

Aus dem Institut für Softwaretechnik und Programmiersprachen
der Universität zu Lübeck
Direktor: Prof. Dr. Martin Leucker

QUBE – Array Programming with Dependent Types

Inauguraldissertation
zur
Erlangung der Doktorwürde
der Universität zu Lübeck
Aus der Sektion Informatik/Technik

vorgelegt von
Dipl.-Inf. Kai Trojahnner
aus Flensburg

Lübeck 2011

1. Berichterstatter: Prof. Dr. Till Tantau
2. Berichterstatter: Dr. Clemens Grell
Vorsitzender des Prüfungsausschusses: Prof. Dr. Martin Leucker
Tag der mündlichen Prüfung: 1. Juni 2011

Walter Dosch

1947 – 2010

Zusammenfassung

Arrayprogrammiersprachen wie APL oder MATLAB verwenden multidimensionale Arrays als grundlegende Datenstrukturen. Rank-generische Operationen sind gleichermaßen auf Vektoren, Matrizen, und höherdimensionale Arrays anwendbar. Bei vielen Operationen unterliegen die Operanden jedoch spezifischen Einschränkungen bezüglich der Typen ihrer Elemente und ihrer Form. Beispielsweise überprüft die MATLAB-Operation $A + B$ zur Laufzeit, ob beide Operanden die gleiche Form haben und ob die Addition der Elemente definiert ist. Im Falle einer Anwendung auf inkompatible Operanden wird die Ausführung des gesamten Programms abgebrochen.

In dieser Arbeit stelle ich die Programmiersprache QUBE vor, welche die statische Verifikation von Arrayprogrammen unterstützt, so dass Fehler schon während der Übersetzung entdeckt werden. Dazu verwendet QUBE abhängige Typen, die sowohl den Elementtyp als auch die Form eines Arrays beschreiben. Weil Arrays unterschiedlicher Form verschiedene Typen haben, können die erlaubten Argumente einer Funktion genau angegeben werden. Das Typsystem kann Elementtypfehler, Formfehler und Indexfehler ausschließen.

Als formales Modell von QUBE definiere ich die Kernsprache $QUBE_{CORE}$. Eine operationelle Semantik definiert die korrekte Auswertung von $QUBE_{CORE}$ und die möglichen Laufzeitfehler. Das Typsystem von $QUBE_{CORE}$ wird durch eine Menge von Ableitungsregeln für wohlgeformte Typen, die Subtyprelation und die Typüberprüfung von Ausdrücken beschrieben. Ich beweise, dass bei der Auswertung von wohlgetypten Ausdrücken keine Laufzeitfehlern auftreten können.

Um zu untersuchen, wie abhängige Typen genutzt werden können, um aus rank-generischen Spezifikationen effiziente Programme zu erzeugen, habe ich zusammen mit meinen Studenten einen Übersetzer für QUBE konstruiert. Der Übersetzer verwendet für die Typüberprüfung einen Theorembeweiser für das SMT Problem. Da QUBE-Programme statisch verifiziert werden, brauchen sie keine dynamischen Tests wie etwa Bereichstests durchzuführen. Multidimensionale Arrays werden als reine Sequenzen von Daten ohne Typannotationen oder Formdeskriptoren repräsentiert. Erste Experimente zeigen, dass für einige interessante Benchmarks QUBE-Programme ähnlich schnell wie C-Programme sind.

Abstract

Array programming languages such as APL or MATLAB use multidimensional arrays as the primary data structures. Rank-generic operations apply transparently to vectors, matrices, and arrays with an even higher rank. Often, these operations require that the ranks, shapes, and the elements of their arguments satisfy certain constraints. For example, in MATLAB, the element-wise addition $A + B$ dynamically checks that both arrays have the same shape. In case of an improper application, the entire program aborts with an error message.

In this thesis, I present QUBE, a new programming language that checks array programs at compile time, such that errors are detected before a program is run. For this purpose, QUBE employs an advanced type system based on dependent types, i. e., types that depend on values. Since dependent types distinguish between arrays of different shapes, the allowed arguments of a function can be precisely characterised. The type system is sufficiently expressive to statically rule out base type errors, shape errors, and even array boundary violations.

As a formal model of QUBE, I define the core language $\text{QUBE}_{\text{CORE}}$. An operational semantics defines both the proper evaluation of $\text{QUBE}_{\text{CORE}}$ as well as the potential run-time errors. The type system of $\text{QUBE}_{\text{CORE}}$ is described by a set of inference rules that formally specify well-formed types, the subtype relation, and type checking of $\text{QUBE}_{\text{CORE}}$ expressions. I provide a proof that $\text{QUBE}_{\text{CORE}}$ is type-safe, i. e., evaluating a well-typed expression will not cause a run-time error.

To explore how the power of dependent types can be harnessed to generate efficient code from rank-generic programs, I, with help from my students, have constructed a compiler for QUBE. The compiler performs type checking in collaboration with an SMT solver. Due to static verification, programs do not need to perform any dynamic checks such as array bounds checks. Moreover, multidimensional arrays are represented as mere sequences of data without additional type tags or shape descriptors. Early experiments show that for some interesting benchmarks, the run-time performance of QUBE programs is comparable with handwritten C code.

Acknowledgements

I dedicate this thesis to Walter Dosch, who was my initial thesis advisor. When I was a freshman at the University of Lübeck, he introduced me to the wonderful world of functional programming. For my dissertation, he encouraged me to develop my own programming language, then called the *Lübeck Array Language*, and patiently guided me through my research. Walter Dosch passed away far too soon in August 2010.

In July 2010, Till Tantau graciously consented to take care of me in the final stages of my thesis. I especially thank him for taking over and providing helpful comments as well as inspiring discussions.

A great deal of thanks goes to Clemens Grell who initiated me to the SAC array programming language and compiler project. He supervised my diploma thesis and motivated me to make my own contributions to the field.

Several students helped me implement the QUBE compiler in its various incarnations. Without Florian Büther, the compiler surely wouldn't be here, but also Markus Weigel, Johannes Blume, and Sebastian Hungerecker made valuable contributions.

I thank my colleagues from the Institute of Software Technology and Programming Languages, namely Bastian Dölle, Hedwig Hellkamp, Annette Stümpel, and Dietmar Wolf, for all the good times and the cakes we had together. Martin Leucker kindly provided me with an office after my contract with the university expired.

Markus Hinkelmann deserves thanks for proofreading this document and for some very enlightening discussions on theoretical computer science.

I am particularly grateful to my parents who supported me during my studies. My wife Silke deserves the most thanks. For cheering me up when my research stalled, for celebrating with me when there was a breakthrough, and for motivating me to finish this thesis at all times.

Contents

1	Introduction	1
I	Foundations	11
2	The λ-Calculus and Type Systems	13
2.1	The λ -Calculus	14
2.2	An Applied λ -Calculus	17
2.3	Simple Types	20
3	Decidable First-Order Theories	29
3.1	Propositional Logic	30
3.2	First-Order Logic	33
3.3	Quantifier-Free Fragments of First-Order Theories	36
3.4	Array Properties	38
II	A Formal Treatment of QUBE	41
4	A Core Language for Array Programming	43
4.1	QUBE $_{\lambda}$: a Functional Foundation	45
4.2	QUBE $_{\rightarrow}$: Integer Vectors	50
4.3	QUBE $_{\square}$: Multidimensional Arrays	53
4.4	Properties of Evaluation	59

5	Type Checking QUBE_{CORE}	61
5.1	Well-Formed Types	63
5.2	Joining Structured Vectors	64
5.3	Subtyping	66
5.4	Type Checking	67
5.5	Correctness of Type Checking	74
5.6	SMT-Based Validity Checking	91
III	The QUBE Programming Language	97
6	The QUBE Programming Language	99
6.1	Expression Syntax	99
6.2	Module System	102
6.3	Stateful Computations	104
7	Language Implementation	107
7.1	Design of the QUBE Compiler	107
7.2	Compilation at a Glance	109
7.3	Descriptor-Free Array Representation	112
8	Rank-Generic Array Operations	117
8.1	Type Abbreviations	118
8.2	Element-Wise Computations	118
8.3	Selection Functions	120
8.4	Structural Functions	121
8.5	Higher-Order Functions	124
9	Evaluation	127
9.1	Matrix Multiplication and Inner Product	127
9.2	Rank-Generic Convolution	129
9.3	Quicksort	132
10	Conclusion and Future Work	135

1

Introduction

Some “very high-level languages”, like APL, are normally interpreted because there are many things about the data, such as size and shape of arrays, that cannot be deduced at compile time.

Aho, Sethi, Ullman: *Compilers: Principles, Techniques, and Tools* [1]

This thesis presents QUBE, a new programming language that combines the expressiveness of array programming with the power of dependent types. I claim that this combination makes particular sense for three reasons: First, dependent array types can distinguish between arrays of different shapes, allowing array operations to be assigned accurate types that precisely specify the allowed arguments and how the type of the result depends on them. Second, dependent types provide static safety guarantees for array programs. QUBE uses a combination of type checking and automatic theorem proving to statically rule out large classes of program errors, in particular array boundary violations. Third, the information provided by dependent types is sufficient to compile array programs into efficient target programs, even if the shapes of the arrays involved cannot be determined at compile time. By virtue of static verification, QUBE programs do not need to check whether an operation has been applied to appropriate arguments or whether an array is accessed outside its boundaries. Furthermore, multidimensional arrays can be represented as mere sequences of elements without additional type or shape tags.

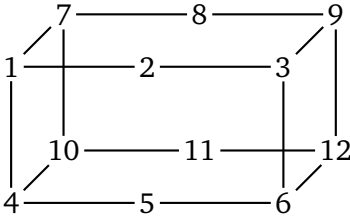
Array	Rank	Shape vector
1	0	[]
[1 2 3]	1	[3]
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	2	[2 3]
	3	[2 2 3]

Figure 1.1: Ranks and shape vectors

Array programming languages like APL [55, 35], J [57], MATLAB [76], ZPL [25], and SAC [89], use multidimensional arrays as the primary data structures. Such arrays may be vectors, matrices, or tensors with an even higher number of axes; degenerate arrays without any axes are isomorphic to scalar values. Formally, an *r-dimensional array* organises a collection of homogeneous elements along *r* orthogonal axes. Each element is identified by a vector of *r* natural numbers called the element's *index vector*.

Multidimensional arrays are characterised by two essential properties, namely their rank and their shape vector. The rank of a multidimensional array is a natural number that denotes its number of axes, i. e., the common length of the elements' index vectors. The shape vector is a vector of natural numbers that describe the extent of each axis. It is thus an element-wise upper bound for all index vectors into the array; the product of the shape components equals the number of array elements. Figure 1.1 shows the basic properties of some example arrays. Rank zero arrays such as **1** do not have any axes and hence their shape vector is empty. Note that shape and index vectors are themselves arrays of rank one.

Array programming is renowned for its conciseness. Programs are composed from general-purpose array operations that apply to entire arrays rather than individual elements. In particular, many array operations are rank-generic, i. e., they are applicable to arrays with an arbitrary number of axes, each of which may have arbitrary length. For example, the expression $A + B$ computes the element-wise sum of the arrays *A* and *B* without explicit loops over the elements. Similarly, the APL inner product $A +. * B$ generalises matrix multiplication to ar-

rays of arbitrary (positive) rank. The high abstraction level of the individual operations allows the programmer to solve problems in large conceptual steps with very few lines of code. Often, programs that would require several pages of code in conventional programming languages can be expressed in a one-line array program. Moreover, many array operations are inherently data parallel as they homogeneously apply to a large number of elements. This makes array programs well-suited for implicit parallelisation [39]. The recent advent of multi-core processors [93] has created new interest in the paradigm [24, 23, 38].

Despite its power and expressiveness, rank-generic programming also introduces a host of subtle programming pitfalls. Typically, array operations can only be evaluated if the arguments satisfy specific constraints between ranks, shapes, and even element values. For example, element-wise arithmetic can only be performed on arrays that have the same number of axes, the same shape, and elements that are compatible with the operation at hand. Similarly, the inner product requires that the last axis of the first array is as long as the first axis of the second array. Even array indexing is more intricate in a rank-generic setting. Array elements are indexed by means of an integer vector whose length must match the number of array axes. Furthermore, each index must range between zero and the corresponding element of the array shape.

Interpreted array languages like APL, J, and MATLAB are dynamically typed. When the interpreter encounters an array operation, it checks whether the operation has been applied to appropriate arguments and, if so, performs the computation, typically by invoking a (well-optimised) native implementation of the operation. In case of an improper application, the program aborts with an error message. The combination of interpretation and dynamic typing allows for rapid program development. The programmer can interact with the programming system, new code can be loaded at run-time, and even an `eval` function, that allows arbitrary data to be executed as code, is a common feature of interpreted array languages. However, the absence of static types makes bugs hard to find because errors are typically reported at a location different from where the programming mistake was made. Thus, long-running or safety-critical applications must be carefully coded and thoroughly tested in order to find bugs and avoid that the program terminates abruptly, potentially after it has been deployed.

Beyond safety considerations, dynamic checks also carry a performance penalty. The run-time system must tag arrays with type and shape information so that these properties can be dynamically inspected. Checking and tagging both add a constant overhead to array operations that can even outweigh the actual computation when small arrays are processed. This is particularly unfavourable for algorithms that loop over individual array elements instead of applying operations to entire arrays.

To counter these issues, compiled array languages such as FORTRAN-90, HPF, FISH, ZPL, and SAC have been developed. Naturally, compilation rules out interactive program development and an `eval` function. But unlike interpreters, compilers can employ a broad range of optimisations to improve program efficiency. In particular, scalars can often be identified statically so that they can be stored in processor registers rather than on the heap and applications of complex array operations can be replaced with simple processor instructions. By means of type checking, compilers can perform some of the required consistency checks at compile time, which eliminates the need to check for these properties dynamically. The amount of bugs that can be statically found and thus the amount of dynamic checks that can be avoided depends on the strength of the type system.

FISH [59, 58] is a compiled array programming language with an impure call-by-value semantics and support for polymorphic higher-order functions. Multidimensional arrays are supported as homogenous nestings of vectors. By means of *shape analysis*, the FISH compiler determines the shapes of all intermediate array expressions such that appropriate amounts of memory can be allocated statically. To describe array shapes, FISH uses expressions of a special kind *size*, which are evaluated by the compiler. Each function `f` is accompanied by a shape function `#f` that maps the size of the arguments to the size of the result. Shape analysis proceeds by first inlining all functions and then evaluating all shape functions. FISH rejects all programs that contain non-constant array shapes. Since arrays are indexed by run-time integers whereas array sizes are determined at compile time, shape-analysis is insufficient to statically capture array boundary violations. Still, applications of functions to arguments of incompatible shape will be reported as shape errors, so that combinations of functions that are free of array bounds errors will produce programs without bounds errors.

SAC [42, 89, 94] is a compiled array programming language with a pure call-by-value semantics. The design of SAC aims at high run-time performance and automatic parallelisation [39]. In SAC, multidimensional arrays are the only available data structures, even scalar values are considered arrays [90]. The language provides just a few array operations as built-in functions. Rank-generic array operations are specified by means of a powerful array comprehension called `WITH-loop`. An extensive standard library provides numerous high-level, general-purpose array operations whose implementations are based on `WITH-loops`. SAC programs are typically assembled from these building blocks. This style of programming leads to lean and concise specifications, but also introduces many intermediate arrays. To achieve competitive run-times, the SAC compiler employs a host of powerful program optimisations that chiefly aim at avoiding the creation of temporary arrays whenever possible [44, 88, 40].

More liberal than FISH, the type system of SAC classifies arrays with a hierar-

chy of types [89]. While the type of array elements is always monomorphic, arrays are described at four different levels of accuracy: there are types for arrays of statically known value (AKV, for example `int[2,2]{1,2,3,4}`), types for arrays of known shape (AKS, for example `int[2,2]`), types for arrays of known dimensionality (AKD, for example `int[.,.]`), and types for arrays of unknown dimensionality (AUD, for example `int[*]`). Via subtyping, an expression of some specific type can be safely used in a position where a less specific type is required. In contrast, when an expression of some unspecific type is used in a position where a more specific type is expected, the compiler inserts a run-time shape check that potentially aborts the program with an error message.

The amount of program errors that can be statically detected by the SAC type system corresponds to the available type information. Array boundary violations will only be captured at compile time when the array has at most an AKS type and when the index vector has an AKV type. Shape errors will only be found when both the actual type and the expected type are at most AKS types. Similarly, rank errors will only be reported by the compiler if both types are at most AKD types. When an expression has an AUD type, or when an AUD type is expected at some position, only base type errors can be detected statically.

Run-time checks that stay prevalent in compiled code cause overhead both directly through their mere execution and indirectly by hampering program optimisation. To improve the available type information and reduce run-time checks, the SAC compiler uses code specialisation [43] and partial evaluation techniques [51]. *Symbolic array attributes* serve as a uniform scheme to infer and represent structural information in shape-generic array programs such that it may be used by optimisations [98]. Recently, the SAC compiler has been extended with a framework for dynamic recompilation at run-time when all structural properties of arrays are known [46].

This thesis contributes QUBE, a new array programming language that verifies rank-generic array programs entirely statically such that no dynamic checks are necessary. For this purpose, QUBE features an advanced type system based on dependent types, i. e., types that are parameterised by values [72, 7, 36, 86]. Like SAC, QUBE is a compiled language with a pure call-by-value semantics that only provides the most essential array operations as language primitives. However, QUBE does not follow the *everything is an array* paradigm. The type system distinguishes between multidimensional arrays and other data structures, namely unboxed scalars, tuples, and first-class functions.

The type system of QUBE classifies arrays using types of the form $[T|e]$ where the type T describes the array elements and the expression e is an integer vector that represents the array shape. For example, a 2×3 integer matrix has type $[\text{int}|[2,3]]$, but also type $[\text{int}|[1+1,1+2]]$, because $[1+1,1+2]$ evaluates to

[2, 3]. Types of the form `intvec e` describe integer vectors of length e that are used as shape vectors and index vectors.

The potential run-time values of an expression can be restricted with refinement types [36, 86]. A type of the form $\{x : T \mid e\}$ describes the set of all values x of type T for which the expression e evaluates to `true`. For example, `nat` is the type of natural numbers and `index n` is the type of all integers that are valid indices into a vector of length n .

```
type nat = { x:int | 0 <= x }
type index n:nat = { x:int | 0 <= x & x < n }
```

Vector types can be refined, too. In such refinements, the vector predicate `vfa v_1, \dots, v_m p` (vector for all) expresses that a property p holds for all corresponding elements of some vectors v_1, \dots, v_m . The property p has the form $(x_1, \dots, x_m \rightarrow e)$ where the variable x_i represents an element from the vector v_i in the boolean expression e . For example, the type `natvec n`, which describes vectors of natural numbers of length n , is defined in terms of `vfa`. Similarly, `indexvec r s` describes valid index vectors into an array of rank r and shape s .

```
type natvec n:nat = { x:intvec n | vfa x (xi → 0 <= xi) }
type indexvec r:nat s:(natvec r) =
  { x:intvec r | vfa x,s (xi,si → 0 <= xi & xi < si) }
```

The type system is sufficient to statically rule out array boundary violations. For all accesses `a.[x]` into an array `a` of rank r and shape s , the type checker verifies that the index x has type `indexvec r s`. For example, the type system rejects the following definition of `foo` because the array access will fail for $m < 2$ or $n < 3$.

```
let foo m:nat n:nat a:[int|[m,n]] = a.[ [1,2] ]
```

```
Type error in file test/abc.q, line 1, column 36:
Index may violate the array boundaries.
```

Dependent function types of the form $x : T \rightarrow T_x$ allow the result type T_x of a function to depend on the argument value x . Together with refinement types and array types, dependent function types can be used to precisely specify the constraints a function imposes on the ranks, shapes, and values of its arguments and the result. Based on this information, the type checker can statically detect base type errors, rank errors, shape errors, and illegal argument values, even in a rank-generic setting. For example, the type of the rank-generic array addition `add` makes clear that the function takes two integer arrays of some arbitrary but equal shape s and yields a result of the same shape.

```
val add : r:nat. s:(natvec r). [int|s]. [int|s] → [int|s]
```

The type of the inner product `ip`, which generalises matrix multiplication to arrays of arbitrary rank, makes the constraints on the arguments explicit: the last axis of the first array must be as long as the first axis of the second array.

```
val ip : m:nat. n:nat. r:(natvec m). s:nat. t:(natvec n).
    [int|r,[s]]. [int|[s],t] → [int|r,t]
```

Dual to dependent function types, QUBE supports dependent tuple types of the form $(x:T, T_x)$, i. e., tuples where the type of the second component depends on the value of the first. Dependent tuples are useful to form packages of arrays and their shape properties, so that arrays of different shape can be stored in a common data structure. QUBE uses dependent tuples to represent strings as pairs of an integer that describes the string length and a vector of characters. The command-line arguments passed to a program are represented as a dependent tuple that combines the number of arguments with an array of strings.

```
type string = ( len:nat, [char|[len]])
val cmdline_args : ( argc: nat, [string|[argc]])
```

As pointed out above, the dependent type of an expression is not unique. The type of an array may be $[int|[2,3]]$, or $[int|[1+1,1+2]]$, or even $[int|f\ x]$ if the expression $f\ x$ happens to evaluate to $[2,3]$. In order to decide whether the array types $[T|e_1]$ and $[T|e_2]$ are equal, the type checker must decide whether the expressions e_1 and e_2 denote the same value. Furthermore, to check whether a refinement type $\{x:T | e_1\}$ is a subtype of some other refinement type $\{x:T | e_2\}$, the type checker must prove that all values x that satisfy e_1 also satisfy e_2 . Since arbitrary expressions are allowed to appear in types, both problems are undecidable.

To sidestep this problem, the QUBE type checker encodes the constraints to be checked as first-order formulas in the decidable fragment of uninterpreted functions and linear arithmetic. The resulting formulas are then verified in collaboration with the YICES theorem prover [33]. The encoding is sound but, naturally, incomplete: if an encoded formula is valid, the original constraint is valid, too. However, not all valid constraints are encoded as valid formulas. In effect, type checking behaves conservatively. It rules out all programs with type errors, but it also rejects some programs that would actually behave well at run-time.

The type system of QUBE provides static guarantees that well-typed array programs do not cause run-time errors. Beyond rendering dynamic checks obsolete, the type system also allows for a particularly efficient run-time representation of multidimensional arrays. To make ranks and shape vectors dynamically accessible, for example to compute memory locations of array elements, language implementations typically associate each array with a shape descriptor. The implementation of QUBE dispenses with shape descriptors and represents arrays as mere sequences of elements [97]. The compiler uses information from the array types to statically annotate programs with expressions that evaluate to ranks and shape vectors wherever these values will be required at run-time.

Other Related Work The contribution of QUBE is positioned at the intersection of array programming, functional programming, and dependently typed programming. The following paragraphs briefly outline work from the different areas of research related to this thesis.

Attempts have been made to compile the classical array languages like APL and MATLAB in order to improve program efficiency, although the flexibility of these languages renders compilation difficult. The APEX compiler [11] translates an extended subset of ISO APL into SISAL, a functional vector language. Recently, APEX has been modified to target SAC, instead. A compiler for MATLAB is commercially available from MathWorks. However, instead of improving efficiency, the goals of the MATLAB compiler are chiefly to create standalone executables or libraries from MATLAB programs, and code obfuscation. Rediscovering array properties for better compilation of untyped array languages such as MATLAB is an area of ongoing research, see for example [31, 74, 61]. In QUBE, the array types contain everything the programmer knows about the structural properties of the program, eliminating the need for such work.

The field of functional array programming was pioneered by SISAL [21] and NESL [14], although neither language supports rank-generic programming. SISAL demonstrated that functional array programming and implicit parallelisation can achieve competitive run-time performance, despite the aggregate update problem [53]. While SISAL restricts itself to (one-dimensional) vectors of homogeneously nested vectors, NESL also supports irregularly nested vectors. Inspired by NESL, work has been going on to integrate nested data-parallelism into HASKELL [24, 23]. Recently, support for rank-generic programming has been added to DATA PARALLEL HASKELL as a library [63]. Although the HASKELL type system cannot detect array boundary violations, many rank and shape errors can be detected statically.

Another field of related work is the research area of dependently typed programming [92]. Dependent types naturally lend themselves for describing arrays as they allow the use of (dynamic) terms to index within families of types. Indeed, the classical example for dependently typed programming is the index family of vectors from which an element with a particular length is selected [83]. The expressive power of dependent types renders deciding type equality generally undecidable as it boils down to deciding whether any two expressions denote the same value. For example, CAYENNE [4] is a fully dependently typed language. Its type system is undecidable and it lacks phase distinction. Both problems can be overcome by restricting the type language as done in EPIGRAM [73, 2], which rules out general recursion in type-forming expressions to retain decidability.

Languages with light-weight forms of dependent types such as DML [102], *applied type system* [100], and *indexed types* [103] have been developed. These

languages allow indexing into type families only with compile-time expressions of certain linear *index sorts*. The problem of deciding whether two types are equal or in a subtype relation is reduced to constraint solving on these sorts, which is decidable. Light-weight dependent types are sufficient to rule out array boundary violations for arrays of fixed rank [101]. An early version of QUBE extended indexed types for rank-generic programming [96]. To automatically infer dependent types for programs, *logically qualified data types*, or LIQUID TYPES, that combine Hindley-Milner type inference with predicate abstraction have been proposed [86].

Outline The remainder of this thesis is organised in three parts.

The first part covers the theoretical foundations on which this work relies. Chapter 2 briefly recapitulates essential concepts of programming languages and type systems. The chapter presents formalisms based on the λ -calculus that allow us to reason about syntax, semantics, and the type system of a language with mathematical rigour. Chapter 3 gives a brief introduction to propositional logic, first-order logic, and the relevant fragments of first-order theories that will be relevant in the remainder of the thesis.

The second parts formally discusses the syntax, semantics, and the type system of QUBE. To achieve a rigorous presentation, the discussion focusses on $\text{QUBE}_{\text{CORE}}$, a simplified language that captures the essential concepts of QUBE without syntactic sugar or convenience features. Chapter 4 presents the syntax and operational semantics of $\text{QUBE}_{\text{CORE}}$ and shows that evaluation of $\text{QUBE}_{\text{CORE}}$ expressions is deterministic. Chapter 5 explains the type system of $\text{QUBE}_{\text{CORE}}$ and provides a formal proof of type safety. The main results are a progress and a preservation theorem for $\text{QUBE}_{\text{CORE}}$. These state that a well-typed expression is either a value or can be further evaluated, and that the type of an expression is preserved under evaluation. Furthermore, the chapter describes how type constraints are encoded as logical formulas.

The third part presents the implementation of QUBE. Chapter 6 describes the syntax of the actual QUBE programming language. Chapter 7 explains the compilation process that translates QUBE programs via a series of intermediate representations into code for the Low-Level Virtual Machine (LLVM), which in turn emits native code. Chapter 8 illustrates the expressiveness of QUBE. A host of rank-generic array operations, that are typically provided as built-ins primitives by interpreted array languages, are defined as type-safe QUBE functions. Chapter 9 evaluates the QUBE language and its implementation by means of more complex example programs. Finally, Chapter 10 concludes the thesis and outlines some directions for future work.

Part I

Foundations

2

The λ -Calculus and Type Systems

This chapter gives an introduction to the formal treatment of programming languages and type systems. The presented formalisms allow us to specify and reason about syntax, semantics, and typing rules of a language with mathematical rigour. The presentation recapitulates concepts from textbooks on programming language theory, mainly from [82] but also from [83, 64].

The λ -calculus [26, 5, 8, 52, 64] is a formal system which was introduced by Church and Kleene in the 1930s to investigate function definition, function application, and recursion. In the 1960s, Landin recognised [67] that the λ -calculus captures the essence of many programming languages and that their more elaborate features may be understood by explaining them in terms of the calculus.

λ -calculi exist in untyped and typed flavours. The untyped λ -calculus was influential in the development of early functional programming languages such as LISP. Typed λ -calculi form the foundation of modern type systems used in both typed programming languages and mechanical proof assistants. By classifying expressions according to the kinds of values they compute, type systems help to identify program errors early in the development cycle [82].

The remainder of this chapter is structured as follows: Section 2.1 introduces the most essential definitions and properties of the untyped λ -calculus. Section 2.2 presents an applied λ -calculus with a call-by-value semantics that resembles a simple programming language and examines its basic properties. Section 2.3 extends the applied calculus with simple types that warrant orderly evaluation of expressions.

2.1 The λ -Calculus

This section formally introduces essentials of the λ -calculus by explaining its grammar, notational conventions, capture-avoiding substitution, β -reduction, normal forms, and fundamental evaluation strategies.

In the following, the metavariables x, y range over variables, and the metavariables e, e', e_i represent λ -expressions.

Definition 2.1 (λ -calculus)

For a countably infinite set of variables V , the set of expressions in the λ -calculus is defined by the following grammar.

$$\begin{array}{lll}
 e ::= & x & \text{(Variable)} \\
 & \lambda x. e & \text{(Abstraction)} \\
 & e e & \text{(Application)}
 \end{array}$$

In the λ -calculus, all variables x are themselves expressions. The *abstraction* $\lambda x. e$ abstracts a variable x from the *body* e , essentially creating a function that depends on x . The *application* $e_1 e_2$ applies the *operator* e_1 to the *operand* e_2 .

The above syntax of the λ -calculus is given as an abstract syntax and thus is inherently ambiguous. We use parentheses to disambiguate the structure of expressions, subject to the following conventions: applications associate to the left, whereas abstraction bodies extend to the utmost right, for example $a b c$ stands for the same expression as $((a b) c)$ and $\lambda f. \lambda x. f x$ abbreviates $(\lambda f. (\lambda x. (f x)))$.

An abstraction $\lambda x. e$ *binds* the variable x in the body e . The latter is also referred to as the *scope* of the *binder* λx . An occurrence of a variable x is said to be *bound* if it appears inside the scope of a binder λx otherwise the occurrence is *free*.

Definition 2.2 (Free variables)

The set $\mathcal{FV}(e)$ of free variables of an expression e is defined inductively:

$$\begin{array}{ll}
 \mathcal{FV}(x) & = \{x\} \\
 \mathcal{FV}(\lambda x. e) & = \mathcal{FV}(e) \setminus \{x\} \\
 \mathcal{FV}(e_1 e_2) & = \mathcal{FV}(e_1) \cup \mathcal{FV}(e_2)
 \end{array}$$

A λ -expression e is *closed* iff $\mathcal{FV}(e) = \emptyset$. Other expressions are called *open*. Closed expressions are also known as *combinators*. A prominent combinator is the *identity function* $\lambda x. x$ which merely yields its argument x .

Expressions that differ only in the names of bound variables are said to be *syntactically equivalent* or α -equivalent, written $e_1 \equiv e_2$. For example, $\lambda x. x \equiv \lambda y. y$. We may freely convert between α -equivalent expressions by consistent renaming of bound variables. Without loss of generality, we adopt the following convention (also known as the *Barendregt convention* [5]):

Convention 2.3 (Variable convention)

If e_1, \dots, e_n appear in a certain mathematical context (definition, proof), then in these expressions all bound variables are chosen to be different from the free variables.

The variable convention allows us to provide a straightforward definition of *capture-avoiding substitution*. The function $e[x \mapsto e']$ replaces all free occurrences of a variable x in an expression e by an expression e' . For example, $(\lambda x. y)[y \mapsto z] = \lambda x. z$, $(\lambda x. x)[y \mapsto z] = \lambda x. x$, and $(f y)[f \mapsto \lambda x. x] = (\lambda x. x) y$.

Definition 2.4 (Substitution)

The *substitution function* $e[x \mapsto e']$ is defined recursively:

$$\begin{aligned} x[x \mapsto e'] &= e' \\ y[x \mapsto e'] &= y && \text{if } y \neq x \\ (\lambda y. e)[x \mapsto e'] &= \lambda y. (e[x \mapsto e']) && \text{if } y \neq x, y \notin \mathcal{FV}(e') \\ (e_1 e_2)[x \mapsto e'] &= (e_1[x \mapsto e'])(e_2[x \mapsto e']) \end{aligned}$$

The variable convention ensures that the condition in the third clause always holds so that no further clauses are required.

Evaluation of λ -expressions is performed via β -reduction which captures the idea of function application. The application of abstractions to operands is explained in terms of substitution.

Definition 2.5 (Redex, β -reduction)

An application of the form $(\lambda x. e_1) e_2$ is called a *reducible expression*, β -*redex*, or simply *redex*. The β -*reduction rule* replaces a redex $(\lambda x. e_1) e_2$ by the abstraction body e_1 in which all free occurrences of x are substituted with the operand e_2 .

$$(\lambda x. e_1) e_2 \rightarrow_{\beta} e_1[x \mapsto e_2]$$

Given two λ -expressions e and e' , we say that e is β -reducible to e' , written $e \rightarrow_{\beta}^* e'$, iff there exists a finite, potentially empty sequence of β -reductions

$e \rightarrow_{\beta} \dots \rightarrow_{\beta} e'$ that transforms e into e' . The evaluation of a λ -expression stops once no further redices remain in an expression. From an operational point of view, such a *normal form* may be regarded as a computational result.

Definition 2.6 (Normal form)

A λ -expression is said to be in *normal form* if it cannot be reduced any further.
 A λ -expression e is said to *have a normal form* if $e \rightarrow_{\beta}^* e'$ holds for some λ -expression e' in normal form.

Not every λ -expression has a normal form. A simple counterexample of this is the *divergent combinator* ω whose evaluation incessantly yields itself.

$$\omega \equiv (\lambda x. x x) (\lambda x. x x) \rightarrow_{\beta} (\lambda x. x x) (\lambda x. x x) \equiv \omega.$$

Typically, an expression under evaluation contains more than one redex to choose from for the next evaluation step. This gives rise to *evaluation strategies* that reduce the individual redices of a λ -expression in a deterministic order. A choice for a particular strategy has significant impact on both the semantics and the implementation of a programming language.

- Under *normal order reduction (leftmost-outermost)*, the leftmost redex of an expression that is not contained in any other redex is reduced first. This means that operations are applied to unevaluated operands thereby deferring the operands' evaluation.
- Under *applicative order reduction (leftmost-innermost)*, the leftmost redex of an expression that does not contain any further redices is reduced first. Therefore, both the operator and the operand of an application are reduced before the application itself.

Normal order evaluation has the advantage that it will reach the normal form of an expression if one exists. As a downside, normal-order evaluation can entail significant overhead as redices in the unevaluated operand may be copied, thereby duplicating computations.

Applicative order will not normalise an expression if the evaluation of the operand does not terminate (for example $(\lambda x. e) \omega$). However, if the evaluation of the operand terminates, the result will only be computed once and can potentially be used many times in the body of the abstraction.

Functional programming languages such as SML [75, 70, 49], OCAML [69, 22], HASKELL [81, 13, 54], and CLEAN [84] employ strategies that, unlike the strategies presented so far, do not evaluate expressions inside of abstractions. Instead of transforming expressions into full normal forms, these strategies merely compute *weak normal forms* [91, 80].

Definition 2.7 (Weak normal form)

A λ -expression is said to be in *weak normal form* iff it is an abstraction that may contain redices in its body.

Restricting the above evaluation strategies to weak normal forms gives rise to the two predominant evaluation strategies used by language implementations.

- The *call-by-name* strategy restricts normal order reduction. The operator is reduced to weak normal form and then applied to the unevaluated operand. To avoid multiple evaluation of the operand, concrete implementations such as HASKELL and CLEAN refine this strategy even further to a variant called *call-by-need* or *lazy evaluation*. Instead of copying the unevaluated argument to all variable locations in the syntax tree, only a pointer to a common *thunk* containing the argument is propagated. Upon first evaluation of the argument, the thunk is updated to hold the appropriate value for future access.
- Similarly, the *call-by-value* regime restricts applicative order reduction such that both the operator and the operand are merely reduced to weak normal forms before β -reduction is performed. Call-by-value is fairly easy to implement in an efficient way and therefore is the most widely used evaluation strategy. For example, SML and OCAML implement call-by-value evaluation.

2.2 An Applied λ -Calculus

This section presents an applied λ -calculus that extends the bare λ -calculus with the usual representations of truth values and integers along with primitive operators. We define δ -reduction, values, the evaluation relation, and formally discuss essential properties of the latter.

Definition 2.8 (δ -redex, δ -reduction)

An application of a primitive operator to legitimate arguments is called a *δ -redex*. *δ -reduction*, written $e \Rightarrow_{\delta} e$, replaces the redex by the result.

Legitimate arguments are those, on which the operator is defined. For example, $* 6 7 \Rightarrow_{\delta} 42$. For the sake of readability, applications of binary operations may also be written in infix notation with the usual precedence rules.

Figure 2.1 defines the syntax and the operational semantics of an applied λ -calculus with a call-by-value semantics that may be regarded as a simple programming language. The top half defines a set of expressions, a set of constant

Syntax

e	$::=$	$x \mid \lambda x. e \mid e e \mid c \mid \text{if } e \text{ then } e \text{ else } e$	Expressions
c	$::=$	$\mathbb{B} \mid \mathbb{Z} \mid f^1 \mid f^2$	Constants
f^1	$::=$	not	Unops
f^2	$::=$	$\leftrightarrow \mid \& \mid \mid \mid = \mid < \mid + \mid - \mid *$	Binops
v	$::=$	$\lambda x. e \mid c \mid f^2 v$	Values

Evaluation

$$\boxed{e \Rightarrow e}$$

$$\frac{e_1 \Rightarrow e'_1}{e_1 e_2 \Rightarrow e'_1 e_2} \text{ (E-APP1)}$$

$$\frac{e_2 \Rightarrow e'_2}{v_1 e_2 \Rightarrow v_1 e'_2} \text{ (E-APP2)}$$

$$(\lambda x. e_1) v_2 \Rightarrow e_1[x \mapsto v_2] \text{ (E-ABSAPP)}$$

$$\frac{f^1(v) \Rightarrow_{\delta} v'}{f^1 v \Rightarrow v'} \text{ (E-PRFAPP1)}$$

$$\frac{f^2(v_1, v_2) \Rightarrow_{\delta} v_3}{(f^2 v_1) v_2 \Rightarrow v_3} \text{ (E-PRFAPP2)}$$

$$\frac{e_p \Rightarrow e'_p}{\text{if } e_p \text{ then } e_t \text{ else } e_e \Rightarrow \text{if } e'_p \text{ then } e_t \text{ else } e_e} \text{ (E-COND)}$$

$$\text{if true then } e_t \text{ else } e_e \Rightarrow e_t \text{ (E-THEN)}$$

$$\text{if false then } e_t \text{ else } e_e \Rightarrow e_e \text{ (E-ELSE)}$$

Figure 2.1: An applied λ -calculus with call-by-value evaluation

symbols and a set of values. In the remainder, the metavariables v, v', v_i range over values, the metavariable c represents constant symbols, and a metavariable f^n represents function symbols of arity n .

The set of expressions consists of the expressions of the λ -calculus, constant symbols c and the conditional expression $\text{if } e_p \text{ then } e_t \text{ else } e_e$, with the *predicate* e_p and the two branch expressions e_t, e_e . The set of constant symbols comprises the truth values $\mathbb{B} = \{\text{true}, \text{false}\}$, the integers $\mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$, and some of the usual logical, relational, and arithmetic operators. A value is either a λ -abstraction, a constant symbol c , or an application $f^2 v$ of a binary operator to a single argument.

Definition 2.9 (Value)

A value v is an expression that is considered a valid evaluation result.

The bottom half of Figure 2.1 defines the operational semantics of the language by means of inference rules.

Definition 2.10 (One-step evaluation relation, multi-step evaluation relation)

The *one-step evaluation relation* $e \Rightarrow e$ is the smallest relation that satisfies the inference rules. The *multi-step evaluation relation* $e \Rightarrow^* e$ is the reflexive, transitive closure of $e \Rightarrow e$.

The evaluation relation captures the call-by-value strategy: the rules E-APP1 and E-APP2 process applications from left to right and bottom-up; when the operator is an abstraction and the operand is a value, E-ABSAPP performs a β -reduction step. The definition of the capture-avoiding substitution carries over from Section 2.1, *mutatis mutandis*. The rules E-APPPRF1 and E-APPPRF2 evaluate applications of primitive functions to legitimate arguments by means of δ -reduction. Evaluation of the conditional is defined by the final three rules. E-COND evaluates the predicate. When it evaluates to either true or false, the entire conditional is evaluated to e_t (E-THEN) or e_e (E-ELSE), respectively. No evaluation takes place inside of abstractions.

Every expression in normal form that is not a value is said to be a *stuck expression* [82]. For example, not 42, $(\lambda x. x) + 0$, if 42 then true else false. The intuition behind stuck expressions is that due to a run-time error, the machine has entered a meaningless state in which no further evaluation is possible.

We can now formally discuss essential properties of the evaluation relation. First, we check that every value is in normal form, a crucial property of every language definition.

Theorem 2.11

Every value v is in normal form.

Proof: Immediate. There are no rules that evaluate abstractions $\lambda x. e$, constant symbols c , or partially applied binary operators $f^2 v$. ■

The next theorem states that one-step evaluation is deterministic, i. e., the result of an expression that makes an evaluation step will always be the same.

Theorem 2.12 (Determinacy of one-step evaluation)

If $e \Rightarrow e'$ and $e \Rightarrow e''$, then $e' = e''$.

Proof: By induction on a derivation of $e \Rightarrow e'$. For every rule E, we assume that e evaluates to e' . We show that no other rule that matches e than E is applicable. In consequence, e'' can only be derived from e by rule E. By the induction hypothesis, evaluation of the subexpressions is deterministic and hence $e' = e''$.

1. Case E-APP1: $e = e_1 e_2$ with $e_1 \Rightarrow e'_1$. Since e_1 is not a value, no other rule matches e . By the hypothesis $e'_1 = e''_1$ and hence $e' = e''$.
2. Case E-APP2: $e = v_1 e_2$ with $e_2 \Rightarrow e'_2$. Similar.
3. Case E-ABSAPP: $e = (\lambda x. e_1) v_2$. Both the operator and the operand are values, so no other rule matches e . Substitution is deterministic, so that $e' = e''$.
4. Case E-PRFAPP1: $e = f^1 v$ with $f^1(v) \Rightarrow_\delta v'$. No other rule matches and δ -reduction is deterministic, so that $e' = e''$.
5. Case E-PRFAPP2: $e = (f^2 v_1) v_2$ with $f^2(v_1, v_2) \Rightarrow_\delta v_3$. Similar.
6. Case E-COND: $e = \text{if } e_p \text{ then } e_t \text{ else } e_e$ with $e_p \Rightarrow e'_p$. Since e_p is not a value, no other rule matches e . By the hypothesis, $e'_p = e''_p$ and thus $e' = e''$.
7. Case E-THEN: $e = \text{if true then } e_t \text{ else } e_e$. No other rule matches e and therefore $e' = e''$.
8. Case E-FALSE: $e = \text{if false then } e_t \text{ else } e_e$. Similar. ■

An immediate corollary of the determinacy of one-step evaluation is that multi-step evaluation is also deterministic.

Corollary 2.13 (Uniqueness of normal forms)

If $e \Rightarrow^* e'$ and $e \Rightarrow^* e''$ where e' and e'' are normal forms, then $e' = e''$.

In addition to the facilities for function definition and application, an applied λ -calculus features built-in constant symbols and operators whose operational semantics is defined in terms of δ -reduction. This section introduced an applied λ -calculus with a call-by-value evaluation regime that serves as a language nucleus in the remainder of the thesis. We showed that evaluation is deterministic and that every value is in normal form. All language extensions must retain these essential properties.

2.3 Simple Types

Every terminating evaluation sequence either yields a value or ends at a stuck expression, i. e., an expression for which no evaluation rule applies although it is not considered a valid result. A type system identifies such errors by classifying expressions according to the values they evaluate to. This section augments the applied λ -calculus from the previous section with simple types. We define the typing relation and discuss essential properties of the typed calculus, such as type safety.

Figure 2.2 shows a typed version of the applied λ -calculus from Section 2.2 by providing a syntax, an evaluation relation, and a typing relation.

Syntax

T	$::= B \mid T \rightarrow T$	Types
B	$::= \text{bool} \mid \text{int}$	Base types
e	$::= x \mid \lambda x : T. e \mid e e \mid c \mid \text{if } e \text{ then } e \text{ else } e$	Expressions
c	$::= \mathbb{B} \mid \mathbb{Z} \mid f^1 \mid f^2$	Constants
f^1	$::= \text{not}$	Unops
f^2	$::= \leftrightarrow \mid \& \mid \mid \mid = \mid < \mid + \mid - \mid *$	Binops
v	$::= \lambda x : T. e \mid c \mid f^2 v$	Values
Γ	$::= \cdot \mid \Gamma, x : T$	Context

Evaluation

$\frac{e_1 \Rightarrow e'_1}{e_1 e_2 \Rightarrow e'_1 e_2} \text{ (E-APP1)}$	$\frac{e_2 \Rightarrow e'_2}{v_1 e_2 \Rightarrow v_1 e'_2} \text{ (E-APP2)}$	$e \Rightarrow e$
$(\lambda x : T. e_1) v_2 \Rightarrow e_1[x \mapsto v_2]$ (E-ABSAPP)		
$\frac{f^1(v) \Rightarrow_{\delta} v'}{f^1 v \Rightarrow v'} \text{ (E-PRFAPP1)}$	$\frac{f^2(v_1, v_2) \Rightarrow_{\delta} v_3}{(f^2 v_1) v_2 \Rightarrow v_3} \text{ (E-PRFAPP2)}$	
$\frac{e_p \Rightarrow e'_p}{\text{if } e_p \text{ then } e_t \text{ else } e_e \Rightarrow \text{if } e'_p \text{ then } e_t \text{ else } e_e} \text{ (E-COND)}$		
if true then e_t else $e_e \Rightarrow e_t$ (E-THEN)		
if false then e_t else $e_e \Rightarrow e_e$ (E-ELSE)		

Typing

$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-VAR)}$	$\Gamma \vdash c : \text{type}(c) \text{ (T-CONST)}$	$\Gamma \vdash e : T$
$\frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x : T_1. e : T_1 \rightarrow T_2} \text{ (T-ABS)}$		
$\frac{\Gamma \vdash e_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash e_2 : T_{11}}{\Gamma \vdash e_1 e_2 : T_{12}} \text{ (T-APP)}$		
$\frac{\Gamma \vdash e_p : \text{bool} \quad \Gamma \vdash e_t : T \quad \Gamma \vdash e_e : T}{\Gamma \vdash \text{if } e_p \text{ then } e_t \text{ else } e_e : T} \text{ (T-COND)}$		

Figure 2.2: The applied λ -calculus with simple types

In addition to expressions, constants, and values, the syntax defines a set of types and a set of typing contexts. In the following, the metavariables T, T', T_i range over types and the metavariables $\Gamma, \Gamma', \Gamma_i$ range over contexts.

The set of *types* consists of the base types `bool` and `int`, that describe boolean and integer expressions, and function types $T_1 \rightarrow T_2$ that describe functions with the *domain* T_1 and *codomain* T_2 . The function type associates to the right, so that $T_1 \rightarrow T_2 \rightarrow T_3$ stands for $T_1 \rightarrow (T_2 \rightarrow T_3)$. The expressions and values resemble those from Section 2.2, except that abstractions $\lambda x : T. e$ are annotated with the type of the bound variable.

A *type context* or *environment* is a sequence $\cdot, x_1 : T_1, \dots, x_n : T_n$ that, starting with the empty context \cdot , associates variables with types. Convention 2.3 extends to the variables bound in a context, so that a variable can appear in a context Γ at most once. The function $\text{dom}(\Gamma)$ yields the set of variables bound in Γ . The notation $x : T \in \Gamma$ means that in Γ , the variable x is associated with type T .

The evaluation relation $e \Rightarrow e$ is, apart from the rule E-ABSAPP the same as in Section 2.2. The bottom of the figure defines the ternary typing relation $\Gamma \vdash e : T$ by a set of inference rules.

Definition 2.14 (Typing relation, well-typed expression)

The *typing relation* $\Gamma \vdash e : T$ is the smallest ternary relation between typing contexts, expressions, and types that satisfies the inference rules. An expression e is said to be *well-typed* under a context Γ iff there is some T such that $\Gamma \vdash e : T$.

The typing rule T-VAR assigns a variable x the type T if $x : T$ appears in the context Γ . The rule T-CONST gives types to constants c according to the table shown in Figure 2.3. The typing rule T-ABS for abstractions assign the abstraction $\lambda x : T. e$ the type $T \rightarrow T'$ when e has type T' under the extended context $\Gamma, x : T$. Applications $e_1 e_2$ are checked by the rule T-APP. When the operator e_1 is a function of type $T \rightarrow T'$ and when the operand e_2 is an argument of type T , then the application result has type T' . Finally, the typing rule T-COND for conditionals `if e_p then e_t else e_e` ensures that the predicate e_p is a boolean expression and that the branch expressions e_t, e_e have the same type T .

The goal of type checking is to rule out programs whose evaluation might get stuck at some point. This property is called *safety* or *soundness*. To formally prove that simple types indeed provide type-safety, two theorems are required. The *progress* theorem states that every (closed) well-typed expression is either a value or able to make an evaluation step. The *preservation* theorem (or subject reduction) states that evaluation preserves the type of an expression such that the progress theorem applies again.

A basic assumption about the constant and function symbols in the calculus is

```

true, false : bool
not : bool → bool
↔, &, | : bool → bool → bool
.., -1, 0, 1, .. : int
=, < : int → int → bool
+, -, * : int → int → int

```

Figure 2.3: Types of the constant symbols

that every symbol has a type and that δ -reduction of a function symbol behaves as declared by its type.

Axiom 2.15 (Constant symbols are well-behaved)

Each constant symbol c has a type $\text{type}(c)$ such that: if $\text{type}(f^1) = T_1 \rightarrow T_2$, then δ -reduction $f^1(v_1) \Rightarrow_{\delta} v_2$ is defined for all values v_1 with $\cdot \vdash v_1 : T_1$ so that $\cdot \vdash v_2 : T_2$. The axiom also applies to binary functions f^2 , mutatis mutandis.

The canonical forms lemma recapitulates the possible forms the values of a given type may have. The proof is omitted as it is straightforward from the syntax of values, the types of constants, and the typing rules.

Lemma 2.16 (Canonical forms)

1. If v is a value with $\cdot \vdash v : \text{bool}$ then $v \in \mathbb{B}$.
2. If v is a value with $\cdot \vdash v : \text{int}$ then $v \in \mathbb{Z}$.
3. If v is a value with $\cdot \vdash v : T_1 \rightarrow T_2$ then $v = f^1$, $v = f^2$, $v = f^2 v_1$, or $v = \lambda x : T. e$.

With the canonical forms lemma, the progress theorem can be formalised and proved.

Theorem 2.17 (Progress)

If $\cdot \vdash e : T$ then e is a value or there is some e' with $e \Rightarrow e'$.

Proof: By induction on type derivations $\cdot \vdash e : T$.

1. Case T-VAR: $e = x$
 e is not closed.
2. Case T-CONST: $e = c$
 e is a value.

3. Case T-ABS: $e = \lambda x : T. e$
 e is a value.
4. Case T-APP: $e = e_1 e_2$ with $\cdot \vdash e_1 : T' \rightarrow T$ and $\cdot \vdash e_2 : T'$
 By the hypothesis, the subexpressions e_1 and e_2 are either values or can make an evaluation step. If e_1 can make a step, E-APP1 applies. If e_1 is a value and e_2 can make a step then E-APP2 applies. If both expressions are values, the application can have four different forms according to the canonical forms lemma:
 - (a) Case $e_1 = f^1$: E-PRFAPP1 applies because of axiom 2.15.
 - (b) Case $e_1 = f^2$: $e = f^2 e_2$ is a value.
 - (c) Case $e_1 = f^2 v_1$: E-PRFAPP2 applies because of axiom 2.15.
 - (d) Case $e_1 = \lambda x : T'. e'$: E-ABSAPP applies.
5. Case T-COND: $e = \text{if } e_p \text{ then } e_t \text{ else } e_e$ with $\cdot \vdash e_p : \text{bool}$
 By the hypothesis, e_p is a value or it can make an evaluation step. If e_p can make a step, then E-COND applies. If e_p is a value, then by the canonical forms lemma, it is either true or false and therefore either E-THEN or E-ELSE applies. ■

In order to prove preservation, some basic lemmas are required. The weakening lemma states that extending a context with a fresh variable x does not change the type of a well-typed expression. The proof is a straightforward induction on typing derivations $\Gamma_1, \Gamma_2 \vdash e : T$.

Lemma 2.18 (Weakening)

If $\Gamma_1, \Gamma_2 \vdash e : T$ then $\Gamma_1, x : T_x, \Gamma_2 \vdash e : T$.

The substitution lemma states that substituting an identifier x of type T_x with an expression e_x of the same type T_x in an expression e preserves the type of e .

Lemma 2.19 (Substitution lemma)

If $\Gamma_1, x : T_x, \Gamma_2 \vdash e : T$ and $\Gamma_1 \vdash e_x : T_x$ then $\Gamma_1, \Gamma_2 \vdash e[x \mapsto e_x] : T$.

Proof: By induction on the derivation of $\Gamma_1, x : T_x, \Gamma_2 \vdash e : T$.

1. Case T-CONST: $\Gamma_1, x : T_x, \Gamma_2 \vdash c : \text{type}(c)$, $c[x \mapsto e_x] = c$
 By T-CONST, $\Gamma_1, \Gamma_2 \vdash c : \text{type}(c)$.
2. Case T-VAR: $\Gamma_1, x : T_x, \Gamma_2 \vdash x' : T$, $x' : T \in (\Gamma_1, x : T_x, \Gamma_2)$.
 - (a) Case $x' = x$: $x'[x \mapsto e_x] = e_x$
 Since $T_x = T$, $\Gamma_1 \vdash e_x : T$. By weakening, $\Gamma_1, \Gamma_2 \vdash e_x : T$.
 - (b) Case $x \neq x'$: $x'[x \mapsto e_x] = x'$
 Since $x \neq x'$, $x' : T \in (\Gamma_1, \Gamma_2)$. By T-VAR, $\Gamma_1, \Gamma_2 \vdash x' : T$.
3. Case T-ABS: $\Gamma_1, x : T_x, \Gamma_2 \vdash \lambda x' : T_1. e : T_1 \rightarrow T_2$, $\Gamma_1, x : T_x, \Gamma_2, x' : T_1 \vdash e : T_2$.
 By the variable convention 2.3, $x \neq x'$ and $x' \notin \mathcal{FV}(e_x)$. By the hypothesis, $\Gamma_1, \Gamma_2, x' : T_1 \vdash e[x \mapsto e_x] : T_2$. By T-ABS, $\Gamma_1, \Gamma_2 \vdash \lambda x' : T_1. e[x \mapsto e_x] : T_1 \rightarrow T_2$.

4. Case T-APP: $e = e_1 e_2$: By the induction hypothesis and T-APP.
5. Case T-COND: $e = \text{if } e_p \text{ then } e_t \text{ else } e_e$: By the induction hypothesis and T-COND. ■

The evaluation of β -redices involves substituting identifiers with expressions. The proof of the preservation theorem thus relies on the substitution lemma.

Theorem 2.20 (Preservation)

If $\Gamma \vdash e : T$ and $e \Rightarrow e'$ then $\Gamma \vdash e' : T$.

Proof: By induction on type derivations $\Gamma \vdash e : T$.

1. Case T-VAR: $e = x$
There is no e' with $x \Rightarrow e'$.
2. Case T-CONST: $e = c$
 e is a value.
3. Case T-ABS: $e = \lambda x : T'. e$
 e is a value.
4. Case T-APP: $e = e_1 e_2$ with $\Gamma \vdash e_1 : T' \rightarrow T$ and $\Gamma \vdash e_2 : T'$
The application can make an evaluation step in five possible ways:
 - (a) Case E-APP1: $e_1 \Rightarrow e'_1$ then $e' = e'_1 e_2$
The result follows from the induction hypothesis and T-APP.
 - (b) Case E-APP2: $e_1 = v_1$ and $e_2 \Rightarrow e'_2$ then $e' = v_1 e'_2$
Similar.
 - (c) Case E-APPAPS: $e_1 = \lambda x : T'. e_b$ and $e_2 = v_2$ then $e' = e_b[x \mapsto v_2]$
By the substitution lemma, $\Gamma \vdash e_b[x \mapsto v_2] : T$.
 - (d) Case E-PRFAPP1: $e_1 = f^1$ and $e_2 = v_2$ with $f^1(v_2) \Rightarrow_\delta v$ then $e' = v$
By axiom 2.15, δ -reduction is type-preserving.
 - (e) Case E-PRFAPP2: $e_1 = f^2 v_1$ and $e_2 = v_2$ with $f^2(v_1, v_2) \Rightarrow_\delta v$ then $e' = v$
Similar.
5. Case T-COND: $e = \text{if } e_p \text{ then } e_t \text{ else } e_e$ with $\Gamma \vdash e_p : \text{bool}$, $\Gamma \vdash e_t : T$, $\Gamma \vdash e_e : T$
The conditional can make an evaluation step in three possible ways:
 - Case E-COND: $e_p \Rightarrow e'_p$ then $e' = \text{if } e'_p \text{ then } e_t \text{ else } e_e$
By the hypothesis and T-COND.
 - Case E-THEN: $e_p = \text{true}$ then $e' = e_t$
Immediate, since $\Gamma \vdash e_t : T$.
 - Case E-ELSE: $e_p = \text{false}$ then $e' = e_e$
Similar. ■

Unlike the untyped λ -calculus, the simply-typed λ -calculus is strongly normalising, i. e., the evaluation of every well-typed expression eventually terminates

yielding a value. For example, there is no typed form of the divergent combinator $\omega = (\lambda x. x x) (\lambda x. x x)$ as the inherent self-application $x x$ cannot be typed.

Lemma 2.21 (Self-application is ill-typed)

There is no context Γ and no type T so that $\Gamma \vdash x x : T$.

Proof: By reductio ad absurdum. Suppose that under some context Γ , there is some type T so that $\Gamma \vdash x x : T$. By inversion of the typing rule T-APP, there must be some T' with $\Gamma \vdash x : T' \rightarrow T$ and simultaneously $\Gamma \vdash x : T'$. Therefore the proposition is wrong. ■

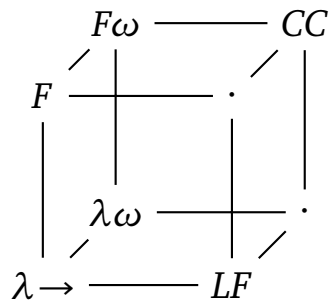
To nonetheless enable the specification of recursive programs, typed λ -calculi are commonly extended with some kind of fixed-point expression that reproduces itself upon evaluation.

Summary

This chapter presented essential concepts of the λ -calculus, operational semantics, and type systems. By restricting the set of allowed expressions, types enable a compiler to statically rule out erroneous programs. Type checking is inherently conservative: only programs for which the absence of certain behaviours can be proved are accepted. As a downside, a type system may also reject well-behaved programs as ill-typed. To increase the number of typeable programs, more advanced typing schemes have been studied.

In the simply-typed λ -calculus, an abstraction $\lambda x : T. e$ may be regarded as an expression that is parameterised by another expression of type T . Figure 2.4 shows Barendregt's λ -cube [6, 82] that nicely visualises three orthogonal extensions of this basic system. The cube places the simply typed λ -calculus in the lower left front corner as $\lambda \rightarrow$. Each of its three axes represents a new form of abstraction.

1. The vertical axis introduces type abstractions that map types to expressions and thereby give rise to polymorphism. The resulting calculus is known as System F or the second-order λ -calculus.
2. The perpendicular axis introduces type operators, i. e., types that depend on types. The system $\lambda\omega$ is thus called the simply-typed λ -calculus with type operators. The combination with System F is known as System $F\omega$.
3. The horizontal axis introduces dependent types, i. e., types that depend on expressions. The λ -calculus with dependent types has become widely known as the logical framework LF . The calculus of constructions (CC) combines all three forms of abstraction.

Figure 2.4: Barendregt's λ -cube [6, 82]

3

Decidable First-Order Theories

The type system of QUBE uses dependent types to accurately model array shapes and to restrict the potential run-time values of expressions. Type checking proceeds in collaboration with an automatic theorem prover for the Satisfiability Modulo Theories problem. This chapter gives a brief introduction to propositional logic, first-order logic, and the relevant fragments of first-order theories that will be used in later chapters. More detailed presentations of these topics can be found in textbooks on logic and computer aided verification, such as [66, 17, 50].

The remainder of this chapter is structured as follows: First, Section 3.1 introduces propositional logic and basic terminology. Section 3.2 then gives a brief account of first-order logic, which extends propositional logic with non-logical function symbols, predicate symbols and quantifiers. Next, Section 3.3 presents some decidable quantifier-free fragments of first-order theories that the QUBE compiler uses to reason about programs, namely the theory of uninterpreted functions and equality as well as linear integer arithmetic. Section 3.4 presents array properties, a decidable fragment of quantified first-order logic that is used by the QUBE compiler to model integer vectors.

3.1 Propositional Logic

This section gives a brief introduction to propositional logic (PL) and explains basic terminology such as validity, satisfiability, essential normal forms, soundness and completeness.

Given an enumerable set V of *propositional variables*, an *atomic formula* is a propositional variable $x \in V$. A *propositional logic formula* is generated from atomic formulae using the logical connectives \neg (negation), \wedge (conjunction), \vee (disjunction), \rightarrow (implication), and \leftrightarrow (equivalence).

Definition 3.1 (Syntax of propositional logic formulas)

The set \mathcal{P} of *propositional logic formulas* φ is defined by the grammar:

$$\begin{array}{lcl} \varphi & ::= & a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi & \text{Formulas} \\ a & ::= & x & \text{Atoms} \end{array}$$

The above syntax of PL formulas is ambiguous. We use parentheses to disambiguate the structure of formulas. To avoid excessive bracketing, the following conventions about the priorities of the logical connectives apply: \neg has the highest priority, \wedge and \vee are next, \rightarrow and \leftrightarrow have the lowest priorities.

The meaning of a propositional logic formula is a truth value, i. e., an element from the set $\mathbb{B} = \{\top, \perp\}$ with $\top \neq \perp$. In general, the truth value of a formula depends on the valuation of its propositional variables.

Definition 3.2 (Assignment)

An *assignment* $\alpha : V \rightarrow \mathbb{B}$ is a function that maps propositional variables to truth values.

Given a formula and an assignment, the truth value of a formula may be evaluated. The evaluation function $\llbracket \cdot \rrbracket (\cdot) : \mathcal{P} \rightarrow (V \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$ is defined inductively. In the definition, the truth functions *not* : $\mathbb{B} \rightarrow \mathbb{B}$, *and* : $\mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$, and *or* : $\mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ have the usual meanings.

$$\begin{aligned} \llbracket x \rrbracket (\alpha) &= \alpha(x) \\ \llbracket \neg\varphi \rrbracket (\alpha) &= \text{not}(\llbracket \varphi \rrbracket (\alpha)) \\ \llbracket \varphi_1 \wedge \varphi_2 \rrbracket (\alpha) &= \text{and}(\llbracket \varphi_1 \rrbracket (\alpha), \llbracket \varphi_2 \rrbracket (\alpha)) \\ \llbracket \varphi_1 \vee \varphi_2 \rrbracket (\alpha) &= \text{or}(\llbracket \varphi_1 \rrbracket (\alpha), \llbracket \varphi_2 \rrbracket (\alpha)) \\ \llbracket \varphi_1 \rightarrow \varphi_2 \rrbracket (\alpha) &= \text{or}(\text{not}(\llbracket \varphi_1 \rrbracket (\alpha)), \llbracket \varphi_2 \rrbracket (\alpha)) \\ \llbracket \varphi_1 \leftrightarrow \varphi_2 \rrbracket (\alpha) &= \text{or}(\text{and}(\llbracket \varphi_1 \rrbracket (\alpha), \llbracket \varphi_2 \rrbracket (\alpha)), \\ &\quad \text{and}(\text{not}(\llbracket \varphi_1 \rrbracket (\alpha)), \text{not}(\llbracket \varphi_2 \rrbracket (\alpha)))) \end{aligned}$$

Given an assignment α , a formula φ is said to *hold under* α , written $\alpha \models \varphi$, iff $\llbracket \varphi \rrbracket (\alpha) = \top$.

Definition 3.3 (Validity, satisfiability)

A formula φ is called *valid*, written $\models \varphi$, iff φ holds under all assignments, otherwise φ is called *invalid*. A formula φ is called *satisfiable* iff φ holds under some assignment α , otherwise φ is called *unsatisfiable*.

Satisfiability and validity are dual concepts. A formula is valid iff its negation is unsatisfiable. Vice versa, a formula is invalid iff its negation is satisfiable. A satisfying assignment of a formula φ is also called a *model* of φ . An assignment that satisfies $\neg\varphi$ is said to be a *counterexample* of φ . Valid formulas are also called *tautologies*.

Definition 3.4 (Equivalence, implication)

Two formulas φ_1 and φ_2 are said to be *equivalent*, written $\varphi_1 \Leftrightarrow \varphi_2$, if they evaluate to the same truth values under all assignments. I. e., for all assignments α , $\alpha \models \varphi_1$ iff $\alpha \models \varphi_2$, i. e., $\models \varphi_1 \leftrightarrow \varphi_2$.

The formula φ_1 is said to *imply* the formula φ_2 , written $\varphi_1 \Rightarrow \varphi_2$, if every assignment α satisfying φ_1 also satisfies φ_2 . I. e., for all assignments α , if $\alpha \models \varphi_1$, then $\alpha \models \varphi_2$, i. e., $\models \varphi_1 \rightarrow \varphi_2$.

A *normal form* imposes syntactic restrictions on formulas. The process of deciding whether a formula is satisfiable usually starts with transforming the formula to some normal form that is appropriate for the decision procedure at hand.

Definition 3.5 (Negation normal form (NNF))

A formula is in *negation normal form (NNF)* if it is generated from literals using the logical connectives \wedge and \vee . A *literal* is an atom or its negation.

$$\begin{array}{ll} \varphi_{nnf} & ::= l \mid \varphi_{nnf} \wedge \varphi_{nnf} \mid \varphi_{nnf} \vee \varphi_{nnf} & \text{NNF formulas} \\ l & ::= a \mid \neg a & \text{Literals} \end{array}$$

Every PL formula can be transformed into an equivalent formula in NNF by exhaustively replacing the left hand sides of the following equivalences with the respective right hand sides.

$$\begin{aligned} \neg\neg\varphi & \Leftrightarrow \varphi \\ \neg(\varphi_1 \wedge \varphi_2) & \Leftrightarrow \neg\varphi_1 \vee \neg\varphi_2 \\ \neg(\varphi_1 \vee \varphi_2) & \Leftrightarrow \neg\varphi_1 \wedge \neg\varphi_2 \\ \varphi_1 \rightarrow \varphi_2 & \Leftrightarrow \neg\varphi_1 \vee \varphi_2 \\ \varphi_1 \leftrightarrow \varphi_2 & \Leftrightarrow (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \end{aligned}$$

Definition 3.6 (Conjunctive normal form (CNF))

A formula is in *conjunctive normal form* if it is a conjunction of clauses. A *clause* is a disjunction of literals.

$$\varphi_{cnf} ::= \bigwedge_i \bigvee_j l_{i,j} \quad \text{CNF formulas}$$

A formula can be transformed into an equivalent formula in CNF by first transforming the formula into NNF and then distributing the conjunction symbols by exhaustively applying the following equivalences from left to right.

$$\begin{aligned} (\varphi_1 \wedge \varphi_2) \vee \varphi_3 &\Leftrightarrow (\varphi_1 \vee \varphi_3) \wedge (\varphi_2 \vee \varphi_3) \\ \varphi_1 \vee (\varphi_2 \wedge \varphi_3) &\Leftrightarrow (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3) \end{aligned}$$

The *decision problem* is to determine whether a formula is valid. Because of the duality of validity and satisfiability, determining the satisfiability of the negated formula is an equivalent problem. We are interested in automatic procedures to determine the validity (or the unsatisfiability) of a given formula.

Definition 3.7 (Decision procedure)

A procedure for the decision problem is *sound* if when it returns **valid (unsatisfiable)**, the formula is indeed valid (unsatisfiable). A procedure for the decision problem is *complete* if it terminates on every input, and when the formula is valid (unsatisfiable), it returns **valid (unsatisfiable)**.

A procedure is called a *decision procedure* if it is sound and complete.

The satisfiability problem of boolean formulas (SAT) is naively decidable because the number of potential assignments is finite. Given a formula with n propositional variables, a decision procedure may simply enumerate all 2^n assignments and report whether any of these satisfies the formula.

SAT was the first known \mathcal{NP} -complete problem [27]. It is thus unknown whether there is an algorithm that finds a satisfying assignment for all satisfiable formulas in polynomial time. Nonetheless, the theoretical and practical significance of the problem has motivated the development of extremely powerful SAT solvers. Based on the *Davis-Putnam-Loveland-Logemann (DPLL)* algorithm [29, 28], solvers like CHAFF [77] and MINISAT [34] are used in the industrial practice to solve CNF formulas with hundreds of thousands or even millions of boolean variables in reasonable amounts of time.

3.2 First-Order Logic

First-order logic (FOL), also called predicate logic, extends propositional logic with non-logical function and predicate symbols and quantification. Terms evaluate to objects, predicates take objects to truth values. Quantifiers express properties of entire sets of objects.

A signature fixes the set of non-logical symbols that may appear in a first-order formula.

Definition 3.8 (Signature)

A signature $\Sigma = (\mathbb{F}, \mathbb{P})$ consists of a family $\mathbb{F} = (\mathbb{F}_n)_{n \in \mathbb{N}}$ of function symbols, and a family $\mathbb{P} = (\mathbb{P}_n)_{n \in \mathbb{N}}$ of predicate symbols. n is called the *arity* of a symbol. We assume all sets to be mutually disjoint.

Nullary function symbols may be regarded as constant symbols. Given a signature $\Sigma = (\mathbb{F}, \mathbb{P})$ and an enumerable set V of variables, a *term* is either a variable $x \in V$, or an application of a function symbol $f_n \in \mathbb{F}_n$ to n terms. An *atomic formula* is an application of a predicate symbol $p_n \in \mathbb{P}_n$ to n terms. A *first-order formula* consists of *atomic formulas*, combined with the logical connectives \neg (negation), \wedge (conjunction), \vee (disjunction), \rightarrow (implication), and \leftrightarrow (equivalence), and the quantifiers $\forall x$ (for all x) and $\exists x$ (for some x).

Definition 3.9 (Syntax of terms and first-order formulas)

The set \mathcal{T} of terms t and the set \mathcal{F} of first-order formulas φ over Σ and V are defined by the following grammar:

φ	$::=$	a	$ $	$\neg\varphi$	$ $	$\varphi \wedge \varphi$	$ $	$\varphi \vee \varphi$	$ $	$\varphi \rightarrow \varphi$	$ $	$\varphi \leftrightarrow \varphi$	Formulas
				$ $	$\forall x. \varphi$	$ $	$\exists x. \varphi$						
a	$::=$	$p_n(t_1, \dots, t_n)$										Atoms	
t	$::=$	x	$ $	$f_n(t_1, \dots, t_n)$									Terms

As before, we use parentheses to disambiguate the structure of first-order formulas. The following conventions about the order of operations apply: \neg is evaluated first, \wedge and \vee are evaluated next, then quantifiers are evaluated, before finally \rightarrow and \leftrightarrow are evaluated.

In analogy with the λ -calculus, a variable is called *free* in a formula φ , iff at least one occurrence is not bound by a quantifier. The set $\mathcal{FV}(\varphi)$ contains all free variables of φ . A formula that contains no free variable is called a *closed formula* or a *sentence*.

An interpretation gives meaning to a signature. It provides a domain for interpreting variables and terms, for example, the integers or real numbers, and

explains function symbols as actual functions, and predicate symbols as actual predicates.

Definition 3.10 (Interpretation, assignment)

An *interpretation* $I = (D, \mathbb{F}^I, \mathbb{P}^I)$ of a signature $\Sigma = (\mathbb{F}, \mathbb{P})$ provides

- a non-empty set of objects D as the *domain of interpretation*,
- for every function symbol $f \in \mathbb{F}_n$ a function $f^I : D^n \rightarrow D$, and
- for every predicate symbol $p \in \mathbb{P}_n$ a predicate $p^I : D^n \rightarrow \mathbb{B}$.

An *assignment* $\alpha : V \rightarrow D$ maps variables to objects from the domain D . The altered assignment $\alpha[x \mapsto d]$ is defined as

$$\alpha[x \mapsto d](y) = \begin{cases} d & \text{if } x = y \\ \alpha(y) & \text{if } x \neq y \end{cases}$$

With an interpretation $I = (D, \mathbb{F}^I, \mathbb{P}^I)$ and an assignment α , we may evaluate formulas by evaluating terms to objects and formulas to truth values. The evaluation function $\llbracket \cdot \rrbracket_I(\cdot) : \mathcal{T} \rightarrow (V \rightarrow D) \rightarrow D$ which determines the value of terms is defined inductively.

$$\begin{aligned} \llbracket x \rrbracket_I(\alpha) &= \alpha(x) \\ \llbracket f(t_1, \dots, t_n) \rrbracket_I(\alpha) &= f^I(\llbracket t_1 \rrbracket_I(\alpha), \dots, \llbracket t_n \rrbracket_I(\alpha)) \end{aligned}$$

Similarly, the evaluation function $\llbracket \cdot \rrbracket_I(\cdot) : \mathcal{F} \rightarrow (V \rightarrow D) \rightarrow \mathbb{B}$ that determines the truth value of a formula in an interpretation I is also defined inductively.

$$\begin{aligned} \llbracket p(t_1, \dots, t_n) \rrbracket_I(\alpha) &= p^I(\llbracket t_1 \rrbracket_I(\alpha), \dots, \llbracket t_n \rrbracket_I(\alpha)) \\ \llbracket \neg \varphi \rrbracket_I(\alpha) &= \text{not}(\llbracket \varphi \rrbracket_I(\alpha)) \\ \llbracket \varphi_1 \wedge \varphi_2 \rrbracket_I(\alpha) &= \text{and}(\llbracket \varphi_1 \rrbracket_I(\alpha), \llbracket \varphi_2 \rrbracket_I(\alpha)) \\ \llbracket \varphi_1 \vee \varphi_2 \rrbracket_I(\alpha) &= \text{or}(\llbracket \varphi_1 \rrbracket_I(\alpha), \llbracket \varphi_2 \rrbracket_I(\alpha)) \\ \llbracket \varphi_1 \rightarrow \varphi_2 \rrbracket_I(\alpha) &= \text{or}(\text{not}(\llbracket \varphi_1 \rrbracket_I(\alpha)), \llbracket \varphi_2 \rrbracket_I(\alpha)) \\ \llbracket \varphi_1 \leftrightarrow \varphi_2 \rrbracket_I(\alpha) &= \text{or}(\text{and}(\llbracket \varphi_1 \rrbracket_I(\alpha), \llbracket \varphi_2 \rrbracket_I(\alpha)), \\ &\quad \text{and}(\text{not}(\llbracket \varphi_1 \rrbracket_I(\alpha)), \text{not}(\llbracket \varphi_2 \rrbracket_I(\alpha)))) \\ \llbracket \forall x. \varphi \rrbracket_I(\alpha) &= \begin{cases} \top & \text{if for all } d \in D : \llbracket \varphi \rrbracket_I(\alpha[x \mapsto d]) \\ \perp & \text{else} \end{cases} \\ \llbracket \exists x. \varphi \rrbracket_I(\alpha) &= \begin{cases} \top & \text{if for some } d \in D : \llbracket \varphi \rrbracket_I(\alpha[x \mapsto d]) \\ \perp & \text{else} \end{cases} \end{aligned}$$

Given an interpretation I and an assignment α , a first-order formula φ is said to

hold in I under α , written $\alpha \models_I \varphi$, iff $\llbracket \varphi \rrbracket_I(\alpha) = \top$. A formula φ is said to hold in I , written $\models_I \varphi$, iff φ holds in I under any assignment.

Definition 3.11 (Validity, satisfiability)

A first-order sentence φ is *valid*, written $\models \varphi$, iff φ holds in all interpretations, otherwise φ is *invalid*. The sentence φ is *satisfiable*, iff φ holds in some interpretation, otherwise φ is *unsatisfiable*.

Validity and satisfiability are only defined for closed formulas. By convention, we say that a (non-closed) formula φ is valid, iff its *universal closure* $\forall * . \varphi$ is valid. Dually, we call φ satisfiable, iff its *existential closure* $\exists * . \varphi$ is satisfiable.

The normal forms of propositional logic have counterparts in first-order logic. We restrict the presentation to the first-order extension of negation normal form as we will use it later in Section 3.4.

Definition 3.12 (Negation normal form (NNF))

A first order formula is in *negation normal form (NNF)* if it contains only \neg , \wedge , and \vee as logical connectives and negation appears only in literals. A *literal* is an atom or its negation.

$$\begin{array}{ll} \varphi_{nnf} & ::= l \mid \varphi_{nnf} \wedge \varphi_{nnf} \mid \varphi_{nnf} \vee \varphi_{nnf} & \text{NNF formulas} \\ & \mid \forall x. \varphi_{nnf} \mid \exists x. \varphi_{nnf} \\ l & ::= a \mid \neg a & \text{Literals} \end{array}$$

In order to transform a first-order formula to an equivalent formula in negation normal form, the equivalences for normalising propositional logic formulas from Section 3.1 may be applied from left to right. Additionally, the following two equivalences push down negation symbols into quantified formulas.

$$\begin{aligned} \neg \forall x. \varphi &\iff \exists x. \neg \varphi \\ \neg \exists x. \varphi &\iff \forall x. \neg \varphi \end{aligned}$$

The function and predicate symbols of a signature are purely syntactic. Their meanings are subject to interpretation. Coincidentally, we may decide to interpret the function symbol $+ \in \mathbb{F}_2$ as integer addition, but entirely different interpretations are possible, too. A first-order *theory* restricts the interpretation of non-logical symbols by specifying additional axioms.

Definition 3.13 (First-order theory)

A theory $T = (\Sigma, \mathcal{A})$ is defined by a signature Σ and a set \mathcal{A} of *axioms* that provide meaning to the non-logical symbols of Σ . Each axiom is a Σ -sentence.

The concepts of validity and satisfiability can be refined to take theory-specific axioms into account.

Definition 3.14 (*T*-validity, *T*-satisfiability)

For a given theory $T = (\Sigma, \mathcal{A})$, a Σ -sentence φ is said to be *T-valid* iff all interpretations that satisfy the axioms in \mathcal{A} also satisfy φ . The sentence φ is called *T-satisfiable*, iff there is an interpretation that satisfies both the axioms of T and φ .

A theory restricts the interpretation of the non-logical symbols in formulas. In contrast, restricting the syntax of a logic language yields a *fragment* of that logic. For example, the *quantifier-free fragment* of a theory T is the set of all T -valid formulas without quantifiers.

3.3 Quantifier-Free Fragments of First-Order Theories

This section briefly describes the quantifier-free fragments of first-order theories that the QUBE compiler uses to reason about expressions. Specifically, these are the theory of equality and uninterpreted functions, and linear integer arithmetic. A simple but useful first-order theory is the theory of equality in conjunction with uninterpreted functions (and predicates).

Definition 3.15 (*Equality with uninterpreted functions*)

The theory of equality with uninterpreted functions T_{EUF} extends an arbitrary signature $\Sigma = (\mathbb{F}, \mathbb{P})$ with a single interpreted equality predicate $= \in \mathbb{P}_2$.

The axioms of equality logic ensure that $=$ denotes an equivalence relation by prescribing that $=$ should be reflexive, symmetric, and transitive.

1. $\forall x. x = x$ (Reflexivity)
2. $\forall x, y. x = y \leftrightarrow y = x$ (Symmetry)
3. $\forall x, y, z. x = y \wedge y = z \rightarrow x = z$ (Transitivity)

Furthermore, all interpretations of function and predicate symbols should be consistent: functions and predicates must always evaluate to equal values when given equal arguments. Apart from functional consistency, symbols are uninterpreted.

The following axioms ensure that term and predicate construction is compatible with equality, such that it becomes a congruence relation.

4. for all function symbols $f \in \mathbb{F}_n$
 $\forall x^n, y^n. \left(\bigwedge_i x_i = y_i \right) \rightarrow (f(x^n) = f(y^n))$ (Function congruence)
5. for all predicate symbols $p \in \mathbb{P}_n$
 $\forall x^n, y^n. \left(\bigwedge_i x_i = y_i \right) \rightarrow (p(x^n) \leftrightarrow p(y^n))$ (Predicate congruence)

Satisfiability checking in the quantifier-free fragment of T_{EUF} is decidable. In computer aided verification, uninterpreted functions are widely used as substitutes for functions whose exact semantics is either too complex for mechanical proof or even entirely irrelevant for proving the formula at hand.

Example 3.16 (Equality checking with uninterpreted functions)

Functional consistency is sufficient to prove the equality of the expressions

1. let $x = a * b$ in let $y = x * c$ in y , and
2. $(a * b) * c$.

Rewrite in T_{EUF} , and prove the validity of

$$x = f_{mul}(a, b) \wedge y = f_{mul}(x, c) \rightarrow y = f_{mul}(f_{mul}(a, b), c),$$

or equivalently, prove the unsatisfiability of

$$x = f_{mul}(a, b) \wedge y = f_{mul}(x, c) \wedge y \neq f_{mul}(f_{mul}(a, b), c).$$

Merely relying on functional consistency is insufficient to prove all correct statements. Thus, validity checking of an arbitrary formula by approximating it in T_{EUF} is sound but not complete. For example, equalities arising from the commutativity of addition such as $a + b = b + a$ cannot be proved in T_{EUF} . However, in the context of array programming, where ranks, shapes, and array indices are all (vectors of) integers, it is particularly important to accurately reason about (in-)equalities of integers. The theory of linear integer arithmetic allows us to express and reason about such constraints.

Definition 3.17 (Linear integer arithmetic)

The signature of the theory of linear integer arithmetic provides an infinite set $\mathbb{F}_0 = \mathbb{Z}$ of constant symbols, an infinite set of unary function symbols $\mathbb{F}_1 = \{c \cdot \mid c \in \mathbb{Z}\}$, two binary function symbols $\mathbb{F}_2 = \{+, -\}$ and two binary predicate symbols $\mathbb{P}_2 = \{=, <\}$.

Intuitively rather than axiomatically, the theory is interpreted in the domain of integers where the constant symbols have interpretations as integer values. Similarly, the function and predicate symbols are interpreted as the corresponding functions and relations over the integers.

Linear integer arithmetic is an extension of *Presburger arithmetic* [85]. Additional functions and predicates can be expressed in terms of the existing, for example, $a \leq b \leftrightarrow a = b \vee a < b$. However, the theory cannot be extended to capture true multiplication as this would yield *Peano arithmetic* which, according to Gödel's first incompleteness theorem [48], is undecidable.

3.4 Array Properties

The theories presented in the previous sections allow us to express linear constraints between integer variables. In the context of rank-generic array programming, we also need to express constraints between entire vectors of integers although their lengths may not be statically known. For example, when an array of rank r and shape vector s is passed to a function that expects an array of shape r' and shape vector s' , the compiler must verify that $r = r'$ and $\forall i. 0 \leq i < r \rightarrow s[i] = s'[i]$. Such vector constraints can be expressed in the array property fragment [18] which is a decidable fragment of quantified array logic. In array logic, there are two basic array operations: reading and writing. The term $a[i]$ reads the element at position i from a . The term $a[i \leftarrow e]$ denotes the array a where the element at position i has been replaced by e . The theories used to reason about the indices and the elements are called the index theory and the element theory, respectively.

Definition 3.18 (Array property)

A formula is called an array property if and only if it has the form

$$\forall x_i. \varphi_I \rightarrow \varphi_V,$$

where the variable x_i belongs to a designated set of index identifiers. The formula must satisfy the following syntactic conditions:

1. The *index guard* φ_I must obey the grammar

$$\begin{array}{lll} \varphi_I & ::= & \varphi_I \wedge \varphi_I \mid \varphi_I \vee \varphi_I \mid t_i \leq t_i \mid t_i = t_i & \text{Index guards} \\ t_i & ::= & x_i \mid t & \text{Index terms} \\ t & ::= & \mathbb{Z} \mid \mathbb{Z} \cdot x \mid t + t & \text{Terms} \end{array}$$

The identifier x used in a term must not be an index identifier.

2. In the *value constraint* φ_V , the index identifier x_i may only be used in array read functions of the form $a[x_i]$. The read cannot be nested, for example, $a[b[x_i]]$ is not allowed.

The satisfiability problem for a formula in the array property fragment is decided

by reduction to an equisatisfiable quantifier-free formula that uses the index and element theories. Given a formula φ , the reduction proceeds in the following steps:

1. Convert φ to NNF.
2. Remove writes from φ by applying the write rule exhaustively:

$$\frac{\varphi(a[i \leftarrow v])}{\varphi(a') \wedge a'[i] = v \wedge (\forall j. j \neq i \rightarrow a[j] = a'[j])} \text{ (WRITE)}$$

where a' is a fresh variable. Rephrase $j \neq i$ to meet the syntax of array properties.

3. Replace all existential quantifications $\exists i. \varphi_E(i)$ in φ by $\varphi_E(j)$, where j is a fresh variable.
4. Instantiate the universal quantifiers. Replace quantifications of the form $\forall i. \varphi_A(i)$ by $\bigwedge_{i \in \mathcal{I}(\varphi)} \varphi_A(i)$, where the *index set* $\mathcal{I}(\varphi)$ contains all relevant symbolic indices that occur in the entire formula φ after step 3:

$$\mathcal{I}(\varphi) = \bigcup \left\{ \begin{array}{l} \{t \mid \cdot[t] \in \varphi \text{ such that } t \text{ is not a universally quantified variable}\} \\ \{t \mid t \text{ occurs as a term in an index guard}\} \end{array} \right.$$

The resulting formula can then be decided with the corresponding decision procedures.

The syntax of the array property fragment demands that universally quantified variables x_i may only be used in the value constraint in array selections of the form $a[x_i]$. Neither nested reads of the form $b[a[x_i]]$ nor any kind of arithmetic expressions involving universally quantified variables such as $x_i + 1$ are allowed. It was shown in [18] that permitting either of the above expressions yields a fragment for which the satisfiability problem is undecidable.

As a consequence, array properties can only be used to compare vector elements at corresponding index positions. The fragment suffices to express that all elements of some vectors v and w of length n are equal. For the sake of brevity, we write $0 \leq i < n$ instead of $0 \leq i \wedge i \leq n + 1$.

Example 3.19 (Vector equality)

$$\boxed{\forall i. 0 \leq i < n \rightarrow v[i] = w[i]}$$

For an access into a multidimensional array a of rank r and shape vector s with an index vector v , all elements of v must range between 0 and the corresponding elements of s . There is an array property for that, too.

Example 3.20 (Bounded vector)

$$\forall i. 0 \leq i < n \rightarrow 0 \leq v[i] \wedge v[i] < s[i]$$

As a negative example, the concatenation of vectors v and v' with respective lengths n and n' cannot be interpreted in the array property fragment as this requires index shifting.

Example 3.21 (Concatenation is not expressible with array properties)

$$\forall i. (0 \leq i < n \rightarrow w[i] = v[i]) \wedge (n \leq i < n + n' \rightarrow w[i] = \underline{v'[i - n]})$$

Similarly, dropping m elements from the beginning of a vector v of length n is also not expressible as an array property.

Example 3.22 (Drop is not expressible with array properties)

$$\forall i. 0 \leq i < n - m \rightarrow w[i] = \underline{v[i + m]}$$

Summary

First-order logic is a framework that provides a generic syntax for defining custom theories that restrict the non-logical function and predicate symbols. The QUBE compiler uses the quantifier-free fragments of the theory of equality and uninterpreted functions as well as linear integer arithmetic to statically reason about programs. Using the Nelson-Oppen procedure [78, 79, 66], the decision procedures for both theories can be combined so that a decision procedure for the combined theory is obtained. Hence, many modern SMT (Satisfiability Modulo Theories) solvers such as YICES [33], Z3 [30] provide support for these theories.

In order to reason about integer vectors, which represent array shapes and index vectors, the QUBE compiler uses the array property fragment which reduces to the aforementioned theories.

Part II

A Formal Treatment of QUBE

4

A Core Language for Array Programming

This chapter formally discusses the syntax and the operational semantics of the QUBE programming language. To achieve a rigorous presentation, we specify a core language $\text{QUBE}_{\text{CORE}}$ that captures the essential concepts of QUBE without syntactic sugar or convenience features. $\text{QUBE}_{\text{CORE}}$ is a typed language that uses type annotations to guide the type system. The syntax of $\text{QUBE}_{\text{CORE}}$ is designed to superficially resemble the syntax of OCAML so that readers familiar with OCAML should recognise many of the language elements.

The language comprises three conceptual layers. Its foundation QUBE_{λ} (QUBE fun) is an applied λ -calculus with dependent types that provides essential programming features like abstractions, applications, tuples and let-bindings. The next layer $\text{QUBE}_{\rightarrow}$ (QUBE vector) adds support for specifying and manipulating integer vectors that serve as shape and index vectors. The topmost layer QUBE_{\square} (QUBE array) provides means for defining, accessing and manipulating multidimensional arrays.

Figure 4.1 shows the abstract syntax of $\text{QUBE}_{\text{CORE}}$ in extended Backus-Naur form. For every layer, the language defines types T , expressions e , and values v . Values form a designated subset of expressions which are accepted as legal results of evaluation when no further step is possible. Types classify expressions according to the values they compute.

The notational conventions and the variable convention carry over from Chap-

Syntax

T	$::= T_\lambda \mid T_\rightarrow \mid T_\square$	Types
e	$::= e_\lambda \mid e_\rightarrow \mid e_\square$	Expressions
v	$::= v_\lambda \mid v_\rightarrow \mid v_\square$	Values
<hr/>		
T_λ	$::= B \mid \{x:T \mid e\} \mid x:T \rightarrow T \mid (x:T, T)$	Types
B	$::= \text{bool} \mid \text{int}$	Base types
e_λ	$::= c \mid x \mid \text{fun } x:T \rightarrow e \mid e e$ $\mid \text{let } x = e \text{ in } e \mid \text{if } :T e \text{ then } e \text{ else } e \mid (e:T)$ $\mid (e, e:(x:T, T)) \mid \text{let } (x, x) = e \text{ in } e$	Expressions
c	$::= \mathbb{B} \mid \mathbb{Z} \mid f^1 \mid f^2$	Constants
f^1	$::= \text{not}$	Unops
f^2	$::= \leftrightarrow \mid \& \mid \mid = \mid != \mid < \mid <= \mid >= \mid > \mid + \mid - \mid * \mid / \mid \%$	Binops
v_λ	$::= c \mid f^2 v \mid \text{fun } x:T \rightarrow e \mid (v, v:(x:T, T))$	Values
<hr/>		
T_\rightarrow	$::= \text{intvec } e$	Vector type
e_\rightarrow	$::= [[e\{, e\}]] \mid e.(e) \mid e.(e) \leftarrow e \mid \text{vec } e e$ $\mid \text{vmap } e e\{, e\} (x\{, x\} \rightarrow e)$ $\mid \text{vfa } e e\{, e\} (x\{, x\} \rightarrow e)$	Vector exprs.
v_\rightarrow	$::= [[\mathbb{Z}\{, \mathbb{Z}\}]]$	Vector value
<hr/>		
T_\square	$::= [T \mid e\{, e\}]$	Array type
e_\square	$::= [[e\{, e\}] : T \mid [[\mathbb{Z}\{, \mathbb{Z}\}]]]$ $\mid e.[e\{, e\}] \mid e.[e\{, e\}] \leftarrow e \mid \text{reshape } e\{, e\} e$ $\mid \text{gen } :T e\{, e\} \text{ with } x\{, x\} \rightarrow e$ $\mid \text{loop } x:T = e; e\{, e\} \text{ with } x\{, x\} \rightarrow e$	Array exprs.
v_\square	$::= [[v\{, v\}] : T \mid [[\mathbb{Z}\{, \mathbb{Z}\}]]]$	Array value

Figure 4.1: Abstract syntax of QUBE_{CORE}

ter 2. As before, $e[x \mapsto e']$ denotes capture-avoiding substitution of all free occurrences of the variable x in the expression e with e' . Similarly, the notation $T[x \mapsto e]$ denotes capture-avoiding substitution of all free occurrences of the variable x in the type T with e . The following table gives an overview of the metavariables that we use in the remainder of the part.

Metavariable	Represented set
e	expressions
T	types
v	values
x	variables
c, f^1, f^2	constant and function symbols
n, i	integer constants

We use overlined metavariables such as \bar{a} to represent finite sequences of comma-separated metavariables $a_0, \dots, a_{|\bar{a}|-1}$ where $|\bar{a}|$ denotes the length of the sequence. The remainder of this chapter is organised as follows: Section 4.1 describes the functional language QUBE_λ that forms the foundation of QUBE_{CORE}. Section 4.2 presents the language layer QUBE_↓, that provides syntax for handling integer vectors. Support for multidimensional arrays in QUBE_□ is described in Section 4.3. After QUBE_{CORE} has been presented entirely, Section 4.4 presents essential properties of the evaluation relation.

4.1 QUBE_λ: a Functional Foundation

The language fragment QUBE_λ (QUBE fun) forms the basis of QUBE_{CORE}. It augments an applied λ -calculus with constant symbols, let-bindings, conditional expressions, and tuples. As its most significant features, QUBE_λ provides dependent types and refinement types.

Fig 4.2 shows the abstract syntax of QUBE_λ in isolation. In the course of this section, we will explain the types T and the expressions e in the same order as they appear in the figure. For every expression, we simultaneously discuss both its syntax and its operational semantics which is shown in Figure 4.3.

The types T_λ are either base types B or aggregate types. The base types `bool` and `int` describe truth values and integers, respectively. Other base types such as `unit`, `double`, or `char` may be added depending of the application.

A refinement type $\{x : T \mid e\}$ describes the subset of values x of type T that satisfy the boolean expression e . In particular, since `true` is satisfied by any x , the type $\{x : T \mid \text{true}\}$ is equivalent to T .

Syntax of QUBE_λ

T_λ	$::= B \mid \{x:T \mid e\} \mid x:T \rightarrow T \mid (x:T, T)$	Types
B	$::= \text{bool} \mid \text{int}$	Base types
e_λ	$::= c \mid x \mid \text{fun } x:T \rightarrow e \mid e e$ $\quad \mid \text{let } x = e \text{ in } e \mid \text{if } :T e \text{ then } e \text{ else } e \mid (e:T)$ $\quad \mid (e, e:(x:T, T)) \mid \text{let } (x, x) = e \text{ in } e$	Expressions
c	$::= \mathbb{B} \mid \mathbb{Z} \mid f^1 \mid f^2$	Constants
f^1	$::= \text{not}$	Unops
f^2	$::= \leftrightarrow \mid \& \mid \mid = \mid != \mid < \mid <= \mid >= \mid > \mid + \mid - \mid * \mid / \mid \%$	Binops
v_λ	$::= c \mid f^2 v \mid \text{fun } x:T \rightarrow e \mid (v, v:(x:T, T))$	Values

Figure 4.2: Abstract syntax of QUBE_λ

To illustrate refinement types, we define two type abbreviations that will be used frequently in the remainder: a type `nat` for natural numbers and a type `index` for integers that range between zero and an upper bound. Note that type abbreviations are not a part of $\text{QUBE}_{\text{CORE}}$. To obtain pure $\text{QUBE}_{\text{CORE}}$ types, we only need to replace the abbreviations with their definitions.

Example 4.1 (Refinement type)

```
type nat = { x:int | 0 <= x }
type index b:nat = { x:int | 0 <= x & x < b }
```

The dependent function type $x:T_1 \rightarrow T_2$ binds the variable x of the domain type T_1 in the codomain type T_2 . This allows the result type of a function to vary according to the supplied argument. For example, when an operator of type $x:T_1 \rightarrow T_2$ is applied to an operand e of type T_1 , the application has type $T_2[x \mapsto e]$. The usual (non-dependent) function type is a special case of the dependent function type whose codomain does not depend on the argument value, i. e., $T_1 \rightarrow T_2$ is equivalent to $x:T_1 \rightarrow T_2$ if $x \notin \mathcal{FV}(T_2)$.

To exemplify dependent function types, we provide types that precisely specify an integer identity function and safe integer division by means of refinement types.

Example 4.2 (Dependent function type)

```
id : x:int → { v:int | v = x }
/ : x:int → y:{ v:int | not (v = 0) } → { v:int | v = x / y }
```

Similar to the dependent function type, the dependent tuple type $(x:T_1, T_2)$

describes pairs (e_1, e_2) whose second components' type depends on the value of the first component, so that if e_1 has type T_1 , then e_2 has type $T_2[x \mapsto e_1]$. The dependent tuple type generalises the conventional Cartesian product: the type (T_1, T_2) is equivalent to $(x : T_1, T_2)$ if $x \notin \mathcal{FV}(T_2)$.

To illustrate dependent tuple types, we define type abbreviations for ascendingly ordered pairs of integers and, as a borderline case, for complex numbers.

Example 4.3 (Dependent tuple type)

```
type ordered_pair = (x:int, { v:int | v >= x })
type cpx = (int,int)
```

We now describe the expressions of QUBE_{CORE}. The constant symbols c comprise the booleans $\mathbb{B} = \{\text{true}, \text{false}\}$, the integers $\mathbb{Z} = \{.., -1, 0, 1, ..\}$, as well as the sets f^1, f^2 of built-in unary and binary functions. The only unary function symbol is the logical negation `not`. Binary function symbols are the logical connectives `&` (and), `|` (or), `↔` (iff), the relational operators `=`, `!=`, `<`, `<=`, `>=`, `>`, and the arithmetic operators `+`, `-`, `*`, `/`, `%`.

An abstraction `fun x : T → e` represents a function. It binds the variable x of type T in the abstraction body e . An abstraction is a value by itself. In particular, abstraction bodies are never evaluated. As a simple example, we specify a function for computing the arithmetic mean of two integers. Note that, like λ abstractions, `fun` associates to the right.

Example 4.4 (Function value: arithmetic mean)

```
fun x:int → fun y:int → (x + y) / 2
```

An application $e_1 e_2$ consists of an operator e_1 and an operand e_2 . Evaluation of applications follows a call-by-value regime. E-APP1 first evaluates the operator e_1 before E-APP2 evaluates the operand e_2 . If the application evaluates to a β -redex $(\text{fun } x : T \rightarrow e) v$, E-ABSAPP β -reduces the application by substituting all free occurrences of x in e with the evaluated operand. Applications of function symbols to legitimate arguments are evaluated via δ -reduction as described by the rules E-PRFAPP1 and E-PRFAPP2. Ill-formed applications of function symbols such as `1/0` or `0 = false` cannot be evaluated and are thus “stuck” expressions. Applications $f^2 v$ of a binary function symbol to a single value $f^2 v$ are considered values.

The conditional expression `if : T ep then et else ee` consists of the predicate e_p and the two branches e_t and e_e . The annotation `: T` guides the type checking mechanism by providing a common supertype for both branches. E-COND evaluates the predicate. If it evaluates to either `true` or `false`, the entire conditional

$e \Rightarrow e$

Evaluation rules

$$\frac{e_1 \Rightarrow e'_1}{e_1 e_2 \Rightarrow e'_1 e_2} \text{ (E-APP1)} \qquad \frac{e_2 \Rightarrow e'_2}{v_1 e_2 \Rightarrow v_1 e'_2} \text{ (E-APP2)}$$

$$(\text{fun } x : T \rightarrow e) v \Rightarrow e[x \mapsto v] \text{ (E-ABSAPP)}$$

$$\frac{f^1(v) \Rightarrow_{\delta} v'}{f^1 v \Rightarrow v'} \text{ (E-PRFAPP1)} \qquad \frac{f^2(v_1, v_2) \Rightarrow_{\delta} v_3}{(f^2 v_1) v_2 \Rightarrow v_3} \text{ (E-PRFAPP2)}$$

$$\frac{e_1 \Rightarrow e'_1}{\text{let } x = e_1 \text{ in } e_2 \Rightarrow \text{let } x = e'_1 \text{ in } e_2} \text{ (E-LETE)}$$

$$\text{let } x = v_1 \text{ in } e_2 \Rightarrow e_2[x \mapsto v_1] \text{ (E-LET)}$$

$$\frac{e_1 \Rightarrow e'_1}{\text{if } :T e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow \text{if } :T e'_1 \text{ then } e_2 \text{ else } e_3} \text{ (E-COND)}$$

$$\text{if } :T \text{ true then } e_2 \text{ else } e_3 \Rightarrow e_2 \text{ (E-CONDT)}$$

$$\text{if } :T \text{ false then } e_2 \text{ else } e_3 \Rightarrow e_3 \text{ (E-CONDE)}$$

$$\frac{e \Rightarrow e'}{(e : T) \Rightarrow (e' : T)} \text{ (E-COERCES)} \qquad (v : T) \Rightarrow v \text{ (E-COERCE)}$$

$$\frac{e_1 \Rightarrow e'_1}{(e_1, e_2 : T) \Rightarrow (e'_1, e_2 : T)} \text{ (E-TUP1)} \qquad \frac{e_2 \Rightarrow e'_2}{(v_1, e_2 : T) \Rightarrow (v_1, e'_2 : T)} \text{ (E-TUP2)}$$

$$\frac{e_1 \Rightarrow e'_1}{\text{let } (x_1, x_2) = e_1 \text{ in } e_2 \Rightarrow \text{let } (x_1, x_2) = e'_1 \text{ in } e_2} \text{ (E-UNPACKE)}$$

$$\text{let } (x_1, x_2) = (v_{11}, v_{12} : T) \text{ in } e_2 \Rightarrow e_2[x_1 \mapsto v_{11}][x_2 \mapsto v_{12}] \text{ (E-UNPACK)}$$

Figure 4.3: Evaluation rules of QUBE_{λ}

evaluates to e_t (E-COND_T) or e_e (E-COND_E), respectively. The following example uses a conditional to define a safe integer division function.

Example 4.5 (Safe division)

```
fun a:int → fun b:int →
  if :int b != 0 then a / b else 0
```

The let-binding `let $x = e_1$ in e_2` binds the variable x to the value of the right-hand side e_1 in the body expression e_2 . E-LETE first evaluates the right-hand side expression. If this yields a value v_1 , E-LET evaluates the entire let-binding to $e_2[x \mapsto v_1]$.

As an example, we define a function `squared_difference` that uses a let-binding to avoid redundant computation of the difference.

Example 4.6 (Let-binding)

```
let squared_difference = fun a:int → fun b:int →
  let d = a - b in
  d * d
in ...
```

The coercion $(e : T)$ serves to up-coerce an expression e to a type T . The expression is evaluated by E-COERCE_S. If e evaluates to v , the entire coercion evaluates to v after rule E-COERCE.

The tuple constructor $(e_1, e_2 : (x : T_1, T_2))$ forms a (dependent) tuple with components e_1 and e_2 . The type annotation $(x : T_1, T_2)$ is necessary as the typing of dependent tuples is ambiguous. For example, the tuple $(0, 1)$ may be regarded as a pair of integers (int, int) , as an ordered pair $(x : \text{int}, \{ v : \text{int} \mid v \geq x \})$, or as any other type that describes the tuple. The rules E-TUP1 and E-TUP2 evaluate the expressions from left to right, yielding the value $(v_1, v_2 : (x : T_1, T_2))$.

The unpack expression `let $(x_1, x_2) = e_1$ in e_2` projects the component values from a tuple. The expression binds the variables x_1 and x_2 to the respective component values of the right-hand side e_1 in the body expression e_2 . E-UNPACK_E evaluates the right-hand side expression. If e_1 evaluates to a tuple value $(v_{11}, v_{12} : (x : T_1, T_2))$, E-UNPACK evaluates the entire expression to e_2 , in the process replacing x_1 with v_{11} and x_2 with v_{12} , respectively. The example adds the corresponding elements of two ordered pairs, which again yields an ordered pair.

Example 4.7 (Adding ordered pairs)

```

let ordered_add = fun a:ordered_pair → fun b:ordered_pair →
  let (a1,a2) = a in
  let (b1,b2) = b in
  (a1 + b1, a2 + b2 : ordered_pair)

```

4.2 QUBE_→: Integer Vectors

Building on QUBE_λ, the language layer QUBE_→ (QUBE vector) adds support for integer vectors. In the context of array programming, these vectors are particularly important as array shapes and as index vectors into multidimensional arrays. QUBE_→ provides language elements for defining, accessing, and manipulating integer vectors. A vector comprehension allows the user to specify arbitrary operations that combine corresponding elements from multiple vectors. These operations can be encoded in the array property fragment of first-order logic, so that the compiler can statically reason about the values of integer vectors.

Figure 4.4 shows the abstract syntax of QUBE_→, its operational semantics is shown in Figure 4.5. The evaluation rule E-SEQ defines the auxiliary relation $\bar{e} \Rightarrow_{seq} \bar{e}$ that evaluates sequences \bar{e} of expressions from left to right.

To describe integer vectors, QUBE_→ introduces a new type `intvec e` where the expression e describes the length of the vector. Since vectors cannot have negative extent, e must denote a natural number. The vector type is refinable so that properties of entire vectors can be encoded in the type.

The vector constructor $[\bar{e}]$ defines a vector with elements \bar{e} . The evaluation rule E-VECTOR makes use of the relation $\bar{e} \Rightarrow_{seq} \bar{e}$ to evaluate the vector elements. If all elements evaluate to integers, the resulting vector $[\bar{\mathbb{Z}}]$ is a value.

The vector selection $e.(e_i)$ selects the element at index e_i from the vector e . The rules E-VSELV and E-VSELI evaluate the vector and the index, respectively. If

Syntax of QUBE_→

T_{\rightarrow}	$::=$	<code>intvec e</code>	Vector type
e_{\rightarrow}	$::=$	<code>[[e{,e}]] e.(e) e.(e) ← e vec e e</code>	Vector exprs.
		<code>vmap e e{,e} (x{,x} → e)</code>	
		<code>vfa e e{,e} (x{,x} → e)</code>	
v_{\rightarrow}	$::=$	<code>[[Z{,Z}]]</code>	Vector value

Figure 4.4: Abstract syntax of QUBE_→

Evaluation rules, continued

 $e \Rightarrow e$

$$\frac{e_i \Rightarrow e'_i}{v_0, \dots, v_{i-1}, e_i, \dots, e_{n-1} \Rightarrow_{seq} v_0, \dots, v_{i-1}, e'_i, \dots, e_{n-1}} \text{ (E-SEQ)}$$

$$\frac{\bar{e} \Rightarrow_{seq} \bar{e}'}{[\bar{e}] \Rightarrow [\bar{e}']} \text{ (E-VECTOR)}$$

$$\frac{e \Rightarrow e'}{e.(e_i) \Rightarrow e'.(e_i)} \text{ (E-VSELV)} \quad \frac{e_i \Rightarrow e'_i}{v.(e_i) \Rightarrow v.(e'_i)} \text{ (E-VSELI)}$$

$$\frac{0 \leq i < |\bar{v}|}{[\bar{v}].(i) \Rightarrow v_i} \text{ (E-VSEL)}$$

$$\frac{e \Rightarrow e'}{e.(e_i) \leftarrow e_e \Rightarrow e'.(e_i) \leftarrow e_e} \text{ (E-VMV)} \quad \frac{e_i \Rightarrow e'_i}{v.(e_i) \leftarrow e_e \Rightarrow v.(e'_i) \leftarrow e_e} \text{ (E-VMI)}$$

$$\frac{e_e \Rightarrow e'_e}{v.(v_i) \leftarrow e_e \Rightarrow v.(v_i) \leftarrow e'_e} \text{ (E-VME)} \quad \frac{0 \leq i < |\bar{v}|}{[\bar{v}].(i) \leftarrow v_e \Rightarrow [\bar{v}[i \mapsto v_e]]} \text{ (E-VMOD)}$$

$$\frac{e_n \Rightarrow e'_n}{\text{vec } e_n e_e \Rightarrow \text{vec } e'_n e_e} \text{ (E-VECL)} \quad \frac{e_e \Rightarrow e'_e}{\text{vec } v_n e_e \Rightarrow \text{vec } v_n e'_e} \text{ (E-VECE)}$$

$$\frac{0 \leq n}{\text{vec } n v \Rightarrow \underbrace{[v, \dots, v]}_n} \text{ (E-VEC)}$$

$$\frac{e_n \Rightarrow e'_n}{\text{vmap } e_n \bar{e} f \Rightarrow \text{vmap } e'_n \bar{e} f} \text{ (E-VMAPL)} \quad \frac{\bar{e} \Rightarrow_{seq} \bar{e}'}{\text{vmap } v_n \bar{e} f \Rightarrow \text{vmap } v_n \bar{e}' f} \text{ (E-VMAPV)}$$

$$\frac{0 \leq n \quad (n = |\bar{v}_i|)_i}{\text{vmap } n [\bar{v}_0], \dots, [\bar{v}_m] (x_0, \dots, x_m \rightarrow e) \Rightarrow [\bar{e}']} \text{ (E-VMAP)}$$

where $e'_j = e[x_0 \mapsto v_{0j}]..[x_m \mapsto v_{mj}]$

$$\frac{e_n \Rightarrow e'_n}{\text{vfa } e_n \bar{e} p \Rightarrow \text{vfa } e'_n \bar{e} p} \text{ (E-VFAL)} \quad \frac{\bar{e} \Rightarrow_{seq} \bar{e}'}{\text{vfa } v_n \bar{e} p \Rightarrow \text{vfa } v_n \bar{e}' p} \text{ (E-VFAV)}$$

$$\frac{0 \leq n \quad (n = |\bar{v}_i|)_i}{\text{vfa } n [\bar{v}_0], \dots, [\bar{v}_m] (x_0, \dots, x_m \rightarrow e) \Rightarrow e'_0 \&.. \& e'_{n-1}} \text{ (E-VFA)}$$

where $e'_j = e[x_0 \mapsto v_{0j}]..[x_m \mapsto v_{mj}]$

Figure 4.5: Evaluation rules of QUBE_→

the vector evaluates to a value $[\bar{v}]$ and the index evaluates to an integer i with $0 \leq i < |\bar{v}|$, then E-VSEL yields the element v_i .

The vector modification $e.(e_i) \leftarrow e_e$ yields a modified version of the vector e in which the element at the index e_i is replaced with the new element e_e . Vector modification is non-destructive, i. e., the original vector is preserved as its elements are copied to the new vector value. The rules E-VMV, E-VMI, and E-VME evaluate the three subexpressions from left to right. If the vector and the index expression evaluate to a vector value $[\bar{v}]$ and an appropriate integer index i , and the new element successfully evaluates to a value v_e , then E-VECE yields a vector with elements \bar{v} except that v_i is replaced with v_e .

The following example illustrates the non-destructive behaviour of the vector modification. Vector b is defined as a modification of vector a . Whereas $b.(2)$ reflects the new element, the corresponding element of a remains unchanged.

Example 4.8 (Non-destructive vector modification)

<pre>let a = [1,2,3] in let b = a.(2) ← a.(0) in a.(2)</pre>	<pre>(* b = [1,2,1] *) ⇒* 3,</pre>
--	------------------------------------

Unlike the vector constructor $[\bar{e}]$ which creates a vector of fixed length with distinct elements, the constant-value vector expression $\text{vec } e_n e_e$ defines a vector of non-constant length e_n that only contains copies of the element e_e . The length and the element are evaluated by the rules E-VECL and E-VECE, respectively. If the length evaluates to a non-negative integer n and the element evaluates to a value v , the entire expression evaluates to a vector that contains n copies of v after rule E-VEC.

To illustrate constant-value vectors, we define a null vector whose length is determined by an expression.

Example 4.9 (Null vector with dynamic length)

<pre>let n = 1+3 in vec n 0</pre>	<pre>⇒* [0,0,0,0]</pre>
-----------------------------------	-------------------------

The vector comprehension $\text{vmap } e_n \bar{e} f$ simultaneously applies a k -ary function f to all corresponding elements of the argument vectors e_1, \dots, e_k of length e_n , where $k \geq 1$. Intuitively, the resulting vector is specified by $(\text{vmap } e_n e_1, \dots, e_k f).(i) = f(e_1.(i), \dots, e_k.(i))$. The function f has the form $(\bar{x} \rightarrow e)$ and may be understood as an in-place abstraction of the variables \bar{x} from the body e . The length and the argument vectors are evaluated by the rules E-VMAPL and E-VMAPV. if the length evaluates to an integer n and all vectors evaluate to vector values of length

n , then V-MAP evaluates the expression to a vector constructor. It applies the function $(\bar{x} \rightarrow e)$ to the respective vector elements by substituting the variables \bar{x} with the actual values.

As an example, we use `vmap` to specify an expression that computes the element-wise sum of two vectors.

Example 4.10 (Vector sum)

```
let av = [1,2,3,4] in
let bv = [1,1,1,1] in
vmap 4 av,bv (a,b → a+b)           ⇒* [2,3,4,5]
```

Similar to `vmap`, the vector predicate $\text{vfa } e_n \bar{e} p$ checks whether a k -ary predicate p holds for all corresponding elements of the vectors e_1, \dots, e_k of length e_n , with $k \geq 1$. The result is specified by the formula $\forall i. 0 \leq i < e_n. p(e_1.(i), \dots, e_k.(i))$. The predicate is an in-place abstraction $(\bar{x} \rightarrow e)$ with variables \bar{x} and a body e . The length and the argument vectors are evaluated by E-VFAL and E-VFAV. If the length evaluates to an integer n and all vectors evaluate to vector values of length n , then V-VFA evaluates the entire expression to a conjunction of applications of p to the corresponding vector elements.

To illustrate how vector predicates may be used to refine vector types, we define the type `natvec` for vectors of natural numbers and the type `indexvec` that describes vectors whose elements range between 0 and the corresponding elements of a boundary vector.

Example 4.11 (Refining vector types)

```
type natvec n:nat = { xv:intvec n | vfa n xv (x → 0 <= x) }
type indexvec n:nat bv:(natvec n) =
  { xv:intvec n | vfa n xv,bv (x,b → 0 <= x & x < b) }
```

4.3 QUBE_□: Multidimensional Arrays

The third language layer QUBE_□ (QUBE array) adds support for multidimensional arrays and rank-generic programming. The layer relies on QUBE_→, since array shapes and indices into multidimensional arrays are integer vectors. QUBE_□ provides means for defining, accessing and modifying arrays. A rank-generic array comprehension allows for the convenient specification of powerful array operations. The abstract syntax of QUBE_□ is shown in Figure 4.6, the corresponding operational semantics is detailed in Figure 4.7.

Syntax of QUBE_□

T_{\square}	$::= [T e\{,e\}]$	Array type
e_{\square}	$::= [[e\{,e\}] : T [[Z\{,Z\}]]]$	
	$e.[e\{,e\}] \mid e.[e\{,e\}] \leftarrow e \mid \text{reshape } e\{,e\} \ e$	Array exprs.
	$\text{gen} : T \ e\{,e\} \ \text{with } x\{,x\} \rightarrow e$	
	$\text{loop } x : T = e ; e\{,e\} \ \text{with } x\{,x\} \rightarrow e$	
v_{\square}	$::= [[v\{,v\}] : T [[Z\{,Z\}]]]$	Array value

Figure 4.6: Abstract syntax of QUBE_□

The type of arrays is $[T | \bar{e}]$ where T is the common type of the array elements and \bar{e} describes the array shape as a structured vector, i. e., a comma-separated sequence of vectors. Structured vectors allow us to specify array shapes and index vectors as compositions of multiple independent vectors. For example, an integer matrix of shape $[2, 3]$ is described by the type $[\text{int} | [2, 3]]$ but also by $[\text{int} | [2], [3]]$ or even $[\text{int} | fs, cs]$ for appropriate expressions fs and cs . The type of the rank-generic outer product `oprod` makes use of a structured array shape: given an array of type $[\text{int} | sa]$ and an array of type $[\text{int} | sb]$, the function yields an array of type $[\text{int} | sa, sb]$.

Example 4.12 (Type of rank-generic outer product)

$$\text{oprod} : ra : \text{nat} \rightarrow rb : \text{nat} \rightarrow sa : (\text{natvec } ra) \rightarrow sb : (\text{natvec } rb) \rightarrow$$

$$[\text{int} | sa] \rightarrow [\text{int} | sb] \rightarrow [\text{int} | sa, sb]$$

The array constructor $[\bar{e} : T | [\bar{n}]]$ defines a multidimensional array with elements \bar{e} of the element type T and shape $[\bar{n}]$ (and thus rank $|\bar{n}|$). As a data type invariant, the array shape must not have negative shape components and the number of array elements must equal the product of the shape vector. Since these restrictions go beyond mere syntax, they are enforced by the type checker. The annotated element type serves two purposes. First, it provides a common supertype for the array elements. Second, it allows the type checker to determine the type of an empty array such as $[: \text{int} | [0, 10]]$. The evaluation rule E-ARR evaluates the elements from left to right by applying the relation $\bar{e} \Rightarrow_{seq} \bar{e}$.

An array of values $[\bar{v} : T | [\bar{n}]]$ is itself a value, where \bar{v} is the data vector and \bar{n} is the shape vector. The data vector contains the array elements in row-major order, i. e., the order of array elements in the data vector is determined by the lexicographic order of their corresponding index vectors. Fig. 4.8 shows the representation of some multidimensional arrays in QUBE_{CORE}.

The array selection $e_a.[\bar{e}]$ selects the element indexed by the structured index

Evaluation rules, continued

 $e \Rightarrow e$

$$\frac{\bar{e} \Rightarrow_{seq} \bar{e}'}{[\bar{e} : T \mid \bar{n}] \Rightarrow [\bar{e}' : T \mid \bar{n}]} \text{ (E-ARR)}$$

$$\frac{e_a \Rightarrow e'_a}{e_a \cdot [\bar{e}] \Rightarrow e'_a \cdot [\bar{e}]} \text{ (E-SELA)} \qquad \frac{\bar{e} \Rightarrow_{seq} \bar{e}'}{v_a \cdot [\bar{e}] \Rightarrow v_a \cdot [\bar{e}']} \text{ (E-SELI)}$$

$$\frac{|\bar{n}| = |\bar{i}_1, \dots, \bar{i}_m| \quad (0 \leq (\bar{i}_1, \dots, \bar{i}_m)_j < n_j)_j}{[\bar{v} : T \mid \bar{n}] \cdot [[\bar{i}_1], \dots, [\bar{i}_m]] \Rightarrow v_{idx}} \text{ (E-SEL)}$$

where $idx = \iota \bar{n} (\bar{i}_1, \dots, \bar{i}_m)$

$$\frac{e_a \Rightarrow e'_a}{e_a \cdot [\bar{e}] \leftarrow e_e \Rightarrow e'_a \cdot [\bar{e}] \leftarrow e_e} \text{ (E-MA)} \qquad \frac{\bar{e} \Rightarrow_{seq} \bar{e}'}{v_a \cdot [\bar{e}] \leftarrow e_e \Rightarrow v_a \cdot [\bar{e}'] \leftarrow e_e} \text{ (E-MI)}$$

$$\frac{e_e \Rightarrow e'_e}{v_a \cdot [\bar{v}] \leftarrow e_e \Rightarrow v_a \cdot [\bar{v}] \leftarrow e'_e} \text{ (E-ME)}$$

$$\frac{|\bar{n}| = |\bar{i}_1, \dots, \bar{i}_m| \quad (0 \leq (\bar{i}_1, \dots, \bar{i}_m)_j < n_j)_j}{[\bar{v} : T \mid \bar{n}] \cdot [[\bar{i}_1], \dots, [\bar{i}_m]] \leftarrow v_e \Rightarrow [\bar{v}[idx \mapsto v_e] : T \mid \bar{n}]} \text{ (E-MOD)}$$

where $idx = \iota \bar{n} (\bar{i}_1, \dots, \bar{i}_m)$

$$\frac{\bar{e} \Rightarrow_{seq} \bar{e}'}{\text{reshape } \bar{e} e_a \Rightarrow \text{reshape } \bar{e}' e_a} \text{ (E-RS)} \qquad \frac{e_a \Rightarrow e'_a}{\text{reshape } \bar{v} e_a \Rightarrow \text{reshape } \bar{v} e'_a} \text{ (E-RA)}$$

$$\frac{(0 \leq i_{jk})_{jk} \quad (0 < n_j)_j}{\text{reshape } [\bar{i}_1], \dots, [\bar{i}_m] \quad [\bar{v} : T \mid \bar{n}] \Rightarrow [\bar{v}' : T \mid [\bar{i}_1, \dots, \bar{i}_m]]} \text{ (E-RSHP)}$$

where $v'_k = v_{k \bmod \prod \bar{n}}$

$$\frac{\bar{e} \Rightarrow_{seq} \bar{e}'}{\text{gen} : T \bar{e} \text{ with } \bar{x} \rightarrow e_b \Rightarrow \text{gen} : T \bar{e}' \text{ with } \bar{x} \rightarrow e_b} \text{ (E-GENS)}$$

$$\frac{(0 \leq n_{ij})_{ij}}{\text{gen} : T [\bar{n}_1], \dots, [\bar{n}_m] \text{ with } x_1, \dots, x_m \rightarrow e_b \Rightarrow [\bar{e} : T \mid [\bar{n}_1, \dots, \bar{n}_m]]} \text{ (E-GEN)}$$

where $|e| = \prod (\bar{n}_1, \dots, \bar{n}_m)$ $e_k = e_b[x_j \mapsto [\bar{i}_j]]_j$ such that $\iota (\bar{n}_1, \dots, \bar{n}_m) (\bar{i}_1, \dots, \bar{i}_m) = k$ Figure 4.7: Evaluation rules of QUBE_□

Array	Uniform array representation
1	[1 : int []]
[1 2 3]	[1,2,3 : int [3]]
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	[1,2,3,4,5,6 : int [2,3]]
	[1,2,3,4,5,6,7,8,9,10,11,12 : int [2,2,3]]

Figure 4.8: Representation of multidimensional arrays in QUBE_{CORE}

vector \bar{e} from the array e_a . The array and the index vector are evaluated by the rules E-SELA and E-SELI, respectively. The selection requires that the array evaluates to a value $[\bar{v} : T \mid [\bar{n}]]$ and that \bar{e} evaluates to a sequence of integer vectors $[\bar{i}_1], \dots, [\bar{i}_m]$ which jointly form a valid index vector $\bar{i}\bar{v} = \bar{i}_1, \dots, \bar{i}_m$ into the array, i. e., the length of $\bar{i}\bar{v}$ must equal the array's rank $|\bar{n}|$ and all elements $i\bar{v}_j$ must range between 0 and the corresponding shape element n_j . If all the preconditions are met, the selection yields the array element v_{idx} . The appropriate linear offset idx into the data vector \bar{v} is determined by the offset computation function ι (iota).

Definition 4.13 (Offset computation function ι)

Let \bar{s} with $|\bar{s}| = r$ be a shape vector and \bar{v} with $|\bar{v}| = r$ be an appropriate index vector into an array of shape \bar{s} . The offset computation function $\iota \bar{s} \bar{v}$ (iota) is defined as follows:

$$\iota \bar{s} \bar{v} = \sum_{j=0}^{r-1} (v_j \cdot \prod_{k=j+1}^{r-1} s_k)$$

The example shows two equivalent selections into a matrix of shape [2,2], one uses a single vector, the other a structured vector.

Example 4.14 (Array selection)

[1,2,3,4 :int [2,2]]. [[0,1]]	⇒ 2
[1,2,3,4 :int [2,2]]. [[0],[1]]	⇒ 2

The array modification $e_a.\bar{e}] \leftarrow e_e$ modifies the array e_a by replacing the element indexed by the (structured) index vector \bar{e} with the new element e_e . Array modification is non-destructive: the original array is preserved as its elements are copied to the new array value. The three subexpressions are evaluated from left to right by the rules E-MA, E-MI, and E-ME. Similar to the selection, the array modification requires that the array evaluates to a value $[\bar{v} : T | [n]]$ and that the structured vector \bar{e} evaluates to a valid index vector into that array. If e_e also evaluates to a value v_e then E-MOD yields an array equal to the original one except that the element idx in \bar{v} is replaced with v_e . The linear offset idx is once more defined by the index computation function ι .

As an example, we show two equivalent modification of a matrix of shape [2,2], one uses a single vector, the other a structured vector.

Example 4.15 (Array modification)

[1,2,3,4 :int [2,2]]. [[0,1]]	← 0	⇒ [1,0,3,4 :int [2,2]]
[1,2,3,4 :int [2,2]]. [[0],[1]]	← 0	⇒ [1,0,3,4 :int [2,2]]

The reshape operation $\text{reshape } \bar{e} e_a$ gives a new shape \bar{e} to an array e_a . The evaluation rules E-RS and E-RA evaluate the new shapes and the array from left to right. Provided that the array evaluates to a non-empty array value $[\bar{v} : T | [n]]$ and that the new shape evaluates to non-negative integer vectors $[\bar{i}_1], \dots, [\bar{i}_m]$, rule E-RSHP yields a new array $[\bar{v}' : T | [\bar{i}_1, \dots, \bar{i}_m]]$ whose elements are a repetition of the data vector \bar{v} : the element v'_k is the element $v_{k \bmod \prod \bar{n}}$.

To illustrate reshape, we use it to conveniently specify an identity matrix of shape [4,4] using a structured shape.

Example 4.16 (Reshape)

reshape [4],[4] [1,0,0,0,0 :int [5]]
⇒ [1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1 :int [4,4]]

The array comprehension $\text{gen } :T \bar{e}$ with $\bar{x} \rightarrow e_b$ defines an array with the structured shape \bar{e} and element type T . For each shape vector e_j , the comprehension binds an index vector x_j in the body e_b . The evaluation rule E-GENS evaluates the shape vectors. If they all evaluate to non-negative integer vectors $[\bar{n}_1], \dots, [\bar{n}_m]$, rule E-GEN evaluates the entire array comprehension to an array

Evaluation rules, continued

$e \Rightarrow e$

$$\frac{\bar{e} \Rightarrow_{seq} \bar{e}'}{\text{loop } x_a : T = e_a ; \bar{e} \text{ with } \bar{x} \rightarrow e_b \Rightarrow \text{loop } x : T = e_a ; \bar{e}' \text{ with } \bar{x} \rightarrow e_b} \text{ (E-LOOPS)}$$

$$\frac{(0 \leq n_{ij})_{ij}}{\text{loop } x_a : T = e_a ; [\bar{n}_1], \dots, [\bar{n}_m] \text{ with } x_1, \dots, x_m \rightarrow e_b \Rightarrow f_{p-1} (\dots (f_0 e_a))} \text{ (E-LOOP)}$$

where $p = \prod (\bar{n}_1, \dots, \bar{n}_m)$

$f_k = \text{fun } x_a : T \rightarrow e_b [x_j \mapsto [\bar{i}_j]]_j$ such that $\iota (\bar{n}_1, \dots, \bar{n}_m) (\bar{i}_1, \dots, \bar{i}_m) = k$

Figure 4.9: Evaluation rules of loop expressions

constructor $[\bar{e} : T \mid [\bar{n}_1, \dots, \bar{n}_m]]$. Each element expression e_k is obtained by instantiating the index vectors x_j in e_b as vectors $[\bar{i}_j]$ with $\iota (\bar{n}_1, \dots, \bar{n}_m) (\bar{i}_1, \dots, \bar{i}_m) = k$.

The following example shows two equivalent array comprehensions that specify an array of shape $[2, 2]$ whose elements equal the sum of their corresponding index positions.

Example 4.17 (Array comprehension)

```
gen :int [2,2] with x → x.(0)+x.(1) ⇒* [0,1,1,2 :int | [2,2]]
gen :int [2],[2] with x,y → x.(0)+y.(0) ⇒* [0,1,1,2 :int | [2,2]]
```

Dual to `gen`, the loop expression `loop $x_a : T = e_a ; \bar{e}$ with $\bar{x} \rightarrow e_b$` specifies a rank-generic reduction operation. Its semantics is shown in Figure 4.9. For each shape vector e_j , `loop` binds an index vector x_j in the body e_b . Starting out with the initial value e_a , the accumulator variable x_a of type T , which is also bound in e_b , represents the intermediate loop result. E-LOOPS evaluates the shape vectors. If all shapes evaluate to non-negative vectors $[\bar{n}_1], \dots, [\bar{n}_m]$, E-LOOP unfolds the loop into a sequence of function applications $f_{p-1} (\dots (f_0 e_a))$ where p is the product of all shape components and each function f_k is defined as `fun $x_a : T \rightarrow e_b [x_j \mapsto [\bar{i}_j]]_j$ with $\iota (\bar{n}_1, \dots, \bar{n}_m) (\bar{i}_1, \dots, \bar{i}_m) = k$` .

To illustrate `loop`, we define an expression that computes the element-wise sum of a multidimensional array. The loop unfolds into an expression that, starting with 0, selects and adds all elements from the array `a` in ascending lexicographic order.

Example 4.18 (Array sum)

```

let a = [1,2,3,4 :int|[2,2]] in
loop sum:int = 0; [2,2] with x → sum + a.[x]
  ⇒ ((fun sum:int → sum + a.[[1,1]])
      ((fun sum:int → sum + a.[[1,0]])
        ((fun sum:int → sum + a.[[0,1]])
          ((fun sum:int → sum + a.[[0,0]]) 0))))
  ⇒* 10

```

4.4 Properties of Evaluation

The previous sections introduced the language $\text{QUBE}_{\text{CORE}}$ and its operational semantics in the form of a small-step evaluation relation. This section shows that the properties introduced in Section 2.2 hold for $\text{QUBE}_{\text{CORE}}$ as well.

The set of values is a subset of expressions in normal form which are accepted as legal results of evaluation, as formalised by the following theorem.

Theorem 4.19 (Every value is in normal form)

For all v there is no e with $v \Rightarrow e$.

Proof: By structural induction over v .

1. Case $v = c$: immediate, there is no rule to evaluate constants.
2. Case $v = \text{fun } x : T \rightarrow e$: similar.
3. Case $v = f^2 v'$: only the rules E-APP1 and E-APP2 match the expression. By the hypothesis, the two subexpression are in normal form, so neither rule is applicable.
4. Case $v = (v_1, v_2 : (x : T_1, T_2))$: only the rules E-TUP1 and E-TUP2 match the expression. By the hypothesis, the two subexpression are in normal form, so neither rule is applicable.
5. Case $v = [\bar{v}]$: only E-VECTOR matches the expression. By the hypothesis, all subexpressions are in normal form, hence E-SEQ and consequently E-VECTOR are not applicable.
6. Case $v = [\bar{v} : T | \bar{Z}]$: only E-ARR matches the expression. By the hypothesis, all subexpressions are in normal form, therefore E-SEQ and consequently E-ARR are not applicable. ■

One-step evaluation of $\text{QUBE}_{\text{CORE}}$ expression is deterministic, i. e., the result of an expression taking an evaluation step will always be the same. The proof is

omitted here because it is a straightforward extension of the proof of the corresponding Theorem 2.12 for the applied λ -calculus.

Theorem 4.20 (*Determinacy of one-step evaluation*)

If $e \Rightarrow e'$ and $e \Rightarrow e''$ then $e' = e''$.

As one-step evaluation is deterministic, so is multi-step evaluation. In consequence, each expression has at most one normal form.

Summary

This chapter presented the syntax and the operational semantics of $\text{QUBE}_{\text{CORE}}$, a variant of the functional array programming language QUBE. The language comprises three conceptual layers: QUBE_λ is a foundation that provides essential features of functional programming, QUBE_\rightarrow adds support for processing integer vectors, and QUBE_\square enables rank-generic programming with multidimensional arrays.

We showed that the evaluation of $\text{QUBE}_{\text{CORE}}$ expressions is deterministic and that every value is in normal form. The converse proposition does not hold: not every expression in normal form is a value, for example $\text{if } :T \ 42 \ \text{then } e_2 \ \text{else } e_3$, or $[1, 2, 3, 4 : \text{int} \mid [2, 2]] \cdot [[0, 3]]$. These expressions are “stuck” in the evaluation process. In the next chapter, we will present typing rules to rule out expressions whose evaluation might get stuck.

5

Type Checking QUBE_{CORE}

This chapter explains how QUBE_{CORE} programs are type checked so that program errors are statically ruled out. Rank-generic programming adds a host of additional constraints that cannot be addressed with conventional type systems. For example, an element can only be selected from an array of rank r and shape vector s with an index vector of length r whose elements all range between 0 and the corresponding element of s .

Type checking of QUBE_{CORE} is performed by several relations that act in concert. Figure 5.1 shows the common typing context and gives an overview of how the relations depend on each other.

The typing context Γ is either empty (\cdot), a context Γ extended with a pair $x : T$ that associates a variable with a type, or a context Γ extended with a boolean guard expression e that is assumed to hold under the context. Guard expressions allow the type system to take path information about the branches of a conditional into account [96, 86].

The typing relation $\Gamma \vdash e : T$ assigns a type T to an expression e under a context Γ . The relation $\Gamma \vdash T$ checks whether a type T is well-formed under Γ . In a type system with dependent types, both relations mutually depend on each other since types appear in expressions and expressions appear in types. The typing relation relies on the subtype relation $\Gamma \vdash T <: T$ to check whether an expression of some type T_1 may be safely used in a position where an expression of some other type T_2 is expected.

To statically ensure that two array types have the same shape and to perform

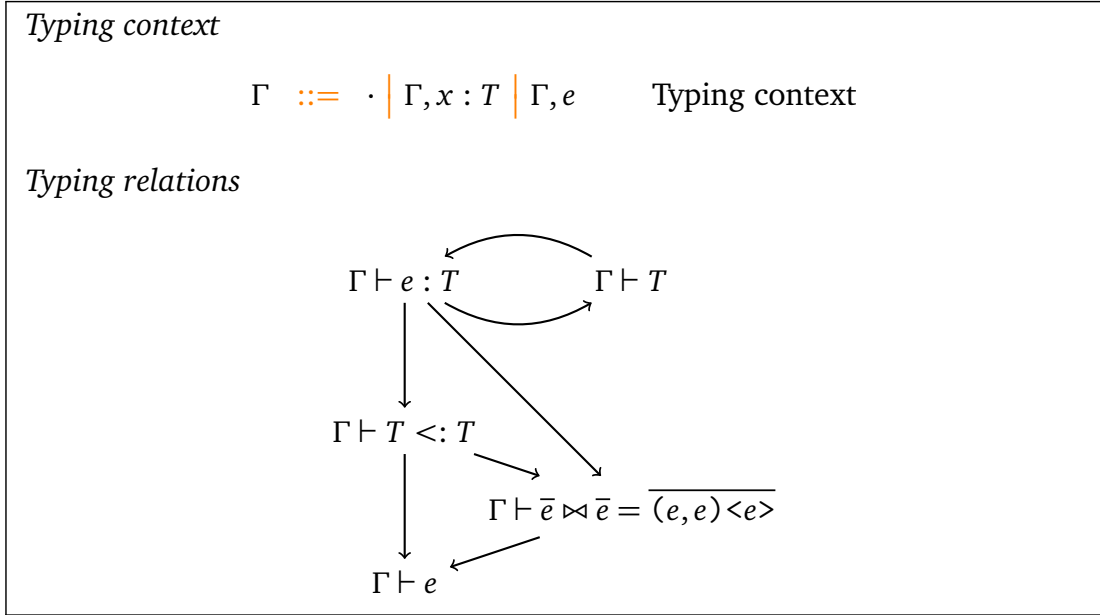


Figure 5.1: Overview of the type checking relations

array bounds checking at compile time, the type checker must reason about how corresponding elements of structured vectors relate to each other. The relation $\Gamma \vdash \bar{e} \bowtie \bar{e} = \overline{(e, e) \langle e \rangle}$ statically joins two structured vectors into a sequence of pairs of vectors of equal length so that corresponding vector elements are aligned.

The proof relation $\Gamma \vdash e$ which statically decides whether a boolean expression e is valid under a context Γ is the spine of the type system. Following the approach taken in [87, 36], this chapter presents two different flavours of the proof relation. In order to study the essential metatheoretic properties of QUBE_{CORE}, a conceptually simple but undecidable formalisation is used. In contrast, the QUBE compiler soundly approximates the proof relation in a decidable fragment of first-order logic.

The remainder of this chapter is structured as follows: Section 5.1 formalises when types are considered well-formed. Section 5.2 explains how corresponding elements of structured vectors are statically aligned. Section 5.3 discusses subtyping. With all pieces in place, Section 5.4 finally explains type checking of QUBE_{CORE} expressions. Section 5.5 proofs essential soundness properties of the type system. Finally, Section 5.6 outlines the actual implementation of the proof relation by means of an SMT solver.

Well-formed types $\boxed{\Gamma \vdash T}$

$$\begin{array}{c}
\Gamma \vdash B \text{ (WF-BTYPE)} \\
\\
\frac{\Gamma \vdash T_1 \quad \Gamma, x : T_1 \vdash T_2}{\Gamma \vdash x : T_1 \rightarrow T_2} \text{ (WF-FUN)} \\
\\
\frac{\Gamma \vdash T_1 \quad \Gamma, x : T_1 \vdash T_2}{\Gamma \vdash (x : T_1, T_2)} \text{ (WF-TUP)} \\
\\
\frac{\Gamma \vdash e : \text{nat}}{\Gamma \vdash \text{intvec } e} \text{ (WF-INTVEC)} \\
\\
\frac{\Gamma \vdash T \quad (\Gamma \vdash e_i : \text{natvec } l_i)_i}{\Gamma \vdash [T | \bar{e}]} \text{ (WF-ARR)} \\
\\
\frac{T \in \{\text{bool}, \text{int}, \text{intvec } _ \} \quad \Gamma \vdash T \quad \Gamma, x : T \vdash e : \text{bool}}{\Gamma \vdash \{x : T | e\}} \text{ (WF-REFTYPE)}
\end{array}$$

Figure 5.2: Well-formed types

5.1 Well-Formed Types

The relation $\Gamma \vdash T$ checks whether a type T is well-formed under a context Γ . The context is needed for type checking the expressions that appear in dependent types. Figure 5.2 shows the rules of the well-formedness relation.

The base types `bool` and `int` are well-formed under any context as they contain no expressions (WF-BTYPE).

Dependent function types $x : T_1 \rightarrow T_2$ and dependent tuple types $(x : T_1, T_2)$ are checked by the rules WF-FUN and WF-TUP, respectively. Both types are well-formed under Γ when T_1 is well-formed under Γ and when T_2 is also well-formed under the additional assumption $x : T_1$.

Integer vectors and array axes cannot have negative length. Consequently, a type `intvec e` is only well-formed after rule WF-INTVEC when e has type `nat`. Similarly, WF-ARR states that an array type $[T | \bar{e}]$ is well-formed if T is a well-formed type and each shape vector e_j has type `natvec`. As pointed out in Chapter 4, the types `nat` and `natvec` are abbreviations for the respective refinement types.

The final rule WF-REFTYPE checks the well-formedness of refinement types. Only the base types `bool`, `int`, and the vector type `intvec e` may be refined. A refinement type $\{x : T | e\}$ is well-formed if T is well-formed and when e has type `bool` under the extended context $\Gamma, x : T$.

5.2 Joining Structured Vectors

The array type and the array primitives make use of structured vectors to describe shapes and index vectors. During type checking, structured vectors must be statically compared, for example, to determine whether an array of type $[T | e_1, \dots, e_m]$ may be used in a position where an array of type $[T | e'_1, \dots, e'_n]$ is required, or to decide whether a selection $a.[e_1, \dots, e_m]$ into an array a of type $[T | e'_1, \dots, e'_n]$ violates the array boundaries.

The generalised selection `gsel` illustrates the problem. Given an array a whose shape fs, cs consists of the *frame shape* fs and the *cell shape* cs , and an index vector x that indexes into the frame shape, the function selects the slice of elements of a whose position is prefixed with x . The example uses an extended form of `let` to conveniently define functions. The notation `let f x:int = e in e'` is equivalent to `let f = fun x:int → e in e'`.

Example 5.1 (Generalised selection)

```
let gsel fr:nat fs:(natvec fr) cr:nat cs:(natvec cr)
    x:(indexvec fr fs) a:[int|fs,cs] =
  gen :int cs with y → a.[x,y]

(gsel 1 [2] 1 [2] [0]) [1,2,3,4 :int|[2,2]]
```

The partially applied function `(gsel 1 [2] 1 [2] [0])` expects an argument of type $[int|[2], [2]]$ but is applied to an array of type $[int|[2,2]]$. The argument is legitimate because the structured vectors $[2], [2]$ and $[2,2]$ have the same elements. In general however, structured vectors are not compile-time constants. In order to check the selection $a.[x,y]$, the type checker must prove that all elements of the structured vector x,y range between 0 and the corresponding elements of the structured shape fs, cs although neither the actual vector elements nor even their number are known at compile time.

Figure 5.3 shows the relation $\Gamma \vdash \bar{e} \bowtie \bar{e} = \overline{(e, e) \langle e \rangle}$ which statically joins two structured vectors so that corresponding elements are aligned. Applied to two structured vectors \bar{e}_V, \bar{e}_W , the relation yields a sequence of pairs of the form $(e_1, e_2) \langle e_l \rangle$ where e_1 and e_2 are vectors of length e_l whose elements stem from \bar{e}_V and \bar{e}_W , respectively. Structured vectors may be joined in either of three cases.

In the first case, handled by J-VECTORS, both structured vectors have the same structure, i.e., they consist of the same number of vectors and corresponding vectors are known to have the same (typically non-constant) length. As its result, the rule merely pairs corresponding vectors and augments each pair with a length expression. In the example, J-VECTORS aligns fs with x and cs with y

Join structured vectors

$$\boxed{\Gamma \vdash \bar{e} \bowtie \bar{e} = \overline{(e, e)} \langle e \rangle}$$

$$\frac{(\Gamma \vdash e_{Vj} : \text{intvec } e_j \quad \Gamma \vdash e_{Wj} : \text{intvec } e_j)_j}{\Gamma \vdash e_{V1}, \dots, e_{Vn} \bowtie e_{W1}, \dots, e_{Wn} = (e_{V1}, e_{W1}) \langle e_1 \rangle, \dots, (e_{Vn}, e_{Wn}) \langle e_n \rangle} \text{ (J-VECTORS)}$$

$$\frac{\Gamma \vdash \oplus e_{V1}, \dots, e_{Vm} = [\bar{e}] \quad \Gamma \vdash \oplus e_{W1}, \dots, e_{Wm} = [\bar{e}'] \quad |\bar{e}| = n = |\bar{e}'|}{\Gamma \vdash e_{V1}, \dots, e_{Vm} \bowtie e_{W1}, \dots, e_{Wm} = ([\bar{e}], [\bar{e}']) \langle n \rangle} \text{ (J-ELEMS)}$$

$$\frac{\Gamma \vdash \text{false}}{\Gamma \vdash e_{V1}, \dots, e_{Vn} \bowtie e_{W1}, \dots, e_{Wn} = ([], []) \langle 0 \rangle} \text{ (J-EXFALSO)}$$

Condense structured vector

$$\boxed{\Gamma \vdash \oplus \bar{e} = [\bar{e}]}$$

$$\frac{(\Gamma \vdash e_j : \text{intvec } e'_j \quad \exists! n_j. \Gamma \vdash n_j = e'_j)_j}{\Gamma \vdash \oplus e_1, \dots, e_m = [e_1 \cdot (0), \dots, e_1 \cdot (n_1 - 1), \dots, e_m \cdot (0), \dots, e_m \cdot (n_m - 1)]} \text{ (CONDENSE)}$$

Figure 5.3: Rules for joining structured vectors

during type checking of the selection.

$$\frac{\dots}{\Gamma \vdash \text{fs}, \text{cs} \bowtie \text{x}, \text{y} = (\text{fs}, \text{x}) \langle \text{fr} \rangle, (\text{cs}, \text{y}) \langle \text{cr} \rangle} \text{ (J-VECTORS)}$$

where $\Gamma = \text{fr} : \text{nat}, \text{fs} : (\text{natvec } \text{fr}), \text{cr} : \text{nat}, \text{cs} : (\text{natvec } \text{cr}),$
 $\text{x} : (\text{indexvec } \text{fr } \text{fs}), \text{a} : [\text{int} | \text{fs}, \text{cs}], \text{y} : (\text{indexvec } \text{cr } \text{cs})$

In the second case, handled by J-ELEMS, both structured vectors are known to have the same (constant) number of elements. In this case, the auxiliary rule CONDENSE yields a set of expressions that represent the individual elements of a structured vector. To deduce the length of each vector, CONDENSE uses the auxiliary relation $\exists! n. \Gamma \vdash e = n$, which determines a *unique* integer constant n such that $\Gamma \vdash e = n$. J-ELEMS yields a single pair of vectors. In the above example, J-ELEMS is used to prove that the array $[1, 2, 3, 4 : \text{int} \mid [2, 2]]$ is an appropriate argument to the function `gsel` that expects an array of type $[\text{int} \mid [2], [2]]$.

$$\frac{\frac{\dots}{\cdot \vdash \oplus [2], [2] = [2, 2]} \text{ (CONDENSE)} \quad \frac{\dots}{\cdot \vdash \oplus [2, 2] = [2, 2]} \text{ (CONDENSE)}}{\cdot \vdash [2], [2] \bowtie [2, 2] = ([2, 2], [2, 2]) \langle 2 \rangle} \text{ (J-ELEMS)}$$

The last rule J-EXFALSO explicitly deals with the case that the context Γ is invalid. Under such a context, for example `false`, we can neither compare length expressions (`false` $\vdash e_1 = e_2$ will always hold), nor can we use the relation

Subtyping

 $\Gamma \vdash T <: T$

$$\begin{array}{c}
\Gamma \vdash B <: B \text{ (SUB-BTYPE)} \\
\\
\frac{\Gamma \vdash T_{21} <: T_{11} \quad \Gamma, x : T_{21} \vdash T_{12} <: T_{22}}{\Gamma \vdash x : T_{11} \rightarrow T_{12} <: x : T_{21} \rightarrow T_{22}} \text{ (SUB-FUN)} \\
\\
\frac{\Gamma \vdash T_{11} <: T_{21} \quad \Gamma, x : T_{11} \vdash T_{12} <: T_{22}}{\Gamma \vdash (x : T_{11}, T_{12}) <: (x : T_{21}, T_{22})} \text{ (SUB-TUP)} \\
\\
\frac{\Gamma \vdash e_1 = e_2}{\Gamma \vdash \text{intvec } e_1 <: \text{intvec } e_2} \text{ (SUB-VEC)} \\
\\
\frac{\Gamma \vdash T_1 <: T_2 \quad \Gamma \vdash \bar{e}_1 \bowtie \bar{e}_2 = \overline{(v, w) \langle l \rangle} \quad (\Gamma \vdash \text{vfa } l_i v_i w_i (x, y \rightarrow x=y))_i}{\Gamma \vdash [T_1 | \bar{e}_1] <: [T_2 | \bar{e}_2]} \text{ (SUB-ARR)} \\
\\
\frac{\Gamma \vdash T_1 <: T_2 \quad \Gamma, x : T_1 \vdash e_1 \Rightarrow e_2}{\Gamma \vdash \{x : T_1 | e_1\} <: \{x : T_2 | e_2\}} \text{ (SUB-REFTYPE)}
\end{array}$$

Figure 5.4: Subtyping rules

$\exists! n. \text{false} \vdash e = n$ to find a unique n equal to e . During type checking, an invalid context can only arise if an expression will never be evaluated at run-time, for example the first branch e_t of the conditional `if false then e_t else e_e` . To allow type checking to proceed in these expressions, J-EXFALSO yields a pair of empty vectors.

5.3 Subtyping

The subtype relation $\Gamma \vdash T <: T$ for comparing (well-formed) types is shown in Figure 5.4. By rule SUB-BTYPE, every base type B is a subtype of itself. For (dependent) function types, SUB-FUN states that subtyping is contravariant for the argument types and covariant for the result types. In contrast, SUB-TUP states that subtyping on dependent tuple types is covariant in both component types. Both rules bind the variable x to the lesser type in the context as x may be referenced in the second part of the respective types.

The subtyping rule SUB-VEC for integer vector types states that a type `intvec e_1` is a subtype of `intvec e_2` when the lengths e_1 and e_2 are provably equivalent. Similarly, the subtyping rule SUB-ARR for array types `[$T_1 | \bar{e}_1$]` and `[$T_2 | \bar{e}_2$]` is covariant on the element types and requires that both structured vectors \bar{e}_1 and

\bar{e}_2 denote the same shape. To compare the array shapes, SUB-ARR joins the structured vectors and compares the corresponding vector segments by proving the validity of a suitable vector predicate.

The subtyping rule for refinement types SUB-REF captures the intuition that a refinement type $\{x : T_1 \mid e_1\}$ is a subtype of $\{x : T_2 \mid e_2\}$ when the former describes a subset of the values described by the latter. The first premise checks that T_1 is a subtype of T_2 . This means that both types are either `bool`, `int`, or `intvec` of equal length, because of the restrictions on which types can be refined and the above subtyping rules. The second premise establishes the subset relation by checking whether e_1 implies e_2 under $\Gamma, x : T_1$. As usual, the implication operator $e_1 \Rightarrow e_2$ is an abbreviation of $(\text{not } e_1) \mid e_2$.

5.4 Type Checking

This section presents the typing relation $\Gamma \vdash e : T$ that associates $\text{QUBE}_{\text{CORE}}$ expressions with types. Given the extensive syntax of $\text{QUBE}_{\text{CORE}}$, we incrementally present the typing rules for QUBE_{λ} , $\text{QUBE}_{\rightarrow}$, and QUBE_{\square} .

Type checking of $\text{QUBE}_{\text{CORE}}$ requires to check the validity of boolean constraints under a context Γ , for example $\Gamma \vdash 0 \leq i \ \& \ i < n$. The implemented decision procedure for the proof relation converts the context and the boolean expression into a logical formula and applies a theorem prover. In particular, the translation puts both the implicit function of `vmap` $e_n \bar{e} (\bar{x} \rightarrow e)$ and the predicate of `vfa` $e_n \bar{e} (\bar{x} \rightarrow e)$ under a universal quantifier. To ensure that the resulting formula stays in the array property fragment [18], we restrict the expressions that may appear in the in-place function definitions $(\bar{x} \rightarrow e)$ in both cases.

- Only constant and function symbols, variables, applications, conditional expressions, up-coercions, and let-bindings are allowed in the function body e .
- Only the bound variables \bar{x} may be referenced in the function body e , i. e., $\mathcal{FV}(e) \subseteq \{\bar{x}\}$.

To enforce these restrictions, we augment the typing relation $\Gamma \vdash_{\varepsilon} e : T$ with an environment parameter $\varepsilon \in \{\circ, \bullet\}$ that indicates whether an expression is checked in a restricted (\bullet) or unrestricted (\circ) environment. Typing rules that are not applicable under `vfa` or `vmap` require the parameter to be \circ in the conclusion. For example, abstractions cannot be typed under `vfa` and `vmap` because the conclusion of the corresponding typing rule T-ABS reads $\Gamma \vdash_{\circ} \text{fun } x : T \rightarrow e : (x : T \rightarrow T')$. In contrast, variables may occur everywhere. Therefore, the typing rule T-VAR does not instantiate the environment parameter in its conclusion

$\Gamma \vdash_{\varepsilon} x : T$. When the typing relation is referred to in other sections as $\Gamma \vdash e : T$, this is equivalent to $\Gamma \vdash_{\circ} e : T$.

Type Checking QUBE _{λ}

The typing rules for QUBE _{λ} are shown in Figure 5.5. As indicated by the environment parameter ε , only constants, variables, applications, let-expressions, conditionals, and coercions may appear under `vmap` and `vfa`. All other kinds of expressions require an unrestricted context (\circ).

The subsumption rule T-SUB connects the subtype relation and the typing relation. It states that if an expression e has a type T' that is a subtype of some arbitrary well-formed type T , then e also has type T .

T-CONST gives types to constant and function symbols according to the table shown in Figure 5.6. To model the semantics of the symbols as closely as possible, the types are defined in terms of the symbols itself. T-VAR projects the type of a variable x from the context Γ .

The typing rule T-ABS for abstractions `fun $x : T \rightarrow e$` checks that T is well-formed under Γ . When e has type T' under the extended context $\Gamma, x : T$ then the entire abstraction has a function type $x : T \rightarrow T'$. Applications $e_1 e_2$ are checked by the rule T-APP. When the operator has a (dependent) function type $x : T_1 \rightarrow T_2$ and when the operand has type T_1 , then the application has type $T_2[x \mapsto e_1]$.

T-LET checks let-bindings `let $x = e_1$ in e_2` . When the right-hand side e_1 has type T_1 under Γ , the rule proceeds to check the body expression e_2 under the additional assumption $x : T_1$. Its type T_2 is also the type of the entire expression. The antecedent $\Gamma \vdash T_2$ enforces that T_2 does not refer to x as it is not bound in the outer context Γ .

The conditional expression `if $: T$ e then e_t else e_e` explicitly provides a common type T for the two branches. The typing rule T-COND checks that T is well-formed under Γ and that the predicate e actually is a boolean expression. The antecedents that check the branches take the path information into account by augmenting Γ with e and `not e` , respectively. When both branches have type T , the entire conditional has type T .

The coercion `($e : T$)` serves to up-cast the type of an expression. Consequently, T-COERCE checks the well-formedness of T and whether e has the annotated type. Then, the entire coercion is associated with type T .

The typing rule T-TUP for tuple constructors `($e_1, e_2 : (x : T_1, T_2)$)` checks whether the type annotation is well-formed. When the components have types T_1 and $T_2[x \mapsto e_1]$, respectively, the tuple constructor has the annotated type.

Type checking

 $\Gamma \vdash_\varepsilon e_\lambda : T$

$$\frac{\Gamma \vdash_\varepsilon e : T' \quad \Gamma \vdash T' <: T \quad \Gamma \vdash T}{\Gamma \vdash_\varepsilon e : T} \text{ (T-SUB)}$$

$$\Gamma \vdash_\varepsilon c : \text{type}(c) \text{ (T-CONST)} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash_\varepsilon x : T} \text{ (T-VAR)}$$

$$\frac{\Gamma \vdash T \quad \Gamma, x : T \vdash_\circ e : T'}{\Gamma \vdash_\circ \text{fun } x : T \rightarrow e : (x : T \rightarrow T')} \text{ (T-ABS)}$$

$$\frac{\Gamma \vdash_\varepsilon e_1 : (x : T_1 \rightarrow T_2) \quad \Gamma \vdash_\varepsilon e_2 : T_1}{\Gamma \vdash_\varepsilon e_1 e_2 : T_2[x \mapsto e_2]} \text{ (T-APP)}$$

$$\frac{\Gamma \vdash_\varepsilon e_1 : T_1 \quad \Gamma, x : T_1 \vdash_\varepsilon e_2 : T_2 \quad \Gamma \vdash T_2}{\Gamma \vdash_\varepsilon \text{let } x = e_1 \text{ in } e_2 : T_2} \text{ (T-LET)}$$

$$\frac{\Gamma \vdash T \quad \Gamma \vdash_\varepsilon e : \text{bool} \quad \Gamma, e \vdash_\varepsilon e_t : T \quad \Gamma, \text{not } e \vdash_\varepsilon e_e : T}{\Gamma \vdash_\varepsilon (\text{if } :T e \text{ then } e_t \text{ else } e_e) : T} \text{ (T-COND)}$$

$$\frac{\Gamma \vdash T \quad \Gamma \vdash_\varepsilon e : T}{\Gamma \vdash_\varepsilon (e : T) : T} \text{ (T-COERCE)}$$

$$\frac{\Gamma \vdash (x : T_1, T_2) \quad \Gamma \vdash_\circ e_1 : T_1 \quad \Gamma \vdash_\circ e_2 : T_2[x \mapsto e_1]}{\Gamma \vdash_\circ (e_1, e_2 : (x : T_1, T_2)) : (x : T_1, T_2)} \text{ (T-TUP)}$$

$$\frac{\Gamma \vdash_\circ e' : (x : T_1, T_2) \quad \Gamma, x_1 : T_1, x_2 : T_2[x \mapsto x_1] \vdash_\circ e : T \quad \Gamma \vdash T}{\Gamma \vdash_\circ \text{let } (x_1, x_2) = e' \text{ in } e : T} \text{ (T-UNPACK)}$$

Figure 5.5: Typing rules for QUBE_λ

$$\begin{array}{l}
\text{true} : \{x:\text{bool} \mid x\} \\
\text{false} : \{x:\text{bool} \mid \text{not } x\} \\
\text{not} : x:\text{bool} \rightarrow \{y:\text{bool} \mid y \leftrightarrow \text{not } x\} \\
\circ \in \{\leftrightarrow, \&, |\} : x:\text{bool} \rightarrow y:\text{bool} \rightarrow \{z:\text{bool} \mid z \leftrightarrow x \circ y\} \\
n \in \mathbb{Z} : \{x:\text{int} \mid x = n\} \\
\circ \in \{=, !=, <, <=, >=, >\} : x:\text{int} \rightarrow y:\text{int} \rightarrow \{z:\text{bool} \mid z \leftrightarrow x \circ y\} \\
\circ \in \{+, -, *\} : x:\text{int} \rightarrow y:\text{int} \rightarrow \{z:\text{int} \mid z = x \circ y\} \\
\circ \in \{/, \%\} : x:\text{int} \rightarrow y:\{y':\text{int} \mid y' \neq 0\} \rightarrow \{z:\text{int} \mid z = x \circ y\}
\end{array}$$

Figure 5.6: Types of constants and function symbols

Unpack expressions $\text{let } (x_1, x_2) = e' \text{ in } e$ are handled similar to let-expressions. First, T-UNPACK ensures that the right-hand side has a tuple type $(x:T_1, T_2)$. Then, the body expression is checked under the context $\Gamma, x_1:T_1, x_2:T_2[x \mapsto x_1]$. When its type T is well-formed under the outer context Γ (and thus does not reference x_1, x_2), then the entire expression has type T .

Type Checking QUBE_↓

The typing rules for the language fragment QUBE_↓ are shown in Figure 5.7. For type checking QUBE_{CORE}, static reasoning about integers and integer vectors is essential. To capture the interrelationships of the expressions as precisely as possible, the typing rules for QUBE_↓ reflect the semantics of each expression at the type level.

None of the expressions in QUBE_↓ may appear under vmap or vfa , as indicated by the environment parameter \circ . We use the notation $0 \leq x < y$ as an abbreviation of $0 \leq x \ \& \ x < y$. Likewise, the notation $a \stackrel{=}{=}_n b$ abbreviates the equality of two vectors a, b of common length n as defined by the vector predicate $\text{vfa } n \ a, b \ (x, y \rightarrow x = y)$.

Vectors $[\bar{e}]$ are typed by the rule T-VECTOR. All subexpressions e_i must be integers. The type of the vector expression is a refinement of $\text{intvec } n$ that specifies the value in terms of the vector constructor $[\bar{e}]$ itself.

T-VSEL and T-VMOD check vector selections $e.(e_i)$ and vector modifications $e.(e_i) \leftarrow e_e$, respectively. In both cases, e has to be a vector and the index e_i must range within the vector bounds. In case of the vector modification, the new ele-

Type checking, continued

$\Gamma \vdash_{\varepsilon} e_{\rightarrow} : T$

$$\frac{(\Gamma \vdash_{\circ} e_j : \text{int})_j}{\Gamma \vdash_{\circ} [\bar{e}] : \{x : \text{intvec } |\bar{e}| \mid x \stackrel{\equiv}{=}_{|\bar{e}|} [\bar{e}]\}} \text{ (T-VECTOR)}$$

$$\frac{\Gamma \vdash_{\circ} e : \text{intvec } e_n \quad \Gamma \vdash_{\circ} e_i : \{x : \text{int} \mid 0 \leq x < e_n\}}{\Gamma \vdash_{\circ} e.(e_i) : \{x : \text{int} \mid x = e.(e_i)\}} \text{ (T-VSEL)}$$

$$\frac{\Gamma \vdash_{\circ} e : \text{intvec } e_n \quad \Gamma \vdash_{\circ} e_i : \{x : \text{int} \mid 0 \leq x < e_n\} \quad \Gamma \vdash_{\circ} e_e : \text{int}}{\Gamma \vdash_{\circ} e.(e_i) \leftarrow e_e : \{x : \text{intvec } e_n \mid x \stackrel{\equiv}{=}_{e_n} e.(e_i) \leftarrow e_e\}} \text{ (T-VMOD)}$$

$$\frac{\Gamma \vdash_{\circ} e_n : \text{nat} \quad \Gamma \vdash_{\circ} e_e : \text{int}}{\Gamma \vdash_{\circ} \text{vec } e_n e_e : \{x : \text{intvec } e_n \mid x \stackrel{\equiv}{=}_{e_n} \text{vec } e_n e_e\}} \text{ (T-VEC)}$$

$$\frac{|\bar{e}| = |\bar{x}| \quad \Gamma \vdash_{\circ} e_n : \text{nat} \quad (\Gamma \vdash_{\circ} e_j : \text{intvec } e_n)_j \quad \overline{x : \text{int} \vdash_{\bullet} e_b : \text{int}}}{\Gamma \vdash_{\circ} \text{vmap } e_n \bar{e} (\bar{x} \rightarrow e_b) : \{x' : \text{intvec } e_n \mid x' \stackrel{\equiv}{=}_{e_n} \text{vmap } e_n \bar{e} (\bar{x} \rightarrow e_b)\}} \text{ (T-VMAP)}$$

$$\frac{|\bar{e}| = |\bar{x}| \quad \Gamma \vdash_{\circ} e_n : \text{nat} \quad (\Gamma \vdash_{\circ} e_j : \text{intvec } e_n)_j \quad \overline{x : \text{int} \vdash_{\bullet} e_b : \text{bool}}}{\Gamma \vdash_{\circ} \text{vfa } e_n \bar{e} (\bar{x} \rightarrow e_b) : \{x' : \text{bool} \mid x' \leftrightarrow \text{vfa } e_n \bar{e} (\bar{x} \rightarrow e_b)\}} \text{ (T-VFA)}$$

where $a \stackrel{\equiv}{=}_n b = \text{vfa } n a, b (x, y \rightarrow x = y)$

Figure 5.7: Typing rules for $\text{QUBE}_{\rightarrow}$

ment e_e has to be an integer, too. The corresponding types $\{x : \text{int} \mid x = e.(e_i)\}$ and $\{x : \text{intvec } e_n \mid x \stackrel{\equiv}{=}_{e_n} e.(e_i) \leftarrow e_e\}$ reflect the expressions at the type level.

The constant-value vector expression $\text{vec } e_n e_e$ is well-typed when the vector length e_n has type nat and when the element e_e has type int . T-VEC associates the expression with type $\{x : \text{intvec } e_n \mid x \stackrel{\equiv}{=}_{e_n} \text{vec } e_n e_e\}$.

The rule T-VMAP checks vector comprehensions of the form $\text{vmap } e_n \bar{e} (\bar{x} \rightarrow e_b)$. Under Γ , the vector length e_n is required to have type nat and all vectors \bar{e} must have type $\text{intvec } e_n$. The body e_b is subject to the restrictions outlined in Section 4.2: only a subset of simple expressions is permitted and no variable apart from \bar{x} may be referenced in e_b . To ensure these requirements, e_b is checked with the environment parameter \bullet under a context that only binds the variables \bar{x} as integers. When e_b has type int , the type of the vector comprehension is a refinement of $\text{intvec } e_n$ that reflects the comprehension itself.

Vector predicates $\text{vfa } e_n \bar{e} (\bar{x} \rightarrow e_b)$ are checked by rule T-VFA which is similar to T-VMAP. The rule checks the same preconditions except that e_b have type bool . The type of the vector predicate is bool refined by the predicate itself.

Type Checking QUBE_□

Figure 5.8 shows the typing rules for the language fragment QUBE_□ that deals with multidimensional arrays. Unlike base and vector types, array types cannot be refined. In consequence, an array type such as $[\text{int} \mid [1024, 1024]]$ abstracts away the potentially huge number of individual array elements. For our goal of type checking array programs, reasoning about individual array elements is inessential. It is sufficient to reason about ranks, shape and index vectors. None of the expressions in QUBE_□ may appear under `vmap` or `vfa`, as indicated by the environment parameter \circ .

Array constructors $[\bar{e} : T \mid [\bar{n}]]$ are checked by T-ARR. The premises ensure that all shape components n_i are non-negative and that the number of elements $|\bar{e}|$ equals the shape product $\prod \bar{n}$. Furthermore, the annotated type T must be well-formed and every element must have type T . Then the array has type $[T \mid [\bar{n}]]$.

Selections $e. [\bar{e}_i]$ and array modifications $e. [\bar{e}_i] \leftarrow e_e$ are checked by the rules E-SEL and E-MOD, respectively. Both rules require that e is an array of some array type $[T \mid \bar{e}_s]$. To avoid array boundary violations, the structured shape vector \bar{e}_s has to be an upper bound for the structured index vector \bar{e}_i . The substituted element e_e is required to have type T , too. The result of a selection has the element type T whereas the result of array modification has the same type as the original array e .

T-RSHP checks reshaping expressions `reshape \bar{e} e_a` . As pointed out in Section 4.3, the array e_a can only be reshaped if it contains at least one element. Thus, the third premise ensures that the array's shape vectors \bar{e}_s are all strictly positive. The new shape vectors \bar{e} must all have type `natvec` to rule out negative shapes. Reshaping yields an array of type $[T \mid \bar{e}]$.

Array comprehensions `gen : T \bar{e} with $\bar{x} \rightarrow e_b$` are checked by T-GEN. The annotated type T has to be well-formed and all shape vectors e_j are required to have type `natvec`. Under the assumption that every x_j denotes a vector that ranges between $\vec{0}$ and e_j , the body expression e_b must have type T . When all preconditions are met, the result array has type $[T \mid \bar{e}]$.

T-LOOP checks loop expressions of the form `loop $x_a : T = e_a ; \bar{e}$ with $\bar{x} \rightarrow e_b$` . The annotated type T must be well-formed and the initial value e_a must of course have type T . Furthermore, the loop bounds \bar{e} have to be non-negative vectors. Assuming that every x_j is a vector that ranges between $\vec{0}$ and e_j , and that the accumulator x_a has type T , the loop body e_b must also have type T . In consequence, the result of the entire reduction also has type T .

Type checking, continued

$\Gamma \vdash_e e_{\square} : T$

$$\frac{(0 \leq n_i)_i \quad |\bar{e}| = \prod \bar{n} \quad \Gamma \vdash T \quad (\Gamma \vdash_e e_j : T)_j}{\Gamma \vdash_e [\bar{e} : T \mid \bar{n}] : [T \mid \bar{n}]} \text{ (T-ARR)}$$

$$\frac{\Gamma \vdash_e e : [T \mid \bar{e}_s] \quad \Gamma \vdash \bar{e}_s \bowtie \bar{e}_i = (s, \nu) \langle r \rangle \quad (\Gamma \vdash \text{vfa } r_j s_j, \nu_j (s, i \rightarrow 0 \leq i < s))_j}{\Gamma \vdash_e e. [\bar{e}_i] : T} \text{ (T-SEL)}$$

$$\frac{\Gamma \vdash_e e : [T \mid \bar{e}_s] \quad \Gamma \vdash_e e_e : T \quad \Gamma \vdash \bar{e}_s \bowtie \bar{e}_i = (s, \nu) \langle r \rangle \quad (\Gamma \vdash \text{vfa } r_j s_j, \nu_j (s, i \rightarrow 0 \leq i < s))_j}{\Gamma \vdash_e e. [\bar{e}_i] \leftarrow e_e : [T \mid \bar{e}_s]} \text{ (T-MOD)}$$

$$\frac{(\Gamma \vdash_e e_i : \text{natvec } _)_i \quad \Gamma \vdash_e e_a : [T \mid \bar{e}_s] \quad (\Gamma \vdash \text{pos } e_{r_j} e_{s_j})_j}{\Gamma \vdash_e \text{reshape } \bar{e} e_a : [T \mid \bar{e}]} \text{ (T-RSHP)}$$

where $\text{pos } n \nu = \text{vfa } n \nu (x \rightarrow 0 < x)$

$$\frac{|\bar{e}| = |\bar{x}| \quad \Gamma \vdash T \quad (\Gamma \vdash_e e_j : \text{natvec } e'_j)_j \quad \Gamma, \bar{x} : \bar{T} \vdash_e e_b : T}{\Gamma \vdash_e \text{gen} : T \bar{e} \text{ with } \bar{x} \rightarrow e_b : [T \mid \bar{e}]} \text{ (T-GEN)}$$

where $T_j = \{x' : \text{intvec } e'_j \mid \text{vfa } e_{n_j} x', e_j (x, y \rightarrow 0 \leq x < y)\}$

$$\frac{|\bar{e}| = |\bar{x}| \quad \Gamma \vdash T \quad \Gamma \vdash_e e_a : T \quad (\Gamma \vdash_e e_j : \text{natvec } e'_j)_j \quad \Gamma, \bar{x} : \bar{T}, x_a : T \vdash_e e_b : T}{\Gamma \vdash_e \text{loop } x_a : T = e_a ; \bar{e} \text{ with } \bar{x} \rightarrow e_b : T} \text{ (T-LOOP)}$$

where $T_j = \{x' : \text{intvec } e'_j \mid \text{vfa } e_{n_j} x', e_j (x, y \rightarrow 0 \leq x < y)\}$

Figure 5.8: Typing rules for QUBE $_{\square}$

5.5 Correctness of Type Checking

The previous sections have formalised the syntax, semantics, and typing rules of QUBE_{CORE}. This section gives proof that QUBE_{CORE} is type-safe, which means that no well-typed program can go wrong. As in Chapter 2.3, type-safety is shown by proving the progress and preservation theorems. However, due to subtyping and dependent types, additional lemmas are required. Apart from the support for rank-generic programming, the metatheory of QUBE_{CORE} is similar to that of other approaches that incorporate type checking and theorem proving, such as [87, 36].

QUBE_{CORE} provides an extensive set of constant and function symbols. The following axiom states that every symbol has a well-formed type and that δ -reduction of a function symbol behaves as declared by its type.

Axiom 5.2 (*Constant symbols are well-behaved*)

Each constant symbol c has a type $type(c)$ such that

1. $\cdot \vdash type(c)$
2. If $type(f^1) = x : T_1 \rightarrow T_2$, then the δ -reduction $f^1(v_1) \Rightarrow_{\delta} v_2$ is defined for all values v_1 with $\cdot \vdash v_1 : T_1$ so that $\cdot \vdash v_2 : T_2[x \mapsto v_1]$.
3. If $type(f^2) = x : T_1 \rightarrow y : T_2 \rightarrow T_3$, then the δ -reduction $f^2(v_1, v_2) \Rightarrow_{\delta} v_3$ is defined for all values v_1, v_2 with $\cdot \vdash v_1 : T_1$ and $\cdot \vdash v_2 : T_2[x \mapsto v_1]$ so that $\cdot \vdash v_3 : T_3[x \mapsto v_1][y \mapsto v_2]$.

Well-formed types and well-typed expressions cannot contain any free variables apart from those declared in the context Γ . As the relations are interdependent, the proof is by simultaneous induction on the derivations.

Lemma 5.3 (*Free variables*)

1. If $\Gamma \vdash T$, then $\mathcal{FV}(T) \subseteq dom(\Gamma)$.
2. If $\Gamma \vdash e : T$, then $\mathcal{FV}(e) \cup \mathcal{FV}(T) \subseteq dom(\Gamma)$.

As part of the proof, it is necessary to reason about the proof relation $\Gamma \vdash e$ that checks the validity of a predicate e under a context Γ . For this purpose, we use the undecidable but intuitive formalisation of the proof relation $\Gamma \vdash e$ shown in Figure 5.9 [87, 36]. The relation is defined in terms of substitutions $\sigma = x_1 \mapsto v_1, x_2 \mapsto v_2, \dots$ that map variables to values. The auxiliary relation $\Gamma \models \sigma$ defines when a substitution σ is permitted by a context Γ . By S-EMPTY, the empty context permits the empty substitution. By S-TYPE, a substitution $\sigma, x \mapsto v$ is permitted by a context $\Gamma, x : T$ if σ is permitted by Γ and there is some

Permitted substitution

$\boxed{\Gamma \models \sigma}$

$$\begin{array}{c} \cdot \models \emptyset \text{ (S-EMPTY)} \\ \\ \frac{\Gamma \models \sigma \quad \cdot \vdash v : T[\sigma]}{\Gamma, x : T \models \sigma, x \mapsto v} \text{ (S-TYPE)} \\ \\ \frac{\Gamma \models \sigma \quad e[\sigma] \Rightarrow^* \text{true}}{\Gamma, e \models \sigma} \text{ (S-GUARD)} \end{array}$$

Proof relation

$\boxed{\Gamma \vdash e}$

$$\frac{\forall \sigma. (\text{if } \Gamma \models \sigma, \text{ then } e[\sigma] \Rightarrow^* \text{true})}{\Gamma \vdash e} \text{ (PROOF)}$$

Figure 5.9: An undecidable formalisation of the proof relation

value v that has type $T[\sigma]$ under the empty context. By S-GUARD, a substitution σ is permitted by Γ, e if σ is permitted by Γ and the instance $e[\sigma]$ of the guard expression e evaluates to `true`. For example, a context $x : \text{nat}, y : \text{int}, x \leq y$ permits the substitutions $x \mapsto 0, y \mapsto 0$ and $x \mapsto 0, y \mapsto 1$, but not $x \mapsto 1, y \mapsto 0$.

According to PROOF, the proof relation $\Gamma \vdash e$ considers a predicate e valid under a context Γ if it evaluates to `true` under all substitutions σ that are permitted by the context. For example, $x : \text{nat}, y : \text{int}, x \leq y \vdash 0 \leq y$ is valid, but $x : \text{nat}, y : \text{int}, x \leq y \vdash y < 0$ is not. In particular, any predicate is vacuously valid if there is no σ with $\Gamma \models \sigma$. This happens when Γ contains a type that is not inhabited by any values or an unsatisfiable guard expression. For example, $\{v : \text{int} \mid 0 \leq v < 0\} \vdash \text{false}$ and $\text{false} \vdash \text{false}$.

As an immediate observation, a substitution σ that has been derived from a context Γ is defined exactly for the variables in Γ . The substitutes are values that are well-typed under the empty context and thus, by Lemma 5.3, contain no free variables.

Lemma 5.4 (*Substitutes are closed*)

$\boxed{\text{If } \Gamma \models \sigma, \text{ then } \text{dom}(\Gamma) = \text{dom}(\sigma) \text{ and } \mathcal{FV}(\text{cod}(\sigma)) = \emptyset.}$

As a corollary of the previous lemma, permuting the individual mappings of a substitution does not affect the substitution result.

$$\begin{array}{c}
\text{Well-formed contexts} \quad \boxed{\vdash \Gamma} \\
\vdash \cdot \text{ (C-EMPTY)} \quad \frac{\vdash \Gamma \quad \Gamma \vdash T}{\vdash \Gamma, x : T} \text{ (C-TYPE)} \quad \frac{\vdash \Gamma \quad \Gamma \vdash e : \text{bool}}{\vdash \Gamma, e} \text{ (C-GUARD)}
\end{array}$$

Figure 5.10: Well-formed contexts

Corollary 5.5 (Permutation of substitution)

If $\Gamma \models \sigma_1, \sigma_2$, then

1. for all expressions e , $e[\sigma_1, \sigma_2] = e[\sigma_2, \sigma_1]$
2. for all types T , $T[\sigma_1, \sigma_2] = T[\sigma_2, \sigma_1]$

The following lemma establishes a link between the structures of substitutions and permitting contexts.

Lemma 5.6 (Structure of substitutions)

1. If $\Gamma \models \sigma_1, \sigma_2$, then there are contexts Γ_1, Γ_2 such that $\Gamma = \Gamma_1, \Gamma_2$ with $\text{dom}(\Gamma_1) = \text{dom}(\sigma_1)$ and $\text{dom}(\Gamma_2) = \text{dom}(\sigma_2)$.
2. If $\Gamma_1, \Gamma_2 \models \sigma$, then there are substitutions σ_1, σ_2 such that $\sigma = \sigma_1, \sigma_2$ with $\text{dom}(\Gamma_1) = \text{dom}(\sigma_1)$ and $\text{dom}(\Gamma_2) = \text{dom}(\sigma_2)$.
3. $\Gamma_1, \Gamma_2 \models \sigma_1, \sigma_2$ with $\text{dom}(\Gamma_1) = \text{dom}(\sigma_1)$ and $\text{dom}(\Gamma_2) = \text{dom}(\sigma_2)$ iff $\Gamma_1 \models \sigma_1, \Gamma_2[\sigma_1] \models \sigma_2$.

Figure 5.10 shows the relation $\vdash \Gamma$ that defines whether a context Γ is considered well-formed. A well-formed context is either empty (\cdot), a well-formed context Γ extended with declaration of a type T that is well-formed under Γ , or a well-formed context Γ extended with a guard expression e that has type `bool` under Γ .

The weakening lemma states that the context under which any given fact has been derived may be freely extended with additional variable bindings.

Lemma 5.7 (Weakening)

If $\vdash \Gamma_1, \Gamma_2$ and $\Gamma_1 \vdash T_x$, then

1. if $\Gamma_1, \Gamma_2 \vdash e : T$, then $\Gamma_1, x : T_x, \Gamma_2 \vdash e : T$.
2. if $\Gamma_1, \Gamma_2 \vdash T$, then $\Gamma_1, x : T_x, \Gamma_2 \vdash T$.
3. if $\Gamma_1, \Gamma_2 \vdash T_1 <: T_2$, then $\Gamma_1, x : T_x, \Gamma_2 \vdash T_1 <: T_2$.
4. if $\Gamma_1, \Gamma_2 \vdash e$, then $\Gamma_1, x : T_x, \Gamma_2 \vdash e$.
5. if $\exists ! n_1. \Gamma_1, \Gamma_2 \vdash n_1 = e$, then either $\exists ! n_2. \Gamma_1, x : T_x, \Gamma_2 \vdash n_2 = e$ with $n_1 = n_2$ or $\Gamma_1, x : T_x, \Gamma_2 \vdash \text{false}$.
6. if $\Gamma_1, \Gamma_2 \vdash \bar{e}_1 \bowtie \bar{e}_2 = \overline{(v, w) \langle l \rangle}$ and $(\Gamma_1, \Gamma_2 \vdash \text{vfa } l_j v_j, w_j f)_j$, then $\Gamma_1, x : T_x, \Gamma_2 \vdash \bar{e}_1 \bowtie \bar{e}_2 = \overline{(v', w') \langle l' \rangle}$ and $(\Gamma_1, x : T_x, \Gamma_2 \vdash \text{vfa } l'_j v'_j, w'_j f)_j$.

Proof: By simultaneous induction over the respective derivations. The proofs of 1.–3. are straightforward inductions.

4. We show that $\forall \sigma. (\text{if } \Gamma_1, x : T_x, \Gamma_2 \models \sigma, \text{ then } e[\sigma] \Rightarrow^* \text{true})$, so that the result follows from PROOF.

Assume $\Gamma_1, x : T_x, \Gamma_2 \models \sigma$. By Lemma 5.6, $\Gamma_1 \models \sigma_1$, $x : T_x[\sigma_1] \models x \mapsto v_x$, and $\Gamma_2[\sigma_1, x \mapsto v_x] \models \sigma_2$. Since $x \notin \mathcal{FV}(\Gamma_2)$, $\Gamma_2[\sigma_1] \models \sigma_2$. By Lemma 5.6, $\Gamma_1, \Gamma_2 \models \sigma_1, \sigma_2$. Since $x \notin e$, $e[\sigma_1, x \mapsto v, \sigma_2] \Rightarrow^* \text{true}$ iff $e[\sigma_1, \sigma_2] \Rightarrow^* \text{true}$. By inversion of PROOF for the assumption $\Gamma_1, \Gamma_2 \vdash e$, we already know that $\forall \sigma. (\text{if } \Gamma_1, \Gamma_2 \models \sigma_1, \sigma_2, \text{ then } e[\sigma_1, \sigma_2] \Rightarrow^* \text{true})$.

5. There is a unique n with $\forall \sigma. (\text{if } \Gamma_1, \Gamma_2 \models \sigma, \text{ then } (n = e[\sigma]) \Rightarrow^* \text{true})$. Necessarily, there must be at least one σ with $\Gamma_1, \Gamma_2 \models \sigma$ and $(n = e[\sigma]) \Rightarrow^* \text{true}$.

By Lemma 5.6, $\Gamma_1, \Gamma_2 \models \sigma$ iff $\Gamma_1 \models \sigma_1$ and $\Gamma_1[\sigma_1] \models \sigma_2$. If there is a value v_x with $\cdot \vdash v_x : T_x[\sigma_1]$, then by S-TYPE, $\Gamma_1, x : T_x \models \sigma_1, x \mapsto v_x$. As $x \notin \mathcal{FV}(\Gamma_2)$, $\Gamma_2[\sigma_1, x \mapsto v_x] \models \sigma_2$. By Lemma 5.6, $\Gamma_1, x : T_x, \Gamma_2 \models \sigma_1, x \mapsto v_x, \sigma_2$. Since $x \notin \mathcal{FV}(e)$, $(n = e[\sigma_1, x \mapsto v_x, \sigma_2]) \Rightarrow^* \text{true}$. If there is no v_x with $\cdot \vdash v_x : T_x[\sigma_1]$, there is no σ satisfying $\Gamma_1, x : T_x, \Gamma_2 \models \sigma$ so that by PROOF, $\Gamma_1, x : T_x, \Gamma_2 \vdash \text{false}$ is vacuously true.

6. By induction on the derivation of $\Gamma_1, \Gamma_2 \vdash \bar{e}_1 \bowtie \bar{e}_2 = \overline{(v, w) \langle l \rangle}$.

(a) Case J-VECTORS:

$$\Gamma_1, \Gamma_2 \vdash e_{v1}, \dots, e_{vn} \bowtie e_{w1}, \dots, e_{wn} = (e_{v1}, e_{w1}) \langle l_1 \rangle, \dots, (e_{vn}, e_{wn}) \langle l_n \rangle,$$

$$(\Gamma_1, \Gamma_2 \vdash e_{vj} : \text{intvec } l_j, \Gamma_1, \Gamma_2 \vdash e_{wj} : \text{intvec } l_j)_j$$

By the hypothesis, the e_{vj}, e_{wj} keep their type under the extended context, so that J-VECTORS applies, again yielding $(e_{v1}, e_{w1}) \langle l_1 \rangle, \dots, (e_{vn}, e_{wn}) \langle l_n \rangle$. By 4., $(\Gamma_1, x : T_x, \Gamma_2 \vdash \text{vfa } l_j e_{vj}, e_{wj} f)_j$.

(b) Case J-ELEMS: $\Gamma_1, \Gamma_2 \vdash e_{v1}, \dots, e_{vm} \bowtie e_{w1}, \dots, e_{wm} = ([\bar{e}], [\bar{e}']) \langle n \rangle,$

$$\Gamma_1, \Gamma_2 \vdash \oplus e_{v1}, \dots, e_{vm} = [\bar{e}], \Gamma_1, \Gamma_2 \vdash \oplus e_{w1}, \dots, e_{wm} = [\bar{e}']$$

J-ELEMS relies on the auxiliary rule CONDENSE to statically form the joined vectors. We inspect the premises of CONDENSE. By the hypothesis, if $\Gamma_1, \Gamma_2 \vdash e_j : \text{intvec } e'_j$, then $\Gamma_1, x : T_x, \Gamma_2 \vdash e_j : \text{intvec } e'_j$. CONDENSE then tries to guess a constant length n_j for each vector e_j under the extended context. By 5., either the same length as before is guessed or the context is contradictory. If $\exists! n_j. \Gamma_1, x : T_x, \Gamma_2 \vdash n_j = e'_j$, then CONDENSE and subsequently J-ELEMS apply as before, so that by 4., $\Gamma_1, x : T_x, \Gamma_2 \vdash \text{vfa } n \ [\bar{e}], [\bar{e}'] f$. If the context is contradictory, then by J-EXFALSO, $\Gamma_1, x : T_x, \Gamma_2 \vdash e_{v1}, \dots, e_{vn} \bowtie e_{w1}, \dots, e_{wn} = ([], []) \langle 0 \rangle$ and $\Gamma_1, x : T_x, \Gamma_2 \vdash \text{vfa } 0 \ [\bar{e}], [\bar{e}'] f$ is trivially true.

(c) Case J-EXFALSO: $\Gamma_1, \Gamma_2 \vdash \bar{e}_1 \bowtie \bar{e}_2 = ([], []) \langle 0 \rangle$, $\Gamma_1, \Gamma_2 \vdash \text{false}$

By 4., $\Gamma_1, x : T_x, \Gamma_2 \vdash \text{false}$, so that J-EXFALSO applies again and $\Gamma_1, x : T_x, \Gamma_2 \vdash \text{vfa } 0 \ [\bar{e}], [\bar{e}'] f$ is trivially true. ■

Similar to weakening, the guard weakening lemma states that the context under which a given fact has been derived may be extended with additional guard expressions. The proof is similar to the proof of the weakening lemma.

Lemma 5.8 (Guard weakening)

If $\vdash \Gamma_1, \Gamma_2$ and $\Gamma_1 \vdash e_G : \text{bool}$, then

1. if $\Gamma_1, \Gamma_2 \vdash e : T$, then $\Gamma_1, e_G, \Gamma_2 \vdash e : T$.
2. if $\Gamma_1, \Gamma_2 \vdash T$, then $\Gamma_1, e_G, \Gamma_2 \vdash T$.
3. if $\Gamma_1, \Gamma_2 \vdash T_1 <: T_2$, then $\Gamma_1, e_G, \Gamma_2 \vdash T_1 <: T_2$.
4. if $\Gamma_1, \Gamma_2 \vdash e$, then $\Gamma_1, e_G, \Gamma_2 \vdash e$.
5. if $\exists! n_1. \Gamma_1, \Gamma_2 \vdash n_1 = e$, then either $\exists! n_2. \Gamma_1, e_G, \Gamma_2 \vdash n_2 = e$ with $n_1 = n_2$ or $\Gamma_1, e_G, \Gamma_2 \vdash \text{false}$.
6. if $\Gamma_1, \Gamma_2 \vdash \bar{e}_1 \bowtie \bar{e}_2 = \overline{(v, w) \langle l \rangle}$, and $(\Gamma_1, \Gamma_2 \vdash \text{vfa } l_j v_j, w_j f)_j$,
then $\Gamma_1, e_G, \Gamma_2 \vdash \bar{e}_1 \bowtie \bar{e}_2 = \overline{(v', w') \langle l' \rangle}$ and $(\Gamma_1, e_G, \Gamma_2 \vdash \text{vfa } l'_j v'_j, w'_j f)_j$

In a similar spirit, the narrowing lemma states that a context under which a fact has been derived may be restricted by confining the types of variables in the context. The proof is similar to the proof of the preceding lemmas.

Lemma 5.9 (Narrowing)

- If $\vdash \Gamma_1, x : T_x, \Gamma_2$ and $\Gamma_1 \vdash T'_x <: T_x$, then
1. if $\Gamma_1, x : T_x, \Gamma_2 \vdash e : T$, then $\Gamma_1, x : T'_x, \Gamma_2 \vdash e : T$.
 2. if $\Gamma_1, x : T_x, \Gamma_2 \vdash T$, then $\Gamma_1, x : T'_x, \Gamma_2 \vdash T$.
 3. if $\Gamma_1, x : T_x, \Gamma_2 \vdash T_1 <: T_2$, then $\Gamma_1, x : T'_x, \Gamma_2 \vdash T_1 <: T_2$.
 4. if $\Gamma_1, x : T_x, \Gamma_2 \vdash e$, then $\Gamma_1, x : T'_x, \Gamma_2 \vdash e$.
 5. if $\exists! n_1. \Gamma_1, x : T_x, \Gamma_2 \vdash n_1 = e$, then either $\exists! n_2. \Gamma_1, x : T'_x, \Gamma_2 \vdash n_2 = e$ with $n_1 = n_2$ or $\Gamma_1, x : T'_x, \Gamma_2 \vdash \text{false}$.
 6. if $\Gamma_1, x : T_x, \Gamma_2 \vdash \overline{e_1} \bowtie \overline{e_2} = \overline{(v, w) \langle l \rangle}$ and $(\Gamma_1, x : T_x, \Gamma_2 \vdash \text{vfa } l_j v_j, w_j f)_j$, then $\Gamma_1, x : T'_x, \Gamma_2 \vdash \overline{e_1} \bowtie \overline{e_2} = \overline{(v', w') \langle l' \rangle}$, $(\Gamma_1, x : T'_x, \Gamma_2 \vdash \text{vfa } l'_j v'_j, w'_j f)_j$

In $\text{QUBE}_{\text{CORE}}$, variables can occur in both expressions and types. Due to the variable convention 2.3, all bound identifiers are different from the variable to be substituted. Therefore, the substitution function for $\text{QUBE}_{\text{CORE}}$ is a straightforward extension of the substitution function for the untyped λ -calculus 2.4.

The small-step semantics of $\text{QUBE}_{\text{CORE}}$ only requires that values are substituted instead of full-blown expressions. We therefore prove a simplified variant of the substitution lemma which states that the type of an expression is preserved when variables are substituted with values of appropriate types.

Lemma 5.10 (Substitution lemma)

- If $\vdash \Gamma_1, x : T_x, \Gamma_2$ and $\Gamma_1 \vdash v_x : T_x$, then
1. if $\Gamma_1, x : T_x, \Gamma_2 \vdash e : T$, then $\Gamma_1, \Gamma_2[x \mapsto v_x] \vdash e[x \mapsto v_x] : T[x \mapsto v_x]$.
 2. if $\Gamma_1, x : T_x, \Gamma_2 \vdash T$, then $\Gamma_1, \Gamma_2[x \mapsto v_x] \vdash T[x \mapsto v_x]$
 3. if $\Gamma_1, x : T_x, \Gamma_2 \vdash T_1 <: T_2$, then $\Gamma_1, \Gamma_2[x \mapsto v_x] \vdash T_1[x \mapsto v_x] <: T_2[x \mapsto v_x]$
 4. if $\Gamma_1, x : T_x, \Gamma_2 \vdash e$, then $\Gamma_1, \Gamma_2[x \mapsto v_x] \vdash e[x \mapsto v_x]$.
 5. if $\exists! n_1. \Gamma_1, x : T_x, \Gamma_2 \vdash n_1 = e$, then $\exists! n_2. \Gamma_1, \Gamma_2[x \mapsto v_x] \vdash n_2 = e[x \mapsto v_x]$ with $n_1 = n_2$.
 6. if $\Gamma_1, x : T_x, \Gamma_2 \vdash \overline{e_1} \bowtie \overline{e_2} = \overline{(v, w) \langle l \rangle}$ and $(\Gamma_1, x : T_x, \Gamma_2 \vdash \text{vfa } l_j v_j, w_j f)_j$, then $\Gamma_1, \Gamma_2[x \mapsto v_x] \vdash \overline{e_1}[x \mapsto v_x] \bowtie \overline{e_2}[x \mapsto v_x] = \overline{((v, w) \langle l \rangle)[x \mapsto v_x]}$ and $(\Gamma_1, \Gamma_2[x \mapsto v_x] \vdash (\text{vfa } l_j v_j, w_j f)[x \mapsto v_x])_j$

Proof: By simultaneous induction over the respective derivations. Most cases are proved by straightforward induction, similar to the proofs of the previous lemmas.

1. Straightforward induction over typing derivations $\Gamma_1, x : T_x, \Gamma_2 \vdash e : T$, except for

the case T-VAR, where $\Gamma_1, x : T_x, \Gamma_2 \vdash x' : T, x' \in \text{dom}(\Gamma_1, x : T_x, \Gamma_2)$

(a) Case $x' = x$: $x'[x \mapsto v_x] = v_x$

Since $T_x = T, \Gamma_1 \vdash v_x : T$. By lemma 5.3, $x \notin \mathcal{FV}(v_x) \cup \mathcal{FV}(T)$, so that $T[x \mapsto v_x] = T$. By weakening and guard weakening, $\Gamma_1, \Gamma_2[x \mapsto v_x] \vdash v_x : T$.

(b) Case $x' \neq x$: $x'[x \mapsto v_x] = x'$

As $x' \neq x, x' \in \text{dom}(\Gamma_1, \Gamma_2)$ and hence, $x' : T[x \mapsto v_x] \in (\Gamma_1, \Gamma_2[x \mapsto v_x])$. By T-VAR, $\Gamma_1, \Gamma_2[x \mapsto v_x] \vdash x' : T[x \mapsto v_x]$.

2. Straightforward induction over type derivations $\Gamma_1, x : T_x, \Gamma_2 \vdash T$.

3. Straightforward induction over subtype derivations $\Gamma_1, x : T_x, \Gamma_2 \vdash T_1 <: T_2$.

4. By $\Gamma_1, x : T_x, \Gamma_2 \vdash e$ and inversion of PROOF, we know that $\forall \sigma$. (if $\Gamma_1, x : T_x, \Gamma_2 \models \sigma$, then $e[\sigma] \Rightarrow^* \text{true}$). We show that for all substitutions σ with $\Gamma_1, \Gamma_2[x \mapsto v_x] \models \sigma$, there is a substitution σ' with $\Gamma_1, x : T_x, \Gamma_2 \models \sigma'$, so that the result follows from PROOF.

Assume $\Gamma_1, \Gamma_2[x \mapsto v_x] \models \sigma_1, \sigma_2$. By Lemma 5.6, $\Gamma_1 \models \sigma_1, \Gamma_2[x \mapsto v_x][\sigma_1] \models \sigma_2$. By permutation of the substitution, $\Gamma_2[\sigma_1, x \mapsto v_x] \models \sigma_2$. From $\Gamma_1 \models \sigma_1$ and $\Gamma_1 \vdash v_x : T_x$ and by S-TYPE, $\Gamma_1, x : T_x \models \sigma_1, x \mapsto v_x$. By Lemma 5.6, $\Gamma_1, x : T_x, \Gamma_2 \models \sigma_1, x \mapsto v_x, \sigma_2$.

5. By 4., if $\Gamma_1, x : T_x, \Gamma_2 \vdash n = e$, then $\Gamma_1, \Gamma_2[x \mapsto v_x] \vdash n = e[x \mapsto v_x]$.

6. By induction on the derivation of $\Gamma_1, x : T_x, \Gamma_2 \vdash \overline{e_1} \bowtie \overline{e_2} = \overline{(v, w) \langle l \rangle}$. Under substitution, all rules apply as before. ■

A subtype relation must be reflexive and transitive, so that subtyping is a pre-order on types.

Theorem 5.11 (*Subtyping is a preorder*)

1. If $\Gamma \vdash T$, then $\Gamma \vdash T <: T$ (reflexivity)
2. If $\Gamma \vdash T_1 <: T_2$ and $\Gamma \vdash T_2 <: T_3$, then $\Gamma \vdash T_1 <: T_3$ (transitivity)

Proof: We show reflexivity and transitivity independently.

1. By induction on the derivation of $\Gamma \vdash T$.

(a) Case WF-BTYPE: $T = B$:

Immediate by SUB-BTYPE.

(b) Case WF-FUN: $T = x : T_1 \rightarrow T_2$ with $\Gamma \vdash T_1$ and $\Gamma, x : T_1 \vdash T_2$:

By the induction hypothesis, $\Gamma \vdash T_1 <: T_1$ and $\Gamma, x : T_1 \vdash T_2 <: T_2$. The result follows from SUB-FUN.

(c) Case WF-TUP: $T = (x : T_1, T_2)$ with $\Gamma \vdash T_1$ and $\Gamma, x : T_1 \vdash T_2$:

Similar.

(d) Case WF-INTVEC: $T = \text{intvec } e$ with $\Gamma \vdash e : \text{nat}$:

By reflexivity of equality, $\Gamma \vdash e = e$. The result follows from SUB-VEC.

- (e) Case WF-ARR: $T = [T | e_1, \dots, e_n]$ with $\Gamma \vdash T$ and $(\Gamma \vdash e_j : \text{natvec } l_j)_j$:
 By the hypothesis, $\Gamma \vdash T <: T$. By J-VECTORS, $\Gamma \vdash e_1, \dots, e_n \bowtie e_1, \dots, e_n = (e_1, e_1) \langle l_1 \rangle, \dots, (e_n, e_n) \langle l_n \rangle$. By reflexivity of equality, $(\Gamma \vdash \text{vfa } l_j e_j, e_j (x, y \rightarrow x = y))_j$. The result follows from SUB-ARR.
- (f) Case WF-REFTYPE: $T = \{x : T' | e\}$ with $\Gamma \vdash T'$ and $\Gamma, x : T' \vdash e : \text{bool}$:
 By the hypothesis, $\Gamma \vdash T' <: T'$. By reflexivity of implication, $\Gamma, x : T' \vdash e \Rightarrow e$.
 The results follows from SUB-REFTYPE.
2. By induction on the derivations of $\Gamma \vdash T_1 <: T_2$ and $\Gamma \vdash T_2 <: T_3$. As each subtyping rule compares two types of the same syntactic form, all three types must have the same syntactic form. We can thus check the rules individually.
- (a) Case SUB-BTYPE: $T_1 = B, T_2 = B, T_3 = B$. Immediate.
- (b) Case SUB-FUN: $T_1 = x : T_{11} \rightarrow T_{12}, T_2 = x : T_{21} \rightarrow T_{22}, T_3 = x : T_{31} \rightarrow T_{32}$ with $\Gamma \vdash T_{21} <: T_{11}, \Gamma, x : T_{21} \vdash T_{12} <: T_{22}$ and $\Gamma \vdash T_{31} <: T_{21}, \Gamma, x : T_{31} \vdash T_{22} <: T_{32}$
 By the hypothesis: $\Gamma \vdash T_{31} <: T_{11}$. By narrowing $\Gamma, x : T_{21} \vdash T_{12} <: T_{22}$ we obtain $\Gamma, x : T_{31} \vdash T_{12} <: T_{22}$. With $\Gamma, x : T_{31} \vdash T_{22} <: T_{32}$ and by the hypothesis, $\Gamma, x : T_{31} \vdash T_{12} <: T_{32}$. By SUB-FUN, $\Gamma \vdash x : T_{11} \rightarrow T_{12} <: x : T_{31} \rightarrow T_{32}$.
- (c) Case SUB-TUP: $T_1 = (x : T_{11}, T_{12}), T_2 = (x : T_{21}, T_{22}), T_3 = (x : T_{31}, T_{32})$ with $\Gamma \vdash T_{11} <: T_{21}, \Gamma, x : T_{11} \vdash T_{12} <: T_{22}$ and $\Gamma \vdash T_{21} <: T_{31}, \Gamma, x : T_{21} \vdash T_{22} <: T_{32}$
 By the hypothesis, $\Gamma \vdash T_{11} <: T_{31}$. By narrowing $\Gamma, x : T_{21} \vdash T_{22} <: T_{32}$, we obtain $\Gamma, x : T_{11} \vdash T_{22} <: T_{32}$. With $\Gamma, x : T_{11} \vdash T_{12} <: T_{22}$ and by the hypothesis, $\Gamma, x : T_{11} \vdash T_{12} <: T_{32}$. By SUB-TUP, $\Gamma \vdash (x : T_{11}, T_{12}) <: (x : T_{31}, T_{32})$.
- (d) Case SUB-VEC: $T_1 = \text{intvec } e_1, T_2 = \text{intvec } e_2, T_3 = \text{intvec } e_3$ with $\Gamma \vdash e_1 = e_2$ and $\Gamma \vdash e_2 = e_3$
 By transitivity of equality, $\Gamma \vdash e_1 = e_3$. By SUB-VEC, $\Gamma \vdash \text{intvec } e_1 <: \text{intvec } e_3$.
- (e) Case SUB-ARR: $T_1 = [T'_1 | \bar{e}_1], T_2 = [T'_2 | \bar{e}_2], T_3 = [T'_3 | \bar{e}_3]$ with $\Gamma \vdash T'_1 <: T'_2, \Gamma \vdash \bar{e}_1 \bowtie \bar{e}_2 = \overline{(u, v) \langle m \rangle}, (\Gamma \vdash \text{vfa } m_i u_i, v_i (x, y \rightarrow x=y))_i$, and $\Gamma \vdash T'_2 <: T'_3, \Gamma \vdash \bar{e}_2 \bowtie \bar{e}_3 = \overline{(v, w) \langle n \rangle}, (\Gamma \vdash \text{vfa } n_i v_i, w_i (x, y \rightarrow x=y))_i$
 By the hypothesis, $\Gamma \vdash T'_1 <: T'_3$. We split cases on how the structured vectors may have been joined and show that all cases satisfy the premises of SUB-ARR, so that $\Gamma \vdash [T'_1 | \bar{e}_1] <: [T'_3 | \bar{e}_3]$.
- i. Case J-EXFALSO, J-EXFALSO: $\Gamma \vdash \text{false}$
 By J-EXFALSO, $\Gamma \vdash \bar{e}_1 \bowtie \bar{e}_3 = ([], []) \langle 0 \rangle$. The element-wise equality of the empty vectors $\Gamma \vdash \text{vfa } 0 [], [] (x, y \rightarrow x=y)$ is trivially valid.
 - ii. Case J-VECTORS, J-VECTORS: $|\bar{e}_1| = |\bar{e}_2| = |\bar{e}_3| = n$,
 $(\Gamma \vdash e_{1i} : \text{intvec } l_i, \Gamma \vdash e_{2i} : \text{intvec } l_i, \Gamma \vdash e_{3i} : \text{intvec } l_i)_i$
 By J-VECTORS, $\Gamma \vdash \bar{e}_1 \bowtie \bar{e}_3 = (e_{11}, e_{31}) \langle l_1 \rangle, \dots, (e_{1n}, e_{3n}) \langle l_n \rangle$ and by transitivity of equality, $(\Gamma \vdash \text{vfa } l_i e_{1i}, e_{3i} (x, y \rightarrow x=y))_i$.
 - iii. Case J-ELEMS, J-ELEMS: $\Gamma \vdash \oplus \bar{e}_1 = [\bar{e}'_1], \Gamma \vdash \oplus \bar{e}_2 = [\bar{e}'_2], \Gamma \vdash \oplus \bar{e}_3 = [\bar{e}'_3]$
 and $|\bar{e}'_1| = |\bar{e}'_2| = |\bar{e}'_3| = n$
 By J-ELEMS, $\Gamma \vdash \bar{e}_1 \bowtie \bar{e}_3 = ([e'_1], [e'_3]) \langle n \rangle$. By transitivity of equality, $\Gamma \vdash \text{vfa } n [e'_1], [e'_3] (x, y \rightarrow x=y)$.

- iv. Case J-VECTORS, J-ELEMS: $|\overline{e_1}| = |\overline{e_2}| = m$, $(\Gamma \vdash e_{1i} : \text{intvec } l_i, \Gamma \vdash e_{2i} : \text{intvec } l_i)_i$, and $\Gamma \vdash \oplus \overline{e_2} = [\overline{e'_2}]$, $\Gamma \vdash \oplus \overline{e_3} = [\overline{e'_3}]$, $|\overline{e'_2}| = |\overline{e'_3}| = n$:
 The shapes $\overline{e_1}, \overline{e_2}$ have the same structure. By CONDENSE, $\Gamma \vdash \oplus \overline{e_1} = [\overline{e'_1}]$.
 By J-ELEMS, $\Gamma \vdash \overline{e_1} \bowtie \overline{e_3} = ([\overline{e'_1}], [\overline{e'_3}]) \langle n \rangle$. By transitivity of equality, $\Gamma \vdash \text{vfa } n [\overline{e'_1}], [\overline{e'_3}] (x, y \rightarrow x=y)$.
- v. Case J-ELEMS, J-VECTORS: Similar
- (f) Case SUB-REF: $T_1 = \{x : T'_1 \mid e_1\}$, $T_2 = \{x : T'_2 \mid e_2\}$, $T_3 = \{x : T'_3 \mid e_3\}$ with $\Gamma \vdash T'_1 <: T'_2$, $\Gamma, x : T'_1 \vdash e_1 \Rightarrow e_2$ and $\Gamma \vdash T'_2 <: T'_3$, $\Gamma, x : T'_2 \vdash e_2 \Rightarrow e_3$:
 By the hypothesis $\Gamma \vdash T'_1 <: T'_3$. By narrowing $\Gamma, x : T'_2 \vdash e_2 \Rightarrow e_3$ we obtain $\Gamma, x : T'_1 \vdash e_2 \Rightarrow e_3$. By transitivity of implication, $\Gamma, x : T'_1 \vdash e_1 \Rightarrow e_3$ and thus by SUB-REF, $\Gamma \vdash \{x : T'_1 \mid e_1\} <: \{x : T'_3 \mid e_3\}$. ■

The following lemma captures the intuition that evaluation does not affect the validity of a predicate.

Lemma 5.12 (Preservation of validity under evaluation)

If $\Gamma \vdash e_E[e]$ and $e \Rightarrow e'$, then $\Gamma \vdash e_E[e']$.

Proof: From the assumption and by inversion of PROOF, we obtain $\forall \sigma$. (if $\Gamma \models \sigma$, then $e_E[e][\sigma] \Rightarrow^* \text{true}$). By Theorem 4.20, evaluation is deterministic and therefore, $e_E[e][\sigma] \Rightarrow^* \text{true}$ iff $e_E[e'][\sigma] \Rightarrow^* \text{true}$. The result follows from PROOF. ■

Similarly, evaluation has no effect on types.

Lemma 5.13 (Equivalence of types under evaluation)

If $\Gamma \vdash T[e]$ and $e \Rightarrow e'$, then $\Gamma \vdash T[e] \equiv T[e']$, i. e., $\Gamma \vdash T[e'] <: T[e]$ and $\Gamma \vdash T[e] <: T[e']$.

Proof: By induction on type derivations $\Gamma \vdash T$.

1. Case WF-BTYPE: $T = B$
 B contains no expressions.
2. Case WF-FUN: $T = x : T_1[e] \rightarrow T_2[e]$, $\Gamma \vdash T_1[e]$, $\Gamma, x : T_1[e] \vdash T_2[e]$
 By the hypothesis, $\Gamma \vdash T_1[e'] \equiv T_1[e]$ and $\Gamma, x : T_1[e] \vdash T_2[e'] \equiv T_2[e]$. By SUB-FUN, $\Gamma \vdash x : T_1[e'] \rightarrow T_2[e'] \equiv x : T_1[e] \rightarrow T_2[e]$.
3. Case WF-TUP: $T = (x : T_1[e], T_2[e])$, $\Gamma \vdash T_1[e]$, $\Gamma, x : T_1[e] \vdash T_2[e]$
 By the hypothesis, $\Gamma \vdash T_1[e'] \equiv T_1[e]$ and $\Gamma, x : T_1[e] \vdash T_2[e'] \equiv T_2[e]$. By SUB-TUP, $\Gamma \vdash (x : T_1[e'], T_2[e']) \equiv (x : T_1[e], T_2[e])$.
4. Case WF-INTVEC: $T = \text{intvec } e_E[e]$, $\Gamma \vdash e_E[e] : \text{nat}$.
 By Theorem 4.20, evaluation is deterministic. Therefore, $e_E[e]$ and $e_E[e']$ will evaluate to the same value under all permitted substitutions, so that $\Gamma \vdash e_E[e] = e_E[e']$.
 By SUB-VEC, $\Gamma \vdash \text{intvec } e_E[e] \equiv \text{intvec } e_E[e']$.

5. Case WF-ARRAY: $T = [T'[e] | \overline{e_s[e]}]$, $\Gamma \vdash T'[e]$, $(\Gamma \vdash e_{s_j}[e] : \text{natvec } e_{l_j})_j$
 By the hypothesis, $\Gamma \vdash T'[e'] \equiv T'[e]$. Evaluation does not affect the structure of a structured vector, so that by J-VECTORS, $\Gamma \vdash e_{s_1}[e'], \dots, e_{s_m}[e'] \bowtie e_{s_1}[e], \dots, e_{s_m}[e] = (e_{s_1}[e'], e_{s_1}[e]) \langle e_{l_1} \rangle, \dots, (e_{s_m}[e'], e_{s_m}[e]) \langle e_{l_m} \rangle$. By Theorem 4.20, evaluation is deterministic. Therefore, for each j , $e_{s_j}[e]$ and $e_{s_j}[e']$ will evaluate to the same value under all permitted substitutions, so that $\Gamma \vdash e_{s_j}[e'] \stackrel{=}{=}_{e_{l_j}} e_{s_j}[e]$. By SUB-ARR, $\Gamma \vdash [T'[e'] | \overline{e_s[e']}] \equiv [T'[e] | \overline{e_s[e]}]$.
6. Case WF-REFTYPE: $T = \{x : T'[e] | e_E[e]\}$, $\Gamma \vdash T'[e]$, $\Gamma \vdash e_E[e] : \text{bool}$.
 By the hypothesis, $\Gamma \vdash T'[e'] \equiv T'[e]$. By Theorem 4.20, evaluation is deterministic. Therefore, $e_E[e]$ and $e_E[e']$ will evaluate to the same value under all permitted substitutions, so that by reflexivity of implication $\Gamma, x : T'[e] \vdash e_E[e] \Rightarrow e_E[e']$ and $\Gamma, x : T'[e] \vdash e_E[e'] \Rightarrow e_E[e]$. By SUB-REFTYPE, $\Gamma \vdash \{x : T'[e'] | e_E[e']\} \equiv \{x : T'[e] | e_E[e]\}$ ■

With all these facts in hand, we can now prove the preservation theorem.

Theorem 5.14 (Preservation of types under evaluation)

If $\Gamma \vdash e : T$ and $e \Rightarrow e'$, then $\Gamma \vdash e' : T$.

Proof: By induction on type derivations $\Gamma \vdash e : T$. We split cases on the potential evaluation steps.

1. Case T-SUB: $\Gamma \vdash e : T'$, $\Gamma \vdash T' <: T$, $\Gamma \vdash T$
 By the hypothesis, $\Gamma \vdash e' : T'$ and thus the result follows from T-SUB.
2. Case T-VAR: $e = x$. There is no e' with $e \Rightarrow e'$.
3. Case T-CONST: $e = c$. e is a value.
4. Case T-ABS: $e = \text{fun } x : T_1 \rightarrow e'$. e is a value.
5. Case T-APP: $e = e_1 e_2$, $\Gamma \vdash e_1 : (x : T_1 \rightarrow T_2)$, $\Gamma \vdash e_2 : T_1$, $T = T_2[x \mapsto e_2]$
 - (a) Case E-APP1: $e_1 \Rightarrow e'_1$, $e' = e'_1 e_2$
 By the hypothesis, $\Gamma \vdash e'_1 : (x : T_1 \rightarrow T_2)$. The result follows from T-APP.
 - (b) Case E-APP2: $e_1 = v_1$, $e_2 \Rightarrow e'_2$, $e' = v_1 e'_2$
 By T-APP, $\Gamma \vdash e' : T_2[x \mapsto e'_2]$. By Lemma 5.13, $\Gamma \vdash T_2[x \mapsto e'_2] <: T_2[x \mapsto e_2]$, so that the result follows from T-SUB.
 - (c) Case E-APPABS: $e_1 = \text{fun } x : T_1 \rightarrow e_b$, $e_2 = v_2$, $\Gamma, x : T_1 \vdash e_b : T_2$, $e' = e_b[x \mapsto v_2]$
 By Lemma 5.10, $\Gamma \vdash e' : T_2[x \mapsto v_2]$.
 - (d) Case E-PRFAPP1: $e_1 = f^1$, $e_2 = v_2$, $f^1(v_2) \Rightarrow_\delta v$, $e' = v$
 By Axiom 5.2, $\Gamma \vdash v : T_2[x \mapsto v_2]$.
 - (e) Case E-PRFAPP2: $e_1 = f^2 v_1$, $e_2 = v_2$, $f^2(v_1, v_2) \Rightarrow_\delta v$, $e' = v$. Similar.
6. Case T-LET: $e = \text{let } x = e_1 \text{ in } e_2$, $\Gamma \vdash e_1 : T_1$, $\Gamma, x : T_1 \vdash e_2 : T$, $\Gamma \vdash T$
 - (a) Case E-LETR: $e_1 \Rightarrow e'_1$, $e' = \text{let } x = e'_1 \text{ in } e_2$
 The result follows from the induction hypothesis and T-LET.

- (b) Case E-LET: $e_1 = v_1, e' = e_2[x \mapsto v_1]$
By the substitution lemma, $\Gamma \vdash e' : T[x \mapsto v_1]$. Since $\Gamma \vdash T$, x is not free in T and thus $\Gamma \vdash e' : T$.
7. Case T-COND: $e = \text{if} : T e_p \text{ then } e_t \text{ else } e_e, \Gamma \vdash T, \Gamma \vdash e_p : \text{bool}, \Gamma \vdash e_t : T, \Gamma \vdash e_e : T$
- (a) Case E-COND: $e_p \Rightarrow e'_p, e' = \text{if} : T e'_p \text{ then } e_t \text{ else } e_e$
The result follows from the induction hypothesis and T-COND.
- (b) Case E-COND1: $e_p = \text{true}, e' = e_t$. Immediate.
- (c) Case E-COND2: $e_p = \text{false}, e' = e_e$. Immediate.
8. Case T-COERCE: $e = (e_1 : T), \Gamma \vdash T, \Gamma \vdash e_1 : T$
- (a) Case E-COERCE1: $e_1 \Rightarrow e'_1, e' = (e'_1 : T)$
The result follows from the induction hypothesis and T-COERCE.
- (b) Case E-COERCE2: $e_1 = v_1, e' = v_1$. Immediate.
9. Case T-TUP: $e = (e_1, e_2 : (x : T_1, T_2)), \Gamma \vdash (x : T_1, T_2), \Gamma \vdash e_1 : T_1, \Gamma \vdash e_2 : T_2[x \mapsto e_1], T = (x : T_1, T_2)$
- (a) Case E-TUP1: $e_1 \Rightarrow e'_1, e' = (e'_1, e_2 : (x : T_1, T_2))$
By the hypothesis, $\Gamma \vdash e'_1 : T_1$. By Lemma 5.13, $\Gamma \vdash T_2[x \mapsto e_1] <: T_2[x \mapsto e'_1]$, so that by T-SUB, $\Gamma \vdash e_2 : T_2[x \mapsto e'_1]$. The result follows from T-TUP.
- (b) Case E-TUP2: $e_1 = v_1, e_2 \Rightarrow e'_2, e' = (v_1, e'_2 : (x : T_1, T_2))$
By the hypothesis, $\Gamma \vdash e'_2 : T_2[x \mapsto v_1]$. The result follows from T-TUP.
10. Case T-UNPACK: $e = \text{let } (x_1, x_2) = e_1 \text{ in } e_2, \Gamma \vdash e_1 : (x : T_1, T_2), \Gamma, x_1 : T_1, x_2 : T_2[x \mapsto x_1] \vdash e_2 : T, \Gamma \vdash T$
- (a) Case E-UNPACK1: $e_1 \Rightarrow e'_1, e' = \text{let } (x_1, x_2) = e'_1 \text{ in } e_2$
The result follows from the induction hypothesis and T-UNPACK.
- (b) Case E-UNPACK2: $e_1 = (v_1, v_2 : (x : T_1, T_2)), e' = e_2[x_1 \mapsto v_1][x_2 \mapsto v_2]$
By inversion of T-TUP, $\Gamma \vdash v_1 : T_1, \Gamma \vdash v_2 : T_2[x \mapsto v_1]$. By the substitution lemma 5.10 and because x_1 is not free in T , $\Gamma, x_2 : T_2[x \mapsto x_1][x_1 \mapsto v_1] \vdash e_2[x_1 \mapsto v_1] : T$. Since $T_2[x \mapsto x_1][x_1 \mapsto v_1] \equiv T_2[x \mapsto v_1]$, and by the substitution lemma, $\Gamma \vdash e_2[x_1 \mapsto v_1][x_2 \mapsto v_2] : T$.
11. Case T-VECTOR: $e = [\bar{e}], (\Gamma \vdash e_j : \text{int})_j, T = \{x : \text{intvec } |\bar{e}| \mid x \stackrel{\#}{=}_{|\bar{e}|} [\bar{e}]\}$
By E-SEQ and E-VECTOR, $\bar{e} \Rightarrow_{\text{seq}} \bar{e}'$ with $|\bar{e}| = |\bar{e}'|$ so that $e' = [\bar{e}']$. By T-VECTOR, $\Gamma \vdash e' : \{x : \text{intvec } |\bar{e}| \mid x \stackrel{\#}{=}_{|\bar{e}|} [\bar{e}']\}$. The result follows from Lemma 5.13 and T-SUB.
12. Case T-VSEL: $e = e_v.(e_i), \Gamma \vdash e_v : \text{intvec } e_n, \Gamma \vdash e_i : \{x : \text{int} \mid 0 \leq x < e_n\}, T = \{x : \text{int} \mid x = e_v.(e_i)\}$
- (a) Case E-VSELV: $e_v \Rightarrow e'_v, e' = e'_v.(e_i)$
By the hypothesis, $\Gamma \vdash e'_v : \text{intvec } e_n$. By T-VSEL, $\Gamma \vdash e' : \{x : \text{int} \mid x = e'_v.(e_i)\}$. The result follows from Lemma 5.13 and T-SUB.
- (b) Case E-VSELI: $e_v = v_v, e_i \Rightarrow e'_i, e' = v_v.(e'_i)$

- By the hypothesis, $\Gamma \vdash e'_i : \{x : \text{int} \mid 0 \leq x < e_n\}$. According to T-VSEL, $\Gamma \vdash e' : \{x : \text{int} \mid x = v_v \cdot (e'_i)\}$. The result follows from Lemma 5.13 and T-SUB.
- (c) Case E-VSEL: $e_v = [\bar{c}]$, $0 \leq e_i = i < |\bar{c}|$, $e' = c_i$
By T-CONST, $\Gamma \vdash e' : \{x : \text{int} \mid x = c_i\}$. The result follows from Lemma 5.13 and T-SUB.
13. Case T-VMOD: $e = e_v \cdot (e_i) \leftarrow e_e$, $\Gamma \vdash e_v : \text{intvec } e_n$, $\Gamma \vdash e_i : \{x : \text{int} \mid 0 \leq x < e_n\}$, $\Gamma \vdash e_e : \text{int}$, $T = \{x : \text{intvec } e_n \mid x \stackrel{\cong}{=}_{e_n} e_v \cdot (e_i) \leftarrow e_e\}$
- (a) Case E-VMV: $e_v \Rightarrow e'_v$, $e' = e'_v \cdot (e_i) \leftarrow e_e$
(b) Case E-VMI: $e_v = v_v$, $e_i \Rightarrow e'_i$, $e' = v_v \cdot (e'_i) \leftarrow e_e$
(c) Case E-VME: $e_v = v_v$, $e_i = v_i$, $e_e \Rightarrow e'_e$, $e' = v_v \cdot (v_i) \leftarrow e'_e$
The results follow from the hypothesis, T-VMOD, Lemma 5.13, and T-SUB.
- (d) Case E-VMOD: $e_v = [\bar{c}]$, $0 \leq e_i = i < |\bar{c}|$, $e_e = c_e$, $e' = [\bar{c}[i \mapsto c_e]]$
By T-VECTOR, $\Gamma \vdash e' : \{x : \text{intvec } |\bar{c}| \mid x \stackrel{\cong}{=}_{|\bar{c}|} [\bar{c}[i \mapsto c_e]]\}$. The result follows from Lemma 5.13 and T-SUB.
14. Case T-VEC: $e = \text{vec } e_n e_e$, $\Gamma \vdash e_n : \text{nat}$, $\Gamma \vdash e_e : \text{int}$, $T = \{x : \text{intvec } e_n \mid x = \text{vec } e_n e_e\}$
- (a) Case E-VECL: $e_n \Rightarrow e'_n$, $e' = \text{vec } e'_n e_e$
By the hypothesis, $\Gamma \vdash e'_n : \text{nat}$. By T-VEC, $\Gamma \vdash e' : \{x : \text{intvec } e'_n \mid x = \text{vec } e'_n e_e\}$. The result follows from Lemma 5.13 and T-SUB.
- (b) Case E-VECE: $e_n = v_n$, $e_e \Rightarrow e'_e$, $e' = \text{vec } v_n e'_e$
By the hypothesis, $\Gamma \vdash e'_e : \text{int}$. By T-VEC, $\Gamma \vdash e' : \{x : \text{intvec } v_n \mid x = \text{vec } v_n e'_e\}$. The result follows from Lemma 5.13 and T-SUB.
- (c) Case E-VEC: $0 \leq e_n = n$, $e_e = c_e$, $e' = [c_e, \dots, c_e]$
By T-VECTOR, $\Gamma \vdash e' : \{x : \text{intvec } n \mid x \stackrel{\cong}{=}_n [c_e, \dots, c_e]\}$. The result follows from Lemma 5.13 and T-SUB.
15. Case T-VMAP: $e = \text{vmap } e_n \bar{e}_v (\bar{x} \rightarrow e_b)$, $\Gamma \vdash e_n : \text{nat}$, $(\Gamma \vdash e_{vj} : \text{intvec } e_n)_j$, $\overline{x : \text{int}} \vdash e_b : \text{int}$, $T = \{x' : \text{intvec } e_n \mid x' \stackrel{\cong}{=}_{e_n} \text{vmap } e_n \bar{e}_v (\bar{x} \rightarrow e_b)\}$
- (a) Case E-VMAPL: $e_n \Rightarrow e'_n$, $e' = \text{vmap } e'_n \bar{e}_v (\bar{x} \rightarrow e_b)$
By the hypothesis, $\Gamma \vdash e'_n : \text{nat}$. According to the typing rule T-VMAP, $\Gamma \vdash e' : \{x' : \text{intvec } e'_n \mid x' \stackrel{\cong}{=}_{e'_n} \text{vmap } e'_n \bar{e}_v (\bar{x} \rightarrow e_b)\}$. The result follows from Lemma 5.13 and T-SUB.
- (b) Case E-VMAPV: $e_n = v_n$, $\bar{e}_v \Rightarrow_{\text{seq}} \bar{e}'_v$, $e' = \text{vmap } v_n \bar{e}'_v (\bar{x} \rightarrow e_b)$
By the hypothesis, $(\Gamma \vdash e'_{vj} : \text{intvec } v_n)_j$. According to T-VMAP, $\Gamma \vdash e' : \{x' : \text{intvec } v_n \mid x' \stackrel{\cong}{=}_{v_n} \text{vmap } v_n \bar{e}'_v (\bar{x} \rightarrow e_b)\}$. The result follows from Lemma 5.13 and T-SUB.
- (c) Case E-VMAP: $0 \leq n = e_n$, $\bar{e}_v = [\bar{c}_0], \dots, [\bar{c}_m]$, $|\bar{c}_i| = n$, $e' = [e_{b0}, \dots, e_{bn-1}]$ where $e_{bj} = e_b[x_0 \mapsto c_{0j}] \dots [x_m \mapsto c_{mj}]$
By Lemma 5.10, $(\Gamma \vdash e_{bj} : \text{int})_j$. According to the typing rule T-VECTOR, $\Gamma \vdash e' : \{x' : \text{intvec } n \mid x' \stackrel{\cong}{=}_n [e_{b0}, \dots, e_{bn-1}]\}$. The result follows from Lemma 5.13 and T-SUB.
16. Case T-VFA: $e = \text{vfa } e_n \bar{e}_v (\bar{x} \rightarrow e_b)$, $\Gamma \vdash e_n : \text{nat}$, $(\Gamma \vdash e_{vj} : \text{intvec } e_n)_j$, $\overline{x : \text{int}} \vdash$

$e_b : \text{bool}, T = \{x' : \text{bool} \mid x' \leftrightarrow \text{vfa } e_n \bar{e}_v (\bar{x} \rightarrow e_b)\}$

(a) Case E-VFAL: $e_n \Rightarrow e'_n, e' = \text{vfa } e'_n \bar{e}_v (\bar{x} \rightarrow e_b)$

By the hypothesis, $\Gamma \vdash e'_n : \text{nat}$. According to the typing rule T-VFA, $\Gamma \vdash e' : \{x' : \text{bool} \mid x' \leftrightarrow \text{vfa } e'_n \bar{e}_v (\bar{x} \rightarrow e_b)\}$. The result follows from Lemma 5.13 and T-SUB.

(b) Case E-VFAV: $e_n = v_n, \bar{e}_v \Rightarrow_{\text{seq}} \bar{e}'_v, e' = \text{vfa } v_n \bar{e}'_v (\bar{x} \rightarrow e_b)$

By the hypothesis, $(\Gamma \vdash e'_{v_j} : \text{intvec } v_n)_j$. According to the typing rule T-VFA, $\Gamma \vdash e' : \{x' : \text{bool} \mid x' \leftrightarrow \text{vfa } v_n \bar{e}'_v (\bar{x} \rightarrow e_b)\}$. The result follows from Lemma 5.13 and T-SUB.

(c) Case E-VFA: $0 \leq n = e_n, \bar{e}_v = [\bar{c}_0], \dots, [\bar{c}_m], e' = e_{b_0} \& \dots \& e_{b_{n-1}}$ where $e_{b_j} = e_b[x_0 \mapsto c_{0j}] \dots [x_m \mapsto c_{mj}]$

By Lemma 5.10, $(\cdot \vdash e_{b_j} : \text{bool})_j$. By T-APP, T-CONST, $\Gamma \vdash e' : \{x' : \text{bool} \mid x' \leftrightarrow e_{b_0} \& \dots \& e_{b_{n-1}}\}$. The result follows from Lemma 5.13 and T-SUB.

17. Case T-ARR: $e = [\bar{e} : T' \mid \bar{n}], \Gamma \vdash T', (\Gamma \vdash e_j : T')_j, T = [T' \mid \bar{n}]$

By E-SEQ and E-ARR, $\bar{e} \Rightarrow_{\text{seq}} \bar{e}'$ with $|\bar{e}| = |\bar{e}'|$. By the induction hypothesis, $(\Gamma \vdash e'_j : T')_j$. Thus, the result follows by T-ARR.

18. Case T-SEL: $e = e_a. [\bar{e}_i], \Gamma \vdash e_a : [T \mid \bar{e}_s], \Gamma \vdash \bar{e}_s \bowtie \bar{e}_i = \overline{(s, v) \langle r \rangle},$

$(\Gamma \vdash \text{vfa } r_j s_j, v_j (s, i \rightarrow 0 \leq i \& i < s))_j$

(a) Case E-SELA: $e_a \Rightarrow e'_a, e' = e'_a. [\bar{e}_i]$

By the hypothesis, $\Gamma \vdash e'_a : [T \mid \bar{e}_s]$, so that the result follows from T-SEL.

(b) Case E-SELI: $\bar{e}_i \Rightarrow_{\text{seq}} \bar{e}'_i, e' = e_a. [\bar{e}'_i]$

By the hypothesis, the type of each e_{ij} is preserved, so that \bar{e}'_i has the same structure as \bar{e}_i . Hence, $\Gamma \vdash \bar{e}_s \bowtie \bar{e}'_i = \overline{(s, v') \langle r \rangle}$. According to Lemma 5.12, $(\Gamma \vdash \text{vfa } r_j s_j, v'_j (s, i \rightarrow 0 \leq i \& i < s))_j$. By T-SEL, $\Gamma \vdash e' : T$.

(c) Case E-SEL: $e_a = [\bar{v} : T \mid \bar{n}], \bar{e}_i = [\bar{i}_1], \dots, [\bar{i}_m], e' = v_{(\iota \bar{n} (\bar{i}_1, \dots, \bar{i}_m))}$.

By inversion of T-ARR, $(\Gamma \vdash v_j : T)_j$ and therefore $\Gamma \vdash e' : T$.

19. Case T-MOD: $e = e_a. [\bar{e}_i] \leftarrow e_e, \Gamma \vdash e_a : [T' \mid \bar{e}_s], \Gamma \vdash e_e : T', \Gamma \vdash \bar{e}_s \bowtie \bar{e}_i = \overline{(s, v) \langle r \rangle},$
 $(\Gamma \vdash \text{vfa } r_j s_j, v_j (s, i \rightarrow 0 \leq i \& i < s))_j, T = [T' \mid \bar{e}_s]$

(a) Case E-MA: $e_a \Rightarrow e'_a, e' = e'_a. [\bar{e}_i] \leftarrow e_e$

By the hypothesis, $\Gamma \vdash e'_a : [T' \mid \bar{e}_s]$, so that the result follows from T-MOD.

(b) Case E-MI: $\bar{e}_i \Rightarrow_{\text{seq}} \bar{e}'_i, e' = e_a. [\bar{e}'_i] \leftarrow e_e$

By the hypothesis, the type of each e_{ij} is preserved, so that \bar{e}'_i has the same structure as \bar{e}_i . Hence, $\Gamma \vdash \bar{e}_s \bowtie \bar{e}'_i = \overline{(s, v') \langle r \rangle}$. By Lemma 5.12, $(\Gamma \vdash \text{vfa } r_j s_j, v'_j (s, i \rightarrow 0 \leq i \& i < s))_j$. The result follows from T-MOD.

(c) Case E-ME: $e_e \Rightarrow e'_e, e' = e_a. [\bar{e}_i] \leftarrow e'_e$

By the hypothesis, $\Gamma \vdash e'_e : T'$, so that the result follows from T-MOD.

(d) Case E-MOD: $e_a = [\bar{v} : T' \mid \bar{n}], \bar{e}_i = [\bar{i}_1], \dots, [\bar{i}_m], e_e = v_e,$

$e' = [\bar{v}[\iota \bar{n} (\bar{i}_1, \dots, \bar{i}_m) \mapsto v_e] : T' \mid \bar{n}]$.

By inversion of T-ARR, $(\Gamma \vdash v_j : T')$. Thus, by T-ARR, $\Gamma \vdash e' : [T' | \bar{n}]$. Since $\Gamma \vdash e_a : [T' | \bar{e}_s]$, we know that $\Gamma \vdash [T' | \bar{n}] <: [T' | \bar{e}_s]$, and therefore by T-SUB, $\Gamma \vdash e' : [T' | \bar{e}_s]$.

20. Case T-RSHP: $e = \text{reshape } \bar{e}_t e_a, (\Gamma \vdash e_{tj} : \text{natvec } e_{lj})_j, \Gamma \vdash e_a : [T' | \bar{e}_s], (\Gamma \vdash \text{vfa } e_{rj} e_{sj} (x \rightarrow 0 < x))_j, T = [T' | \bar{e}_t]$
- (a) Case E-RS: $\bar{e}_t \Rightarrow_{\text{seq}} \bar{e}'_t, e' = \text{reshape } \bar{e}'_t e_a$
By the hypothesis, $(\Gamma \vdash e'_{tj} : \text{natvec } e_{lj})_j$. By T-RESHAPE, $\Gamma \vdash e' : [T' | \bar{e}'_t]$. By Lemma 5.13, $\Gamma \vdash [T' | \bar{e}'_t] <: [T' | \bar{e}_t]$. The result thus follows from T-SUB.
- (b) Case E-RA: $e_a \Rightarrow e'_a, e' = \text{reshape } \bar{e}_t e'_a$
By the hypothesis, $\Gamma \vdash e'_a : [T' | \bar{e}_s]$. The result follows from T-RESHAPE.
- (c) Case E-RSHP: $\bar{e}_t = [\bar{i}_1], \dots, [\bar{i}_m], e_a = [\bar{v} : T' | \bar{n}], (0 < n)_j,$
 $e' = [\bar{v}' : T' | [\bar{i}_1, \dots, \bar{i}_m]], v'_k = v_{k \bmod \prod \bar{n}}$
By T-ARR, $\Gamma \vdash e' : [T' | [\bar{i}_1, \dots, \bar{i}_m]]$. By J-ELEMS, $\Gamma \vdash [\bar{i}_1, \dots, \bar{i}_m] \bowtie [\bar{i}_1], \dots, [\bar{i}_m] = ([\bar{i}_1, \dots, \bar{i}_m], [\bar{i}_1, \dots, \bar{i}_m]) <r>$ and $\Gamma \vdash \text{vfa } r [\bar{i}_1, \dots, \bar{i}_m], [\bar{i}_1, \dots, \bar{i}_m] (x, y \rightarrow x = y)$ holds because of the reflexivity of equality. By SUB-ARR, $\Gamma \vdash [T' | [\bar{i}_1, \dots, \bar{i}_m]] <: [T' | [\bar{i}_1], \dots, [\bar{i}_m]]$, so that by T-SUB, $\Gamma \vdash e' : T$.
21. Case T-GEN: $e = \text{gen} : T_b \bar{e}_s$ with $\bar{x} \rightarrow e_b, \Gamma \vdash T_b, (\Gamma \vdash e_{sj} : \text{natvec } e_{rj})_j, \Gamma, x_1 : \text{indexvec } e_{r1} e_{s1}, \dots, x_m : \text{indexvec } e_{rm} e_{sm} \vdash e_b : T_b, T = [T_b | \bar{e}_s]$
- (a) E-GENS: $\bar{e}_s \Rightarrow_{\text{seq}} \bar{e}'_s, e' = \text{gen} : T_b \bar{e}'_s$ with $\bar{x} \rightarrow e_b$
By the hypothesis, $(\Gamma \vdash e'_{sj} : \text{natvec } e_{rj})_j$. By T-GEN, $\Gamma \vdash e' : [T_b | \bar{e}'_s]$. By Lemma 5.13, $\Gamma \vdash [T_b | \bar{e}'_s] <: [T_b | \bar{e}_s]$. The result follows from T-SUB.
- (b) E-GEN: $\bar{e}_s = [\bar{n}_1], \dots, [\bar{n}_m], e' = [e'_b : T_b | [\bar{n}_1, \dots, \bar{n}_m]], e'_{bk} = e_b[x_j \mapsto [\bar{i}_j]]_j$
By construction, $(\Gamma \vdash [\bar{i}_j] : \text{indexvec } e_{rj} e_{sj})_j$. By the substitution lemma 5.10, $(\Gamma \vdash e_{bk} : T_b)_k$. By T-ARR, $\Gamma \vdash e' : [T_b | [\bar{n}_1, \dots, \bar{n}_m]]$. By rule J-ELEMS, $\Gamma \vdash [\bar{n}_1, \dots, \bar{n}_m] \bowtie [\bar{n}_1], \dots, [\bar{n}_m] = ([\bar{n}_1, \dots, \bar{n}_m], [\bar{n}_1, \dots, \bar{n}_m]) <r>$ and $\Gamma \vdash \text{vfa } r [\bar{n}_1, \dots, \bar{n}_m], [\bar{n}_1, \dots, \bar{n}_m] (x, y \rightarrow x = y)$ holds because of the reflexivity of equality. By SUB-ARR, $\Gamma \vdash [T_b | [\bar{n}_1, \dots, \bar{n}_m]] <: [T_b | [\bar{n}_1], \dots, [\bar{n}_m]]$, so that by T-SUB, $\Gamma \vdash e' : T$.
22. Case T-LOOP: $e = \text{loop } x_a : T = e_a; \bar{e}_s$ with $\bar{x} \rightarrow e_b, \Gamma \vdash T, (\Gamma \vdash e_{sj} : \text{natvec } e_{rj})_j, \Gamma \vdash e_a : T, \Gamma, x_1 : \text{indexvec } e_{r1} e_{s1}, \dots, x_m : \text{indexvec } e_{rm} e_{sm}, x_a : T \vdash e_b : T$
- (a) Case E-LOOPS: $\bar{e}_s \Rightarrow_{\text{seq}} \bar{e}'_s, e' = \text{loop } x_a : T = e_a; \bar{e}'_s$ with $\bar{x} \rightarrow e_b$
By the hypothesis, $(\Gamma \vdash e'_{sj} : \text{natvec } e_{rj})_j$. By T-LOOP, $\Gamma \vdash e' : T$.
- (b) Case E-LOOP: $(e_{sj} = [\bar{n}_j])_j, e' = f_{p-1} (\dots (f_0 e_a))$ with $p = \prod (\bar{n}_1, \dots, \bar{n}_m)$ and $f_k = \text{fun } x : T \rightarrow e_b[x_j \mapsto [\bar{i}_j]]_j$ such that $\iota (\bar{n}_1, \dots, \bar{n}_m) (\bar{i}_1, \dots, \bar{i}_m) = k$
By construction, $(\Gamma \vdash [\bar{i}_j] : \text{indexvec } e_{rj} e_{sj})_j$. By the substitution lemma 5.10, $\Gamma, x_a : T \vdash e_b[x_j \mapsto [\bar{i}_j]]_j : T$. By T-FUN, $\Gamma \vdash f_k : T \rightarrow T$. By several applications of T-APP, $\Gamma \vdash e' : T$. ■

The canonical forms lemma recapitulates the possible forms the values of a given type may have. The lemma is a straightforward observation from the syntax of values, the types of constants, and the typing rules.

Lemma 5.15 (Canonical forms)

1. If v is a value with $\cdot \vdash v : \text{bool}$ then v is either `true` or `false`.
2. If v is a value with $\cdot \vdash v : \text{int}$ then $v \in \mathbb{Z}$.
3. If v is a value with $\cdot \vdash v : x : T_1 \rightarrow T_2$ then $v = f^1$, $v = f^2$, $v = f^2 v_1$, or $v = \text{fun } x : T \rightarrow e$.
4. If v is a value with $\cdot \vdash v : (x : T_1, T_2)$ then $v = (v_1, v_2 : (x : T_1, T_2))$.
5. If v is a value with $\cdot \vdash v : \text{intvec } e$ then $v = [\bar{n}]$ with $e \Rightarrow^* |\bar{n}|$ and $(n_i \in \mathbb{Z})_i$.
6. If v is a value with $\cdot \vdash v : [T | e_1, \dots, e_m]$ then $v = [\bar{v} : T | [\bar{n}]]$ where each element v_i is a value with $\cdot \vdash v_i : T$ and $(n_i \geq 0)_i$, $|\bar{v}| = \prod \bar{n}$. Furthermore, $(e_i \Rightarrow^* [\bar{s}_i])_i$ with $\bar{s}_1, \dots, \bar{s}_m = \bar{n}$.
7. If v is a value with $\cdot \vdash v : \{x : T | e\}$ then v is any value v' with $\cdot \vdash v' : T$ and $e[x \mapsto v'] \Rightarrow^* \text{true}$.

Finally, we are ready to prove the progress theorem for QUBE_{CORE}.

Theorem 5.16 (Progress)

If $\cdot \vdash e : T$, then e is a value or there is some e' with $e \Rightarrow e'$.

Proof: By induction on typing derivations $\cdot \vdash e : T$.

1. Case T-SUB: $e = e$, $\cdot \vdash e : T'$, $\cdot \vdash T' <: T$, $\cdot \vdash T$
The result follows directly from the induction hypothesis.
2. Case T-VAR: $e = x$
 e is not closed.
3. Case T-CONST: $e = c$
 e is a value.
4. Case T-ABS: $e = \text{fun } x : T \rightarrow e'$
 e is a value.
5. Case T-APP: $e = e_1 e_2$, $\cdot \vdash e_1 : x : T_1 \rightarrow T_2$, $\cdot \vdash e_2 : T_1$
By the hypothesis, either e_1 is a value or it can make an evaluation step; e_2 similar. If e_1 is not a value, then E-APP1 applies. If e_1 is a value and e_2 is not, then E-APP2 applies. If both expressions are values, then by the canonical forms lemma either
 - $e_1 = f^2$ then $e = f^2 v_2$ is a value, or
 - $e_1 = f^1$ and E-PRFAPP1 applies because of axiom 5.2, or

- $e_1 = f^2 v_1$ and E-PRFAPP2 applies because of axiom 5.2, or
 - $e_1 = \text{fun } x : T' \rightarrow e'$ and E-ABSAPP applies.
6. Case T-LET: $e = \text{let } x = e_1 \text{ in } e_2$
By the hypothesis, e_1 is a value or it can make an evaluation step. If e_1 is not a value, then E-LETE applies, else E-LET applies.
 7. Case T-COND: $e = \text{if } e_1 \text{ then } e_2 \text{ else } e_3, \cdot \vdash e_1 : \text{bool}$
By the hypothesis, e_1 is a value or it can make an evaluation step. If e_1 is not a value, then E-COND applies. If e_1 is a value, then by the canonical forms lemma, it is either `true` or `false`. Thus either E-CONDT or E-CONDE applies.
 8. Case T-COERCE: $e = (e_1 : T_1)$
By the hypothesis, e_1 is a value or it can make an evaluation step. If e_1 is not a value, then E-COERCES applies, else E-COERCE applies.
 9. Case T-TUP: $e = (e_1, e_2 : (x : T_1, T_2))$
By the hypothesis, e_1, e_2 are values or they can make an evaluation step. If e_1 is not a value, then E-TUP1 applies. If e_1 is a value and e_2 is not, then E-TUP2 applies. If both expressions are values, then e is a value.
 10. Case T-UNPACK: $e = \text{let } (x_1, x_2) = e_1 \text{ in } e_2$
By the hypothesis, e_1 is a value or it can make an evaluation step. If e_1 is not a value, then E-UNPACKE applies. If e_1 is a value, then by the canonical forms lemma $e_1 = (v_1, v_2 : (x : T_1, T_2))$ and hence, E-UNPACK applies.
 11. Case T-VECTOR: $e = [\bar{e}], (\cdot \vdash e_j : \text{int})_j$
By the hypothesis, the expressions \bar{e} are either values or they can make an evaluation step. When an expression is not a value, then E-SEQ and E-VECTOR apply. When all expressions are values, they must all be integers (canonical forms lemma) and therefore, the vector is a value.
 12. Case T-VSEL: $e = e_v . (e_i), \cdot \vdash e_v : \text{intvec } e_l, \cdot \vdash e_i : \{x : \text{int} \mid 0 \leq x < e_l\}$
By the hypothesis, e and e_i are either values or they can make an evaluation step. If e_v is not a value, then E-VSELV applies. If e_v is a value and e_i is not, then E-VSELI applies. If both expressions are values, then, by the canonical forms lemma, e_v must be a vector $[\bar{n}]$ with $e_l \Rightarrow^* |\bar{n}|$ and e_i must be an integer i with $0 \leq i < |\bar{n}|$, so that E-VSEL applies.
 13. Case T-VMOD: Similar.
 14. Case T-VEC: $e = \text{vec } e_n e_e, \cdot \vdash e_n : \text{nat}, \cdot \vdash e_e : \text{int}$
By the hypothesis, e_n and e_e are either values or they can make an evaluation step. If e_n is not a value, then E-VECL applies. If e_n is a value and e_e is not, then E-VECE applies. If both expressions are values, then by the canonical forms lemma, e_n must be an integer n with $0 \leq n$ and e_e must be an integer, so that E-VEC applies.
 15. Case V-MAP: $e = \text{vmap } e_n \bar{e} (\bar{x} \rightarrow e_b), |\bar{e}| = |\bar{x}|, \cdot \vdash e_n : \text{nat}, (\cdot \vdash e_j : \text{intvec } e_n)_j$
By the hypothesis, e_n and the expressions \bar{e} are either values or they can make an evaluation step. If e_n is not a value, then E-VMAPL applies. If e_n is a value, and an e_i is not, then E-SEQ and E-VMAPV apply. If all expressions are values then, by the

canonical forms lemma, e_n must be an integer n with $0 \leq n$ and each expression e_j must be an integer vector $[\overline{m}_j]$ with $|\overline{m}_j| = n$. Since also $|\overline{e}| = |\overline{x}|$, E-VMAP applies.

16. Case T-VFA: $e = \text{vfa } e_n \overline{e} (\overline{x} \rightarrow e_b)$. Similar.
17. Case T-ARRAY: $e = [\overline{e} : T \mid \overline{n}]$, $(\cdot \vdash e_j : T)_j$
By the hypothesis, each expressions e_i is either a value of can make an evaluation step. When some expression e_i is not a value, then E-SEQ and E-ARR apply. When all expressions are values, the array is itself a value.
18. Case T-SEL: $e = e_a.[e_1, \dots, e_m]$, $\cdot \vdash e_a : [T \mid e_{s_1}, \dots, e_{s_n}]$, $\cdot \vdash e_{s_1}, \dots, e_{s_n} \bowtie e_1, \dots, e_m = (s, v) \langle r \rangle$, $(\cdot \vdash \text{vfa } r_i s_j, v_j (s, i \rightarrow 0 \leq i \& i < s))_j$
By the hypothesis, e_a and the e_i are either values or they can make an evaluation step. If e_a is not a value, E-SELA applies. If e_a is a value, but some e_i is not, then E-SEQ and E-SEL1 apply. If all subexpressions are values, then, by the canonical forms lemma, e_a must be an array value $[\overline{v} : T \mid \overline{n}]$ with $(e_{s_i} \Rightarrow^* [\overline{s}_i])_i$ so that $\overline{s}_1, \dots, \overline{s}_m = \overline{n}$. Moreover, each index vector e_j must be an integer vector $[\overline{i}_j]$. The remaining preconditions ensure that the shapes \overline{e}_s and \overline{e}_i can be joined and that the index vectors describe a valid position in e_a , i. e., $|i_1, \dots, i_n| = |n|$ and $(0 \leq (i_1, \dots, i_n)_j < n_j)_j$. Thus, E-SEL applies.
19. Case T-MOD: $e = e_a.[\overline{e}_i] \leftarrow e_e$. Similar.
20. Case T-RSHP: $e = \text{reshape } e_1, \dots, e_m e_a$, $(\cdot \vdash e_i : \text{natvec } _)_i$, $\cdot \vdash e_a : [T' \mid e_{s_1}, \dots, e_{s_n}]$, $(\cdot \vdash \text{vfa } e_{r_j} e_{s_j} (x \rightarrow 0 < x))_j$
By the hypothesis, e_a and the e_i are either values or they can make an evaluation step. If some e_i is not a value, then E-SEQ and E-RS apply. If all e_i are values, but e_a is not, then E-RA applies. If all subexpressions are values, then, by the canonical forms lemma, e_a must be an array value $[\overline{v} : T \mid \overline{n}]$ with $(e_{s_i} \Rightarrow^* [\overline{s}_i])_i$ so that $\overline{s}_1, \dots, \overline{s}_m = \overline{n}$. By the precondition, $(0 < n_j)_j$. Moreover, each shape vector e_j must be a vector of natural numbers $[\overline{i}_j]$. Therefore, E-RSHP applies.
21. Case T-GEN: $e = \text{gen } : T_b \overline{e}_s$ with $\overline{x} \rightarrow e_b$, $\cdot \vdash T_b$, $(\cdot \vdash e_{s_j} : \text{natvec } e_{r_j})_j$, $x_1 : (\text{indexvec } e_{r_1} e_{s_1}), \dots, x_m : (\text{indexvec } e_{r_m} e_{s_m}) \vdash e_b : T_b$
By the hypothesis, the e_{s_i} are either values or they can make an evaluation step. If some e_i is not a value, then E-SEQ and E-GENS apply. If all subexpressions are values, then by the canonical forms lemma, each e_{s_i} must be a vector of natural numbers and therefore E-GEN applies.
22. Case T-LOOP: $e = \text{loop } x_a : T = e_a ; \overline{e}_s$ with $\overline{x} \rightarrow e_b$, $\cdot \vdash T$, $(\cdot \vdash e_{s_j} : \text{natvec } e_{r_j})_j$, $\cdot \vdash e_a : T$, $x_1 : (\text{indexvec } e_{r_1} e_{s_1}), \dots, x_m : (\text{indexvec } e_{r_m} e_{s_m}), x_a : T \vdash e_b : T$
By the hypothesis, the e_{s_i} are either values or they can make an evaluation step. If some e_i is not a value, then E-SEQ and E-LOOPS apply. If all subexpressions are values, then by the canonical forms lemma, each e_{s_i} must be a vector of natural numbers and therefore E-LOOP applies. ■

The progress theorem states that each closed and well-typed term is either a value or can make an evaluation step. As stated by the preservation theorem,

each evaluation step preserves the type of the expression, so that the progress theorem applies to the result again. In our context, where we did not provide facilities for general recursion, this means that any well-typed $\text{QUBE}_{\text{CORE}}$ program will terminate and yield a value.

5.6 SMT-Based Validity Checking

The previous section used a straightforward yet undecidable formalisation of the proof relation $\Gamma \vdash e$ to show that the type system is safe. This section describes the actual, decidable implementation $\Gamma \vdash_D e$ of the proof relation. The implementation maps information from the context and the expression to a formula in the quantifier-free fragment of equality logic with uninterpreted functions and linear integer arithmetic (UFLIA). Moreover, array properties [18, 17, 66] are used to encode properties of integer vectors. The resulting formula can be decided by current automatic theorem provers, namely solvers for the Satisfiability Modulo Theories (SMT) problem of first-order logic [30, 33]. The relation $\Gamma \vdash_D e$ is a sound approximation of $\Gamma \vdash e$, so that $\Gamma \vdash_D e$ implies $\Gamma \vdash e$. On the other hand, we sacrifice completeness, so that $\Gamma \vdash e$ does not imply $\Gamma \vdash_D e$. In effect, the QUBE type checker behaves conservative: it rules out all programs that might get stuck during evaluation, but it also rejects some programs that actually behave well at run-time.

The rule DPROOF explains the implementation of $\Gamma \vdash_D e$. The rule forms a $\text{QUBE}_{\text{CORE}}$ expression that will evaluate to true when the guard expressions $\{e_G \mid e_G \in \Gamma\}$ and the refinements $\{e_R[x' \mapsto x] \mid x : \{x' : T \mid e_R\} \in \Gamma\}$ imply e . Then, the expression is encoded as a first-order formula using the function $\mathcal{C}[\![e]\!]$.

$$\frac{\text{Valid } \mathcal{C}[\![\bigwedge (\{e_G \mid e_G \in \Gamma\} \cup \{e_R[x' \mapsto x] \mid x : \{x' : T \mid e_R\} \in \Gamma)\}] \Rightarrow e]\!]}{\Gamma \vdash_D e} \quad (\text{DPROOF})$$

As an example, we show that the context $x : \{x' : \text{int} \mid 0 \leq x'\}, y : \text{int}, x \leq y$ implies $0 \leq y$. The encoding takes the refinement type of x into account. All occurrences of x' in the refinement predicate $0 \leq x'$ are substituted with x . The guard expression $x \leq y$ and the constraint $0 \leq y$ are translated straightforwardly.

Example 5.17 (Proving simple implications)

$$\frac{\text{Valid } (0 \leq x \wedge x \leq y \rightarrow 0 \leq y)}{x : \{x' : \text{int} \mid 0 \leq x'\}, y : \text{int}, x \leq y \vdash_D 0 \leq y}$$

The example only contains variables and applications of function symbols that are readily available in the quantifier-free UFLIA fragment of first-order logic. In general however, the encoding must also translate the other QUBE_{CORE} expressions, such as let-bindings, function definitions as well as vector and array expressions. We now explain the translation in more detail.

The languages of SMT solvers such as YICES [33] or Z3 [30] are variants of many-sorted first-order logic which generalises first-order logic as presented in Chapter 3. Similar to programming languages that use types to classify expressions, many-sorted logic uses sorts to classify terms. For the sake of simplicity, we do not consider sorts or well-sorted terms. We just assume that there are primitive sorts `bool` and `int` and that all our terms are well-sorted. Terms of sort `bool` are called formulas, so that a syntactic distinction between terms and formulas is redundant. Although the syntax of many-sorted logic permits arbitrary nestings of terms and formulas, both can be disentangled so that the conventional structure of first-order formulas can be restored.

In many-sorted logic, a term t is either a variable x , an application $f(t_1, \dots, t_n)$ of a function symbol f to terms, a conditional term $ite(t_1, t_2, t_3)$, or a quantified term $\forall x. t$ or $\exists x. t$. Constants are nullary function symbols. The boolean function symbols \top , \perp , \neg , \vee , \wedge , \vee , \rightarrow , and \leftrightarrow have the usual meaning. The sort `int` introduces function symbols for the integers \mathbb{Z} , the predicate symbols $=$, \neq , $<$, \leq , \geq , $>$, and the linear arithmetic functions $+$, $-$, and $\mathbb{Z}\cdot$. Moreover, there is a special binary function symbol $t_1[t_2]$ for reading an element with the integer index t_2 from an array t_1 .

Figure 5.11 shows the encoding function $\llbracket e \rrbracket(\kappa)$. The function takes a QUBE_{CORE} expression e and a function κ , which maps QUBE_{CORE} variables to logic terms, for processing local variable definitions. The result (t, T) consists of a term t that approximates the expression and a set T of additional constraints, which are also terms. The encoding function is defined using pattern matching: the first equation whose left-hand side matches the expression applies.

The encoding function $\mathcal{C}\llbracket \cdot \rrbracket$ used in DPROOF is defined in terms of $\llbracket \cdot \rrbracket(\cdot)$:

$$\mathcal{C}\llbracket e \rrbracket = \bigwedge (\{t\} \cup T) \text{ where } (t, T) = \llbracket e \rrbracket(\kappa_0)$$

The function $\kappa_0 : x \rightarrow t$ yields a logical variable \hat{x} for every QUBE_{CORE} variable x . The operation $\cdot[\cdot \leftarrow \cdot]$ alters a function κ such that $\kappa[x \leftarrow t](x) = t$. A variable \hat{x} always represents a fresh uninterpreted function symbol.

$\llbracket x \rrbracket(\kappa)$ encodes the variable x by applying κ . For let-bindings `let $x = e_1$ in e_2` , the encoding follows the operational semantics. First, e_1 is translated to (t_1, T_1) . Then, e_2 is encoded under $\kappa[x \leftarrow t_1]$ which gives (t_2, T_2) so that the let-expression can be encoded as $(t_2, T_1 \cup T_2)$.

Logical encoding of $\text{QUBE}_{\text{CORE}}$ expressions

$$\boxed{\llbracket e \rrbracket(\kappa) = (t, T)}$$

$$\begin{aligned} \llbracket x \rrbracket(\kappa) &= (\kappa(x), \emptyset) \\ \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket(\kappa) &= (t_2, T_1 \cup T_2) \\ &\quad \text{where } (t_1, T_1) = \llbracket e_1 \rrbracket(\kappa), \\ &\quad \quad (t_2, T_2) = \llbracket e_2 \rrbracket(\kappa[x \leftarrow t_1]) \\ \llbracket \text{true} \rrbracket(\kappa) &= (\top, \emptyset) \\ \llbracket \text{false} \rrbracket(\kappa) &= (\perp, \emptyset) \\ \llbracket n \rrbracket(\kappa) &= (n, \emptyset) \\ \llbracket \text{not } e \rrbracket(\kappa) &= (\neg t, T) \text{ where } (t, T) = \llbracket e \rrbracket(\kappa) \\ \circ \in \{\leftrightarrow, \&, |\}. \llbracket e_1 \circ e_2 \rrbracket(\kappa) &= (t_1 \hat{\circ} t_2, T_1 \cup T_2) \text{ where } (t_i, T_i) = \llbracket e_i \rrbracket(\kappa) \\ \circ \in \{=, <, \dots\}. \llbracket e_1 \circ e_2 \rrbracket(\kappa) &= (t_1 \hat{\circ} t_2, T_1 \cup T_2) \text{ where } (t_i, T_i) = \llbracket e_i \rrbracket(\kappa) \\ \circ \in \{+, -\}. \llbracket e_1 \circ e_2 \rrbracket(\kappa) &= (t_1 \hat{\circ} t_2, T_1 \cup T_2) \text{ where } (t_i, T_i) = \llbracket e_i \rrbracket(\kappa) \\ \llbracket n * e \rrbracket(\kappa) &= (n * t, T) \text{ where } (t, T) = \llbracket e \rrbracket(\kappa) \\ \circ \in \{*, /, \%\}. \llbracket e \circ n \rrbracket(\kappa) &= (t \hat{\circ} n, T) \text{ where } (t, T) = \llbracket e \rrbracket(\kappa) \\ \llbracket e_1 e_2 \rrbracket(\kappa) &= (t_1(t_2), T_1 \cup T_2) \text{ where } (t_i, T_i) = \llbracket e_i \rrbracket(\kappa) \\ \llbracket \text{if } :T e_1 \text{ then } e_2 \text{ else } e_3 \rrbracket(\kappa) &= (\text{ite}(t_1, t_2, t_3), T_1 \cup T_2 \cup T_3) \\ &\quad \text{where } (t_i, T_i) = \llbracket e_i \rrbracket(\kappa) \\ \llbracket (e_1, e_2 : T) \rrbracket(\kappa) &= (\tilde{x}, \emptyset) \\ \llbracket \text{let } (x_1, x_2) = e_1 \text{ in } e_2 \rrbracket(\kappa) &= \llbracket e_2 \rrbracket(\kappa) \\ \llbracket (e : T) \rrbracket(\kappa) &= \llbracket e \rrbracket(\kappa) \\ \llbracket e_1 . (e_2) \rrbracket(\kappa) &= (t_1[t_2], T_1 \cup T_2) \text{ where } (t_i, T_i) = \llbracket e_i \rrbracket(\kappa) \\ \llbracket e_1 . (e_2) \leftarrow e_3 \rrbracket(\kappa) &= (\tilde{x}, \{\tilde{x}[t_2] = t_3 \wedge \forall j. j \neq t_2 \rightarrow \tilde{x}[j] = t_1[j]\} \cup T) \\ &\quad \text{where } (t_i, T_i) = \llbracket e_i \rrbracket(\kappa), T = T_1 \cup T_2 \cup T_3 \\ \llbracket [\bar{e}] \rrbracket(\kappa) &= (\tilde{x}, \{\tilde{x}[i] = t_i\}_i \cup \bigcup_i T_i) \text{ where } (t_i, T_i) = \llbracket e_i \rrbracket(\kappa) \\ \llbracket \text{vec } e_1 e_2 \rrbracket(\kappa) &= (\tilde{x}, \{\tilde{y} = t_2 \wedge \forall j. 0 \leq j < t_1 \rightarrow \tilde{x}[j] = \tilde{y}\} \cup T_1 \cup T_2) \\ &\quad \text{where } (t_i, T_i) = \llbracket e_i \rrbracket(\kappa) \\ \llbracket \text{vmap } e_n \bar{e} (\bar{y} \rightarrow e_b) \rrbracket(\kappa) &= (\tilde{x}, \{\forall j. 0 \leq j < t_n \rightarrow \tilde{x}[j] = t_b[y_i \mapsto t_i[j]]\}_i \cup T) \\ &\quad \text{where } T = T_n \cup \bigcup_i T_i, (t_i, T_i) = \llbracket e_i \rrbracket(\kappa) \\ \llbracket \text{vfa } e_n \bar{e} (\bar{y} \rightarrow e_b) \rrbracket(\kappa) &= (\tilde{x}, \{\tilde{x} \leftrightarrow \forall j. 0 \leq j < t_n \rightarrow t_b[y_i \mapsto t_i[j]]\}_i \cup T) \\ &\quad \text{where } T = T_n \cup \bigcup_i T_i, (t_i, T_i) = \llbracket e_i \rrbracket(\kappa) \\ \llbracket e \rrbracket(\kappa) &= (\tilde{x}, \emptyset) \end{aligned}$$

Figure 5.11: A mapping from $\text{QUBE}_{\text{CORE}}$ expressions to logical formulas. The first equation whose left-hand side matches the expression applies.

Booleans and integer constants map to the corresponding logical constants. Similarly, we encode applications of boolean operators, (linear) arithmetic operators, and conditional expressions as applications of the equivalent interpreted functions. All other functions map to uninterpreted function symbols by the bottom-most rule.

Tuples $(e_1, e_2 : T)$ are abstractly represented as uninterpreted functions \tilde{x} . Similarly, the encoding of unpack-expressions $\text{let } (x_1, x_2) = e_1 \text{ in } e_2$ merely takes e_2 into account, abstracting away from the relationship between the variables x_1, x_2 and the expression e_1 . The encoding function ignores type annotations $(e : T)$.

In contrast, the encoding function emphasises the accurate representation of vectors and vector operations. Selections $e_1.(e_2)$ are encoded as logical array reads $t_1[t_2]$. The vector modification $e_1.(e_2) \leftarrow e_3$ represents a vector that is equal to e_1 at all position except e_2 , where it is e_3 . We thus encode the vector modification as a new logical array \tilde{x} with the additional constraints $\tilde{x}[t_2] = t_3$ and $\forall j. j \neq t_2 \rightarrow \tilde{x}[j] = t_1[j]$. In a similar spirit, we encode vector constructors $[\bar{e}]$ as fresh arrays \tilde{x} with $x[j] = t_j$ for all j and constant-value vector expressions $\text{vec } e_1 e_2$ as arrays \tilde{x} with $\tilde{y} = t_2$ and $\forall j. 0 \leq j < t_1 \rightarrow \tilde{x}[j] = \tilde{y}$.

Vectors defined by map expressions $\text{vmap } e_n \bar{e} (\bar{y} \rightarrow e_b)$ are also converted to new logical arrays \tilde{x} . Modelling the operational semantics of vmap , the array is constrained such that each element $\tilde{x}[j]$ must equal $t_b[y_i \mapsto t_i[j]]_i$, i. e., every y_i in t_b gets replaced with $t_i[j]$. The encoding of vector predicates $\text{vfa } e_n \bar{e} (\bar{x} \rightarrow e_b)$ is similar, but introduces \tilde{x} as a truth variable that is constrained by the universally quantified formula $\forall j. 0 \leq j < t_n \rightarrow t_b[x_i \mapsto t_i[j]]_i$.

All other expressions, namely abstractions, and the expressions from QUBE_□, are encoded as uninterpreted functions \tilde{x} .

We now prove some essential properties of the transformation. Given an expression, the encoding function yields a term and a set of constraints that are lifted out of their original context. In the presence of universal quantifiers, it is crucial that no term that depends on a quantified variable is moved out of the scope of the quantifier.

Proposition 5.18 (Expressions under vmap and vfa yield no constraints)

For map operations $\text{vmap } e_n \bar{e} (\bar{x} \rightarrow e_b)$ and predicates $\text{vfa } e_n \bar{e} (\bar{x} \rightarrow e_b)$, encoding of the e_b yields an empty constraint set, i. e.,

$$\llbracket e_b \rrbracket (\kappa) = (t_b, \emptyset)$$

Proof: Constraints are only generated by the encoding rules for $[\cdot]$, $\cdot(\cdot) \leftarrow \cdot$, vec , vmap , and vfa . The typing rules treat e_b as a restricted expression and thereby ensure that none of the above expressions may appear in e_b . ■

An encoded formula may contain some universal quantifiers and thus may not be immediately processed by an SMT solver. We show that the universally quantified terms all belong to the decidable array property fragment [18], so that the entire formula can be transformed into an equisatisfiable quantifier-free formula by means of the decision procedure outlined in Section 3.4.

Proposition 5.19 (Quantified terms are array properties)

The universally quantified terms that result from encoding vector expressions are array properties.

Proof: Only the encoding rules for $\cdot.(\cdot) \leftarrow \cdot$, vec , vmap , and vfa emit universally quantified terms. We inspect the rules individually.

1. Case $e_1.(e_2) \leftarrow e_3$:
The quantified term is $\forall j. j \neq t_2 \rightarrow \tilde{x}[j] = t_1[j]$. The index guard $j \neq t_2$ is a shorthand for $j \leq t_2 - 1 \vee t_2 + 1 \leq j$. By construction, e_2 and thus t_2 cannot depend on j , which allows us to abstract out t_2 out of the index guard as a new variable. In the value constraint $\tilde{x}[j] = t_1[j]$, the index variable j is only used to select into arrays.
2. Case $\text{vec } e_1 e_2$:
The quantified term is $\forall j. 0 \leq j < t_1 \rightarrow \tilde{x}[j] = \tilde{y}$. Similar to the above case, the index guard may be rewritten as $0 \leq j \leq t_1 - 1$ and t_1 may be abstracted out as it cannot depend on j . The value constraint obeys the required grammar.
3. Case $\text{vmap } e_n \bar{e} (\bar{x} \rightarrow e)$:
The quantified term is $\forall j. 0 \leq j < t_n \rightarrow \tilde{x}[j] = t_b[y_i \rightarrow t_i[j]]_i$. The index guard may be transformed to meet requirements as in case 2. Since e_b is a restricted expression, the term t_b can only contain constants, the variables \bar{x} as integers, logical and arithmetic functions and if-then-else terms. In particular, t_b cannot by itself reference j . Therefore, substituting the variables x_i with $t_i[j]$ yields a well-formed value constraint.
4. Case $\text{vfa } e_n \bar{e} (\bar{x} \rightarrow e)$: Similar. ■

The encoding function is accurate for boolean expressions, integers and linear integer operations, vector expressions and let-bindings. All other expressions are soundly abstracted away with uninterpreted function symbols. When only expressions of the former group are used in program positions where the proof relation is used to reason about values, the type checker will not report any false type errors. Specifically, these positions are type refinements, length expressions in vector types, shape expressions in array types, conditional predicates, function arguments or tuple components when a value of some refinement type is expected, length expressions in vec , vmap , and vfa , vector indices, shape

expressions in `reshape`, `gen`, `loop`, and array index vectors. Despite the extensive list, many interesting array programs can be typed. Some examples will be shown in Section 8. Moreover, in cases where the proof machinery fails to accurately reason about the value of a complex expression, appropriate run-time checks can be inserted manually.

Summary

This chapter presented the type system of QUBE_{CORE} and gave proof of its safety. However, even with SMT-based constraint proving, the presented rules are not immediately suitable for implementation. Responsible for this is the subsumption rule T-SUB which applies to every expression arbitrarily often. In contrast, all other rules are syntax directed: for every expression, there is one and only one type checking rule that matches that expression. Thus, to implement a terminating type checking algorithm, it is only necessary to tame the subsumption rule.

QUBE_{CORE} is constructed such that type checking never requires to guess an appropriate supertype for an expression. Instead, the language elements are annotated with these types. For example the conditional `if :T e then et else ee` provides a common supertype T for both branches. Thus, in all situations where the declarative rules use the subsumption rule to appropriately elevate the type of an expression, it is sufficient to check whether the type is a subtype of the required type and we can therefore eliminate T-SUB.

Part III

The QUBE Programming Language

6

The QUBE Programming Language

This chapter describes the actual syntax of the QUBE programming language. The syntax of QUBE is heavily inspired by OCAML [69], a popular dialect of the ML family of functional languages. Programmers familiar with OCAML should be able to quickly learn QUBE, too. Aimed to be a practical language for array programming with dependent types, QUBE extends $\text{QUBE}_{\text{CORE}}$ with a richer expression syntax, more base types, some syntactic sugar, an ML-style module system [83], and support for stateful computations and I/O. The latter two extensions were added to the QUBE compiler by Florian Büther in the course of his Diploma thesis [20]. Hence, this chapter only gives a brief overview of these features and refers the reader to the aforementioned thesis for details.

The remainder of the chapter is organised as follows: Section 6.1 explains the expression syntax of QUBE and how it extends $\text{QUBE}_{\text{CORE}}$. Section 6.2 outlines the module system of QUBE. Finally, Section 6.3 explains how QUBE integrates stateful computations and I/O operations by means of uniqueness types.

6.1 Expression Syntax

This section explains the expression syntax of QUBE and how it is different from $\text{QUBE}_{\text{CORE}}$. Fig. 6.1 specifies an (abstract) grammar of QUBE expressions in extended Backus-Naur form.

To facilitate general-purpose computations, QUBE provides the additional base

types `char`, `float`, and `double` along with the required constants, arithmetic operations, and conversion operations.

QUBE generalizes the unary functions of `QUBECORE` to functions of arbitrary arity. The n -ary function type has the form $x_1:T_1 \dots x_n:T_n \rightarrow T_{n+1}$. The syntax of function types allows the compiler to statically determine how many arguments are required for an exact application of a given operator. The corresponding abstraction has the form `fun $x_1:T_1 \dots x_n:T_n : T_{n+1} \rightarrow e$` , where the declaration of the return type $:T_{n+1}$ is optional. The syntax of the extended application that allows an operator to be applied to multiple operands is $e e_1 \dots e_m$, where m may differ from the arity of e . The following example illustrates the definition of a ternary function and its function type.

Example 6.1 (Function with multiple arguments)

```
(fun x:int y:int z:int :int → x+y+z) : x:int. y:int. z:int → int
```

Variables that denote binary functions can be enclosed in backticks and be used as infix operators. For example, `x `add` y` is equivalent to `add x y`.

Similar to the generalisation of functions, QUBE generalises pairs to (dependent) tuples with n components for $n \geq 2$. The dependent tuple type of length n has the form $(x_1:T_1, \dots, x_{n-1}:T_{n-1}, T_n)$. The syntax of the dependent tuple constructor is $(e_1, \dots, e_n : (x_1:T_1, \dots, x_{n-1}:T_{n-1}, T_n))$, whereas a non-dependent tuple may simply be specified as (e_1, \dots, e_n) . The example shows how dependent triples can be used to represent arrays of different rank and shape in a uniform way.

Example 6.2 (Dependent triples)

```
(1, [5] , [1,2,3,4,5| [5]] : (r:nat, s:natvec r, [int|s]))
(2, [2,2], [1,2,3,4| [2,2]] : (r:nat, s:natvec r, [int|s]))
```

The type annotation at the conditional expression is optional in QUBE. In most expressions `if e_p then e_t else e_e` , either the type of e_t is a supertype of the type of e_e , or vice versa. In these cases, the compiler will automatically assume the greater type as the type of the conditional.

The `let` construct of QUBE combines the `let` and `unpack` expressions from `QUBECORE` and adds support for mutually recursive function definitions. The non-recursive binding `let $p_1 = e_1$ and ... and $p_n = e_n$ in e_{n+1}` evaluates the expressions e_1, \dots, e_n in arbitrary order and matches the results with the patterns p_1, \dots, p_n such that all variables from the patterns are bound to values in e_{n+1} . Patterns may be variables x and tuple patterns (p_1, \dots, p_m) for $m \geq 2$.

The recursive binding `let rec $p_1 = e_1$ and ... and $p_n = e_n$ in e_{n+1}` defines a set of mutually recursive functions under the restriction that all patterns p_i are

$T ::= B \mid \{x:T \mid e\} \mid \{[x:]T.\} [x:]T \rightarrow T$ $\mid (\{[x:]T,\}^+ T) \mid \text{intvec } e \mid [T \mid e\{,e\}] \mid X\{e\}$	Types
$B ::= \text{bool} \mid \text{int} \mid \text{char} \mid \text{float} \mid \text{double}$	Base types
$e ::= c \mid x \mid \text{fun } \{x:T\}^+ [:T] \rightarrow e \mid e\{e\}^+$ $\mid (e:T) \mid (e\{,e\}^+ [:T]) \mid \text{if } [:T] e \text{ then } e \text{ else } e$ $\mid \text{let } [\text{rec}] lb \{ \text{and } lb \} \text{ in } e$ $\mid e.(e) \mid e.(e) \leftarrow e \mid \text{vec } e \ e$ $\mid \text{vmap } e\{,e\} (x\{,x\} \rightarrow e) \mid \text{vfa } e\{,e\} (x\{,x\} \rightarrow e)$ $\mid [[e\{,e\}] [:T] [[[Z\{,Z\}]]]]$ $\mid e.[e\{,e\}] \mid e.[e\{,e\}] \leftarrow e \mid \text{reshape } e\{,e\} \ e$ $\mid \text{gen } [:T] e\{,e\} \text{ with } g$ $\mid \text{loop } x [:T] = e; e\{,e\} \text{ with } g$	Expressions
$lb ::= p = e \mid x \{x:T\}^+ [:T] = e$	Let bindings
$p ::= x \mid (p\{,p\}^+)$	Patterns
$g ::= vp\{,vp\} \rightarrow e \mid rp\{,rp\} \rightarrow e \mid vp\{,vp\} \rightarrow e$	Generators
$vp ::= x \mid [[x\{,x\}]]$	Vector patterns
$rp ::= [\text{@r as}] vp$	Range patterns
$r ::= e \mid e.. \mid ..e \mid e..e$	Ranges

Figure 6.1: The expression syntax of QUBE

mere variables x_i and that all right-hand side expressions e_i are abstractions $\text{fun } x_1 : T_1 \dots x_n : T_n : T_{n+1} \rightarrow e$ that provide a result type.

As a convenience notation, QUBE provides a special let binding for function definitions. The binding $x_f x_1 : T_1 \dots x_n : T_n : T_{n+1} = e$ is equivalent to writing $x_f = \text{fun } x_1 : T_1 \dots x_n : T_n : T_{n+1} \rightarrow e$. In both notations, the result type is only required for declaring recursive functions. To illustrate the let expression, we define the well-known mutually recursive functions even and odd.

Example 6.3 (Mutually recursive functions)

```

let rec even n:nat :bool = if n = 0 then true  else odd  (n-1)
and      odd  n:nat :bool = if n = 0 then false else even (n-1)
in even 4

```

The vector and array expressions of QUBE differ very little from their counterparts in QUBE_{CORE}. Most notably, the vector constructor and the array constructor have been fused into a common expression. The element type and shape anno-

tations are optional. If the type annotation is missing, the compiler determines a common supertype of all elements. For empty array constructors, the element type defaults to `int`. If the shape annotation is missing, the array constructor will be assumed to define a rank 1 array. Vectors with integer elements will be given an `intvec` type, as described by rule T-VECTOR in Figure 5.7. Via an additional subtyping rule, the compiler allows integer vectors to be used as rank 1 integer arrays. In QUBE, `vmap` and `vfa` do not require the vector length as an argument.

The `gen` and `loop` expressions of QUBE employ a restricted form of pattern matching on index vectors to allow the elements in a convex sub-region of the index space to be computed from a different expression than all other elements. An array comprehension of the form

```
gen s with @lb..ub as x → e | x' → e'
```

specifies an array of shape vector `s`, whose elements are either computed from the expression `e` or from the default expression `e'`. Which of these two expressions is chosen depends on the element's index vector. If the index vector is within the range specified by the *range pattern* `@lb..ub`, where the lower bound vector `lb` is inclusive and the upper bound vector `ub` is exclusive, `e` is chosen, otherwise `e'` is taken. In situations where the length of the index vector is constant, QUBE allows the index vector to be alternatively specified as a *vector pattern* `[x̄]`. The element type of `gen` and the accumulator type of `loop` are optional as the compiler can in many cases determine an appropriate type itself. To illustrate range and vector patterns, we define an operation that concatenates two vectors of doubles.

Example 6.4 (Range and vector patterns)

```
let concat m:nat n:nat a:[double|[m]] b:[double|[n]] =
  gen [m+n] with
  | @[0]..[m] as x → a.[x]
  | [i] → b.[[i-m]]
```

6.2 Module System

To support large-scale programming beyond individual expressions, QUBE features an elaborate module system [20, 83]. The system supports separate namespaces, separate compilation, information hiding, and interfacing with foreign functions.

A module consists of a signature that describes all the types and values provided by the module and a structure that contains the actual definitions. Each source

M	$::=$	<code>struct {b} end</code> <code>{m.}m</code> <code>M:I</code>	Module expressions
b	$::=$	<code>type X {x:T} = T</code> <code>let [rec] lb {and lb}</code> <code>structure m = M</code> <code>signature i = I</code> <code>open {m.}m</code> <code>external x : T = symbol</code>	Bindings
I	$::=$	<code>sig {d} end</code> <code>{m.}i</code>	Interface expressions
d	$::=$	<code>type X {x:T} [= T]</code> <code>val x : T</code> <code>structure m : I</code> <code>signature i = I</code> <code>open {m.}m</code> <code>external x : T = symbol</code>	Declarations
T	$::=$	<code>...</code> <code>{m.}X</code>	Types
e	$::=$	<code>...</code> <code>{m.}x</code>	Expressions

Figure 6.2: The module syntax of QUBE

code file `A.q` implicitly defines a structure `A`. Similarly, a source code file `A.qi` implicitly defines a signature `A`. Alternatively, structures and signatures can be defined explicitly, even inside of other structures.

Figure 6.2 shows the module syntax of QUBE. In the figure, four different kinds of identifiers are used. As before, the metavariables x and X represent value and type identifiers, respectively. The metavariable m stands for a module identifier and i stands for an interface identifier. Module identifiers must begin with an uppercase letter, all other identifiers must begin with a lowercase letter.

To navigate through the module hierarchy, both value identifiers and type identifiers are extended with a module path. For example, `Cpx.make` selects the value `make` from a module `Cpx`.

A module expression M is either a structure definition `struct bs end` that packages together related bindings bs , a path $m_1 \dots m_n . m$ that selects a structure m from a module tree, or a module expression of the form $M : I$ that restricts the interface of the module M to the interface I .

The binding type $X \ x_1 : T_1 \dots x_n : T_n = T_{n+1}$ defines a new type X that takes arguments x_i of type T_i as an abbreviation of the right-hand side type T_{n+1} . Similar to the `let` expression, the `let` binding simultaneously defines a number of (potentially recursive) values.

A binding of the form `structure m = M` binds a structure m that is defined by the module expression M . Similarly, a binding of the form `signature i = I` binds a signature i that is defined by the interface expression I .

A binding `open m_1 \dots m_n . m` makes all names from the module $m_1 \dots m_n . m$ available in the current structure without the module path.

Finally, a binding of the form `external x : T = symbol` declares an externally

```

structure Cpx = struct
  type t = (double,double)
  let make re:double im:double = (re,im)
  let re c:t = let (re,_) = c in re
  let im c:t = let (_,im) = c in im
  (* ... *)
end : sig
  type t
  val make : double. double → t
  val re : t → double
  val im : t → double
  (* ... *)
end

```

Figure 6.3: An example module for complex numbers

defined function x of type T that is implemented by the linker symbol *symbol*.

Analogue to module expressions, an interface expression is either a signature definition `sig ds end` that packages a list ds of declarations, or a path $m_1 \dots m_n . i$ that selects an interface i from a module tree.

The declarations of a signature abstractly describe the bindings of a structure. A type declaration may omit the implementation of the type to create an abstract data type. A declaration of the form `val x : T` merely indicates the existence of a value x of type t . Similarly, a structure can only be declared to have a certain interface. In contrast, the `signature`, `open`, and `external` declarations are equivalent to the corresponding bindings.

To illustrate some of the module syntax, Figure 6.3 shows an example module `Cpx` that provides basic support for complex numbers. The structure implements complex numbers as pairs of `double` and defines the necessary constructor and destructor functions. The module is sealed by a signature that hides the implementation type of complex numbers such that values of type `Cpx.t` can only be manipulated by functions from `Cpx`.

6.3 Stateful Computations

QUBE expressions are referentially transparent, i. e., an expression will always evaluate to the same result when applied to the same arguments. While the absence of state and side-effects makes it easy to reason about QUBE expressions, additional measures are required to mimic the behaviour of imperative programs

T	$::=$	$T_C \mid *T_C \mid ?T_C \mid u \wedge T_C$	Types
T_C	$::=$	$B \mid \{x:T \mid e\} \mid \{[x:]T.\} [x:]T \rightarrow T$ $\mid (\{[x:]T, \}^+ T) \mid \text{intvec } e \mid [T \mid e\{, e\}] \mid X \{e\}$	Core Types
e	$::=$	$\dots \mid e!(e) \mid e!(e) \leftarrow e \mid e![\{e, \}e] \mid e![\{e, \}e] \leftarrow e$	Expressions

Figure 6.4: Syntax for uniqueness types and destructive array operations

that manipulate the machine state or interact with the execution environment. Similar to the programming languages CLEAN and SAC, QUBE uses *uniqueness types* [9] to incorporate side-effects into the referentially transparent setting. The implementation most closely resembles the approach described in [32].

A variable with a unique type must be referenced at most once. Unique objects are thus explicit representations of state. To manipulate such a state, a function must consume the corresponding unique object and produce a new unique object that represents the new state. The data dependencies between producers and consumers of unique objects then implicitly enforce the desired sequential evaluation of unique expressions.

Figure 6.4 shows the syntax extensions for uniqueness types. The types have been refined such that every type consists of a core type T_C and a uniqueness attribute. Core types are essentially the types as defined in Section 6.1. A type $*T_C$ is *unique*. The absence of a uniqueness attribute implicitly makes a type *shared*. A uniqueness variable u makes the type $u \wedge T_C$ polymorphic with respect to the uniqueness attribute. The type checker determines an appropriate instance of u depending on how the corresponding value is used. The notation $?T_C$ is a convenience notation for $u \wedge T_C$ where u is a fresh variable.

Each element of a data structure has its own uniqueness. In order to avoid illegal sharing of unique elements, the uniqueness propagation rule must be obeyed [84]: if a unique object is stored in a data structure, the data structure becomes unique as well.

To enable interactive programs, QUBE defines an abstract type `*world` whose values represent states of the execution environment. The function `main`, which serves as the entry point of a QUBE program, must have type `*world → *world`. The type captures the intuition that an interactive program transforms the execution environment from an initial to a final state. Using the library function `Io.println` of type `string. *world → *world`, we can write a Hello World program.

Example 6.5 (Hello World)

```
let main w:*world = Io.println "Hello World" w
```

```

let swap n:nat arr:[int|[n]] i:(index n) j:(index n) =
  let (arr,ai) = arr![[i]] in
  let (arr,aj) = arr![[j]] in
  let arr = arr![[i]] ← aj in
  let arr = arr![[j]] ← ai in
  arr

```

Figure 6.5: Destructive swapping of array elements

In the context of array programs, uniqueness types have a desirable second application. Since a unique array type describes an array that has at most one reference to it, modifications of this array may be performed destructively. Thus, the asymptotic complexity of array modification drops from $\mathcal{O}(n)$ to $\mathcal{O}(1)$.

QUBE provides a custom set of expressions to access and modify unique arrays. The unique vector selection $e!(e)$ and the unique array selection $e![e, \dots, e]$ take unique arrays and shared indices. In order to preserve the array reference, the expressions return a fresh reference to the array in addition to the selected element. The unique vector modification $e!(e) \leftarrow e$ and the unique array modification $e![e, \dots, e] \leftarrow e$ take unique arrays, shared indices, and shared substitute elements.

To illustrate destructive array modification, Figure 6.5 defines a function `swap` that takes a unique array and two valid array indices `i` and `j`. The operation uses unique array selections to select the elements at the positions `i` and `j`, respectively. Afterwards, the array is updated destructively and the final reference to the array is returned.

Summary

This chapter presented the syntax of the QUBE programming language. Based on `QUBECORE`, the language provides a number of features for practical array programming with dependent types. The module system of QUBE allows complex programs to be decomposed into smaller and more manageable modules that assemble related definitions. QUBE uses the module system to provide an extensive standard library of predefined functions. Via uniqueness types, QUBE supports stateful computations such as I/O and destructive array updates without losing referential transparency.

7

Language Implementation

QUBE is implemented by a compiler that translates QUBE programs into code for the Low-Level Virtual Machine (LLVM) which in turn emits machine-specific assembly language. The compiler was implemented by Florian Büther, Markus Weigel, Sebastian Hungerecker, and myself.

The remainder of this chapter is organized as follows: Section 7.1 characterises the design goals of the QUBE compiler. Section 7.2 gives an overview of the compiler architecture and describes the individual phases of the compilation process. The implementation dynamically represents multidimensional arrays as mere sequences of elements. Section 7.3 explains how the QUBE compiler uses the dependent array types to statically augment intermediate programs with information about array ranks and shapes wherever these are required for evaluation.

7.1 Design of the QUBE Compiler

The design of the QUBE compiler aims at translating QUBE programs into well-performing executables. A secondary goal was to explore how to use the rich type information to produce even better code.

Most visible to the user, the compiler uses the types to statically detect program errors such as array-boundary violations and give appropriate type error messages. Type checking is performed in collaboration with a Satisfiability Modulo

Theories (SMT) solver. Programs that pass the type checker are known not to exhibit program errors. In consequence, the compiler can avoid inserting dynamic checks such as array bounds checks into the program.

QUBE's type system is monomorphic. Therefore, all basic values have unboxed representations. Only closures, tuples, and arrays are allocated on the heap. Multidimensional arrays are represented solely by their linear data vectors. Instead of also providing the corresponding shape vectors at run-time, the compiler uses the shape expressions from the array types to statically annotate the program with shape information wherever necessary. The process will be described in more detail in Section 7.3.

Integer vectors have a pre-eminent role in array programming. They represent shapes and allow us to index elements from multidimensional arrays. Unfortunately, the overhead of allocating a vector on the heap just to select a single array element is prohibitive. To avoid integer vectors from manifesting at run-time, the QUBE compiler employs two strategies. First, whenever the length of an integer vector is statically known, the compiler represents the vector by its individual elements. Operations on these vectors can in most cases be unrolled, resulting in a number of scalar computations. Second, the compiler implements *index vector elimination* (IVE) [12], a technique developed in the context of the SAC compiler [42] that replaces linear operations on index vectors with equivalent operations on scalar offsets.

QUBE is a higher-order language that supports first-class functions and nested function declarations. Inner functions may refer to variables that are bound in the surrounding function. The QUBE compiler uses lambda lifting [60] to eliminate free variables from inner functions. For each free variable, the procedure introduces an additional function argument that must be passed at every application. The closed functions are moved to the top-level.

The implementation of curried function applications takes advantage of QUBE's n -ary function types. As the arity of any given function f is statically known, the compiler can determine whether an application $f e_1 .. e_n$ provides too few, too many, or the right number of arguments. If the number of arguments matches the function's arity, an exact call is made. If there are too many arguments, the compiler splits the application so that first an exact call is made whose result is then applied to the remaining arguments. If the application has too few arguments, the compiler generates a closure that combines a tuple (e_1, \dots, e_n) with a pointer to a new function that takes the tuple and the remaining arguments so that it can make an exact call to f .

Another important feature of functional programming languages is automatic memory management. The language implementation is responsible for allocating space for new data structures and releasing the memory once the data struc-

tures are no longer used. Currently, the QUBE compiler relies on the Boehm-Dehmers-Weiser conservative garbage collector [16] for storage reclamation. In the future however, we would like to replace it with a memory management scheme based on reference counting, since this would allow us to safely perform destructive array updates even in the context of immutable arrays [45, 94].

The QUBE compiler targets the Low-Level Virtual Machine (LLVM) [68], a compiler infrastructure that supports a variety of processor architectures. LLVM takes programs in a machine-independent typed assembly language and performs extensive compile-time optimisations. The result is either statically compiled to machine code or stored as bytecode for interpretation (aided by just-in-time compilation). A feature that makes compiling to LLVM particularly interesting is its support for tail-call optimisation (TCO) [62], a technique that substitutes function calls in tail position with more efficient jump instructions into the called functions.

As LLVM can use the C calling conventions, the generated object files can be linked with native libraries. Interfacing between QUBE and C programs is straightforward due to the similarities between the representations of QUBE data structures and the corresponding data structures in C. The interoperation works in both directions: QUBE programs can access library functions written in C and, vice versa, QUBE functions can serve as the trusted computational kernels of programs written in any other language that can interface with C.

The QUBE compiler itself is written in OCAML [69, 22], a popular language in the ML family of programming languages [70, 75]. OCAML offers inductive data types and facilities to define recursive functions over these types by pattern matching. In conjunction with type inference and automatic memory management, these features make OCAML well-suited for compiler construction.

7.2 Compilation at a Glance

This section describes the compilation process of the QUBE compiler. Fig. 7.1 gives an overview of the compilation. The blocks represent the individual compilation phases that lower the source program to a native object file `a.out` through a series of intermediate representations.

The compiler accepts a QUBE program as a source file suffixed with `.q`. Scanning and parsing of the program yields a *Parsetree*, an abstract syntax tree that closely resembles the source program.

Type checking is performed in collaboration with the YICES theorem prover [33]. The implementation of the proof relation $\Gamma \vdash_D e$ encodes the context and the

Compilation process

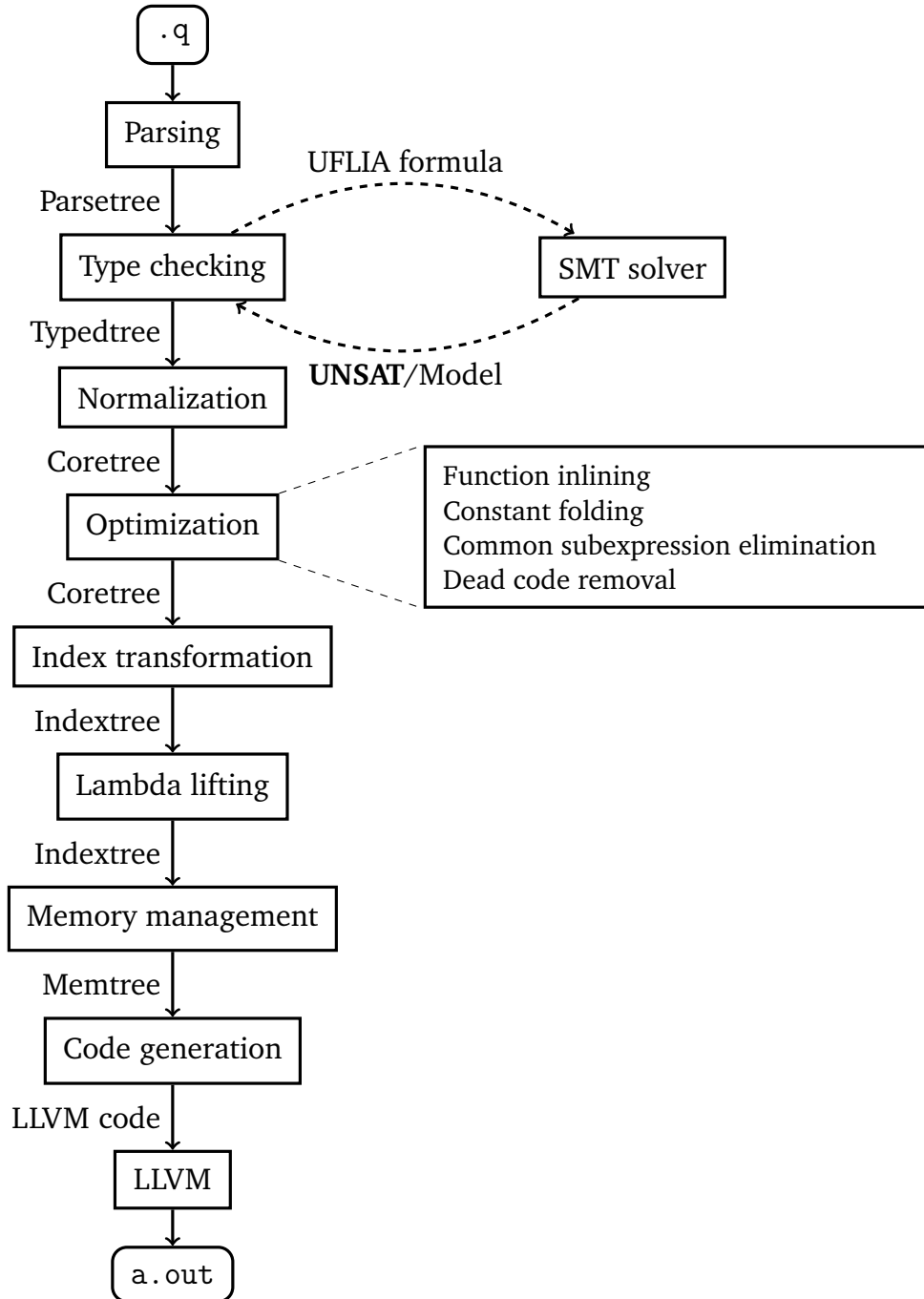


Figure 7.1: The compilation process of QUBE

negated property $\text{not } e$ as a logical formula in the decidable UFLIA¹ fragment of first-order logic as described in Section 5.6. When the SMT solver yields *unsatisfiable* then e is valid under Γ and thus the program is type correct. In case that the SMT solver finds the encoded formula satisfiable, it reports a model, i. e., an assignment of the program variables in Γ that satisfies the encoding of $\text{not } e$. Since e may thus be invalid under Γ , the program is rejected with an appropriate type error message.

Well-typed programs are emitted by the type checker in an intermediate representation called *Typedtree*. This abstract syntax tree still largely resembles the source program with a few notable differences. First, all user-defined types are replaced by their definitions. Second, all bound identifiers are uniquely renamed. Third, to admit descriptor-free representation, all language elements that deal with vectors or arrays are annotated with expressions that represent the vector lengths, array rank and shapes, respectively. The descriptor-free array representation and the required compile-time annotations will be described in more detail in section 7.3.

The *Normalisation* phase systematically eliminates complex subexpressions from the syntax tree, yielding the intermediate representation *Coretree*. Similar to administrative normal form [37], *Coretree* only allows functions to be applied to trivial arguments, called atoms, which are either constants or identifiers. More complex expressions must only appear on the right-hand side of a let-binding. Thus, complex subexpressions are abstracted out of their original context and bound to fresh variables.

Due to its regular structure, the *Coretree* representation is well-suited for *program optimisation*. The side-effect free nature of (well-typed) QUBE programs allows to straightforwardly apply a host of compiler optimisations without changing the program semantics. Currently, the QUBE compiler performs the following high-level program optimisations most of which are well-known from compiler construction text books [1, 3, 47]:

- **Function inlining:** non-recursive functions whose size do not exceed a certain threshold are inlined.
- **Partial evaluation:** the compiler reduces applications of the built-in logic and arithmetic operations to constant arguments. Conditionals with constant predicates are reduced to the adequate branch. Unpack expressions are reduced when the definition of the tuple is known. For vectors and arrays that are defined by a constructor, selections and modifications with constant index vectors and reshape operations with constant shape vectors are reduced. For vectors of known length, instances of `vmap` and `vfa`

¹uninterpreted functions and linear integer arithmetic

are expanded. Similarly, instances of `gen` and `loop` are expanded for small shapes.

- **Common subexpression elimination:** identical right-hand sides of let-bindings are eliminated to avoid redundant computations.
- **Dead code removal:** the compiler eliminates all bindings of unreferenced variables as these cannot contribute to the program result.

After program optimisation, the compiler phase *index transformation* lowers the multidimensional QUBE arrays to linear arrays that can be mapped to actual computing hardware. In particular, index vectors into multidimensional arrays are transformed into scalar offsets into the linear array representation. By means of *index vector elimination* [12], the compiler converts many linear operations on heap-allocated integer vectors into equivalent scalar operations that can be performed in processor registers. The compiler phase maps the syntax tree to a representation called *Indextree* that dispenses with dependent types in favour of simple types.

The next phase *Lambda lifting* [60] eliminates nested function definitions. The procedure augments inner functions with additional arguments for each free variable and modifies all applications of these functions to pass the necessary value. The closed functions are then moved out of their original context to the top-level.

The *Memory management* phase augments the intermediate code with instructions that explicitly allocate and modify memory. In effect, the purely functional QUBE programs are transformed into stateful programs that perform computations as a sequence of modifications of the machine state.

As the final stage of the QUBE compiler, *code generation* transforms the syntax tree into code for the Low-Level Virtual Machine (LLVM) [68] which in turn emits machine-specific object code.

7.3 Descriptor-Free Array Representation

Array ranks and shape vectors are essential for the evaluation of array programs: for a matrix `a: [int | [m,n]]` stored in row-major order, the element `a. [[i,j]]` is located in the linear memory representation of `a` at the offset $i \cdot n + j$. To provide rank and shape vector wherever necessary, language implementations typically associate each array with an array descriptor that is maintained at run-time in addition to the array itself. A descriptor-based array representation of `a` takes the form of a tuple `<2, [m,n], [d_1, ..., d_p]>` where the first component

2 is the array rank, the second component $[m, n]$ is the shape vector, and the third component $[d_1, \dots, d_p]$ is the data vector whose length p equals the product of the shape vector elements. To the programmer, these arrays appear as abstract data types that provide means for obtaining the array shape and (safely) accessing the individual elements.

In the context of QUBE, a more compact representation of arrays can be employed [97]. The type of an array captures both its element type and its shape as a structured vector. The QUBE compiler uses this information to statically annotate the program with rank and shape information wherever these are required for evaluation. The above selection thus becomes $a. [([m, n], [i, j]) \langle 2 \rangle]$, removing the need for dynamically looking up the shape of a . The language implementation can thus dispense with array descriptors and instead represent arrays as mere sequences of elements.

Once all structural properties of arrays are represented explicitly in intermediate code, they become subject to program optimisation. For example, dead code removal eliminates superfluous computations of and on structural properties. Likewise, common subexpression elimination avoids repeated computation of identical shape information. Last not least, constant folding and algebraic simplification contribute their share to optimise computations on rank and shape expressions.

Fig. 7.2 shows the rules of the relation $e \uparrow e$ which statically annotates programs with the required array properties. Program annotation takes place during type checking where the type of each expressions is determined. For the sake of clarity, we avoid reformulating the entire set of typing rules from Chapter 5 and omit the typing context Γ .

Each shape segment e_i of an array type $[T | e_1, \dots, e_n]$ is annotated with its length l_i . The array type becomes $[T | e_1 \langle l_1 \rangle, \dots, e_n \langle l_n \rangle]$.

The vector expressions $\text{vfa } \bar{e} (\bar{x} \rightarrow e)$ and $\text{vmap } \bar{e} (\bar{x} \rightarrow e)$ implicitly loop over all input vectors \bar{e} simultaneously to compute the result. Since type checking ensures that all vectors have the same length l , it is sufficient to annotate the expressions with l once, yielding $\text{vfa} \langle l \rangle \bar{e} (\bar{x} \rightarrow e)$ and $\text{vmap} \langle l \rangle \bar{e} (\bar{x} \rightarrow e)$.

Array selections $e_a. [\bar{e}]$ are annotated with the result of joining the structured shape vector of e_a and the structured index vector \bar{e} . Both vectors are required to compute the offset into the linear representation of e_a . As the join operation ensures that corresponding shape and index segments have the same length, the offset computation can be carried out without dynamically aligning the structured vectors. Array modifications $e_a. [\bar{e}] \leftarrow e_e$ are annotated in the same way.

Evaluation of a reshape expression $\text{reshape } \bar{e} e_a$ involves computing the number of elements of e_a as well as the number of elements in the result which is given

$$\begin{array}{c}
\frac{(e_i : \text{intvec } l_i)_i}{[T | e_1, \dots, e_n] \uparrow [T | e_1 \langle l_1 \rangle, \dots, e_n \langle l_n \rangle]} \text{ (A-TARRAY)} \\
\\
\frac{(e_i : \text{intvec } l_i)_i}{\text{vfa } e_1, \dots, e_n f \uparrow \text{vfa} \langle l \rangle e_1, \dots, e_n f} \text{ (A-VFA)} \\
\\
\frac{(e_i : \text{intvec } l_i)_i}{\text{vmap } e_1, \dots, e_n f \uparrow \text{vmap} \langle l \rangle e_1, \dots, e_n f} \text{ (A-VMAP)} \\
\\
\frac{(e_i : \text{intvec } l_i)_i \quad e_a : [T | e'_1 \langle l'_1 \rangle, \dots, e'_m \langle l'_m \rangle] \quad e'_1, \dots, e'_m \bowtie e_1, \dots, e_n = (s_1, v_1) \langle r_1 \rangle, \dots, (s_o, v_o) \langle r_o \rangle}{e_a. [e_1, \dots, e_n] \uparrow e_a. [(s_1, v_1) \langle r_1 \rangle, \dots, (s_o, v_o) \langle r_o \rangle]} \text{ (A-SEL)} \\
\\
\frac{(e_i : \text{intvec } l_i)_i \quad e_a : [T | e'_1 \langle l'_1 \rangle, \dots, e'_m \langle l'_m \rangle] \quad e'_1, \dots, e'_m \bowtie e_1, \dots, e_n = (s_1, v_1) \langle r_1 \rangle, \dots, (s_o, v_o) \langle r_o \rangle}{e_a. [e_1, \dots, e_n] \leftarrow e_e \uparrow e_a. [(s_1, v_1) \langle r_1 \rangle, \dots, (s_o, v_o) \langle r_o \rangle] \leftarrow e_e} \text{ (A-MOD)} \\
\\
\frac{(e_i : \text{intvec } l_i)_i \quad e_a : [T | e'_1 \langle l'_1 \rangle, \dots, e'_m \langle l'_m \rangle]}{\text{reshape } e_1, \dots, e_n e_a \uparrow \text{reshape} \langle e'_1 \langle l'_1 \rangle, \dots, e'_m \langle l'_m \rangle \rangle e_1 \langle l_1 \rangle, \dots, e_n \langle l_n \rangle e_a} \text{ (A-RSHP)} \\
\\
\frac{(e_i : \text{intvec } l_i)_i}{\text{gen } e_1, \dots, e_n \text{ with } g \uparrow \text{gen } e_1 \langle l_1 \rangle, \dots, e_n \langle l_n \rangle \text{ with } g} \text{ (A-GEN)} \\
\\
\frac{(e_i : \text{intvec } l_i)_i}{\text{loop } x_a = e_a ; e_1, \dots, e_n \text{ with } g \uparrow \text{loop } x_a = e_a ; e_1 \langle l_1 \rangle, \dots, e_n \langle l_n \rangle \text{ with } g} \text{ (A-LOOP)}
\end{array}$$

Figure 7.2: Static annotation of the required array properties

by the product of the elements from \bar{e} . Thus, we annotate the expression with the entire shape of e_a and the length of each vector e_i .

The shape vectors of the array comprehension `gen` and the `loop` expression are also annotated with their respective lengths, since these are required to determine the lengths of the corresponding index vectors and for computing the number of elements in the index space.

The idea of descriptor-free array representation is related to the compilation scheme devised for rank-generic SAC programs [65]. The SAC compiler uses different representations for arrays of statically known shape, arrays of statically known rank (but unknown shape), and arrays of statically unknown rank and shape to avoid creating run-time descriptors whenever possible. Thus the efficiency of compiled code depends on the available shape information. To improve the amount of static shape information, the SAC compiler uses a combination of partial evaluation and function specialisation [43]. Recent work [98] proposed *symbolic array attributes* as a uniform scheme to infer and represent structural information in rank-generic array programs such that it may be used by optimisations. Descriptor-free array representation takes all these ideas to their conclusion.

Summary

This chapter presented the implementation of the QUBE programming language. The QUBE compiler translates programs to LLVM programs through a series of intermediate representations. Most of the program transformations involved are similar to those employed by compilers for other strict, higher-order functional programming languages, with a few notable exceptions.

Type checking is performed in collaboration with an SMT solver to soundly decide whether two expressions denote the same value and, in extension, whether two types are equal.

Dependent array types represent array ranks and shape vectors in the type system. The QUBE compiler uses this information to statically annotate intermediate programs with array properties wherever these are required for evaluation. This allows the program to dispense with dynamic representations of ranks and shape vectors. Instead, arrays are stored as mere sequences of elements. Furthermore, exposing rank and shape computations explicitly in intermediate code makes them subject to compile time optimisation. In effect, the compiled programs contain very little overhead.

QUBE's n -ary function types provide static information about the arity of a (potentially unknown) function. The compiler uses this information to determine

whether an application is partial, exact, or an over application of a function. For each situation, appropriate code that does not require dynamic branching is generated. In contrast, compilers for languages such as HASKELL, SML, or OCAML that only have unary function types (despite allowing to define n -ary functions) must deal with function application dynamically by either of two basic approaches [71]. In the *push/enter* model, the caller pushes all arguments to the stack. The called function, which knows its own arity, inspects the stack to determine how many arguments it has been passed. If there are too few arguments, the function must return an object representing a partial application, if there are too many arguments, some of them must be left on the stack for later consumption. In the *eval/apply* model, the caller, which statically knows the number of arguments, evaluates the function and inspects the resulting closure to determine its arity. By case distinction, the caller chooses appropriate code that either extends the closure, makes an exact call, or makes an exact call followed by another application.

8

Rank-Generic Array Operations

Traditional array languages such as APL [55], J [56], and MATLAB [76] provide a large number of built-in array operations from which more complex programs are assembled. In contrast, QUBE allows the user to define her own set of type-safe array operations by means of the versatile `gen` and `loop` expressions.

This chapter introduces a number of essential rank-generic array operations. The assortment of operations is inspired by those available in SAC and the interpreted array languages mentioned above. As the current implementation of QUBE does not support polymorphism, we define all operations for arrays of integers. They may be instantiated for other types as deemed necessary.

The remainder of the chapter is organised as follows: Section 8.1 defines some common type abbreviations that will be used throughout the chapter. Section 8.2 illustrates element-wise array operations that apply a function to all elements of an array or pairs of corresponding elements from two arrays. Section 8.3 defines selection functions that select larger chunks of array elements instead of a single element. Section 8.4 presents structural array operations that affect the organisation of elements inside an array. Finally, Section 8.5 presents powerful higher-order array functions.

8.1 Type Abbreviations

The program shown in Example 8.1 defines several type abbreviations that will be used throughout the remainder of the chapter.

The type `nat` describes non-negative integers. `natvec n` is the type of vectors of length `n` that only contain non-negative integers. The type `index n` describes valid indices into a vector of length `n`. `indexvec r s` is the type of valid index vectors into an array of rank `r` and shape `s`.

The types `nat_le b` and `natvec_le n b` are the types of naturals and vectors of naturals of length `n`, respectively, that do not exceed the boundary (vector) `b`. Conversely, the types `int_ge b` and `intvec_ge n b` describe integers and integers vectors of length `n` whose elements are greater or equal to the boundary (vector) `b`.

Example 8.1 (Type abbreviations)

```

type nat = { v:int | 0 <= v }
type natvec n:nat = { v:intvec n | vfa v (vi → 0 <= vi) }

type index n:nat = { v:int | 0 <= v & v < n }
type indexvec r:nat s:(natvec r) =
  { v:intvec r | vfa v,s (vi,si → 0 <= vi & vi < si) }

type nat_le b:int = { v:int | 0 <= v & v <= b }
type natvec_le n:nat b:(intvec n) =
  { v:intvec r | vfa v,b (vi,bi → 0 <= bi & vi <= bi) }

type int_ge b:int = { v:int | b <= v }
type intvec_ge n:nat b:(intvec n) =
  { v:intvec n | vfa b,v (bi,vi → bi <= vi) }

```

8.2 Element-Wise Computations

An important class of array operations are element-wise operations that apply a function to each element of an array or to pairs of corresponding elements from two arrays of equal shape. In this section, we develop the rank-generic addition operation `add` by incrementally generalising a shape-specific implementation.

We start with a shape-specific function that adds matrices of shape `[2,2]` by selecting and adding the corresponding elements. The type system ensures that

`add` can only be applied to matrices of the required shape. As the new shape also has shape `[2,2]`, all array accesses are known to be correct.

Example 8.2 (*Shape-specific Add*)

```
let add a:[int|[2,2]] b:[int|[2,2]] =
  [ a.[[0,0]] + b.[[0,0]], a.[[0,1]] + b.[[0,1]],
    a.[[1,0]] + b.[[1,0]], a.[[1,1]] + b.[[1,1]] ] | [2,2]
```

Using dependent types, we can generalise `add` such that it becomes applicable to arbitrary matrices of shape `[m,n]`. For this purpose, we abstract the shape components `m:nat` and `n:nat` from the shape of the array types. Furthermore, we replace the array constructor with an array comprehension that creates an array of shape `[m,n]` and at each index `x` adds the corresponding elements from `a` and `b`.

Example 8.3 (*Shape-generic Add*)

```
let add m:nat n:nat a:[int|[m,n]] b:[int|[m,n]] =
  gen [m,n] with x → a.[x] + b.[x]
```

Even more general, by specifying the shape as a vector `s:(natvec r)` where `r:nat`, we obtain a variant of `add` that is applicable to any two integer arrays of shape `s`, no matter whether these are vectors, matrices, or anything else. It is noteworthy that this generalisation does not require to change the function body of `add` apart from the shape.

Example 8.4 (*Rank-generic Add*)

```
let add r:nat s:(natvec r) a:[int|s] b:[int|s] =
  gen s with x → a.[x] + b.[x]
```

Other element-wise operations, like element-wise negation, subtraction or multiplication can be defined in the same way.

A slight variation of this scheme is the function `sum` that adds all elements of an array. The rank-generic operation is implemented by means of a loop expression that, starting with the initial value `0`, iterates over the array indices and successively adds all array elements to the intermediate result.

Example 8.5 (*Sum*)

```
let sum r:nat s:(natvec r) a:[int|s] =
  loop acc:int = 0; s with x → acc + a.[x]
```

8.3 Selection Functions

The array selection `a.[x]` selects a single element from an array `a`. In many situations it is desirable to simultaneously select larger chunks of data. With the help of `gen`, many useful and type-safe generalisations of scalar selection can be specified.

The generalised selection `gsel` and its dual `fsel` perform rank-generic array slicing. Given an array `a` whose shape `fs,cs` consists of the *frame shape* `fs` and the *cell shape* `cs`, and an index vector `x` that indexes into the frame shape, `gsel` selects the slice of elements whose position in `a` is prefixed with `x`. Similarly, given the array `a` and a vector `y` that indexes into the cell shape, the operation `fsel` selects the slice of element whose position in `a` is suffixed with `y`.

Example 8.6 (*Gsel and fsel: array slicing*)

```
let gsel fr:nat fs:(natvec fr) cr:nat cs:(natvec cr)
    x:(indexvec fr fs) a:[int|fs,cs] =
  gen cs with y → a.[x,y]

let fsel fr:nat fs:(natvec fr) cr:nat cs:(natvec cr)
    y:(indexvec cr cs) a:[int|fs,cs] =
  gen fs with x → a.[x,y]
```

The function `take` selects an interval of items from an array. The operation takes an array `a` of some shape `s` and a vector `t` whose magnitude defines the extent of the interval along each axis of `a`. If `t.(i)` is positive, the interval starts at the beginning of axis `i`. Otherwise, the interval ends at the tail of the axis. The dual function `drop` takes an array `a` and a vector `d`. Depending on the sign of `d.(i)`, the operation discards the leading or the trailing `|d.(i)|` items along the axis `i`.

Example 8.7 (*Take and drop*)

```
let take r:nat s:(natvec r) a:[int|s]
    t:{ v:intvec r | vfa s,v (si,vi → -si<=vi & vi<=si) } =
  let off = vmap s,t (si,ti → if ti < 0 then si+ti else 0) in
  gen vmap t (ti → if ti < 0 then -ti else ti) with x →
  a.[vmap x,off (xi,oi → xi+oi)]

let drop r:nat s:(natvec r) a:[int|s]
    d:{ v:intvec r | vfa s,v (si,vi → -si<=vi & vi<=si) } =
  let off = vmap d (di → if di<0 then 0 else di) in
  gen vmap s,d (si,di → if di<0 then si+di else si-di) with x →
  a.[vmap x,off (xi,oi → xi+oi)]
```

Using `take` and `drop`, we can conveniently define the function `tile` that, starting at a position `p`, selects a block of elements of size `ts` from an array `a`. To achieve the desired effect, the definition first drops `p` items from `a` and then takes `ts` items from the result.

Example 8.8 (Tile)

```
let tile r:nat s:(natvec r) a:[int|s]
    ts:(natvec_le r s)
    p:(natvec_le r (vmap s,ts (si,tsi → si-tsi))) =
  take r (vmap s,p (si,pi → si-pi)) ts (drop r s p a)
```

The indirect selection `mselect` simultaneously performs multiple independent selections. The function takes an array `a` and an array `idxs` of index vectors into `a`. For every vector in `idxs`, `mselect` performs a selection into `a` and yields the array of results. It is worth noting that the arrays `a` and `idxs` may have different ranks and shapes as long as all elements of `idxs` are appropriate index vectors into `a`. The index generator `iota` computes the set of all possible index vectors into an array of a given shape `s`. For an array `a` of rank `r` and shape `s`, `mselect r s r s (iota r s) a` is equivalent to `a`.

Example 8.9 (Indirect selection and the index generator)

```
let mselect di:nat si:(natvec di) r:nat s:(natvec r)
    idxs:[indexvec r s | si] a:[int|s] =
  gen si with x → let y = idxs.[x] in a.[y]

let iota r:nat s:(natvec r) =
  gen s with x → (x : indexvec r s)
```

8.4 Structural Functions

This section defines functions that affect the structure of arrays, i. e. the organization of elements inside the array.

The operation `reverse` takes an array and reverses the order of its items by subtracting the current index from the maximal array index.

Example 8.10 (Reverse)

```
let reverse r:nat s:(natvec r) a:[int|s] =
  gen s with x → a.[vmap s,x (si,xi → si-xi-1)]
```

The function `condense` drops every second item along each axis of a given array, thereby halving the extent of each axis. The dual function `scatter` creates an array whose extent in each axis is twice that of the corresponding axis from the given array `a`. The elements of `a` are copied into every even index position of the result array, the other elements initialised with a specified default element `def`.

Example 8.11 (*Condense and scatter*)

```
let condense r:nat s:(natvec r) a:[int|s] =
  gen vmap s (si → si/2) with x → a.[vmap x (xi → 2 * xi)]

let scatter r:nat s:(natvec r) def:int a:[int|s]=
  gen vmap s (si → 2*si) with x →
    if vfa x (xi → xi % 2 = 0)
    then a.[vmap x (xi → xi / 2)]
    else def
```

Variants of `condense` or `scatter` for other constant strides can be specified when required. However, generalising either function to some abstract stride `str` requires that we perform an array bounds check for every selection. The reason is that neither the multiplication of `x` by `str` nor the division of `x` by `str` is expressible in linear integer arithmetic and thus the compiler cannot statically verify the access into `a`.

The operation `cat` (*catenate*) joins two arrays `a` and `b` by appending the array `b` at the end of the outermost axis of array `a`. Therefore, the shapes of both arrays must have a common suffix `s` of length `r`.

Example 8.12 (*Catenate*)

```
let cat m:nat n:nat r:nat s:(natvec r)
  a:[int|[m],s] b:[int|[n],s] =
  gen [m+n],s with
  | @[0]..[m] as [i],j → a.[[i],j]
  | [i],j → b.[[i-m],j]
```

The function `embed` allows us to embed an array into a larger array of the same rank and is thus complementary to `tile`. The operation takes an array `a`, a new shape `ns` that must be at least as large as the shape of `a`, an outer array `o` of shape `ns` and an index position `p`. It creates an array of shape `ns` into which the elements of `a` are copied starting from the index `p`. The other elements are initialised with the corresponding elements from `o`.

Example 8.13 (*Embed*)

```

let embed r:nat s:(natvec r) a:[int|s] ns:(intvec_ge r s)
      o:[int|ns] p:(natvec_le r (vmap ns,s (ni,si→ni-si))) =
  let ub = vmap p,s (pi,si → pi+si) in
  gen ns with
  | @p..ub as x → a.[vmap x,p (xi,pi → xi-pi)]
  | x → o.[x]

```

The function `shift` moves the elements of a given array `a` as specified by the offset vector `off`, whose length must equal the rank of `a`. A positive entry in `off` means that the elements are shifted to the left (i. e. towards the beginning) of the corresponding axis. Vice versa, the elements are shifted to the right if the entry in `off` is negative. The emerging vacancies are initialised with the default element `def`.

The type of `off` statically ensures that not all elements can be shifted out of the array. In case this functionality is required, a wrapper function can be written that either shifts the elements or yields a new array entirely filled with `def`.

Example 8.14 (*Shift*)

```

let shift r:nat s:(natvec r) a:[int|s] def:int
      off:{ v:intvec r | vfa s,v (si,vi → -si<vi & vi<si) } =
  let lb = vmap off (oi → if oi < 0 then -oi else 0) in
  let ub = vmap s,off (si,oi → if oi < 0 then si else si-oi) in
  gen s with
  | @lb..ub as x → a.[vmap x,off (xi,oi → xi+oi)]
  | x → def

```

Similar to `shift` is the function `rotate`. As indicated by its name, the function rotates each axis of a given array by as many elements to the left as are specified by corresponding element of the rotation vector `rot`. The elements of `rot` must range between 0 and the length of the axis. To achieve unbounded rotation, the rotation vector would need to be pre-processed, for example, by element-wise computation of `rot % s`.

The implementation realises multidimensional rotation as a sequence of rotations along the individual axes. For each axis `ax`, the items at positions below the boundary `b = s.(ax)-rot.(ax)` are obtained by adding `rot.(ax)` to the current index. The items beyond `b` are obtained by subtracting `b` from the current index.

Example 8.15 (Rotate)

```

let rotate r:nat s:(natvec r) a:[int|s] rot:(indexvec r s) =
  let null = vec r 0 in
  loop a = a; [r] with [ax] →
    let off = null.(ax) ← rot.(ax) in
    let bound = null.(ax) ← (s.(ax) - rot.(ax)) in
    gen s with
      | @bound..s as x → a.[vmap x,bound (xi,bi → xi-bi)]
      | y → a.[vmap y,off (xi,oi → xi+oi)]

```

8.5 Higher-Order Functions

A powerful feature of functional programming is the ability to define higher-order functions, or functionals. These take functions as arguments or have a function as a result. Higher-order functions are useful as they allow us to encapsulate programming patterns that repeat across different functions.

The higher-order functions `map` and `map2` capture the pattern underlying element-wise operations such as `add` that apply a function to every element of an array and return the array of results. `map` takes a unary function `f` and applies it to each element of an array. Similarly, `map2` applies a binary operation to corresponding elements of two arrays.

Example 8.16 (Map and Map2)

```

let map f:(int → int) r:nat s:(natvec r) a:[int|s] =
  gen s with x → f a.[x]

let map2 f:(int. int → int) r:nat s:(natvec r)
  a:[int|s] b:[int|s] =
  gen s with x → f a.[x] b.[x]

```

By partial application of `map2`, we obtain an alternative rank-generic definition of element-wise addition.

```
let add = map2 (fun x:int y:int → x+y)
```

In the same style, the construction principle underlying functions such as `sum` can be condensed into a higher-order function. The operation `fold_left` uses a loop to traverse an index space in ascending lexicographic order. Starting with an abstract identity element `id`, the function uses an abstract binary operation `f` to combine the intermediate result `acc` with the current loop element. Vice versa, `fold_right` traverses the index space in descending lexicographic order.

Example 8.17 (Fold)

```

let fold_left f:(int. int → int) id:int
    r:nat s:(natvec r) a:[int|s] =
  loop acc : int = id; s with x → f acc a.[x]

let fold_right f:(int. int → int) id:int
    r:nat s:(natvec r) a:[int|s] =
  loop acc : int = id; s with x →
    f a.[vmap s,x (si,xi → si-xi-1)] acc

```

We use `fold_left` to simplify the definition of `sum` and to provide a function `prod` that computes the product of all elements of an array.

```

let sum = fold_left (fun x:int y:int → x+y) 0
let prod = fold_left (fun x:int y:int → x*y) 1

```

The inner product `ip` is a generalisation of the matrix multiplication. Instead of restricting its arguments to (suitable) matrices, `ip` allows the arguments to have arbitrary shapes and an arbitrary number of axes as long as the last axis of the first argument is as long as the first axis of the second argument. The function combines from each vector along the last axis (rows) of the first array with each vector along the first axis (columns) of the second array by applying the abstract function `g` to corresponding elements. Each result vector is reduced using the function `f` with identity `id`.

Example 8.18 (Inner product)

```

let ip f:(int. int → int) id:int g:(int. int → int)
    m:nat n:nat r:(natvec m) s:nat t:(natvec n)
    a:[int|r,[s]] b:[int|[s],t] =
  gen r,t with i,j →
    loop sum :int = id; [s] with [k] → sum 'f'
      a.[i,[k]] 'g' b.[[k],j]

```

We can specialise `ip` to the common matrix multiplication by appropriately instantiating the shape vectors and the functions `f` as addition with the identity element 0 and `g` as multiplication, respectively.

```

let matmul r:nat s:nat t:nat =
  let add x:int y:int = x+y in
  let mul x:int y:int = x*y in
  ip add 0 mul 1 1 [r] s [t]

```

Similar to `ip`, the outer product `op` combines each element from an array `a` of shape `s` with each element from array `b` of shape `t` by applying the abstract

function g , yielding an integer array of shape s, t .

Example 8.19 (Outer product)

```
let op g:(int. int → int)
    m:nat n:nat s:(natvec m) t:(natvec n)
    a:[int|s] b:[int|t] =
  gen s,t with x,y → a.[x] 'g' b.[y]
```

Summary

This chapter presented how rank-generic operations that are usually provided as built-in functions by array programming systems can be specified in QUBE itself. By virtue of dependent types, these functions are automatically verified by the compiler and can thus be trusted even in safety-critical applications. Moreover, the argument types precisely document the sets of allowed function parameters.

9

Evaluation

This chapter evaluates the suitability of QUBE for practical program development by considering a number of example programs. The programs chosen are all well-known and nicely demonstrate QUBE language features. Ideally, the evaluation provides insights that motivate potential future extensions of the QUBE language and compiler.

All measurements were performed on an Apple iMac with a 2.8 GHz Core 2 Duo processor and 4 GB RAM. The QUBE compiler uses LLVM version 2.8 as the backend compiler. The reference C programs were compiled with GCC version 4.2.1.

The remainder of this chapter is structured as follows: Section 9.1 and Section 9.2 compare the efficiency of rank-generic implementations of the inner product and the convolution with C implementations of the algorithms. Section 9.3 investigates a QUBE implementation of Quicksort that makes use of destructive array updates to achieve better run-time performance.

9.1 Matrix Multiplication and Inner Product

Matrix multiplication is an important operation in linear algebra. The inner product generalises matrix multiplication to arrays of arbitrary rank. As a first micro-benchmark, this section compares the rank-generic inner product with a naive (but reasonable) implementation of matrix multiplication in C.

The inner product `ip` shown in Example 9.1 takes two arrays `a` and `b`, where the last axis of `a` has to be as long as the first axis of `b`. The type of `ip` precisely documents this constraint: for two vectors `r` and `t` and a natural number `s`, the shape of `a` must be `r, [s]`, and the shape of `b` has to be `[s], t`. The result has shape `r, t`.

The body of `ip` consists of a `gen` expression that creates a new array where the element at position `i, j` is the scalar product of row `i` from `a` and column `j` from `b` that is computed by an appropriate loop.

Example 9.1 (Inner product (generalised matrix multiplication))

```
let ip m:nat n:nat r:(natvec m) s:nat t:(natvec n)
    a:[double|r,[s]] b:[double|[s],t] :[double|r,t] =
  gen r,t with i,j →
    loop sum:double = 0.; [s] with [k] →
      sum + (a.[i,[k]] * b.[[k],j])
```

When applied to arrays of some specific rank, the QUBE compiler will inline the body of `ip` and create a rank-specific instance of the `gen` expression that computes the result.

Example 9.2 shows a straightforward C implementation of matrix multiplication that serves as a reference for comparing the run-time performance of `ip`. Unlike the QUBE program, the types of the function arguments do not communicate the shape constraints. Memory management and index computations must be performed explicitly.

Example 9.2 (Matrix multiplication in C)

```
double *matmul(int m, int n, int o, double *a, double *b) {
  int i,j,k;
  double *c = malloc(m*o*sizeof(double));
  for (i = 0; i < m; i++) {
    for (j = 0; j < o; j++) {
      double sum = 0.0;
      for (k = 0; k < n; k++) {
        sum += a[i*n+k] * b[k*o+j];
      }
      c[i*o+j] = sum;
    }
  }
  return(c);
}
```

Figure 9.1 compares the run-times of the rank-generic inner product, a rank-specific instance of the inner product and the C matrix multiplication when applied to arrays of shape $[1024, 1024]$.

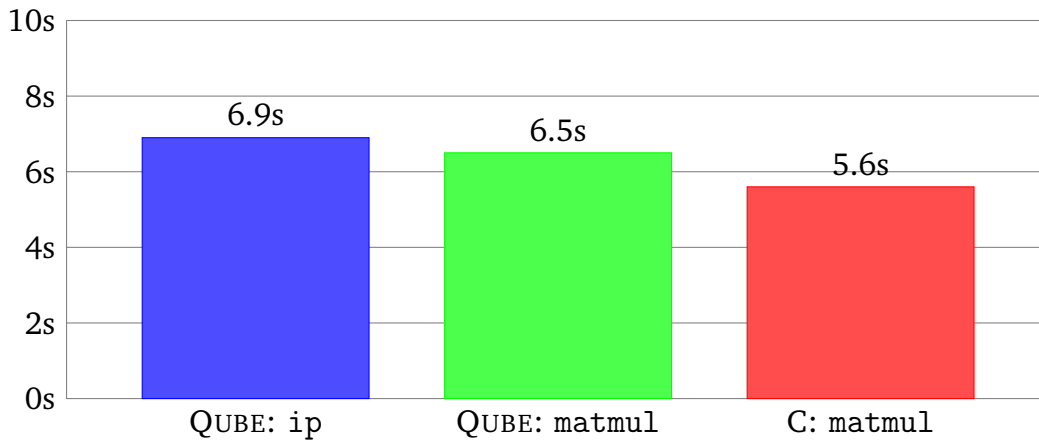


Figure 9.1: Run-times of inner product in QUBE, matrix multiplication in QUBE, and matrix multiplication in C for arrays of shape $[1024, 1024]$.

All programs perform roughly equally well. In particular, the rank-generic instance performs so well because the compiler’s index vector elimination can replace the offset computations $\iota r i$ and $\iota t j$ with scalar indices such that neither of the index vectors i and j manifests at run-time. The rank-specific QUBE program performs somewhat better because the function is inlined into the applying context where all shapes are constants. Somewhat mysteriously, the C program performs even better. We attribute these improvements to differences between the gcc and LLVM optimisation and code generation schemes.

9.2 Rank-Generic Convolution

As a second micro-benchmark, this section investigates array convolution. Rank-generic convolution is a straightforward extension of the well-known convolution on time-discrete 1D signals f and g of length m and $n \leq m$, respectively.

$$\left[(f * g)(i) = \sum_{j=0}^m (f(i+j) \cdot g(m-j)) \right]_{i=0}^{m-n+1}$$

Example 9.3 shows a rank-generic convolution function written in QUBE. For any rank r , the function takes a rank r array f of some shape f_s and another rank r

array g of some other shape gs that must not exceed fs . First, g is reverted by means of the function `reverse` defined in Section 8.4. Then, a `gen` expression moves the resulting array g' over all subarrays of f of shape gs . At each position, a loop computes the sum of the element-wise product of g' and the underlying subarray.

Example 9.3 (Rank-generic convolution)

```
let convolve r:nat fs:(natvec r) f:[double|fs]
      gs:(natvec_le r fs) g:[double|gs] =
  let g' = reverse r gs g in
  gen vmap fs,gs (fsi,gsi → fsi-gsi+1) with x →
    loop sum:double = 0.; gs with y →
      sum + f.[vmap x,y (xi,yi → xi+yi)] * g'.[y]
```

Again, we use a rank-specific C implementation of the algorithm as a reference to evaluate the efficiency of `convolve`. Example 9.4 shows the corresponding C program that computes a 2D convolution. With four nested `for`-loops and extensive index computations, the program is substantially more complicated than the QUBE program.

Example 9.4 (2D Convolution in C)

```
double *convolve( int fm, int fn, double *f,
                 int gm, int gn, double *g) {
  int ri,rj,gi,gj;
  double *gr = reverse(gm,gn,g);
  int rm = fm-gm+1;
  int rn = fn-gn+1;
  double *r = malloc(rm*rn*sizeof(double));
  for (ri = 0; ri < rm; ri++) {
    for (rj = 0; rj < rn; rj++) {
      double sum = 0.0;
      for (gi = 0; gi < gm; gi++) {
        for (gj = 0; gj < gn; gj++) {
          sum += f[(ri+gi)*fn+rj+gj] * gr[gi*gn+gj];
        }
      }
      r[ri*rn+rj] = sum;
    }
  }
  free(gr);
  return(r);
}
```

Figure 9.2 compares the run-times of the rank-generic convolution, a rank-specific convolution in QUBE, and the C program. All programs have been applied to an array of shape $[10240, 10240]$ and a $[3, 3]$ filter kernel.

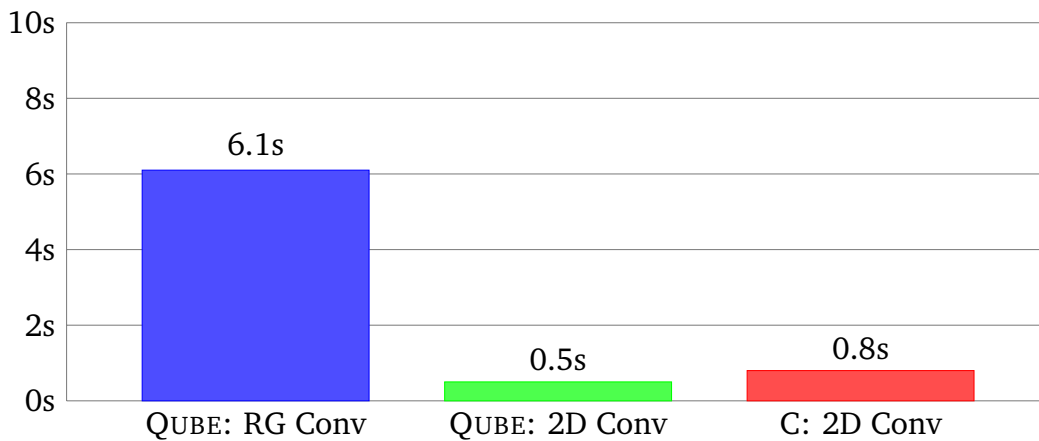


Figure 9.2: Run-times of rank-generic convolution in QUBE, 2D convolution in QUBE, and 2D convolution in C for arrays of shape $[10240, 10240]$ and $[3, 3]$.

Almost eight times slower than the reference program, the rank-generic convolution turns out to be a stress-test for the language implementation for three reasons. First, since r is unknown, the `gen` and `loop` expressions cannot be mapped to finite nestings of `for`-loops. Instead, both expressions are implemented by means of recursive functions which causes some function call overhead. Second, although the compiler's index vector elimination scheme simplifies the offset computation $\iota fs (vmap\ x,y (xi,yi \rightarrow xi+yi))$ into $(\iota fs\ x)+(\iota fs\ y)$, it cannot further simplify the two offset computations such that they have been computed by means of separate `for`-loops. Third, as both index vectors x and y are required to determine the offset into f , memory for the two vectors must be dynamically allocated. Since the amount of work performed by each instance of the innermost loop is small, the memory management overhead is significant.

As a more positive result, the rank-specific instance of `convolve`, which is the typical case, only consumes 63% of the time required by the reference program. One reason for the good performance is that none of the problems hampering the rank-generic program occurs in the rank-specific setting. Moreover, the QUBE compiler inlines the definition of `convolve` into the applying context where the loop boundaries and array sizes are constants instead of variables that must be kept in registers.

9.3 Quicksort

Fig. 9.3 shows a type-safe implementation of Hoare’s quicksort algorithm for sorting integer arrays. The program aims to mimic the well-known imperative implementations of quicksort as closely as possible. To achieve good performance, the program makes extensive use of uniqueness types that allow for destructive array updates.

The type bounded $l\ u$ describes all integers in the inclusive interval $[l, u]$. The auxiliary function `swap` takes a unique array and destructively exchanges the elements at two given index positions `i` and `j`. The types of the indices ensure that no out-of-bounds accesses can occur.

The function `qsort` takes a unique array `arr` of length `n` and two integers `l` and `r` that specify the indices of the leftmost and the rightmost array element to be sorted. When there are at least two elements to be sorted, the program determines a pivot element and partitions the array such that all elements less than or equal to the pivot element are moved to its left and all other elements are moved to its right. Both parts of the array are then sorted recursively. The result of `qsort` has polymorphic uniqueness, such that it may be used in places where either a unique array or a shared array is expected.

The partitioning function `partition` takes the unique array, the interval of elements to be sorted given by `l` and `r` and the initial position `pivot_idx` of the pivot element in the interval. The function swaps the pivot element to the rightmost position in the interval. Thereafter, it uses the tail-recursive function `ploop` to rearrange the array elements. The function traverses the array with a read index `read` and swaps all elements less than or equal to the pivot elements to a store index `store` that marks the boundary between both parts of the interval. At the end of the loop, the pivot element is swapped to the final index `store`. `partition` returns the partitioned array and the new location of the pivot element.

The second function `qsort` is merely a wrapper function that parameterises the recursive quicksort function.

Figure 9.4 compares the run-times of a QUBE implementation of quicksort with a corresponding reference implementation in C. As a worst-case input, both programs are applied to arrays with ten million integers that are initially sorted in reverse order. The QUBE program takes 50% more time to sort the array than the C program. The overhead stems from the memory management required to allocate and garbage collect the tuples returned by `partition`. Since `ploop` is a tail-recursive function that has no further call-sites, the LLVM inlines the function into the body of `partition` and subsequently converts the tail-recursion into a loop.


```

type bounded l:int u:int = { v:int | l <= v & v <= u }

let swap n:nat arr:[int|[n]] i:(index n) j:(index n) =
  let (arr,ai) = arr![[i]] in
  let (arr,aj) = arr![[j]] in
  let arr = arr![[i]] ← aj in
  let arr = arr![[j]] ← ai in
  arr

let partition n:nat arr:[int|[n]]
  l:nat r:{v:int|v<n} pivot_idx:(bounded l r) =
  let (arr,pivot) = arr![[pivot_idx]] in
  let arr = swap n arr pivot_idx r in
  let rec ploop arr:[int|[n]] read:(bounded l r)
    store:(bounded l read) :*(?[int|[n]],bounded l r) =
    if read < r
    then
      let (arr,arr_read) = arr![[read]] in
      if arr_read <= pivot
      then ploop (swap n arr read store) (read+1) (store+1)
      else ploop arr (read+1) store
    else (swap n arr store r,store)
  in
  ploop arr l l

let rec qsort n:nat arr:[int|[n]] l:nat r:{v:int|v<n} :?[int|[n]] =
  if l < r
  then
    let pivot_idx = (l+r)/2 in
    let (arr,pivot_idx) = partition n arr l r pivot_idx in
    let arr = qsort n arr l (pivot_idx-1) in
    let arr = qsort n arr (pivot_idx+1) r in
    arr
  else arr

let qsort n:nat a:[int|[n]] = qsort n a 0 (n-1)

```

Figure 9.3: Quicksort in QUBE

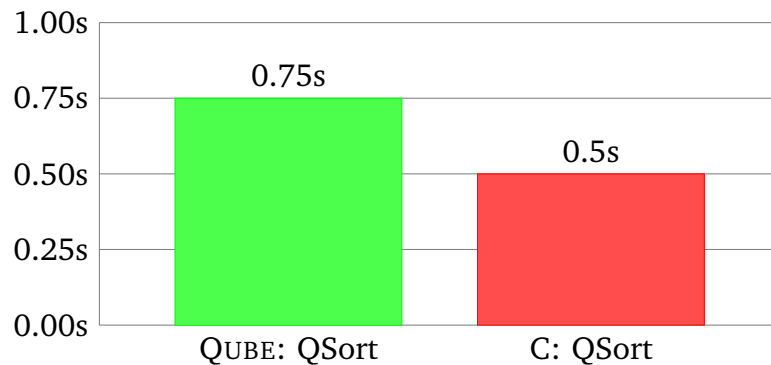


Figure 9.4: Run-times of quicksort in QUBE and in C for ten million elements.

Summary

This chapter evaluated the suitability of the QUBE programming language and its implementation for actual program development by means of some well-known example programs.

The rank-generic implementations of the inner product and the convolution are at the same time simpler and more powerful than the corresponding rank-specific C programs. In both cases, the dependent types are unobtrusive and naturally document the constraints the functions impose on the arguments. The QUBE implementation of quicksort uses uniqueness types and destructive array updates to mimic the well-known imperative implementations of the algorithm. The combination of dependent types and uniqueness types causes some notational overhead, in particular since the partitioning loop maps to a tail-recursive function that requires its own set of typed arguments.

At least in a rank-specific context, which is the typical case, the run-time performance of the QUBE programs for inner product and convolution is on par with the C functions. In the case of quicksort, we have identified tuple construction as a source of overhead. An optimisation that replaces passing boxed tuples between functions with passing the individual components could eliminate large parts of this overhead.

10

Conclusion and Future Work

Making the expressive power of dependent types available for practical program development is a subject of ongoing research. It is a particular challenge to design programming systems with dependent types in a way such that the user is not required to have expert knowledge in type theory. QUBE is a new programming language that employs dependent types in the context of array programming where their use is both intuitive and beneficial.

Dependent types are intuitive for array programs because rank and shape are inherent properties of multidimensional arrays. Scientific programmers are used to specifying their algorithms in terms of array shapes: each undergraduate course on linear algebra teaches the function type of matrix multiplication as $\mathbb{R}^{m \times n} \times \mathbb{R}^{n \times p} \rightarrow \mathbb{R}^{m \times p}$. Dependent types allow the developer to concisely express the constraints on the array shapes.

Dependent types are beneficial for array programs because base type errors, rank errors, shape errors, and even array boundary violations can be statically ruled out. Array programs with dependent types are thus correct by construction. Since static verification makes dynamic checks such as array bounds checks obsolete, dependent types also help to improve the run-time efficiency of array programs.

The syntax, semantics, and type system of QUBE have been formalised by means of the core language $\text{QUBE}_{\text{CORE}}$ that captures the essential concepts of QUBE without any syntactic sugar. Multidimensional array types are parameterised by integer vectors that represent the array shape. Integer vector types are in turn

parameterised by integers that represent the vector length. As a significant result of this thesis, the type system of `QUBECORE` has a formal correctness proof, i. e., a proof that well-typed expressions cannot exhibit any run-time errors.

Based on `QUBECORE`, the actual QUBE programming language adds a host of features to facilitate practical programming, such as an ML-style module system and support for stateful computations. QUBE is implemented by a compiler that translates programs into efficient native code. In order to decide whether two dependent types are equivalent or in a subtype relation, the compiler performs type checking in collaboration with a theorem prover for the Satisfiability Modulo Theories problem of first-order logic. Since SMT solving proceeds fully automatically, the QUBE type checker behaves very much like a (very powerful) type checker for a mainstream programming language that either accepts or rejects a program without further interaction with the user. Moreover, the QUBE compiler uses the rank and shape information from the dependent types to instrument programs with corresponding expressions wherever these properties are required at run-time, for example to compute memory offsets. The benefits of this strategy are twofold: First, once the structural properties of arrays are encoded explicitly in the intermediate program, they become subject to compiler optimisation. Second, multidimensional arrays may be represented as mere sequences of data without additional shape descriptors or type tags. In effect, the compiled programs contain very little overhead.

Unlike interpreted languages, QUBE does not provide a large number of built-in array operations. Instead, the standard array operations can be defined in QUBE itself by means of the versatile `gen` and `loop` expressions. Due to static typing, these array operations are inherently type-safe. QUBE appears to be well-suited for developing practical array programs. For most rank-generic functions, dependent types are unobtrusive as they naturally express the constraints a function imposes on its arguments. For the programs investigated, the run-time performance of QUBE programs is roughly on par with corresponding C programs.

The development of QUBE has focused on static verification of rank-generic array programs. There are several potential directions for future extensions of both the language and the compiler.

The most glaring omission in QUBE's type system is missing support for polymorphism. In the current system, functions must be re-implemented for each argument type. Adding support for parametric polymorphism would allow the user to define functions that operate on values of arbitrary types.

Furthermore, the QUBE compiler would greatly benefit from additional program optimisation schemes. QUBE encourages a programming style where programs are composed from predefined rank-generic array operations. Although elegant, the resulting code is plagued with excessive creation of temporary arrays that

increase both the program's time and memory demands. In order to avoid temporary arrays, QUBE could employ similar techniques as the SAC compiler. WITH-LOOP-FOLDING [88] is an effective deforestation technique that replaces selections into arrays with the selected element's definition. In QUBE, it should be possible to perform deforestation even in a rank-generic setting by using the SMT solver to determine the code that defines a given element.

Finally, due to its functional and side-effect free semantics and its focus on array operations that uniformly apply to a large number of elements, QUBE lends itself well for non-sequential evaluation on multi-core processors or even general-purpose graphics processing units. Since almost all array operations are defined in terms of `gen` and `loop` expressions, the implementation efforts can be concentrated on these expressions. The evaluation of `gen` can be trivially distributed over multiple worker threads. In contrast, the accumulator variable and the deterministic evaluation order of `loop` preclude a naive approach to concurrent evaluation. Again, taking a leaf out of SAC's book, a potential remedy could be to define a `parallel` loop that does not guarantee a certain evaluation order and uses an additional function to combine the partial results.

The current QUBE compiler may be used as a basis to explore all the directions of future work outlined above.

Bibliography

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, USA, 1986.
- [2] Thorsten Altenkirch, Conor McBride, and James McKinna. *Why Dependent Types Matter*. Manuscript, available online, April 2005.
- [3] Andrew W. Appel. *SSA is Functional Programming*. ACM SIGPLAN Notices, vol. 33(4), pp. 17–20, 1998.
- [4] Lennart Augustsson. *Cayenne – A Language with Dependent Types*. In Proceedings of the third ACM SIGPLAN International Conference on Functional Programming, ICFP '98, pp. 239–250. ACM Press, New York, NY, USA, 1998.
- [5] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, vol. 103 of Studies in Logics and the Foundations of Mathematics. North Holland, Amsterdam, The Netherlands, 1981.
- [6] Henk Barendregt. *Introduction to Generalized Type Systems*. Journal of Functional Programming, vol. 1(2), pp. 125–154, 1991.
- [7] Henk Barendregt. *Lambda Calculi with Types*, vol. 2 of Handbook of Logic in Computer Science, pp. 117–309. Oxford University Press, Inc. New York, NY, USA, 1992.
- [8] Henk Barendregt and Erik Barendsen. *Introduction to Lambda Calculus*. Technical report, Department of Computer Science, Catholic University of Nijmegen, 1991.
- [9] Erik Barendsen and Sjaak Smetsers. *Conventional and Uniqueness Typing in Graph Rewrite Systems*. In Rashmi K. Shyamasundar, editor, Foundations of Software Technology and Theoretical Computer Science, 13th Conference, FSTTCS'93, Lecture Notes in Computer Science, vol. 761, pp. 41–51. Springer Verlag, Berlin, Germany, 1993.

- [10] Robert Bernecky. *Compiling APL*. In Lenore M. Restifo Mullin et. al., editor, *Arrays, Functional Languages, and Parallel Systems*, pp. 19–33. Kluwer Academic Publishers, 1991.
- [11] Robert Bernecky. *An Overview of the APEX Compiler*. Technical Report 305/97, University of Toronto, Toronto, Canada, 1997.
- [12] Robert Bernecky, Stephan Herhut, Sven-Bodo Scholz, Kai Trojahner, Clemens Grelck, and Alex Shafarenko. *Index Vector Elimination: Making Index Vectors Affordable*. In Zoltán Horváth and Viktória Zsók, editors, *Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages, IFL'06*, Technical Report, vol. 2006-S01. Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary, 2006.
- [13] Richard S. Bird. *An Introduction to Functional Programming using Haskell*. Series in Computer Science. Prentice Hall Europe, London, UK, second edition, 1998.
- [14] Guy E. Blelloch. *Programming Parallel Algorithms*. *Communications of the ACM*, vol. 39(3), pp. 85–97, 1996.
- [15] Johannes Blume. *A Decision Procedure for Linear Vector Arithmetic*. Bachelor's thesis, Institut für Softwaretechnik und Programmiersprachen, Universität zu Lübeck, Lübeck, 2008.
- [16] Hans Boehm, Alan Demers, and Scott Shenker. *Mostly Parallel Garbage Collection*. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Languages Design and Implementation, PLDI '91*, ACM SIGPLAN Notices, vol. 26, pp. 157–164. ACM Press, New York, NY, USA, 1991.
- [17] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation*. Springer-Verlag, Berlin, Heidelberg, Germany, 2007.
- [18] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. *What's Decidable About Arrays?* In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Lecture Notes in Computer Science*, vol. 3855, pp. 427–442. Springer-Verlag, Berlin, Heidelberg, Germany, 2006.
- [19] Florian Büther. *A Compiler for Qube, a Functional Array Programming Language with Dependent Types*. Studienarbeit, Institut für Softwaretechnik und Programmiersprachen, Universität zu Lübeck, Lübeck, 2008.

-
- [20] Florian Büther. *Modules and State for the Functional Programming Language Qube*. Diploma thesis, Institut für Softwaretechnik und Programmiersprachen, Universität zu Lübeck, Lübeck, 2010.
- [21] David Cann. *Retire Fortran? A Debate Rekindled*. *Communications of the ACM*, vol. 35(8), pp. 81–89, 1992.
- [22] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Developing Applications with Objective Caml*, 2000.
- [23] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. *Data Parallel Haskell: a Status Report*. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP '07*, pp. 10–18. ACM Press, New York, NY, USA, 2007.
- [24] Manuel M.T. Chakravarty and Gabriele Keller. *An Approach to Fast Arrays in Haskell*. In Johan Jeuring and Simon L. Peyton Jones, editors, *Advanced Functional Programming, 4th International School, AFP 2002, Lecture Notes in Computer Science*, vol. 2638, pp. 27–58. Springer-Verlag, Berlin, Heidelberg, Germany, 2003.
- [25] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. *The High-Level Parallel Language ZPL Improves Productivity and Performance*. In *Proceedings of the First Workshop on Productivity and Performance in High-End Computing (PPHEC-04)*, pp. 66–75, 2004.
- [26] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, NJ, USA, 1941.
- [27] Stephen A. Cook. *The Complexity of Theorem-Proving Procedures*. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71*, pp. 151–158. ACM Press, New York, NY, USA, 1971.
- [28] Martin Davis, George Logemann, and Donald Loveland. *A Machine Program for Theorem-Proving*. *Communications of the ACM*, vol. 5(7), pp. 394–397, 1962.
- [29] Martin Davis and Hilary Putnam. *A Computing Procedure for Quantification Theory*. *Journal of the ACM*, vol. 7(3), pp. 201–215, 1960.
- [30] Leonardo M. de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference*,

- TACAS'08, Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer-Verlag, Berlin, Heidelberg, Germany, 2008.
- [31] Luiz de Rose and David Padua. *Techniques for the Translation of MATLAB Programs into Fortran 90*. ACM Transactions on Programming Languages and Systems, vol. 21(2), pp. 286–323, 1999.
- [32] Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. *Uniqueness Typing Simplified*. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, Implementation and Application of Functional Languages, 19th International Symposium, IFL'07, Lecture Notes in Computer Science, vol. 5083, pp. 201 – 218. Springer-Verlag, Berlin, Heidelberg, Germany, 2008.
- [33] Bruno Dutertre and Leonardo M. de Moura. *A Fast Linear-Arithmetic Solver for DPLL(T)*. In Thomas Ball and Robert B. Jones, editors, Computer Aided Verification, 18th International Conference, CAV 2006, Lecture Notes in Computer Science, vol. 4144, pp. 81–94. Springer Verlag, Heidelberg, Germany, 2006.
- [34] Niklas Eén and Niklas Sörensson. *An Extensible SAT-solver*. In Enrico Giunchiglia and Armando Tacchella, editors, SAT '03, Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer-Verlag, Berlin, Heidelberg, Germany, 2003.
- [35] Adin D. Falkoff and Kenneth E. Iverson. *The Design of APL*. IBM Journal of Research and Development, vol. 17(4), pp. 324–334, 1973.
- [36] Cormac Flanagan. *Hybrid Type Checking*. In Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '06, pp. 245–256. ACM Press, New York, NY, USA, 2006.
- [37] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. *The Essence of Compiling with Continuations*. In Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI '93, pp. 237–247. ACM Press, New York, NY, USA, 1993.
- [38] Anwar Ghuloum, Terry Smith, Gansha Wu, Xin Zhou, Jesse Fang, Peng Guo, Byoungro So, Mohan Rajagopalan, Yongjian Chen, and Biao Chen. *Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architecture*. Intel Technology Journal, vol. 11(4), pp. 333–346, 2007.
- [39] Clemens Grelck. *Shared Memory Multiprocessor Support for Functional Array Processing in SAC*. Journal of Functional Programming, vol. 15(3), pp. 353–401, 2005.

- [40] Clemens Grelck, Karsten Hinckfuß, and Sven-Bodo Scholz. *With-Loop Fusion for Data Locality and Parallelism*. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, Implementation and Application of Functional Languages, 17th International Workshop, IFL'05, Lecture Notes in Computer Science, vol. 4015, pp. 178–195. Springer-Verlag, Berlin, Heidelberg, Germany, 2006.
- [41] Clemens Grelck, Frank Penczek, and Kai Trojahnner. *CAOS: A Domain-Specific Language for the Parallel Simulation of Cellular Automata*. In Viktor Malyshkin, editor, Parallel Computing Technologies, 9th International Conference, PaCT'07, Lecture Notes in Computer Science, vol. 4671, pp. 410–417. Springer-Verlag, Berlin, Heidelberg, Germany, 2007.
- [42] Clemens Grelck and Sven-Bodo Scholz. *SAC: A Functional Array Language for Efficient Multithreaded Execution*. International Journal of Parallel Programming, vol. 34(4), pp. 383–427, 2006.
- [43] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. *A Binding Scope Analysis for Generic Programs on Arrays*. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, Implementation and Application of Functional Languages, 17th International Workshop, IFL'05, Lecture Notes in Computer Science, vol. 4015, pp. 212–230. Springer-Verlag, Berlin, Heidelberg, Germany, 2006.
- [44] Clemens Grelck, Sven-Bodo Scholz, and Kai Trojahnner. *With-Loop Scalarization: Merging Nested Array Operations*. In Philip W. Trinder, Greg Michaelson, and Ricardo Pena, editors, Implementation of Functional Languages, 15th International Workshop, IFL'03, Lecture Notes in Computer Science, vol. 3145, pp. 118–134. Springer-Verlag, Berlin, Heidelberg, Germany, 2004.
- [45] Clemens Grelck and Kai Trojahnner. *Implicit Memory Management for SAC*. In Clemens Grelck and Frank Huch, editors, Implementation and Application of Functional Languages: 16th International Workshop, IFL'04, Technical Report, vol. 0408, pp. 335–348. University of Kiel, 2004.
- [46] Clemens Grelck, Tim van Deurzen, Stephan Herhut, and Sven-Bodo Scholz. *An Adaptive Compilation Framework for Generic Data-Parallel Array Programming*. In Andreas Krall and Gergö Barany, editors, 15th Workshop on Compilers for Parallel Computing, CPC'10. Institute of Computer Languages, Vienna University of Technology, Vienna, Austria, 2010.

- [47] Dick Grune, Henri E. Bal, Cerial J. H. Jacobs, and Koen G. Langendoen. *Modern Compiler Design*. John Wiley & Sons, West Sussex, England, 2000.
- [48] Kurt Gödel. *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme*. Monatshefte für Mathematik und Physik, vol. 38, pp. 173–198, 1931.
- [49] Robert Harper. *Introduction to Standard ML*. Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1993.
- [50] John R. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.
- [51] Stephan Herhut, Sven-Bodo Scholz, Robert Bernecky, Clemens Grellck, and Kai Trojahner. *From Contracts Towards Dependent Types: Proofs by Partial Evaluation*. In Olaf Chitil, Zoltan Horváth, and Viktória Zsók, editors, *Implementation and Application of Functional Languages: 19th International Workshop, IFL'07, Lecture Notes in Computer Science*, vol. 5083, pp. 254–273. Springer-Verlag, Berlin, Heidelberg, Germany, 2008.
- [52] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda Calculus*. London Mathematical Society Student Texts. Cambridge University Press, Cambridge, UK, 1986.
- [53] Paul Hudak and Adrienne Bloss. *The Aggregate Update Problem in Functional Programming Systems*. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL'85*, pp. 300–313. ACM Press, New York, NY, USA, 1985.
- [54] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, et al. *Report on the Programming Language Haskell, A Non-strict, Purely Functional Language*. SIGPLAN Notices, vol. 27(5), 1992.
- [55] Kenneth E. Iverson. *A Programming Language*. John Wiley, New York, NY, USA, 1962.
- [56] Kenneth E. Iverson. *Programming in J*. Iverson Software Inc., Toronto, Canada, 1991.
- [57] Kenneth E. Iverson. *J Introduction and Dictionary*. Iverson Software Inc., Toronto, Canada, 1995.
- [58] C. Barry Jay. *The FISh Language Definition*. Technical report, School of Computing Sciences, University of Technology, Sidney, Australia, 1998.

- [59] C. Barry Jay and Paul A. Steckler. *The Functional Imperative: Shape!* In Chris Hankin, editor, Proceedings of the 7th European Symposium on Programming, ESOP'98, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS'98, Lecture Notes in Computer Science, vol. 1381, pp. 139–53. Springer-Verlag, Berlin, Heidelberg, Germany, 1998.
- [60] Thomas Johnsson. *Lambda Lifting: Transforming Programs to Recursive Equations*. In Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science, vol. 201. Springer-Verlag, Berlin, Heidelberg, Germany, 1985.
- [61] Pramod G. Joisha and Prithviraj Banerjee. *An Algebraic Array Shape Inference System for MATLAB*. ACM Transactions on Programming Languages and Systems, vol. 28(5), pp. 848–907, 2006.
- [62] Guy L. Steele Jr. *Debunking the “Expensive Procedure Call” Myth, or Procedure Call Implementations Considered Harmful, or LAMBDA: The Ultimate GOTO*. In Proceedings of the 1977 Annual Conference, ACM'77, pp. 153–162. ACM Press, New York, NY, USA, 1977.
- [63] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. *Regular, Shape-Polymorphic, Parallel Arrays in Haskell*. In Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP'10, pp. 261–272. ACM Press, New York, NY, USA, 2010.
- [64] Werner Kluge. *Abstract Computing Machines: A Lambda Calculus Perspective*. Springer-Verlag Berlin Heidelberg, 2005.
- [65] Dietmar Kreye. *A Compilation Scheme for a Hierarchy of Array Types*. In Thomas Arts and Markus Mohnen, editors, Implementation of Functional Languages, 13th International Workshop, IFL'02, Lecture Notes in Computer Science, vol. 2312, pp. 18–35. Springer-Verlag, Berlin, Heidelberg, Germany, 2002.
- [66] Daniel Kroening and Ofer Strichman. *Decision Procedures*. Springer-Verlag, Berlin, Heidelberg, 2008.
- [67] Peter J. Landin. *The Mechanical Evaluation of Expressions*. Computer Journal, vol. 6(4), pp. 308–320, 1964.

- [68] Chris Lattner. *LLVM: An Infrastructure for Multi-Stage Optimization*. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.
- [69] Xavier Leroy. *The Objective Caml System release 3.11*. INRIA, Rocquencourt, France, 2008.
- [70] David MacQueen, Robert Harper, Robin Milner, et al. *Functional Programming in ML*. LFCS education, University of Edinburgh, Edinburgh, Scotland, UK, 1987.
- [71] Simon Marlow and Simon Peyton Jones. *Making a Fast Curry: Push/Enter vs. Eval/Apply for Higher-Order Languages*. In ICFP '04: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, pp. 4–15. ACM Press, New York, NY, USA, 2004.
- [72] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.
- [73] Conor McBride and James McKinna. *The View from the Left*. Journal of Functional Programming, vol. 14(1), pp. 69–111, January 2004.
- [74] Cheryl McCosh. *Type-Based Specialization in a Telescoping Compiler for MATLAB*. Master Thesis TR03-412, Rice University, Houston, Texas, USA, 2003.
- [75] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, USA, 1990.
- [76] Cleve Moler, John Little, and Steve Bangert. *Pro-Matlab User's Guide*. The MathWorks, Cochituate Place, 24 Prime Park Way, Natick, MA, USA, 1987.
- [77] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. *Chaff: Engineering an Efficient SAT Solver*. In Proceedings of the 38th Annual Design Automation Conference, DAC'01, pp. 530–535. ACM Press, New York, NY, USA, 2001.
- [78] Greg Nelson and Derek C. Oppen. *Simplification by Cooperating Decision Procedures*. ACM Transactions on Programming Languages and Systems, vol. 1(2), pp. 245–257, 1979.
- [79] Derek C. Oppen. *Complexity, Convexity and Combinations of Theories*. Theoretical Computer Science, vol. 12(3), pp. 291 – 302, 1980.

-
- [80] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ, USA, 1987.
- [81] Simon L. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003.
- [82] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.
- [83] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2004.
- [84] Marinus J. Plasmeijer and Marko van Eekelen. *Concurrent Clean 2.0 Language Report*. University of Nijmegen, The Netherlands, 2001.
- [85] Mojżesz Presburger. *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt*. Comptes Rendus du I congrès de Mathématiciens des Pays Slaves, Warszawa, vol. , pp. 92–101, 1929.
- [86] Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. *Liquid Types*. In Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, PLDI’08, SIGPLAN Notices, vol. 43(6), pp. 159–169. ACM Press, New York, NY, USA, 2008.
- [87] Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. *Liquid Types*. CSE Techreport, University of California, San Diego, California, USA, 2008.
- [88] Sven-Bodo Scholz. *With-loop-folding in SAC — Condensing Consecutive Array Operations*. In Chris Clack, Tony Davie, and Kevin Hammond, editors, Selected Papers from the 9th International Workshop on Implementation of Functional Languages, IFL’97, Lecture Notes in Computer Science, vol. 1467, pp. 72–92. Springer-Verlag, Berlin, Heidelberg, Germany, 1998.
- [89] Sven-Bodo Scholz. *Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting*. Journal of Functional Programming, vol. 13(6), pp. 1005–1059, 2003.
- [90] Sven-Bodo Scholz, Stephan Herhut, Clemens Grellck, and Frank Penczek. *SAC 1.0 – Single Assignment C – Tutorial*. Technical Report 498, School of Computer Science, University of Hertfordshire, Hatfield, United Kingdom, 2010.

- [91] Peter Sestoft. *Demonstrating Lambda Calculus Reduction*. In Torben Mogensen, David Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, Lecture Notes in Computer Science, vol. 2566, pp. 420–435. Springer-Verlag, Berlin, Heidelberg, Germany, 2002.
- [92] Jan Smith, Bengt Nordström, and Kent Petersson. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.
- [93] Herb Sutter. *A Fundamental Turn Toward Concurrency in Software*. *Dr. Dobb's Journal*, vol. 30(3), pp. 16–23, 2005.
- [94] Kai Trojahner. *Implicit Memory Management for a Functional Array Processing Language*. Diploma thesis, Institut für Softwaretechnik und Programmiersprachen, Universität zu Lübeck, Lübeck, Germany, 2005.
- [95] Kai Trojahner. *Assembling Concurrent Programs Correctly from Data-Parallel Program Bricks*. In Viktor Malyshkin, editor, *Parallel Computing Technologies, 9th International Conference, PaCT'07*, Lecture Notes in Computer Science, vol. 4671, pp. 410–417. Springer-Verlag, Berlin, Heidelberg, Germany, 2007.
- [96] Kai Trojahner and Clemens Grelck. *Independently Typed Array Programs Don't Go Wrong*. *Journal of Logic and Algebraic Programming*, vol. 78(7), pp. 643–664, 2009. The 19th Nordic Workshop on Programming Theory (NWPT 2007).
- [97] Kai Trojahner and Clemens Grelck. *Descriptor-Free Representation of Arrays with Dependent Types*. In Sven-Bodo Scholz and Olaf Chitil, editors, *Implementation and Application of Functional Languages, 20th international symposium, IFL'08*, Lecture Notes in Computer Science, vol. 5836. Springer-Verlag, Berlin, Heidelberg, Germany, 2010.
- [98] Kai Trojahner, Clemens Grelck, and Sven-Bodo Scholz. *On Optimising Shape-Generic Array Programs using Symbolic Structural Information*. In Zoltán Horváth and Viktória Zsóka, editors, *Implementation and Application of Functional Languages, 18th International Symposium, IFL'06*, Lecture Notes in Computer Science, vol. 4449, pp. 1–18. Springer-Verlag, Berlin, Heidelberg, Germany, 2007.
- [99] Markus Weigel. *Facilitating Array Programming with Dependent Types – Precise Error Messages and Implicit Index Arguments for Qube*. Master's thesis, Institut für Softwaretechnik und Programmiersprachen, Universität zu Lübeck, Lübeck, Germany, 2008.

-
- [100] Hongwei Xi. *Applied Type System (extended abstract)*. In S. Baradi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs, Third International Workshop, TYPES'03*, Lecture Notes in Computer Science, vol. 3085, pp. 394–408. Springer Verlag, Berlin, Germany, 2004.
- [101] Hongwei Xi and Frank Pfenning. *Eliminating Array Bound Checking through Dependent Types*. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'98*, ACM SIGPLAN Notices, vol. 33(5), pp. 249–257. ACM Press, New York, NY, USA, 1998.
- [102] Hongwei Xi and Frank Pfenning. *Dependent Types in Practical Programming*. In Alexander Aiken, editor, *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL'99*, pp. 214–227. ACM Press, New York, NY, USA, 1999.
- [103] Christoph Zenger. *Indexed Types*. *Theoretical Computer Science*, vol. 187(1-2), pp. 147–165, 1997.