Indexed Types in Object-Oriented Programming

Joana Campos and Vasco T. Vasconcelos

University of Lisbon, Faculty of Sciences, LaSIGE

Abstract. Dependent type systems allow semantic properties to be expressed in types that carry important information about program values. The type systems in mainstream languages such as Java are effective but have a limited expressive power. We propose to extend a simple Java-like language with indexed types, a form of dependent types defined on index expressions that can statically detect many programming errors. Index types take the form of type annotations in the generics style, so as to express semantic properties in a fashion familiar to object-oriented programmers. For example, Polynomial⟨3⟩ is an instance of Polynomial⟨**nat** degree⟩, where degree has type **nat**, which is the type of all polynomials of some degree. Expressions in index types currently represent integer values only. Index types may be associated with class definitions, and may be used to constrain base types in fields or local variables, in arguments to methods or in return types. As opposed to conventional types, indexed types may change over a program lifetime. We discuss introducing indirection into type environments to provide support to type check references to mutable objects.

1 Introduction

Iterating through the elements of an array to find the one that matches a certain criterion is a pattern common to all programming languages with support for arrays. This pattern opens the opportunity for frequent and tedious mistakes in the use of array indices. The type systems in mainstream languages such as Java are effective but have a limited expressive power, not enforcing compile-time index checking against the bounds of the array. Instead, the Java interpreter automatically does a range-check, suspends execution and throws an exception if the index is out of range. Other languages, such as C, never perform any sort of automatic bounds checking. Speed and efficiency are the arguments presented by those who advocate that compilers should not offer the option to capture these errors, leaving them, if any, for a late run-time checking. However, many compilers can currently detect this sort of error using efficient type-checking algorithms. The frequency of such errors, and the time programmers spend detecting and correcting them, are strong reasons to make bound errors detectable at compile time in mainstream programming languages.

Dependently typed programming languages have for long been used to capture this and other sorts of errors at compile time. Dependent type systems allow

semantic properties to be expressed in types that carry important information about program values. The practical advance of these systems has been hindered by complexity issues revealed during type checking. In the last years, however, impressive advances have been made introducing more or less restrictive forms of dependent types into programming languages while retaining practical type-checking. Xi and Pfenning's functional DML (Dependent ML) [21] and Xi's imperative version Xanadu [18] have been designed with the concern of narrowing the gap between dependent types and realistic programming. Both languages offer a domain-specific application of dependent types in the form of index types that can be used to specify and infer more precise properties about programs, which are statically enforced by the compiler.

Motivated by Xi and Pfenning's work, we propose to extend a simple Java-like language with support for a light-weight, restricted form of dependent types defined on index expressions that represent integer values, allowing for safer implementations of data structures. The programmer needs to supply type annotations in the form of index types, refining base types with program properties that can be statically verified by the type checker. We have designed these type annotations using the generics style, so as to express semantic properties in a familiar fashion for object-oriented programmers. We change the class declaration to allow it to be parametrized by one or more index variables that can be used anywhere inside the class, in fields and in methods. This same technique can be applied to create indexed methods, but the scope of the index variables is now restricted to the method in which it is introduced, and they can be used to express constrains on its arguments, return type or on local variables.

A type system that supports value-dependent types in object-oriented programming is also proposed in the X10 programming language. Nystrom *et al.* [12] present an expressive framework of constrained types, where constrained variables can have any type, not just the basic **int**, and constraints include functions. X10's support of constrained types focuses on the immutable state of objects. The challenge of our work stems from preparing the type system to support aliasing of objects whose states may change over a program lifetime. We intend to extend indexed types with a form of indirection in type environments for keeping track of shared object references. Notice that this is still a work in progress. We are building our ideas from singleton type systems—*aliasing constraints*—a notion proposed by Smith *et al.* [16] in the context of a typed assembly language. We are mainly interested in adapting the idea of a pointer type to track references to an object whose type changes as the program evolves.

Indexed types with support for mutable objects can introduce into object-oriented programming the following properties: (1) robustness as only certain safe operations are allowed on some data structures, (2) flexibility as aliasing is allowed but no inconsistencies occur, and (3) readability since design decisions are documented in the language types.

The remaining sections are organized as follows: Section 2 introduces the main features of the language and the form of our indexed types; Section 3 presents a programming example to illustrate the expressive power of indexed

$$
\begin{array}{rrcl}
\text{(class declarations)} & D & ::= & \mathsf{class}\ C\ \langle \vec{\gamma}\ \vec{a} \rangle\ \{\vec{F}; \vec{M}\} \\
\text{(field declarations)} & F & ::= & T\ f \\
\text{(method declarations)} & M & ::= & \langle \vec{\gamma}\ \vec{a} \rangle\ T\ m(T\ x)\ \{e\} \\
\text{(values)} & v & ::= & \mathsf{unit} \mid \mathsf{true} \mid \mathsf{false} \mid \ldots -1 \mid 0 \mid 1 \mid \ldots \mid o \\
\text{(expressions)} & e & ::= & v \mid x \mid o.f \mid e; e \mid o.f = e \mid \mathsf{new}\ C\ \langle \vec{i} \rangle\ (\vec{e}) \\
 & & \mid & o.f.m(e) \mid \mathsf{if}\ (e)\ e\ \mathsf{else}\ e \mid \mathsf{while}\ (e)\ e \\
\text{(types)} & T & ::= & \mathsf{unit} \mid \mathsf{boolean}\ \langle i \rangle \mid \mathsf{int}\ \langle i \rangle \mid C\ \langle \vec{i} \rangle \mid \langle \gamma\ a \rangle\ T
\end{array}
$$

Fig. 1. The user syntax

$$
\begin{array}{rrcl}
\text{(index types)} & \gamma & ::= & \mathsf{integer} \mid \{a{:}\gamma \mid p\} \\
\text{(index expressions)} & i, j & ::= & a \mid i + j \mid i - j \mid i * j \mid i \div j \mid i \,\%\, j \\
 & & \mid & \mathit{min}(i, j) \mid \mathit{max}(i, j) \\
\text{(index propositions)} & p & ::= & i < j \mid i \leq j \mid i > j \mid i \geq j \mid i = j \mid i \neq j \\
 & & \mid & p_1 \wedge p_2 \mid p_1 \vee p_2
\end{array}
$$

Fig. 2. The syntax of index expressions

types and concludes with an informal discussion of how the language can be extended to handle type changes in mutable objects; Section 4 reviews related work, and finally Section 5 concludes with several ideas for future work.

2 Dependently typed classes

The goal of this section is to provide some intuition about indexed types in our language before presenting a complete example in Section 3. Figure 1 presents the syntax of our language extended with index types in Figure 2, and is followed by a brief explanation of some technical details. Notice that we distinguish between the program language and the index language whose variables are used in types, but cannot be used in computations as they are not available for run-time processing. We have taken Xi and Pfenning's syntax for type index expressions [18, 20, 21] and have adapted it to a simple object-oriented language.

We begin with some general explanations about the syntax. The metavariables C, f, x, and m range over class, field, parameter and method names, respectively. We write $\vec{F}$ as short for $F_1; \ldots; F_n;$ (a sequence of field declarations), $\vec{M}$ as short for $M_1 \ldots M_n$ (a sequence of method declarations), where $n \geq 0$ in all cases. We abbreviate all sort of sequences using a similar pattern. We assume that class identifiers in a sequence of declarations $\vec{D}$ are all distinct, and that the

set of method and field names declared in each class contains only distinct names as well. Object references o include the keyword this, which refers to the current instance. Class, method and field declarations are standard in object-oriented languages, as are expressions. We have defined (in order of appearance) values, parameters, fields, the sequential expression composition, assignment to fields, object creation, the method call, and control flow expressions.

The novelty of our language is related to types and appears as shaded in Figure 1. We use a to range over index variables, whose value is domain-constrained. The index type $\{a{:}\gamma \mid p\}$ in Figure 2 denotes a type γ refined by proposition p. Variable a may occur in p denoting an arbitrary value of type γ. For example, {b:**int** | b≥0} denotes the type of non-negative integer values, which we usually abbreviate to **nat**, and {a:**nat** | a%2=0} is the type of the non-negative even numbers.

A class declaration may specify one or more index variables that constrain its fields and methods, and a method may be parametrized by one or more index variables that can be used to express constraints on its arguments, return type or local variables. For example, class DoubleArray⟨**nat** size⟩ describes a class named DoubleArray parametrized by the subset of natural numbers, possibly to denote the length of the array. The body of the class may use the index variable size to refer, *within types*, to the length of the array.

Types include the unit type with the single value unit, and the *singleton* types int⟨i⟩ and boolean⟨i⟩. The int⟨i⟩ type denotes the set of integer values equal to (the integer value of) expression i, while boolean⟨1⟩ and boolean⟨0⟩ are used for expressions with the true and false values. A type of the form $C\langle i\rangle$ describes an object of class C where the formal parameter has been replaced by the value of index expression i. For example, DoubleArray⟨max(m, n)⟩ describes an array of length max(m,n).

A type $\langle \gamma\, a\rangle T$ introduces an index variable a of index type γ that can be used in type T. For example, a method signature of the form

```
⟨{a:nat | a<size} b⟩ double get (int⟨b⟩ index) {...}
```

says that method get expects a natural number smaller than size. Index variable b (as well as size) may be used in the body of the method to declare further types. On the other hand, a signature of the form:

```
double get (⟨{a:nat | a<size} b⟩ int⟨b⟩ index) {...}
```

would have the same meaning but would not allow to use variable b in the body of the method. The situation must be confronted with the signature of one of the sorting methods in class java.util.Arrays

```
public static ⟨T⟩ void sort(T[] a, Comparator⟨? super T⟩ c)
```

where type variable T is introduced before the signature of the method and further used in the signature and possible in the body.

When programming with index types, certain types are abbreviated as illustrated in Figure 3. If referring to the natural numbers, we can use **nat** for the type $\{a{:}\,\mathsf{integer} \mid a \geq 0\}$. A range of integer values between m and n can be

$$
\begin{aligned}
\mathsf{nat} &\triangleq \{a{:}\,\mathsf{integer} \mid a \geq 0\} \\
\mathsf{int} &\triangleq \langle \mathsf{integer}\ a \rangle\ \mathsf{int}\langle a \rangle \\
\mathsf{int}[m..n] &\triangleq \langle \{b{:}\,\mathsf{integer} \mid m \leq b \wedge b \leq n\}\, a \rangle\ \mathsf{int}\langle a \rangle
\end{aligned}
$$

Fig. 3. Short forms for index types (nat) and types (int)

```
public class DoubleArray⟨nat size⟩ {                                    1
                                                                        2
  public DoubleArray⟨size⟩ (int⟨size⟩ length) { <native> }              3
                                                                        4
  public double get (int[0..size[ index) { <native> }                   5
                                                                        6
  public void set (int[0..size[ index, double item) { <native> }        7
                                                                        8
  public int⟨size⟩ length() { <native> }                                9
}                                                                       10
```

Fig. 4. The DoubleArray native implementation

written as **int**[m..n]. We use open and closed brackets to signify which indices are included in the range. **int**]m..n[can be used to define the integer i satisfying $m < i < n$, and **int**]m..n] is equivalent to the range in $m < i \leq n$.

It is important to distinguish the conventional int value type from the integer index type. On the one hand, int (as an abbreviation) is a type, and can be used to declare fields, local variables and method parameters. On the other hand, integer is an index type and can be used to parametrize types, including class types and integer types, but cannot be used to declare, say, a local variable. In particular we cannot declare a field with the index type nat. Instead, one must write ⟨**nat** b⟩**int**⟨b⟩ to denote the set of the integer values b that happen to be natural numbers.

3 Example

In this section, we present a complete example of an indexed class implementation. We begin by sketching an indexed DoubleArray⟨**nat** size⟩ class that is used by a Polynomial⟨**nat** degree⟩ class, which we explain later in detail. We conclude this section with some considerations about indexed types.

3.1 The DoubleArray⟨nat size⟩ class

Figure 4 shows how the DoubleArray class looks like when indexed by a natural number that happens to denote the array size. Rather than setting up a concrete syntax for array operations (as one would find in C or Java), and in order to

obtain a smaller and more compact user syntax, we establish that our language natively offers classes such as this one, providing the utility methods get, set and length. To create an array of a given size, such as 10, one would simply write:

```
DoubleArray array = new DoubleArray(10);
```

When the constructor is invoked, the compiler can infer the correct type based on the parameter passed in; it allows omitting the type specification. The full type declaration might still be written DoubleArray⟨10⟩array = **new** DoubleArray⟨10⟩(10); The index type 10 may seem redundant, but suppose the class had a constructor that only builds arrays of even length:

```
public DoubleArray⟨size + size % 2⟩ (int⟨size⟩ n) { ... }
```

In this case, it might be useful to document the explicit type as

```
DoubleArray⟨10⟩ array = new DoubleArray⟨10⟩(9);
```

Notice that the language distinguishes between the index variable size and the run-time variable length, defined in the constructor method (line 3), although they may take the same value. The index variable cannot be used in computations inside the class or its methods, while the length variable may. Different values for the index variable size define different DoubleArray classes and can be used to constrain computation values, namely in array subscripting in the get and set methods.

3.2 The Polynomial⟨nat degree⟩ class

We now turn our attention to a class describing a polynomial. The class definition, in Figure 5, is indexed by the degree of the polynomial, and uses an instance of DoubleArray, created in line 6, to hold its terms. The constructor that takes a single parameter (line 5) expects a non-negative number representing the degree of the polynomial, while the one that takes two parameters (line 9) also expects the coefficient of the term of the polynomial's degree.

The sum method signature in line 22 presents some interesting features: it expects another Polynomial of some degree n and returns a Polynomial⟨max(degree,n)⟩ whose degree is the maximum between the current polynomial degree and the one given as parameter. Notice also the loop that starts in line 25. The declared type of i enforces the invariant that the array subscripting (the set and get operations in line 28) is safe, stating that it may range between 0 and the lowest of the two degrees.

Finally, the copy method signature in line 36 states that it expects two DoubleArray, and additionally it expects the index first from where to start the copy. The method signature is indexed by two variables representing two natural numbers: the length n of both arrays and the number m of index first. The proviso that $m \leq n$ ensures that the loop variable i is always within the range of both arrays, which in turn allows the type checker to validate the array access operations in line 39.

```
public class Polynomial⟨nat degree⟩ {

  private DoubleArray⟨degree + 1⟩ terms; // the underlying array object

  public Polynomial⟨degree⟩ (int⟨degree⟩ d) {
    terms = new DoubleArray⟨degree + 1⟩(d + 1);
  }

  public Polynomial⟨degree⟩ (int⟨degree⟩ d, double coefficient) {
    this(d);
    terms.set(d + 1, coefficient);
  }

  public int coefficient (int[0..degree] d) {
    return terms.get(d);
  }

  public int⟨degree⟩ degree() {
    return terms.length() − 1;
  }

 public ⟨nat n⟩ Polynomial⟨max(degree,n)⟩ sum (Polynomial⟨n⟩ other) {
    Polynomial⟨max(degree,n)⟩ result =
      new Polynomial⟨max(degree,n)⟩ (Math.max(this.degree(), other.degree()));
    for (int[0..min(degree, n)] i = 0;
          i ≤Math.min(this.degree(), other.degree());
          i++)
      result.terms.set(i, this.terms.get(i) + other.terms.get(i));
    if (this.degree() > other.degree())
      copy (this.terms, result.terms, other.degree());
    else
      copy (other.terms, result.terms, this.degree());
    return result;
  }

  private ⟨nat n, {m:nat|m ≤ n} m⟩ void copy (
            DoubleArray⟨n⟩ from, DoubleArray⟨n⟩ to, int⟨m⟩ first) {
    for (int[m..n[ i = first; i < from.length(); i++)
      to.terms.set(i, from.terms.get(i));
  }
}
```

Fig. 5. A polynomial implemented with indexed types

3.3 Types that change

A distinctive feature of imperative dependent types frameworks is that types for references may change as computation progresses [18]. In class Polynomial, the loop variable i declared in Figure 5, line 24, becomes of type **int**⟨0⟩ after the initialization. After one pass through the loop (including the i++ instruction), the type becomes **int**⟨1⟩, then **int**⟨2⟩, and so forth, until **int**⟨min(degree,n)⟩.

Local attributes may be subject to the same behaviour. We have carefully crafted our code in order for field array to always be of type DoubleArray⟨degree + 1⟩; the field is indeed of a (Java) **final** nature. To illustrate how this could be otherwise, let us reprogram method sum by taking advantage of a clone() method (not shown).

```
public ⟨nat n⟩ Polynomial⟨max(degree,n)⟩ sum(Polynomial⟨n⟩ other) {
    Polynomial result = clone();
    if (other.degree() > result.degree())
      result.grow(other.array.length());
    for (int[0..n] i = 0; i < other.array.length(); i++)
      result.array.set(i, result.array.get(i) + other.array.get(i));
  }
```

Method grow, not shown, allocates an array of a length passed in the parameter, then copies the entries in array to the new array, and finally assigns the new array to field array. We then see that the type of the local variable result, initially Polynomial⟨degree⟩, may change to Polynomial⟨n⟩ when n>degree, and the same goes for the array within the result object.

A final variant of our example implements method sum in a mutable style: sum is a procedure that changes the state of the target object.

```
  public ⟨nat n⟩ void sum(Polynomial⟨n⟩ other)
    becomes Polynomial⟨max(degree,n)⟩ {
    if (other.array.length() > this.array.length())
      grow(other.array.length());
    for (int[0..n] i = 0; i < other.array.length(); i++)
      array.set(i, array.get(i) + other.array.get(i));
  }
```

From the previous discussion, it should be clear that the changed state induced by a call to the method may have impact on the type of the method: if n>degree the type of **this** changes from Polynomial⟨degree⟩ to Polynomial⟨n⟩. We have annotated such a possibility with a **becomes** clause in the signature of the method.

3.4 Type checking

In general, type checking the code in Figure 5 should be straightforward (and decidable) by using the results of Pfenning and Xi [18, 21], and a Presburger constraint system, allowing as constraints only linear inequalities over the integers.

After type-checking basic types and index types, the constraints generated are extracted and verified by the constraint system, which reports an error if they

cannot be solved. As an example, we discuss why is the array access operation in line 11 Figure 5 safe. The type checker knows that variable d is of value degree and that degree is a **nat** and hence non-negative; it also knows that array terms is of size degree+1. It remains to show that d + 1 is of type **int**[0..degree+1], as per method set in line 7, Figure 4. An adequate constraint solver can easy check that 0 ≤degree + 1 and degree + 1 ≤degree + 1.

The challenge is to type check references whose type may change over time. In this case, a tight control of aliasing is needed. A possibility is to apply the results of previous work on aliasing control, while annotating types with a **un** qualifier to mean that the object may be shared by multiple references, and **lin** to mean that there is a single reference to the object [2, 8, 17]. Except for the last variant of the sum method, all references in the code shown in this paper are linear in this sense.

Another approach that we are investigating is the introduction of indirection into type environments. We are building our ideas from an extension to linear reasoning presented by Smith *et al.* [16]. We intend to extend indexed types with a pointer type in the run-time syntax. To track the evolution of types across a program, the pointer types create an indirection to the environment of indexed types, ensuring simultaneous updates of references to the same object when its type changes.

4 Related Work

The notion of dependent types as types which contain values has been originally proposed by Martin-Löf [10] in his Type Theory, and has had important developments in functional programming languages.

Xi and Pfenning's DML (Dependent ML) [21] extends a real functional programming language with type index expressions that capture many program invariants in data structures and detect those errors at compile time. A particular application of the proposed type system is the array bound check elimination [20]. Their work relates closely to Zenger's indexed types [22]. Those ideas were later introduced into an imperative language, Xanadu [18], designed to demonstrate the benefits of combining imperative programming with dependent types. In this work, we attempt a similar approach in the object-oriented paradigm. The caml-based implementation of Xanadu [19] incorporates dependent record types which are in a sense similar to our dependent classes.

In the past years, there has been great interest in the notion of path-dependent types as those provided by Scala [13, 14]. Several programming languages employ types to express dependencies on enclosing objects, and many provide variants of virtual classes [3, 4, 6, 7, 11]. Our work is also closely related to X10's constrained types [12] in which constraints are defined over final access paths. Nystrom *et al.* define predicates over the immutable state of objects that can capture many invariants in classes. The most important difference between X10's constrained types and our types (and DML's) is that in X10 there is no need for a language of constrained types since they are defined as program properties in the form

of final instance fields. This approach is very appealing and the framework is powerful. To our knowledge, support for aliasing on objects with mutable types has not yet been provided.

In the Cayenne [1] programming language, the distinction between dynamic and static types is not made. Cayenne uses full type dependency and type checking is undecidable. Cyclone [9] came later as a type-safe extension of the C programming language, combining static analysis and run-time checks. It also adds annotations with the purpose of providing more information needed for the program verification.

Dependent types are also the focus of languages such as Ωmega [15], a combination between a purely functional programming language and a theorem prover, and Epigram. The latter follows an approach similar to that of Cayenne, expressing programs as proofs in type theory, but allowing only provably well-founded recursion so as to guarantee decidability.

5 Conclusion and Future Work

We have presented our approach to introduce indexed types into object-oriented programming so as to provide safer implementations of data structures. We have attempted to design an intuitive syntax to indexed types in the form of Java generic-like type declarations.

Our immediate goal is to provide a full formal treatment of the proposed language. We want further study dependently typed mutable classes and present results in a near future. The challenge is to prepare our type system to support aliasing while allowing type changes in mutable classes. We also envisage the possibility of generating run-time checks for the assertions than cannot be statically verified [5].

Acknowledgements This work was funded by project "Assertion-Types for Object-Oriented Programming", FCT (PTDC/EIA-CCO/105359/2008). The authors would like to thank Dimitris Mostrous for insightful comments.

References

1. Augustsson, L.: Cayenne a language with dependent types. In: ICFP. pp. 239–250. ACM (1998)
2. Campos, J., Vasconcelos, V.T.: Channels as objects in concurrent object-oriented programming. In: PLACES 2010. EPTCS (2011), to appear
3. Clarke, D., Drossopoulou, S., Noble, J., Wrigstad, T.: Tribe: a simple virtual class calculus. In: AOSD. pp. 121–134. ACM (2007)
4. Ernst, E.: gbeta: A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance. Ph.D. thesis, University of Aarhus, Denmark (1999)
5. Flanagan, C.: Hybrid type checking. In: POPL. pp. 245–256. ACM (2006)
6. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: PLDI. pp. 1–12. ACM (2002)

7. Gasiunas, V., Mezini, M., Ostermann, K.: Dependent classes. In: OOPSLA. pp. 133–152. ACM (2007)
8. Gay, S., Vasconcelos, V.T., Ravara, A., Gesbert, N., Caldeira, A.Z.: Modular session types for distributed object-oriented programming. In: POPL. pp. 299–312. ACM (2010)
9. Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., Wang, Y.: Cyclone: A safe dialect of c. In: USENIX Annual Technical Conference, General Track. pp. 275–288. USENIX (2002)
10. Martin-Löf, P.: Intuitionistic type theory. Bibliopolis-Napoli (1984)
11. Nystrom, N., Qi, X., Myers, A.C.: J&: nested intersection for scalable software composition. In: OOPSLA. pp. 21–36. ACM (2006)
12. Nystrom, N., Saraswat, V., Palsberg, J., Grothoff, C.: Constrained types for object-oriented languages. In: OOPSLA. pp. 457–474. ACM (2008)
13. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the scala programming language. Tech. Rep. 001, EPFL (2006)
14. Odersky, M., Zenger, M.: Scalable component abstractions. In: OOPSLA. pp. 41–57. ACM (2005)
15. Sheard, T.: Languages of the future. In: OOPSLA. pp. 116–119. ACM (2004)
16. Smith, F., Walker, D., Morrisett, J.G.: Alias types. In: ESOP. pp. 366–381. Springer (2000)
17. Vasconcelos, V.T.: 9th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Web Services (SFM 2009), LNCS, vol. 5569, chap. Fundamentals of Session Types, pp. 158–186. Springer (2009)
18. Xi, H.: Imperative programming with dependent types. In: LICS. pp. 375–387. IEEE Computer Society (2000)
19. Xi, H.: Facilitating program verification with dependent types. In: SEFM. pp. 72–81. IEEE Computer Society (2003)
20. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: PLDI. pp. 249–257. ACM (1998)
21. Xi, H., Pfenning, F.: Dependent types in practical programming. In: POPL. pp. 214–227. ACM (1999)
22. Zenger, C.: Indexed types. Theoretical Computer Science - Elsevier 187(1-2), 147–165 (1997)