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ p.2\ (\mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \underline{\times} \mathcal{A}[\![\tau_2]\!] \rangle}\ \langle 1,\ \langle z, \mathcal{V}[\![e_2]\!][x/\bullet] \rangle \rangle) \rangle \\
&\cong \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ p.2\ (\mathtt{zipP}_{\langle \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!] \rangle}\ \langle \mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \rangle}\ \langle 1,\ z \rangle,\ \mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e_2]\!][x/\bullet] \rangle \rangle) \rangle \\
& && \text{by Law } \textit{rep/zip} \\
&\cong \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1, \\
& \qquad \langle\!\langle p, \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \mathtt{repP}_{\langle \mathcal{A}[\![\tau_1]\!] \rangle}\ \langle 1,\ z \rangle \rangle\!\rangle \ddagger \mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e_2]\!][x/\bullet] \rangle \rangle \\
& && \text{by semantics of } \ddagger \\
&\cong \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1, \\
& \qquad \mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!] \underline{\Rightarrow} \mathcal{A}[\![\tau_3]\!] \rangle}\ \langle 1,\ \langle\!\langle p, \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], z \rangle\!\rangle \rangle \ddagger \mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e_2]\!][x/\bullet] \rangle \rangle \\
& && \text{by definition of repP} \\
&\cong \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ \mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!] \underline{\Rightarrow} \mathcal{A}[\![\tau_3]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e_1]\!][x/\bullet] \rangle \ddagger \mathtt{repP}_{\langle \mathcal{A}[\![\tau_2]\!] \rangle}\ \langle 1,\ \mathcal{V}[\![e_2]\!][x/\bullet] \rangle \rangle \\
& && \text{by assumption} \\
&\cong \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle 1,\ \mathcal{L}[\![1, e_1]\!][\mathtt{repP}_{\langle \mathcal{A}[\![v]\!] \rangle} \langle 1, x \rangle / \bullet] \ddagger \mathcal{L}[\![1, e_2]\!][\mathtt{repP}_{\langle \mathcal{A}[\![v]\!] \rangle} \langle 1, x \rangle / \bullet] \rangle \\
& && \text{by ind. hyp. 2 on } e_1 \text{ and } e_2 \\
&\cong \mathcal{L}[\![1, e_1 \dagger e_2]\!][\mathtt{repP}_{\langle \mathcal{A}[\![v]\!] \rangle} \langle 1, x \rangle / \bullet] && \text{by definition of } \mathcal{L}[\![\cdot, \cdot]\!]
\end{aligned}$$

Case $e = \langle\rangle$ Then, $\tau = \langle\rangle$ and we have

$$\begin{aligned}
&\mathtt{repP}_{\langle \underline{\langle\rangle} \rangle}\ \langle 1,\ \langle\rangle[x/\bullet] \rangle \\
&= \mathtt{repP}_{\langle \underline{\langle\rangle} \rangle}\ \langle 1,\ \langle\rangle \rangle \\
&= \mathcal{L}[\![1, \underline{\langle\rangle}]\!] && \text{by definition of } \mathcal{L}[\![\cdot, \cdot]\!] \\
&= \mathcal{L}[\![1, \langle\rangle]\!][\mathtt{repP}_{\langle \mathcal{A}[\![v]\!] \rangle} \langle 1, x \rangle / \bullet]
\end{aligned}$$

The remaining cases are analogous to the above.

A.3 Proof of Lemma 6.25

By complete structural induction on e.

Case $e = \bullet$ Then, $\tau = \upsilon$ and we have

$$
\begin{aligned}
& \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, \bullet]\!][x/\bullet] \\
&= x && \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!] \\
&\sim y && \text{by assumption} \\
&= \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, \bullet]\!][y/\bullet] && \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!]
\end{aligned}
$$

Case $e = e_1\, e_2$ Then, $\bullet : \upsilon \vdash e_1 : \tau' \to \tau$ and $\bullet : \upsilon \vdash e_2 : \tau'$ for some τ'. By induction hypothesis on e_1, $\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet].2 \sim \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle} \bullet, e_1]\!][y/\bullet].2$. Thus, the two terms must either diverge or evaluate to the same function tuple by definition of $\sim$ for function types. Due to the $\mathcal{V}$-compatibility of the terms involved, if $\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet].2$ converges, then it evaluates to either $\mathcal{V}[\![\lambda \bullet : \tau'.\ e']\!]$ for some e' or $c^n_{\langle \upsilon_1,\ldots,\upsilon_n\rangle}$ for some c^n. We consider these three cases separately.

Case $\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\tau]\!]\rangle} \bullet, e_1]\!][x/\bullet].2\Uparrow \wedge \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\tau]\!]\rangle} \bullet, e_1]\!][y/\bullet].2\Uparrow$ Then,

$$
\begin{aligned}
& \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \bullet, e_1\ e_2]\!][x/\bullet] && \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!] \\
&= \mathtt{attachP}_{\langle\tau\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ x, \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet].2.2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet]\rangle \\
&\sim \mathtt{attachP}_{\langle\tau\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ x, \bot\rangle && \text{by Lemma 6.24} \\
&\sim \mathtt{attachP}_{\langle\tau\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ y, \bot\rangle && \text{by Lemma 6.24} \\
&\sim \mathtt{attachP}_{\langle\tau\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ y, \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][y/\bullet].2.2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet]\rangle \\
& && \text{by Lemma 6.24} \\
&= \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \bullet, e_1\ e_2]\!][y/\bullet] && \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!]
\end{aligned}
$$

Case $\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle} \bullet, e_1]\!][x/\bullet].2 \Downarrow \mathcal{V}[\![\lambda \bullet : \tau'.\ e']\!] \wedge \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle} \bullet, e_1]\!][y/\bullet].2 \Downarrow \mathcal{V}[\![\lambda \bullet : \tau'.\ e']\!]$ By immutability of computation and since the result of evaluating x does not contain functions, $\lambda \bullet : \tau'.\ e' \preceq e_1$

$$
\begin{aligned}
& \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1\ e_2]\!][x/\bullet] \\
&= \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ x,\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet].2.2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet]\rangle \\
& && \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!] \\
&\sim \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ x,\ \mathcal{V}[\![\lambda \bullet : \tau'.\ e']\!].2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet]\rangle \\
& && \text{by Proposition 6.23 and Lemma 6.24} \\
&\sim \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ x, (\lambda \bullet : \mathcal{F}[\![[:\tau':]]\!].\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\tau']\!]\rangle}\ \bullet, e']\!])\ \mathcal{L}[\![l_x, e_2]\!][x/\bullet]\rangle \\
& && \text{by definition of } \mathcal{V}[\![\cdot]\!] \\
&\sim \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ x,\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\tau']\!]\rangle}\ \bullet, e']\!][\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\tau']\!]\rangle}\ \bullet, e_2]\!][x/\bullet]/\bullet]\rangle \\
&\sim \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ y,\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\tau']\!]\rangle}\ \bullet, e']\!][\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\tau']\!]\rangle}\ \bullet, e_2]\!][y/\bullet]/\bullet]\rangle \\
& && \text{by Lemma 6.24 and ind. hyp. on } e_2 \text{ and then } e' \\
&\sim \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1\ e_2]\!][y/\bullet] && \text{(inverse to the above)}
\end{aligned}
$$

Case $\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle} \bullet, e_1]\!][x/\bullet].2 \Downarrow \mathcal{V}[\![c_{\langle \upsilon_1,\ldots,\upsilon_n\rangle}]\!] \wedge \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle} \bullet, e_1]\!][y/\bullet].2 \Downarrow \mathcal{V}[\![c_{\langle \upsilon_1,\ldots,\upsilon_n\rangle}]\!]$ Follows by requirements on primitives.

Case $e = \langle e_1, e_2 \rangle$ Then, $\tau = \tau_1 \times \tau_2$ such that $\bullet : \upsilon \vdash e_1 : \tau_1$ and $\bullet : \upsilon \vdash e_1 : \tau_2$. We have

$$
\begin{aligned}
& \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, \langle e_1, e_2 \rangle]\!][x/\bullet] \\
&= \mathtt{zipP}_{\langle \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!] \rangle}\ \langle \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, e_1]\!][x/\bullet],\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, e_2]\!][x/\bullet] \rangle \\
& \qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!] \\
&\sim \mathtt{zipP}_{\langle \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!] \rangle}\ \langle \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, e_1]\!][y/\bullet],\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, e_2]\!][y/\bullet] \rangle \\
& \qquad \text{by Lemma 6.24 and ind. hyp. on } e_1 \text{ and } e_2 \\
&= \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, \langle e_1, e_2 \rangle]\!][y/\bullet] \qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!]
\end{aligned}
$$

Case $e = e'.i$ Assume without loss of generality that $i = 1$. Then, $\bullet : \upsilon \vdash e' : \tau \times \tau'$ and we have

$$
\begin{aligned}
& \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, e'.i]\!][x/\bullet] \\
&= \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ x, \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, e']\!][x/\bullet].2.1 \rangle \qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!] \\
&\sim \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ y, \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, e']\!][y/\bullet].2.1 \rangle \qquad \text{by Lemma 6.24 and ind. hyp. on } e' \\
&= \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, e'.i]\!][y/\bullet] \qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!]
\end{aligned}
$$

Case $e = e_1 \dagger e_2$. Then, $\bullet : \upsilon \vdash e_1 : \tau' \Rightarrow \tau$ and $\bullet : \upsilon \vdash e_2 : \tau'$. We have

$$
\begin{aligned}
& \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, e_1 \dagger e_2]\!][x/\bullet] \\
&= \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ x,\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, e_1]\!][x/\bullet] \ddagger \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, e_2]\!][x/\bullet] \rangle \\
& \qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!] \\
&\sim \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!] \rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ y,\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, e_1]\!][y/\bullet] \ddagger \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, e_2]\!][y/\bullet] \rangle \\
& \qquad \text{by Lemma 6.24 and ind. hyp. on } e_1 \text{ and } e_2 \\
&= \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!] \rangle}\ \bullet, e_1 \dagger e_2]\!][y/\bullet] \qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!]
\end{aligned}
$$

The remaining cases are immediate from the definition of $\sim$.

A.4 Proof of Lemma 6.29

The proof is by complete structural induction on e.

Case $e = \bullet$ Then $\tau = \upsilon$ and we have

$$\begin{aligned}
&\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, \bullet]\!][x/\bullet] \mathbin{+\!\!+\!\!+} \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, \bullet]\!][y/\bullet] \\
&= x \mathbin{+\!\!+\!\!+} y && \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!] \\
&= \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, \bullet]\!][x \mathbin{+\!\!+\!\!+} y/\bullet] && \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!]
\end{aligned}$$

Case $e = e_1\ e_2$ Then, $\bullet : \upsilon \vdash e_1 : \tau' \to \tau$ and $\bullet : \upsilon \vdash e_2 : \tau'$. We have

$$\begin{aligned}
&\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1\ e_2]\!][x/\bullet] \mathbin{+\!\!+\!\!+} \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1\ e_2]\!][y/\bullet] \\
&= \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ x,\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet].2.2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet]\rangle \\
&\qquad \mathbin{+\!\!+\!\!+} \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ y,\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][y/\bullet].2.2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet]\rangle \\
&\qquad\qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!] \\
&\cong \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ (x \mathbin{+\!\!+\!\!+} y),\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet].2.2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet] \\
&\qquad\qquad \mathbin{+\!\!+\!\!+} \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][y/\bullet].2.2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet]\rangle \\
&\qquad\qquad\qquad \text{by Laws } \textit{attach/concat} \text{ and } \textit{len/concat}
\end{aligned}$$

Moreover,

$$\begin{aligned}
&\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1\ e_2]\!][x \mathbin{+\!\!+\!\!+} y/\bullet] \\
&= \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ (x \mathbin{+\!\!+\!\!+} y), \\
&\qquad\qquad \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x \mathbin{+\!\!+\!\!+} y/\bullet].2.2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x \mathbin{+\!\!+\!\!+} y/\bullet]\rangle \\
&\qquad\qquad\qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!]
\end{aligned}$$

By Lemma 6.25 we have $\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, \bullet]\!][x/\bullet].2 \sim \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, \bullet]\!][y/\bullet].2$ and, by Lemma 6.27 and Lemma 6.25, $\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, \bullet]\!][x/\bullet].2 \sim \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, \bullet]\!][x \mathbin{+\!\!+\!\!+} y/\bullet].2$. By definition of data-erasing equivalence and due to the $\mathcal{V}$-compatibility of the terms involved, the three terms either diverge or evaluate to some $\mathcal{V}[\![\lambda\bullet : \tau'.\ e']\!]$ or to some $\mathcal{V}[\![c_{\langle \upsilon_1,\ldots,\upsilon_n\rangle}]\!]$. We consider the three cases separately.

Case $\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, \bullet]\!][z/\bullet].2\Uparrow$ *for all* $z \in \{x, y, x \mathbin{+\!\!+\!\!+} y\}$ Then,

$$\begin{aligned}
&\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet].2.2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet] \\
&\qquad\qquad \mathbin{+\!\!+\!\!+} \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][y/\bullet].2.2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet] \\
&\cong \bot \mathbin{+\!\!+\!\!+} \bot \\
&\cong \bot && \text{by definition of } \mathbin{+\!\!+\!\!+} \\
&\cong \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x \mathbin{+\!\!+\!\!+} y/\bullet].2.2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x \mathbin{+\!\!+\!\!+} y/\bullet]
\end{aligned}$$

Case $\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_1]\!][z/\bullet].2 \Downarrow \mathcal{V}[\![\lambda\bullet : \tau'.\ e']\!]$ *for all* $z \in \{x, y, x \mathbin{+\!\!+\!\!+} y\}$ Due to the fact that none of x, y and $x \mathbin{+\!\!+\!\!+} y$ evaluate to structures containing functions, $\mathcal{V}[\![\lambda\bullet : \tau'.\ e']\!] \preceq e_1$. Then,

$$\begin{aligned}
&\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet].2.2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet] \\
&\qquad\qquad \mathbin{+\!\!+\!\!+} \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][y/\bullet].2.2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet] \\
&\cong \mathcal{V}[\![\lambda\bullet : \tau'.\ e']\!].2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet] \mathbin{+\!\!+\!\!+} \mathcal{V}[\![\lambda\bullet : \tau'.\ e']\!].2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet] \\
&\cong \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\tau']\!]\rangle}\ \bullet, e']\!][\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][x/\bullet]/\bullet] \\
&\qquad \mathbin{+\!\!+\!\!+} \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\tau']\!]\rangle}\ \bullet, e']\!][\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][y/\bullet]/\bullet] && \text{by definition of } \mathcal{V}[\![\cdot]\!]
\end{aligned}$$

$$
\begin{array}{ll}
\cong & \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\tau']\!]\rangle}\bullet, e']\!][\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][x/\bullet] \mathbin{+\!\!+\!\!+} \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][y/\bullet]/\bullet] \\
& \qquad \text{by Lemma 6.25 and ind. hyp. on } e' \\
\cong & \mathcal{V}[\![\lambda\bullet : \tau'.\ e']\!].2\ (\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][x/\bullet] \mathbin{+\!\!+\!\!+} \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][y/\bullet]) \\
& \qquad \text{by definition of } \mathcal{V}[\![\cdot]\!] \\
\cong & \mathcal{V}[\![\lambda\bullet : \tau'.\ e']\!]\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][x \mathbin{+\!\!+\!\!+} y/\bullet] \qquad \text{by ind. hyp. on } e_2 \\
\cong & \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_1]\!][x \mathbin{+\!\!+\!\!+} y/\bullet].2.2\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][x \mathbin{+\!\!+\!\!+} y/\bullet]
\end{array}
$$

Case $\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_1]\!][z/\bullet].2 \Downarrow \mathcal{V}[\![c_{\langle \upsilon_1,\ldots,\upsilon_n\rangle}]\!]$ *for all* $z \in \{x, y, x \mathbin{+\!\!+\!\!+} y\}$ Follows by requirements on primitives.

Case $e = \langle e_1, e_2\rangle$ Then, $\tau = \tau_1 \times \tau_2$ such that $\bullet : \upsilon \vdash e_1 : \tau_1$ and $\bullet : \upsilon \vdash e_2 : \tau_2$. We have

$$
\begin{array}{ll}
& \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, \langle e_1, e_2\rangle]\!][x/\bullet] \mathbin{+\!\!+\!\!+} \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, \langle e_1, e_2\rangle]\!][y/\bullet] \\
= & \mathtt{zipP}_{\langle \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_1]\!][x/\bullet],\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][x/\bullet]\rangle \\
& \qquad \mathbin{+\!\!+\!\!+} \mathtt{zipP}_{\langle \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_1]\!][y/\bullet],\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][y/\bullet]\rangle \\
& \qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!] \\
\cong & \mathtt{zipP}_{\langle \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_1]\!][x/\bullet] \mathbin{+\!\!+\!\!+} \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_1]\!][y/\bullet], \\
& \qquad \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][x/\bullet] \mathbin{+\!\!+\!\!+} \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][y/\bullet]\rangle \qquad \text{by Law } \textit{zip/concat} \\
\cong & \mathtt{zipP}_{\langle \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!]\rangle}\ \langle \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_1]\!][x \mathbin{+\!\!+\!\!+} y/\bullet],\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][x \mathbin{+\!\!+\!\!+} y/\bullet]\rangle \\
& \qquad \text{by ind. hyp. on } e_1 \text{ and } e_2 \\
= & \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, \langle e_1, e_2\rangle]\!][x \mathbin{+\!\!+\!\!+} y/\bullet] \qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!]
\end{array}
$$

Case $e = e'.i$ Assume without loss of generality that $i = 1$. Then, $\bullet : \upsilon \vdash e' : \tau \times \tau'$. We have

$$
\begin{array}{ll}
& \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e'.i]\!][x/\bullet] \mathbin{+\!\!+\!\!+} \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e'.i]\!][y/\bullet] \\
= & \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ x,\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e']\!][x/\bullet].2.1\rangle \\
& \qquad \mathbin{+\!\!+\!\!+} \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ y,\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e']\!][y/\bullet].2.1\rangle \\
& \qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!] \\
\cong & \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ (x \mathbin{+\!\!+\!\!+} y), \\
& \qquad \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e']\!][x/\bullet].2.1 \mathbin{+\!\!+\!\!+} \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e']\!][y/\bullet].2.1\rangle \\
& \qquad \text{by Laws } \textit{attach/concat} \text{ and } \textit{len/concat} \\
\cong & \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ (x \mathbin{+\!\!+\!\!+} y), \\
& \qquad (\mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e']\!][x/\bullet] \mathbin{+\!\!+\!\!+} \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e']\!][y/\bullet]).2.1\rangle \\
& \qquad \text{by definition of } \mathbin{+\!\!+\!\!+} \\
\cong & \mathtt{attachP}_{\langle \mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ (x \mathbin{+\!\!+\!\!+} y),\ \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e']\!][x \mathbin{+\!\!+\!\!+} y/\bullet]\rangle.2.1 \text{ by ind. hyp. on } e' \\
= & \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e'.1]\!][x \mathbin{+\!\!+\!\!+} y/\bullet] \qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!]
\end{array}
$$

Case $e = \langle\!\langle e_1, \tau_1, \tau_2, \tau_3, e_2\rangle\!\rangle$ Then, $\tau = \tau_2 \Rightarrow \tau_3$. Below, we use the following abbreviations:

$$
\begin{array}{rcl}
c_x & \equiv & \langle\!\langle : \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_1]\!][x/\bullet].2, \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][x/\bullet] : \rangle\!\rangle \\
c_y & \equiv & \langle\!\langle : \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_1]\!][y/\bullet].2, \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][y/\bullet] : \rangle\!\rangle \\
c_{xy} & \equiv & \langle\!\langle : \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_1]\!][x \mathbin{+\!\!+\!\!+} y/\bullet].2, \mathcal{A}[\![\tau_1]\!], \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \\
& & \qquad \mathcal{L}[\![\mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_2]\!][x \mathbin{+\!\!+\!\!+} y/\bullet] : \rangle\!\rangle \\
\mathit{sel} & \equiv & \mathtt{repP}_{\langle \underline{\mathtt{Bool}}\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ x, \mathit{False}\rangle \mathbin{+\!\!+\!\!+} \mathtt{repP}_{\langle \underline{\mathtt{Bool}}\rangle}\ \langle \mathtt{lenP}_{\langle \mathcal{A}[\![\upsilon]\!]\rangle}\ x, \mathit{True}\rangle
\end{array}
$$

Then,

$$
\begin{aligned}
&\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, \langle\!\langle e_1, \tau_1, \tau_2, \tau_3, e_2\rangle\!\rangle]\!][x/\bullet] \ +\!\!+\!\!+\ \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, \langle\!\langle e_1, \tau_1, \tau_2, \tau_3, e_2\rangle\!\rangle]\!][y/\bullet] \\
&= c_x \ +\!\!+\!\!+\ c_y && \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!] \\
&\cong \langle\!\!|\langle \mathtt{cappP}, \mathtt{cappP}^{\uparrow}\rangle, (\mathcal{A}[\![\tau_2]\!] \underline{\Rightarrow} \mathcal{A}[\![\tau_3]\!]) \underline{+} (\mathcal{A}[\![\tau_2]\!] \underline{\Rightarrow} \mathcal{A}[\![\tau_3]\!]), \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \langle sel, \langle c_x, c_y\rangle\rangle |\!\!\rangle && \text{by definition of } +\!\!+\!\!+
\end{aligned}
$$

Moreover,

$$
\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, \langle\!\langle e_1, \tau_1, \tau_2, \tau_3, e_2\rangle\!\rangle]\!][x +\!\!+\!\!+ y/\bullet] \ = \ c_{xy} \qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!]
$$

The two array closures are equivalent if they yield equivalent results when applied to an array argument. Let z be an array of type $\mathcal{F}[\![[:\tau_2:]]\!]$ and of suitable length. Then,

$$
\begin{aligned}
&\langle\!\!|\langle \mathtt{cappP}, \mathtt{cappP}^{\uparrow}\rangle, (\mathcal{A}[\![\tau_2]\!] \underline{\Rightarrow} \mathcal{A}[\![\tau_3]\!]) \underline{+} (\mathcal{A}[\![\tau_2]\!] \underline{\Rightarrow} \mathcal{A}[\![\tau_3]\!]), \mathcal{A}[\![\tau_2]\!], \mathcal{A}[\![\tau_3]\!], \langle sel, \langle c_x, c_y\rangle\rangle |\!\!\rangle \ddagger z \\
&\cong \mathtt{cappP}^{\uparrow}\ (\mathtt{zipP}\ \langle\langle sel, \langle c_x, c_y\rangle\rangle, z\rangle) \\
&\cong \mathtt{combineP}\ \langle sel,\ \langle c_x \ddagger \mathtt{packP}\ \langle \mathtt{not}^{\uparrow}\ sel, z\rangle,\ c_y \ddagger \mathtt{packP}\ \langle sel, z\rangle\rangle && \text{by definition of } \mathtt{cappP}^{\uparrow} \\
&\cong (c_x \ddagger \mathtt{packP}\ \langle \mathtt{not}^{\uparrow}\ sel, z\rangle) \ +\!\!+\!\!+\ (c_y \ddagger \mathtt{packP}\ \langle sel, z\rangle) && \text{by Law } \textit{combine/rep} \\
&\cong (c_x \ddagger \mathtt{takeP}\ \langle \mathtt{lenP}\ x, z\rangle) \ +\!\!+\!\!+\ (c_y \ddagger \mathtt{dropP}\ \langle \mathtt{lenP}\ x, z\rangle) && \text{by Laws } \textit{pack/take} \text{ and } \textit{pack/drop} \\
&\cong \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet].2.2\ (\mathtt{zipP}\ \langle \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet], \mathtt{takeP}\ \langle \mathtt{lenP}\ x, z\rangle\rangle) \\
&\qquad +\!\!+\!\!+\ \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][y/\bullet].2.2\ (\mathtt{zipP}\ \langle \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet], \mathtt{dropP}\ \langle \mathtt{lenP}\ x, z\rangle\rangle)
\end{aligned}
$$

By the same argument as in the case $e = e_1\ e_2$, we have

$$
\begin{aligned}
&\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet].2.2\ (\mathtt{zipP}\ \langle \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet], \mathtt{takeP}\ \langle \mathtt{lenP}\ x, z\rangle\rangle) \\
&\qquad +\!\!+\!\!+\ \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][y/\bullet].2.2\ (\mathtt{zipP}\ \langle \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet], \mathtt{dropP}\ \langle \mathtt{lenP}\ x, z\rangle\rangle) \\
&\cong \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x +\!\!+\!\!+ y/\bullet].2.2\ (\mathtt{zipP}\ \langle \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet], \mathtt{takeP}\ \langle \mathtt{lenP}\ x, z\rangle \\
&\qquad\qquad +\!\!+\!\!+\ \mathtt{zipP}\ \langle \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet], \mathtt{dropP}\ \langle \mathtt{lenP}\ x, z\rangle\rangle) \\
&\cong \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x +\!\!+\!\!+ y/\bullet].2.2 \\
&\qquad\qquad (\mathtt{zipP}\ \langle \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet] \ +\!\!+\!\!+\ \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet], \\
&\qquad\qquad\qquad \mathtt{takeP}\ \langle \mathtt{lenP}\ x, z\rangle \ +\!\!+\!\!+\ \mathtt{dropP}\ \langle \mathtt{lenP}\ x, z\rangle\rangle) \\
&\qquad\qquad && \text{by Law } \textit{concat/zip} \\
&\cong \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x +\!\!+\!\!+ y/\bullet].2.2\ (\mathtt{zipP}\ \langle \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x +\!\!+\!\!+ y/\bullet], \\
&\qquad\qquad\qquad \mathtt{takeP}\ \langle \mathtt{lenP}\ x, z\rangle \ +\!\!+\!\!+\ \mathtt{dropP}\ \langle \mathtt{lenP}\ x, z\rangle\rangle) \\
&\qquad\qquad && \text{by ind. hyp. on } e_2 \\
&\cong \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x +\!\!+\!\!+ y/\bullet].2.2\ (\mathtt{zipP}\ \langle \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x +\!\!+\!\!+ y/\bullet], z\rangle) \\
&\qquad\qquad && \text{by Law } \textit{take/drop} \\
&\cong c_{xy} \ddagger z
\end{aligned}
$$

Case $e = e_1 \dagger e_2$ Then, $\bullet : \upsilon \vdash e_1 : \tau' \Rightarrow \tau$ and $\bullet : \upsilon \vdash e_2 : \tau'$. We have

$$
\begin{aligned}
&\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1 \dagger e_2]\!][x/\bullet] \ +\!\!+\!\!+\ \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1 \dagger e_2]\!][y/\bullet] \\
&= \mathtt{attachP}_{\langle\mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ x,\ \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet] \ddagger \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet]\rangle \\
&\qquad +\!\!+\!\!+\ \mathtt{attachP}_{\langle\mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ y,\ \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][y/\bullet] \ddagger \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet]\rangle \\
&\qquad\qquad && \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!] \\
&\cong \mathtt{attachP}_{\langle\mathcal{A}[\![\tau]\!]\rangle}\ \langle \mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ (x +\!\!+\!\!+ y),\ (\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet] \ddagger \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet]) \\
&\qquad\qquad +\!\!+\!\!+\ (\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][y/\bullet] \ddagger \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet])\rangle \\
&\qquad\qquad && \text{by Laws } \textit{attach/concat} \text{ and } \textit{len/concat}
\end{aligned}
$$

By Lemma 6.25, $\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet]$ and $\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][y/\bullet]$ exhibit the same convergence beahviour. Using a similar argument as in the previous case, it is easy to show that

$$
\begin{aligned}
&(\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet] \ddagger \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet]) \\
&\qquad\qquad +\!\!+\!\!+\ (\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][y/\bullet] \ddagger \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet]) \\
\cong\ &(\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_1]\!][x/\bullet] +\!\!+\!\!+ \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\bullet, e_1]\!][y/\bullet]) \\
&\qquad\qquad \ddagger\ (\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet] +\!\!+\!\!+ \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet])
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
&\mathtt{attachP}_{\langle\mathcal{A}[\![\tau]\!]\rangle}\ \langle\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ (x +\!\!+\!\!+ y),\ (\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet] \ddagger \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet]) \\
&\qquad\qquad +\!\!+\!\!+\ (\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][y/\bullet] \ddagger \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet])\rangle \\
\cong\ &\mathtt{attachP}_{\langle\mathcal{A}[\![\tau]\!]\rangle}\ \langle\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ (x +\!\!+\!\!+ y),\ (\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x/\bullet] +\!\!+\!\!+ \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][y/\bullet]) \\
&\qquad\qquad \ddagger\ (\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x/\bullet] +\!\!+\!\!+ \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][y/\bullet])\rangle \\
\cong\ &\mathtt{attachP}_{\langle\mathcal{A}[\![\tau]\!]\rangle}\ \langle\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ (x +\!\!+\!\!+ y), \\
&\qquad\qquad \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1]\!][x +\!\!+\!\!+ y/\bullet] \ddagger \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_2]\!][x +\!\!+\!\!+ y/\bullet]\rangle \\
&\qquad\qquad\qquad\qquad \text{by ind. hyp. on } e_1 \text{ and } e_2 \\
=\ &\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, e_1 \dagger e_2]\!][x +\!\!+\!\!+ y/\bullet] \qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!]
\end{aligned}
$$

Case $e = \langle\rangle$ Then, $\tau = \langle\rangle$ and we have

$$
\begin{aligned}
&\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, \langle\rangle]\!][x/\bullet] +\!\!+\!\!+ \mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, \langle\rangle]\!][y/\bullet] \\
=\ &\mathtt{repP}_{\langle\underline{\langle\rangle}\rangle}\ \langle\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ x,\ \langle\rangle\rangle +\!\!+\!\!+ \mathtt{repP}_{\langle\underline{\langle\rangle}\rangle}\ \langle\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ y,\ \langle\rangle\rangle \qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!] \\
\cong\ &\mathtt{repP}_{\langle\underline{\langle\rangle}\rangle}\ \langle\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ (x +\!\!+\!\!+ y),\ \langle\rangle\rangle \qquad \text{by Laws } \textit{rep/concat} \text{ and } \textit{len/concat} \\
=\ &\mathcal{L}[\![\mathtt{lenP}_{\langle\mathcal{A}[\![\upsilon]\!]\rangle}\ \bullet, \langle\rangle]\!][x +\!\!+\!\!+ y/\bullet] \qquad \text{by definition of } \mathcal{L}[\![\cdot,\cdot]\!]
\end{aligned}
$$

The remaining cases are analogous to the above.

Appendix B

Primitives in $\lambda^{\mathcal{C}}$

lenP

$$
\begin{array}{lll}
\texttt{lenP}_{\langle\tau\rangle} & :: & [:\tau:] \to \texttt{Int} \\
\texttt{lenP} \quad (\texttt{nilP}\,\langle\rangle) & = & 0 \\
\texttt{lenP} \quad (\texttt{repP}\,\langle n, x\rangle) & = & n \\
\texttt{lenP} \quad (xs \mathbin{+\!\!+} ys) & = & \texttt{lenP}\,xs + \texttt{lenP}\,ys
\end{array}
$$

(!:)

$$
\begin{array}{lll}
(!:_{\langle\tau\rangle}) & :: & [:\tau:] \times \texttt{Int} \to \tau \\
\texttt{repP}\,\langle n, x\rangle \;!:\; i & = & x \\
(xs \mathbin{+\!\!+} ys) \;!:\; i & = & \textbf{if}\; i < \texttt{lenP}\,xs\; \textbf{then}\; xs\;!:\;i\; \textbf{else}\; ys\;!:\;(i - \texttt{lenP}\,xs)
\end{array}
$$

mapP

$$
\begin{array}{lll}
\texttt{mapP}_{\langle\tau,\upsilon\rangle} & :: & (\tau \Rightarrow \upsilon) \times [:\tau:] \to [:\upsilon:] \\
\texttt{mapP}\,\langle c,\, \texttt{nilP}\,\langle\rangle\rangle & = & \texttt{nilP}\,\langle\rangle \\
\texttt{mapP}\,\langle c,\, \texttt{repP}\,\langle n, x\rangle\rangle & = & \texttt{repP}\,\langle n,\; c\,\dagger\,x\rangle \\
\texttt{mapP}\,\langle c,\; xs \mathbin{+\!\!+} ys\rangle & = & \texttt{mapP}\,\langle c, xs\rangle \mathbin{+\!\!+} \texttt{mapP}\,\langle c, ys\rangle
\end{array}
$$

Other array primitives like `packP` are defined analogously by distributing over $(+\!\!+)$ and performing the obvious computations for `repP` and `nilP`.

$(\rhd)$

$$
\begin{array}{lll}
x & \rhd_{\langle () \rangle}\ y & = \mathtt{seq}\ \langle x,\ y \rangle \\
x & \rhd_{\langle \mathtt{Bool} \rangle}\ y & = \mathtt{seq}\ \langle x,\ y \rangle \\
x & \rhd_{\langle \mathtt{Int} \rangle}\ y & = \mathtt{seq}\ \langle x,\ y \rangle \\
\langle x_1, x_2 \rangle & \rhd_{\langle \tau_1 \times \tau_2 \rangle}\ \langle y_1, y_2 \rangle & = \langle x_1 \rhd_{\langle \tau_1 \rangle}\ y_1,\ x_2 \rhd_{\langle \tau_2 \rangle}\ y_2 \rangle \\
\mathtt{Left}\ x & \rhd_{\langle \tau_1 + \tau_2 \rangle}\ \mathtt{Left}\ y & = \mathtt{Left}\ (x \rhd_{\langle \tau_1 \rangle}\ y) \\
\mathtt{Left}\ x & \rhd_{\langle \tau_1 + \tau_2 \rangle}\ \mathtt{Right}\ y & = \mathtt{Right}\ y \\
\mathtt{Right}\ x & \rhd_{\langle \tau_1 + \tau_2 \rangle}\ \mathtt{Left}\ y & = \mathtt{Left}\ y \\
\mathtt{Right}\ x & \rhd_{\langle \tau_1 + \tau_2 \rangle}\ \mathtt{Right}\ y & = \mathtt{Right}\ (x \rhd_{\langle \tau_1 \rangle}\ y) \\
\mathtt{nilP}\ \langle\rangle & \rhd_{\langle [:\tau:] \rangle}\ xs & = xs \\
\mathtt{repP}\ \langle m, x \rangle & \rhd_{\langle [:\tau:] \rangle}\ \mathtt{nilP}\ \langle\rangle & = \mathtt{nilP}\ \langle\rangle \\
\mathtt{repP}\ \langle m, x \rangle & \rhd_{\langle [:\tau:] \rangle}\ \mathtt{repP}\ \langle n, y \rangle & = \mathtt{repP}\ \langle \mathtt{seq}\ \langle m, n \rangle,\ x \rhd_{\langle \tau \rangle}\ y \rangle \\
\mathtt{repP}\ \langle m, x \rangle & \rhd_{\langle [:\tau:] \rangle}\ (ys +\!\!+\!\!+ zs) & = (\mathtt{repP}\ \langle m, x \rangle \rhd_{\langle [:\tau:] \rangle}\ ys) +\!\!+\!\!+ (\mathtt{repP}\ \langle m, x \rangle \rhd_{\langle [:\tau:] \rangle}\ zs) \\
(xs +\!\!+\!\!+ ys) & \rhd_{\langle [:\tau:] \rangle}\ zs & = xs \rhd_{\langle [:\tau:] \rangle}\ (ys \rhd_{\langle [:\tau:] \rangle}\ zs)
\end{array}
$$

Appendix C

Primitives in $\lambda^{\mathcal{A}}$

lenP

$$
\begin{array}{lll}
\texttt{lenP}_{\langle\tau\rangle} & :: \tau.2 \to \texttt{Int} & \\
\texttt{lenP}_{\langle\underline{\langle\rangle}\rangle} & \langle n, u\rangle & = n \\
\texttt{lenP}_{\langle\underline{\texttt{Int}}\rangle} & \langle n, is\rangle & = n \\
\texttt{lenP}_{\langle\underline{\texttt{Bool}}\rangle} & \langle n, bs\rangle & = n \\
\texttt{lenP}_{\langle\tau_1 \underline{\times} \tau_2\rangle} & \langle n, p\rangle & = n \\
\texttt{lenP}_{\langle\tau_1 \underline{+} \tau_2\rangle} & \langle sel, p\rangle & = \texttt{lenP}_{\langle\underline{\texttt{Bool}}\rangle}\ sel \\
\texttt{lenP}_{\langle\tau_1 \underline{\to} \tau_2\rangle} & \langle n, p\rangle & = n \\
\texttt{lenP}_{\langle\tau_1 \underline{\Rightarrow} \tau_2\rangle} & \langle\!|p, \upsilon_1, \upsilon_2, \upsilon_3, e|\!\rangle & = \texttt{lenP}_{\langle\upsilon_1\rangle}\ e \\
\texttt{lenP}_{\langle\tau^{\uparrow}\rangle} & \langle ss, xs\rangle & = \texttt{lenP}_{\langle\underline{\texttt{Int}}\rangle}\ ss
\end{array}
$$

attachP

$$
\begin{array}{lll}
\texttt{attachP}_{\langle\tau\rangle} & :: \texttt{Int} \times \tau.2 \to \tau.2 & \\
\texttt{attachP}_{\langle\underline{\langle\rangle}\rangle} & \langle !n, xs\rangle & = \langle n, xs.2\rangle \\
\texttt{attachP}_{\langle\underline{\texttt{Int}}\rangle} & \langle !n, xs\rangle & = \langle n, xs.2\rangle \\
\texttt{attachP}_{\langle\underline{\texttt{Bool}}\rangle} & \langle !n, xs\rangle & = \langle n, xs.2\rangle \\
\texttt{attachP}_{\langle\tau_1 \underline{\times} \tau_2\rangle} & \langle !n, xs\rangle & = \langle n, xs.2\rangle \\
\texttt{attachP}_{\langle\tau_1 \underline{+} \tau_2\rangle} & \langle !n, xs\rangle & = \langle \texttt{attachP}_{\langle\underline{\texttt{Bool}}\rangle}\ \langle n, xs.1\rangle, xs.2\rangle \\
\texttt{attachP}_{\langle\tau_1 \underline{\to} \tau_2\rangle} & \langle !n, xs\rangle & = \langle n, xs.2\rangle \\
\texttt{attachP}_{\langle\tau_1 \underline{\Rightarrow} \tau_2\rangle} & \langle !n, \langle\!|p, \overline{\tau}, e|\!\rangle\rangle & = \langle\!|p, \mathcal{A}[\![\overline{\tau}]\!], \texttt{attachP}\ \langle n, e\rangle|\!\rangle \\
\texttt{attachP}_{\langle\tau^{\uparrow}\rangle} & \langle !n, xss\rangle & = \langle \texttt{attachP}_{\langle\underline{\texttt{Int}}\rangle}\ \langle n, xss.1\rangle, xss.2\rangle
\end{array}
$$

nilP

$$
\begin{array}{lll}
\mathtt{nilP}_{\langle\tau\rangle} & :: \langle\rangle \to \tau.2 & \\
\mathtt{nilP}_{\langle\underline{\langle\rangle}\rangle} & \langle\rangle & = \langle 0, \langle\rangle\rangle \\
\mathtt{nilP}_{\langle\underline{\mathtt{Int}}\rangle} & \langle\rangle & = \langle 0, \mathtt{nil}_{\mathtt{Int}}\rangle \\
\mathtt{nilP}_{\langle\underline{\mathtt{Bool}}\rangle} & \langle\rangle & = \langle 0, \mathtt{nil}_{\mathtt{Bool}}\rangle \\
\mathtt{nilP}_{\langle\tau_1 \underline{\times} \tau_2\rangle} & \langle\rangle & = \langle 0, \langle \mathtt{nilP}_{\langle\tau_1\rangle}\ \langle\rangle, \mathtt{nilP}_{\langle\tau_2\rangle}\ \langle\rangle\rangle\rangle \\
\mathtt{nilP}_{\langle\tau_1 \underline{+} \tau_2\rangle} & \langle\rangle & = \langle \mathtt{nilP}_{\langle\underline{\mathtt{Bool}}\rangle}\ \langle\rangle, \langle \mathtt{nilP}_{\langle\tau_1\rangle}\ \langle\rangle, \mathtt{nilP}_{\langle\tau_2\rangle}\ \langle\rangle\rangle\rangle \\
\mathtt{nilP}_{\langle\tau_1 \underline{\to} \tau_2\rangle} & \langle\rangle & = \langle 0, \langle \lambda\bullet : (\tau_1.1).\ \bot, \lambda\bullet : (\tau_1.2).\ \mathtt{nilP}_{\langle\tau_2\rangle}\ \langle\rangle\rangle\rangle \\
\mathtt{nilP}_{\langle\tau_1 \underline{\Rightarrow} \tau_2\rangle} & \langle\rangle & = \langle\!|(\mathtt{nilP}_{\langle\underline{\langle\rangle} \underline{\times} \tau_1 \underline{\to} \tau_2\rangle}\ \langle\rangle).2, \underline{\langle\rangle}, \tau_1, \tau_2, \mathtt{nilP}_{\langle\underline{\langle\rangle}\rangle}\ \langle\rangle|\rangle\!\rangle
\end{array}
$$

repP

$$
\begin{array}{lll}
\mathtt{repP}_{\langle\tau\rangle} & :: \mathtt{Int} \times \tau.1 \to \tau.2 & \\
\mathtt{repP}_{\langle\tau\rangle} & \langle 0, u\rangle & = \mathtt{nilP}_{\langle\tau\rangle}\ \langle\rangle \\
\mathtt{repP}_{\langle\underline{\langle\rangle}\rangle} & \langle !n, u\rangle & = \langle n,\ u\rangle \\
\mathtt{repP}_{\langle\underline{\mathtt{Int}}\rangle} & \langle !n, i\rangle & = \langle n,\ \mathtt{rep}_{\mathtt{Int}}\ \langle n, i\rangle\rangle \\
\mathtt{repP}_{\langle\underline{\mathtt{Bool}}\rangle} & \langle !n, b\rangle & = \langle n,\ \mathtt{rep}_{\mathtt{Bool}}\ \langle n, b\rangle\rangle \\
\mathtt{repP}_{\langle\tau_1 \underline{\times} \tau_2\rangle} & \langle !n, p\rangle & = \langle n,\ \langle \mathtt{repP}_{\langle\tau_1\rangle}\ \langle n, p.1\rangle,\ \mathtt{repP}_{\langle\tau_2\rangle}\ \langle n, p.2\rangle\rangle\rangle \\
\mathtt{repP}_{\langle\tau_1 \underline{+} \tau_2\rangle} & \langle !n, p\rangle & = \langle \mathtt{repP}_{\langle\mathtt{Bool}\rangle}\ \langle n,\ \mathit{isRight}\ p\rangle,\ \mathit{rep}\ \langle n, p\rangle\rangle \\
\multicolumn{3}{l}{\quad \textbf{where}} \\
\multicolumn{3}{l}{\quad\quad \mathit{isRight}\ (\mathtt{Left}\ x) = \mathit{False}} \\
\multicolumn{3}{l}{\quad\quad \mathit{isRight}\ (\mathtt{Right}\ y) = \mathit{True}} \\
\multicolumn{3}{l}{\quad\quad \mathit{rep}\ \langle n,\ \mathtt{Left}\ x\rangle = \langle \mathtt{repP}_{\langle\tau_1\rangle}\ \langle n, x\rangle,\ \mathtt{nilP}_{\langle\tau_2\rangle}\ \langle\rangle\rangle} \\
\multicolumn{3}{l}{\quad\quad \mathit{rep}\ \langle n,\ \mathtt{Right}\ y\rangle = \langle \mathtt{nilP}_{\langle\tau_1\rangle}\ \langle\rangle,\ \mathtt{repP}_{\langle\tau_2\rangle}\ \langle n, y\rangle\rangle} \\
\mathtt{repP}_{\langle\tau_1 \underline{\to} \tau_2\rangle} & \langle !n, p\rangle & = \langle n, p\rangle \\
\mathtt{repP}_{\langle\tau_1 \underline{\Rightarrow} \tau_2\rangle} & \langle !n, \langle\!|p, \upsilon_1, \upsilon_2, \upsilon_3, e|\rangle\!\rangle & = \langle\!|p, \upsilon_1, \upsilon_2, \upsilon_3, \mathtt{repP}_{\langle\upsilon_1\rangle}\ \langle n, e\rangle|\rangle\!\rangle \\
\mathtt{repP}_{\langle\tau^{\uparrow}\rangle} & \langle !n, xs\rangle & = \langle \mathtt{repP}_{\langle\underline{\mathtt{Int}}\rangle}\ \langle n, \mathtt{lenP}_{\langle\tau\rangle}\ xs\rangle,\ \mathtt{repeatP}_{\langle\tau\rangle}\ \langle n, xs\rangle\rangle
\end{array}
$$

$(+\!\!+)$

$$
\begin{array}{llll}
(+\!\!+_{\langle\tau\rangle}) :: \tau.2 \times \tau.2 \to \tau.2 & & & \\
\langle !m, x\rangle & +\!\!+_{\langle\underline{\langle\rangle}\rangle} & \langle !n, y\rangle & = \langle m+n, \mathtt{seq}_{\langle\underline{\langle\rangle}\rangle}\ \langle x, y\rangle\rangle \\
\langle !m, is\rangle & +\!\!+_{\langle\underline{\mathtt{Int}}\rangle} & \langle !n, js\rangle & = \langle m+n, \mathtt{concat}_{\mathtt{Int}}\ \langle is, js\rangle\rangle \\
\langle !m, bs\rangle & +\!\!+_{\langle\underline{\mathtt{Bool}}\rangle} & \langle !n, cs\rangle & = \langle m+n, \mathtt{concat}_{\mathtt{Bool}}\ \langle bs, cs\rangle\rangle \\
\langle !m, ps\rangle & +\!\!+_{\langle\tau_1 \underline{\times} \tau_2\rangle} & \langle !n, qs\rangle & = \langle m+n, \langle ps.1 +\!\!+_{\langle\tau_1\rangle} qs.1, ps.2 +\!\!+_{\langle\tau_2\rangle} qs.2\rangle\rangle \\
\langle !bs, ps\rangle & +\!\!+_{\langle\tau_1 \underline{\times} \tau_2\rangle} & \langle !cs, qs\rangle & = \langle bs +\!\!+_{\langle\underline{\mathtt{Bool}}\rangle} cs, \langle ps.1 +\!\!+_{\langle\tau_1\rangle} qs.1, ps.2 +\!\!+_{\langle\tau_2\rangle} qs.2\rangle\rangle \\
fs & +\!\!+_{\langle\tau_1 \underline{\to} \tau_2\rangle} & gs & = \bot \\
!c & +\!\!+_{\langle\tau_1 \underline{\Rightarrow} \tau_2\rangle} & !d & = \langle\!|\langle \mathtt{cappP}, \mathtt{cappP}^{\uparrow}\rangle, (\tau_1 \underline{\Rightarrow} \tau_2) \underline{+} (\tau_1 \underline{\Rightarrow} \tau_2), \tau_1, \tau_2, \\
 & & & \quad \langle \mathtt{repP}_{\langle\underline{\mathtt{Bool}}\rangle}\ \langle \mathtt{lenP}_{\langle\tau_1 \underline{\Rightarrow} \tau_2\rangle}\ c, \mathit{False}\rangle \\
 & & & \quad\quad +\!\!+_{\langle\underline{\mathtt{Bool}}\rangle}\ \langle \mathtt{lenP}_{\langle\tau_1 \underline{\Rightarrow} \tau_2\rangle}\ d, \mathit{True}\rangle\rangle, \\
 & & & \quad \langle c, d\rangle|\rangle\!\rangle \\
\langle !ss, xs\rangle & +\!\!+_{\langle\tau^{\uparrow}\rangle} & \langle !ts, ys\rangle & = \langle ss +\!\!+_{\langle\underline{\mathtt{Int}}\rangle} ts, xs +\!\!+_{\langle\tau\rangle} ys\rangle
\end{array}
$$

(!:)

$$
\begin{array}{lll}
(!:_{\langle\tau\rangle}) & :: \tau.2 \times \texttt{Int} \to \tau.1 & \\
\langle !n, u\rangle & !:_{\langle\underline{()}\rangle} & i = u \\
\langle !n, is\rangle & !:_{\langle\underline{\texttt{Int}}\rangle} & i = \texttt{index}_{\texttt{Int}}\ \langle is, i\rangle \\
\langle !n, bs\rangle & !:_{\langle\underline{\texttt{Bool}}\rangle} & i = \texttt{index}_{\texttt{Bool}}\ \langle bs, i\rangle \\
\langle !n, xs\rangle & !:_{\langle\tau_1 \underline{\times} \tau_2\rangle} & i = \langle xs.1\ !:_{\langle\tau_1\rangle}\ i, xs.2\ !:_{\langle\tau_2\rangle}\ i\rangle \\
\langle !bs, xs\rangle & !:_{\langle\tau_1 \underline{+} \tau_2\rangle} & i = \mathit{embed}\ \langle bs\ !:_{\langle\underline{\texttt{Bool}}\rangle}\ i,\ xs.1\ !:_{\langle\tau_1\rangle}\ i_1,\ xs.2\ !:_{\langle\tau_2\rangle}\ i_2\rangle \\
\quad \textbf{where} & & \\
\quad\quad \mathit{embed}\ \langle \mathit{False}, x, y\rangle & = \texttt{Left}_{\langle\tau_1,\tau_2\rangle}\ x & \\
\quad\quad \mathit{embed}\ \langle \mathit{True}, x, y\rangle & = \texttt{Right}_{\langle\tau_1,\tau_2\rangle}\ y & \\
\quad\quad i_1 & = \texttt{falsesP}\ (\texttt{takeP}_{\langle\underline{\texttt{Bool}}\rangle}\ \langle i, bs\rangle) & \\
\quad\quad i_2 & = \texttt{truesP}\ (\texttt{takeP}_{\langle\underline{\texttt{Bool}}\rangle}\ \langle i, bs\rangle) & \\
\langle !n, p\rangle & !:_{\langle\tau_1 \underline{\to} \tau_2\rangle} & i = p \\
\langle !n, \langle\!|p, \upsilon_1, \upsilon_2, \upsilon_3, xs|\!\rangle\rangle & !:_{\langle\tau_1 \underline{\Rightarrow} \tau_2\rangle} & i = \langle\!|p, \upsilon_1, \upsilon_2, \upsilon_3, xs\ !:_{\langle\upsilon_1\rangle}\ i|\!\rangle \\
\langle ss, xs\rangle & !:_{\langle\tau^{\uparrow}\rangle} & i = \texttt{attachP}_{\langle\tau\rangle}\ \langle n, \langle n, \mathit{chunk}\ \langle i, ss, xs\rangle\rangle\rangle \\
\quad \textbf{where} & & \\
\quad\quad n & = ss\ !:_{\langle\underline{\texttt{Int}}\rangle}\ i & \\
\quad\quad \mathit{chunk}\ \langle i, ss, xs\rangle & = \texttt{takeP}_{\langle\tau\rangle}\ \langle n, \texttt{dropP}_{\langle\tau\rangle}\ \langle \texttt{sumP}\ (\texttt{takeP}_{\langle\underline{\texttt{Int}}\rangle}\ \langle i, ss\rangle), xs\rangle\rangle &
\end{array}
$$

packP

$$
\begin{array}{lll}
\texttt{packP}_{\langle\tau\rangle} & :: \underline{\texttt{Bool}}.2 \times \tau.2 \to \tau.2 & \\
\texttt{packP}_{\langle\underline{()}\rangle} & \langle\langle !m, !bs\rangle, \langle !n, u\rangle\rangle & = \langle \texttt{truesP}\ \langle m, bs\rangle,\ u\rangle \\
\texttt{packP}_{\langle\underline{\texttt{Int}}\rangle} & \langle\langle !m, !bs\rangle, \langle !n, is\rangle\rangle & = \langle \texttt{truesP}\ \langle m, bs\rangle,\ \texttt{pack}_{\texttt{Int}}\ \langle bs, is\rangle\rangle \\
\texttt{packP}_{\langle\underline{\texttt{Bool}}\rangle} & \langle\langle !m, !bs\rangle, \langle !n, cs\rangle\rangle & = \langle \texttt{truesP}\ \langle m, bs\rangle,\ \texttt{pack}_{\texttt{Bool}}\ \langle bs, cs\rangle\rangle \\
\texttt{packP}_{\langle\tau_1 \times \tau_2\rangle} & \langle\langle !m, !bs\rangle, \langle !n, ps\rangle\rangle & = \langle \texttt{truesP}\ \langle m, bs\rangle,\ \langle \texttt{packP}_{\langle\tau_1\rangle}\ \langle\langle m, bs\rangle, ps.1\rangle, \\
 & & \qquad \texttt{packP}_{\langle\tau_2\rangle}\ \langle\langle m, bs\rangle, ps.2\rangle\rangle \\
\texttt{packP}_{\langle\tau_1 \times \tau_2\rangle} & \langle\langle !m, !bs\rangle, \langle !cs, ps\rangle\rangle & = \langle \texttt{packP}_{\langle\underline{\texttt{Bool}}\rangle}\ \langle\langle m, bs\rangle, cs\rangle,\ \langle \texttt{packP}_{\langle\tau_1\rangle}\ \langle ds, ps.1\rangle, \\
 & & \qquad \texttt{packP}_{\langle\tau_2\rangle}\ \langle es, ps.2\rangle\rangle \\
\quad \textbf{where} & & \\
\quad\quad ds & = \texttt{packP}_{\langle\underline{\texttt{Bool}}\rangle}\ \langle \texttt{not}^{\uparrow}\ cs, \langle m, bs\rangle\rangle & \\
\quad\quad es & = \texttt{packP}_{\langle\underline{\texttt{Bool}}\rangle}\ \langle cs, \langle m, bs\rangle\rangle & \\
\texttt{packP}_{\langle\tau_1 \underline{\to} \tau_2\rangle} & \langle\langle !m, !bs\rangle, \langle !n, p\rangle\rangle & = \langle \texttt{truesP}\ \langle !m, !bs\rangle, p\rangle \\
\texttt{packP}_{\langle\tau_1 \underline{\Rightarrow} \tau_2\rangle} & \langle\langle !m, !bs\rangle, \langle\!|p, \upsilon_1, \upsilon_2, \upsilon_3, xs|\!\rangle\rangle & = \langle\!|p, \upsilon_1, \upsilon_2, \upsilon_3, \texttt{packP}_{\langle\upsilon_1\rangle}\ \langle\langle m, bs\rangle, xs\rangle|\!\rangle \\
\texttt{packP}_{\langle\tau^{\uparrow}\rangle} & \langle\langle !m, !bs\rangle, \langle !ss, xs\rangle\rangle & = \langle \texttt{packP}_{\langle\underline{\texttt{Int}}\rangle}\ \langle\langle m, bs\rangle, ss\rangle,\ \texttt{packP}_{\langle\tau\rangle}\ \langle cs, xs\rangle\rangle \\
\quad \textbf{where} & & \\
\quad\quad cs & = \texttt{expandP}_{\langle\texttt{Bool}\rangle}\ \langle ss, \langle m, bs\rangle\rangle &
\end{array}
$$

zipP

$$\begin{aligned}&\texttt{zipP}_{\langle\tau,\upsilon\rangle} \;::\; \tau.2 \times \upsilon.2 \to (\tau \underline{\times} \upsilon).2\\ &\texttt{zipP}_{\langle\tau,\upsilon\rangle}\ \langle xs, ys\rangle \;=\; \texttt{attachP}_{\langle\tau\underline{\times}\upsilon\rangle}\ \langle \texttt{seq}_{\langle\underline{\texttt{Int}},\underline{\texttt{Int}}\rangle}\ \langle \texttt{lenP}_{\langle\tau\rangle}\ xs, \texttt{lenP}_{\langle\upsilon\rangle}\ ys\rangle,\\ &\qquad\qquad\qquad\qquad\qquad\qquad \langle \texttt{lenP}_{\langle\tau\rangle}\ xs,\ \langle xs, ys\rangle\rangle\rangle\end{aligned}$$

mapP

$$\begin{aligned}&\texttt{mapP}_{\langle\tau,\upsilon\rangle} \;::\; (\tau.1 \Rightarrow \upsilon.1) \times \tau.2 \to \upsilon.2\\ &\texttt{mapP}_{\langle\tau,\upsilon\rangle}\ \langle c, !xs\rangle \;=\; \texttt{repP}_{\langle\tau\underline{\Rightarrow}\upsilon\rangle}\ \langle \texttt{lenP}_{\langle\tau\rangle}\ xs, c\rangle\ \ddagger\ xs\end{aligned}$$

Bibliography

Martin Abadi, Luca Cardelli, Pierre Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 4(1):375–416, 1991.

Samson Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Welsey, Reading, MA, 1990.

Peter Achten, John H. G. van Groningen, and Rinus Plasmeijer. High level specification of I/O in functional languages. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 1–17, London, UK, 1993. Springer-Verlag. ISBN 3-540-19820-2.

S. Aditya, L. Arvind, L. Augustsson, J.-W. Maessen, and R. S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In P. Hudak, editor, *Proc. Haskell Workshop*, pages 35–49, 1995.

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986.

Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):575–631, September 1993.

Andrew W. Appel and Trevor Jim. Optimizing closure environment representations. Technical Report 168, Princeton University, Princeton, NJ, 1988.

Lennart Augustsson. Cayenne – a language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250, New York, NY, USA, 1998. ACM Press.

Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1984.

R. S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.

Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, 1990.

Guy E. Blelloch. Nesl: A Nested Data-Parallel Language (Version 3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University, September 1995.

Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3): 85–97, 1996.

Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, 1996.

Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.*, 8(2):119–134, 1990. ISSN 0743-7315.

F. Cardone and M. Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92(1):48–80, May 1991.

Manuel M. T. Chakravarty and Gabriele Keller. More types for nested data parallel programming. In Philip Wadler, editor, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 94–105. ACM Press, 2000.

Manuel M. T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. Nepal - nested data parallelism in Haskell. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 524–534, London, UK, 2001. Springer-Verlag. ISBN 3-540-42495-4.

Manuel M.T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2005a.

Manuel M.T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *Proceedings of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 1–13. ACM Press, 2005b.

A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, 1941.

M. Cole. Algorithmic skeletons. In K. Hammond and G. Michaelson, editors, *Research Directions in Parallel Functional Programming*, pages 289–303. Springer, 1999.

G. Cousineau, P. L. Curien, and M. Mauny. The categorical abstract machine. In *Functional programming languages and computer architecture*, pages 50–64, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

John Darlington, Yi ke Guo, Hing Wing To, and Jin Yang. Parallel skeletons for structured composition. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 19–28, New York, NY, USA, 1995. ACM Press.

John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, Oslo, 1971. North-Holland.

Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM TOPLAS*, 26(1):47–56, 2004.

M. Hamdan. A survey of cost models for algorithmic skeletons". Technical report, Heriot-Watt University, 1999.

K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer, 1999.

Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–176, 1998.

Y. Hayashi. *Shape-based cost analysis of skeletal parallel programs*. PhD thesis, College of Science and Engineering, University of Edinburgh, 2001.

C. A. Herrmann and C. Lengauer. Hdc: A higher-order language for divide-and-conquer. *Parallel Processing Letters*, 10(2–3):239–250, 2000.

J. Hill. *Data-Parallel Lazy Functional Programming*. PhD thesis, Queen Mary and Westfield College, University of London, 1994.

Ralf Hinze. A new approach to generic functional programming. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-125-9.

J. Holmerin and B. Lisper. Development of parallel algorithms in Data Field Haskell. In *EuroPar'00 – European Conference on Parallel Processing*, volume 1900 of *LNCS*, pages 762–766. Springer-Verlag, 2000.

J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

C. B. Jay. Shape analysis for parallel computing. In J. Darlington, editor, *Proceedings of the fourth international parallel computing workshop: Imperial College London, 25–26 September, 1995*, pages 287–298, 1995a.

C. Barry Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995b.

C. Barry Jay and Paul Steckler. The Functional Imperative: Shape! In *ESOP '98: Proceedings of the 7th European Symposium on Programming*, pages 139–153, London, UK, 1998. Springer-Verlag.

C.B. Jay and M. Sekanina. Shape checking of array programs. Technical report, University of Technology, Sydney, 1996.

Patricia Johann and Janis Voigtländer. Free theorems in the presence of *seq*. In Neil D. Jones and Xavier Leroy, editors, *31st Symposium on Principles of Programming Languages, Venice, Italy*, volume 39 of *SIGPLAN Notices*, pages 99–110. ACM Press, January 2004.

Gabriele Keller. *Transformation-based implementation of nested data parallelism for distributed memory machines*. PhD thesis, Technische Universität Berlin, 1999.

Gabriele Keller and Manuel M. T. Chakravarty. Flattening trees. In David Pritchard and Jeff Reeve, editors, *Euro-Par'98, Parallel Processing*, number 1470 in LNCS, pages 709 – 719. Springer-Verlag, 1998.

Gabriele Keller and Manuel M. T. Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In José Rolim et al., editors, *Parallel and Distributed Processing, Fourth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'99)*, number 1586 in Lecture Notes in Computer Science, pages 108–122, Berlin, Germany, 1999. Springer-Verlag.

Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 281–292, Austin, Texas, 1989.

John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 21st ADM Symposium on Principles of Programming Languages*, San Francisco, California, 1993. ACM Press.

John Launchbury and Ross Paterson. Parametricity and unboxing with unpointed types. In *ESOP '96: Proceedings of the 6th European Symposium on Programming Languages and Systems*, pages 204–218, London, UK, 1996. Springer-Verlag. ISBN 3-540-61055-3.

Roman Lechtchinsky, Manuel M. T. Chakravarty, and Gabriele Keller. Costing nested array codes. *Parallel Processing Letters*, 12(2):249–266, 2002.

Björn Lisper and Per Hammarlund. The data field model. Technical report, Mälardalen Real-Time Research Centre, Mälardalen University, 2001.

H-W. Loidl. The Virtual Shared Memory Performance of a Parallel Graph Reducer. In *DSM 2002 — International Workshop on Distributed Shared Memory on Clusters*, Berlin, Germany, May 2002.

Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Pena-Marí. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

G. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen, 1990.

John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.

Robin Milner. Fully abstract models of types lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 271–283, New York, NY, USA, 1996. ACM Press. ISBN 0-89791-769-3.

John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(3):470–502, 1988. ISSN 0164-0925.

Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38:114 – 117, 1965.

J. H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, Massachusets Institute of Technology, 1968.

G. Morrisett and R. Harper. Typed closure conversion for recursively-defined functions. In Carolyn Talcott, editor, *Higher Order Operational Techniques in Semantics*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

Eric Nocker, Sjaak Smetsers, Marko van Eekelen, and Rinus Plasmeijer. Concurrent clean. In Leeuwen Aarts and Rem, editors, *Proc. of Parallel Architectures and Languages Europe (PARLE '91)*, volume 505 of *LNCS*, pages 202–219. Springer-Verlag, 1991.

S. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, pages 249–257, 1993.

S. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98: A Non-strict, Purely Functional Language. Technical report, February 1999. URL `http://www.haskell.org`.

Simon L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *European Symposium on Programming*, pages 18–44, 1996.

Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666, Cambridge, Massachussets, USA, 1991. Springer-Verlag.

Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84, New York, NY, USA, 1993. ACM Press. ISBN 0-89791-560-7.

Wolf Pfannenstiel, Markus Dahm, Manuel M. T. Chakravarty, Stefan Jähnichen, Gabriele Keller, Friedrich-Wilhelm Schröer, and Martin Simons. Aspects of the compilation of nested parallel imperative languages. In J. Darlington, editor, *Proceedings of the Third International Conference on Programming Models for Massively Parallel Computers (MPPM '98)*, pages 102–109. IEEE Computer Society Press, 1998.

G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

F. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002.

R. Rangaswami. *A Cost Analysis for a Higher-order Parallel Programming Model*. PhD thesis, Department of Computer Science, Edinburgh University, 1996.

John H. Reppy. CML: A higher concurrent language. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 293–305, New York, NY, USA, 1991. ACM Press.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, New York, NY, USA, 1972. ACM Press.

John C. Reynolds. Towards a theory of type structure. In *Colloq. sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423, London, UK, 1974. Springer-Verlag.

James Riely and Jan Prins. Flattening is an improvement. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 360–376, London, UK, 2000. Springer-Verlag.

S.-B. Scholz. On defining application-specific high-level array operations by means of shape-invariant programming facilities. In *Proceedings of APL'98*, pages 40–45. ASM Press, 1998.

Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *LISP and Functional Programming*, pages 150–161, 1994.

Guy L. Steele, Jr. Rabbit: A compiler for Scheme. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

P. W. Trinder, H. W. Loidl, and R.F. Pointon. Parallel and distributed Haskells. *Journal of Functional Programming*, 12(4–5):469–510, 2002.

Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., A. S. Partridge, and Simon L. Peyton Jones. GUM: A portable parallel implementation of haskell. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 79–88, 1996.

P. W. Trindler, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.

Philip Wadler. Theorems for free! In *FPCA'89: Proceedings 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359, New York, 1989. ACM Press.

C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.

Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 435–445, 1994.

H. H. Wang. A parallel method for tridiagonal equations. *ACM Transactions on Mathematical Software*, 7(2):170–183, June 1981. ISSN 0098-3500.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, 1999.