Safe Arrays via Regions and Dependent Types

Christian Grothoff[1] Jens Palsberg[1] Vijay Saraswat[2]

[1] UCLA Computer Science Department
University of California, Los Angeles
christian@grothoff.org, palsberg@ucla.edu
[2] IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
vsaraswa@us.ibm.com

Abstract. Arrays over *regions* of points were introduced in ZPL in the late 1990s and later adopted in Titanium and X10 as a means of simplifying the programming of high-performance software. A region is a set of points, rather than an interval or a product of intervals, and enables the programmer to write a loop that iterates over a region. While convenient, regions do not eliminate the risk of array bounds violations. Until now, language implementations have resorted to checking array accesses dynamically or to warning the programmer that bounds violations lead to undefined behavior. In this paper we show that a type system for a language with arrays over regions can guarantee that array bounds violations cannot occur. We have developed a core language and a type system, proved type soundness, settled the complexity of the key decision problems, implemented an X10 version which embodies the ideas of our core language, written a type checker for our X10 version, and experimented with a variety of benchmark programs. Our type system uses dependent types and enables safety without dynamic bounds checks.

1 Introduction

Type-safe languages allow programmers to be more productive by eliminating such difficult-to-find errors as memory corruption. However, type safety comes at a cost in runtime performance due to the need for dynamic safety checks. Traditionally, compilers use static analysis to eliminate some of these checks; in contrast, our work uses a more powerful type system based upon dependent types to prove the program safe in the absence of such checks. Our type system allows the programmer to provide intuitive annotations which allow the compiler to deduce the desired safety properties. Our work is heavily influenced by X10 [6], a new type-safe object-oriented programming language for distributed high-performance computing. In X10, array accesses are required to be not only in-bounds but also local with respect to the place of execution. This data locality aspect is important for distributed algorithms running on high-performance clusters where remote data access can be costly. Eliminating safety checks for array accesses is important; array accesses are among the most frequent operations in scientific applications, and thus their performance is critical. Experiments show

that simple bounds checks in languages like Java can cause performance hits of up to a factor of two. Dynamic checks can be even more costly in X10, since arrays are allowed to be sparse and distributed.

Modern high performance computing (HPC) languages must contain a rich language for arrays, a dominant data-structure in the HPC space. Chamberlain et al. [4,3,5,7] proposed regions as a construct for specifying array operations in languages for parallel programming. A region is a set of points, rather than an interval or a product of intervals, and enables the programmer to define an array over a region and to write a loop that iterates over a region. Regions were later adopted in Titanium [9] and X10 [6]; in X10, a regions is a first-class value. All of ZPL, Titanium, and X10 provide the programmer with a rich algebra of region operators to manipulate arrays. A programmer can use regions to specify computations on dense or sparse, multidimensional and hierarchical arrays. While convenient, regions do not eliminate the risk of array bounds violations. Until now, language implementations have resorted to checking array accesses dynamically or to warning the programmer that bounds violations lead to undefined behavior. For performance and productivity, we prefer that array computations are statically checked to be safe.

We have developed an extension of the type system for X10 that allows the programmer to express that accesses to X10 arrays are both in-bounds and place-local. Our type system enables the programmer to use concise type annotations to provide useful documentation that allows the compiler to eliminate all safety checks, resulting in faster, statically-checked code. Code type-checks if accesses are performed in a context in which the index can be statically established to be in the region over which the array is defined. X10's region-based iterators such as `for` and `foreach` often provide such a context. For instance in the statement `for (x : r) s` it is the case that within `s` one may assume that `x` lies in the region `r`. The type system uses the operations of the region algebra as a high-level abstraction that it can exploit for its reasoning. This is in contrast to previous work which used decision procedures based on Pressburger Arithmetic in order to statically eliminate checks. Using the new type system requires programmers to write code in such a way that it explicitly uses regions and region operations instead of the traditional integer arithmetic. However, this is a small price to pay; in our experience, refactoring programs to use regions often results in code that is easier to understand and more generic.

We have formalized the core of our type system in the context of an applied dependently-typed lambda calculus [13]. We have proved type soundness and settled the complexity of the key decision problems. We will illustrate the type system with various examples. We have implemented the type system in an experimental compiler for X10, XTC-X10, and type checked a variety of programs.

Our type system is inspired by that of Xi and Pfenning [16,17]. Like Xi and Pfenning, we use dependent types to avoid array-bounds checks. Xi and Pfenning use a decision procedure based on Pressburger arithmetic [12] in order to show the safety of array accesses. In contrast to Xi and Pfenning's language and type system, we study a programming model and type system based on regions. Our

type system uses types that are parameterized over regions. Operations on region values are mapped to corresponding operations on region types. The mapping is defined such that subset relations for regions values corresponds to subtyping of their respective types. Establishing that an index is in-bounds for a particular array is equivalent to establishing that region over which the index may range is a subset of the region over which the array is defined. A subtyping relationship between the respective region types implies the desired subset relationship and can thus be used to statically prove the safety of the access.

An alternative to using types to eliminate bounds checks is the use of static analysis. Early work on using static analysis to eliminate bounds checks investigated the use of theorem proving [14] to eliminate checks. Our work is related in that we use types to guide a decision procedure, a technique that is also used in proof carrying code [8]. For just-in-time compiled languages such as Java where compile time is crucial, the ABCD algorithm [2] describes a light-weight analysis based on interval constraints that is capable of eliminating on average 45% of the array bounds checks. However, the results range from 0 to 100% for the various individual benchmarks, which may make it hard for programmers to write code that achieves consistently good performance.

When speed is of utmost concern, a language designer may decide to not require any bounds checks altogether. For example, the reference manual for Titanium [9], a modern language for high-performance computing, defines that operations which cause bounds violations result in the behavior of the rest of the program being *undefined*. The semantics of our core language is similar: a bounds violation results in the semantics getting stuck. The contribution of our paper is a type system which guarantees statically that bounds violations cannot occur. Thus, our type system enables us to have both safety and high performance.

2 Example Programs

We will present our core language and type system via six example programs which we show in Figure 1. The first five example programs all type check, while the sixth program (`shift`) does not type check.

The function `init` initializes all points in an array to 1. The function `init` takes two arguments, namely a region α and an array over region α. The use of the dependent type α makes `init` polymorphic: `init` can initialize any array without the need for any bounds checking. The expression `a.reg` has type `reg` α and `p` which ranges over `a.reg` has type `pt` α. At the time of the assignment to `a[p]`, we have that the type of `p` matches the type of the region of `a`.

The function `partialinit` allows partial initialization of an array. It takes two extra arguments, namely a region value `x` and a corresponding region type variable β which represents the same region as a type. The body of `partialinit` initializes those points in the argument array which can be found in the region `x`. The argument β comes with the constraint $\beta \subseteq_t \alpha$, which means that the region β must be a subset of the region α. This constraint is necessary to type check that the assignment to `a[p]` is safe; we have that the type of `p` is β

which according to the constraint a subset of α, thus proving that the partial initialization can be performed safely. The call `partialinit <reg 0:9>(new int[0:9])<reg 1:8>(reg 1:8)` is a good example of the kind of reasoning that the programmer has to do when programming directly in the core language; the call satisfies the constraint $\beta \subseteq_t \alpha$ because $[0:8] \subseteq [0:9]$.

The function `copy` takes two arrays `a` and `b` with regions α and β, respectively. The body of `copy` copies elements from `b` to `a`, but only for common points. The type of `p` is $\alpha \cap_t \beta$, and each of the array accesses `a[p]` and `b[p]` will type check because $\alpha \cap_t \beta \subseteq \alpha$ and $\alpha \cap_t \beta \subseteq \beta$.

The function `partialcopy` is a variation of `copy` which takes two extra arguments which specify a subset of the intersection of the regions of `a` and `b`. Like for `copy`, the subset relationship between the type of `x` and the regions of `a` and `b` enables the type checker to prove that bounds checks are not required.

The function `expand` takes an array `a` and region `x`, where `x` must be a superset of the region of `a`, and creates and returns a new array `b` over the region `x`. The function `expand` partially initializes the new array `b` with values from `a` at overlapping points. `expand` is interesting in that it highlights the importance of keeping upper and lower bounds for the region of arrays during type checking.

The function `shiftleft` takes an argument `a` with region α and shifts all elements one position to the left, while leaving the rightmost element unchanged. In more detail, `shiftleft` first creates an inner region of α shifting all elements of α by one to the right $(\alpha + 1)$ and then intersecting the result with α. If α is simply an interval, this effectively removes the first element from α. Then `shiftleft` proceeds with doing `a[p-1] = a[p]` for each point `p` in the inner region. The inner region has type `reg` $(\alpha + 1) \cap_t \alpha)$. The expression `p-1` is always within the region of `a` because `p-1` has type `pt` $((\alpha + 1) \cap_t \alpha) - 1)$ and therefore also, via subtyping, the type `pt` α (because $+1$ and -1 cancel each other out). Similarly, the expression `p` is always within the region of `a` because `p` has type `pt` $(\alpha + 1) \cap_t \alpha)$ and therefore also, again via subtyping, the type `pt` α

The program `shift` is a small variation of `shiftleft` that contains a bug which would result in an array bounds violation – and that consequently does not type check. The problem with `shift` is that the array access `a[p+1]` will be out of bounds when `p` reaches the end of the array.

3 The Core Language

We now present the syntax, semantics, and type system of our core language. In the Appendix of the full version of the paper (available from our webpage), we prove type soundness using the standard technique of Nielson [11] and others that was popularized by Wright and Felleisen [15].

Syntax. We use c to range over integer constants, d to range over region constants, l to range over array labels drawn from a set `Label`, p to range over point constants, x to range over variable names, and α to range over region-variable names. In our core language, points are integers, and we will occasionally

```
let init = lam α.λa:int[α]. for (p in a.reg){ a[p]=1 }
in init<reg 0:9>(new int[0:9])
init : Πα. int[α] → int


let partialinit = lam α.λa:int[α].lam β:(β ⊆t α).λx:reg β
                          for (p in x){ a[p]=1 }
in partialinit <reg 0:9>(new int[0:9])<reg 1:8>(reg 1:8)
partialinit : Πα.int[α] → (Πβ:(β ⊆t α).reg β → int)


let copy = lam α.λa:int[α].lam β.λb:int[β].
                          for (p in (a.reg ∩s b.reg)){a[p]=b[p]}
in copy<reg 0..7>(new int[0..7])<reg 3..10>(new int[3..10])
copy : Πα.int[α] → (Πβ.int[β] → int)


let partialcopy =
       lam α.λa:int[α].lam β.λb:int[β].lam γ:(γ ⊆t α ∩t β).λx:(reg γ).
                          for (p in x){ a[p] = b[p] }
in partialcopy<reg 0..7>(new int[0..7])<reg 3..10>(new int[3..10])
                                          <reg 4..6>(reg 4..6)
partialcopy :
      Πα.int[α] → (Πβ.int[β] → (Πγ:(γ ⊆t α ∩t β).(reg γ → int))


let expand = lam α.λa:int[α].lam β:(α ⊆t β).λx:(reg β).
                          let b = new int[x]
                          in { for (p in a.reg){ b[p] = a[p] } ; b }
in expand<reg 3..7>(new int[3..7])<reg 0..10>(int[0..10])
expand : Πα.int[α] → (Πβ:(α ⊆t β).reg β → int[β])


let shiftleft = lam α.λa:int[α].
                          let inner = (α + 1) ∩s α
                          in { for (p in inner) { a[p-1] = a[p] } }
in shiftleft<reg 3..7>(new int[3..7])
shiftleft : Πα.int[α] → int


let shift = lam α.λa:int[α].
                          let inner = (α + 1) ∩s α
                          in { for (p in inner) { a[p+1] = a[p] } }
in ...
```

Fig. 1. Example programs

write a point constant as c. For shifting a region by a constant we use the notation $\{c_1, \ldots, c_n\} + c = \{c_1 + c, \ldots, c_n + c\}$.

$$
\begin{array}{lll}
\texttt{(Type)} & t & ::= \texttt{int} \mid \texttt{pt}\ r \mid \texttt{reg}\ r \mid t[r] \mid t \to t \mid \Pi\alpha : \varphi.t \\
\texttt{(Region)} & r & ::= \alpha \mid d \mid r \cup_t r \mid r \cap_t r \mid r \setminus_t r \mid r +_t c \\
\texttt{(Constraint)} & \varphi & ::= \texttt{true} \mid r \subseteq_t r \mid \varphi \wedge \varphi \\
 & & \\
\texttt{(Value)} & v & ::= c \mid p \mid d \mid l \mid \lambda x : t.e \mid \texttt{lam}\ \alpha : \varphi.e \\
\texttt{(Expression)} & e & ::= v \mid x \mid e_1\ e_2 \mid e\texttt{<}\rho\texttt{>} \\
 & & \quad \mid \texttt{new}\ t[e] \mid e_1[e_2] \mid e_1[e_2] = e_3 \mid e\texttt{.reg} \\
 & & \quad \mid e_1 \cup_s e_2 \mid e_1 \cap_s e_2 \mid e \setminus_s r \mid e +_s c \mid e +\!\!+_s c \\
 & & \quad \mid \texttt{for}\ (x\ \texttt{in}\ e_1)\{e_2\} \mid e_1; e_2 \\
\texttt{(Region Arg)} & \rho & ::= \alpha \mid d
\end{array}
$$

The language has six data types, namely integers, points, regions, arrays, functions, and dependently-typed functions. The types of points, regions, and arrays are defined in terms of set expressions that involve constants, variables, union, intersection, set difference, and the unusual $r +_t c$. Given an interpretation of the variables, a set expression denotes a set, that is, a region. The type of a point is a region that contains that point. The type of a region is a singleton type consisting of that region itself. A dependently-typed function $\texttt{lam}\ \alpha : \varphi.e$ has its argument constraint by the set constraint φ; its type is $\Pi\alpha : \varphi.t$. The idea is that if we call $\texttt{lam}\ \alpha : \varphi.e$ with a region d, then the set constraint φ must be satisfied for $\alpha = d$ for the call to be type correct.

The expression language contains syntax for creating and calling functions, for creating, accessing, and updating arrays, for computing with regions, and for iterating over regions. The expression $e\texttt{.reg}$ returns the region of an array. The expression $e +\!\!+_s c$ adds a constant c to the point to which e evaluates. The expression $e +_s c$ adds a constant to each of the points in the region to which e evaluates.

We need the set operators to work both on types, expressions, and actual sets. In order to avoid confusion, we give each operator on types the subscript t, on expressions the subscript s, and on sets no subscript at all.

In the example programs earlier in the paper, we used the syntactic sugar $\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ \{\ e'\ \}$ in order to represent $(\lambda x.e')e$.

Semantics. Our core language has a small-step operational semantics. We use H to range over heaps: $H \in \texttt{Label} \to \texttt{Int} \to \texttt{Value}$.

We use $\mathcal{D}(H)$ to denote the domain of a partial function H. A state in the semantics is a pair (H, e). We say that (H, e) can *take a step* if we have H', e' such that $(H, e) \rightsquigarrow (H', e')$, using the rules below. We say that (H, e) is *stuck* if e is not a value and (H, e) cannot take a step. We say that (H, e) *can go wrong* if we have H', e' such that $(H, e) \rightsquigarrow^* (H', e')$ and (H', e') is stuck.

We assume a function `default` which maps a closed type to a value with the property that $\Psi; \varphi; \Gamma \vdash \texttt{default}(t) : t$ for a Ψ that contains suitable definitions of the labels used in $\texttt{default}(t)$, and for any φ and Γ. The idea is that we will use $\texttt{default}(t)$ as the initial value at all points in an array with element type t. While we can easily define examples of such a function `default`, we will not show a specific one, simply because all we need to know about it is the property $\Psi; \varphi; \Gamma \vdash \texttt{default}(t) : t$.

In order to specify the execution order for the loop construct, Rule ((12)) uses a function $\text{order}(\{c_1, \ldots, c_n\}) = \langle c_1, \ldots, c_n \rangle$, where $c_1 < \ldots < c_n$.

Below we show the main rules of the call-by-valae semantics; the rules are mostly standard and the full set of rules can be found in the full version of the paper. The key rules (4) and (5) both have the side condition that $l \in \mathcal{D}(H)$ and $p \in \mathcal{D}(H(l))$. The condition $p \in \mathcal{D}(H(l))$ is the array-bounds check; p must be in the region of the array. If the side condition is not met, then the semantics will get stuck. Notice that in Rule (7) we evaluate the syntactic expression $d_1 \cup_s d_2$ to the value $d_1 \cup d_2$. Rule (12) unrolls the for loop and replaces the loop variable with an appropriate point in each copy of the body of the loop.

$$(H, (\lambda x.e)v) \rightsquigarrow (H, e[x := v]) \tag{1}$$

$$(H, (\texttt{lam}\ \alpha : \varphi.v)\texttt{<}\rho\texttt{>}) \rightsquigarrow (H, v[\alpha := \rho]) \tag{2}$$

$$(H, \texttt{new}\ t[d]) \rightsquigarrow (H[l \mapsto \lambda p \in d.\texttt{default}(t)], l) \quad \text{where } l \text{ is fresh} \tag{3}$$

$$(H, l[p]) \rightsquigarrow (H, H(l)(p)) \quad \text{if } l \in \mathcal{D}(H) \text{ and } p \in \mathcal{D}(H(l)) \tag{4}$$

$$(H, l[p] = v) \rightsquigarrow (H[l \mapsto (H(l))[p \mapsto v]], v) \quad \text{if } l \in \mathcal{D}(H) \text{ and } p \in \mathcal{D}(H(l)) \tag{5}$$

$$(H, l.\texttt{reg}) \rightsquigarrow (H, \mathcal{D}(H(l))) \quad \text{if } l \in \mathcal{D}(H) \tag{6}$$

$$(H, d_1 \cup_s d_2) \rightsquigarrow (H, d_1 \cup d_2) \tag{7}$$

$$(H, d_1 \cap_s d_2) \rightsquigarrow (H, d_1 \cap d_2) \tag{8}$$

$$(H, d_1 \setminus_s d_2) \rightsquigarrow (H, d_1 \setminus d_2) \tag{9}$$

$$(H, d +_s c) \rightsquigarrow (H, d + c) \tag{10}$$

$$(H, p +\!+_s c) \rightsquigarrow (H, p + c) \tag{11}$$

$$(H, \texttt{for}\ (x\ \texttt{in}\ d)\{e\}) \rightsquigarrow (H, e[x := c_1]; \ldots; e[x := c_n]) \quad \text{where } \text{order}(d) = \langle c_1, \ldots, c_n \rangle \tag{12}$$

$$(H, v; e) \rightsquigarrow (H, e) \tag{13}$$

Satisfiability and Entailment. We use ρ to range over mappings from variables to sets. We say that ρ *satisfies* a constraint φ if for all $r_1 \subseteq_t r_2$ in φ we have $r_1\rho \subseteq r_2\rho$. We say that a constraint φ is *satisfiable* if there exists a satisfying assignment for φ.

We say that a constraint is *valid* if all variable assignments satisfy the constraint. We say that φ *entails* φ' if the implication $\varphi \Rightarrow \varphi'$ is valid, and write $\varphi \models \varphi'$.

The *satisfiability problem* is: given a constraint φ, is φ satisfiable? The *entailment problem* is: given two constraints φ, φ', is $\varphi \models \varphi'$ true?

For our notion of set constraints, the satisfiability problem is NP-complete. To see that, first note that already for the fragment of set constraints without the $r +_t c$ expression, the satisfiability problem is NP-hard [1]. Second, to show that the satisfiability problem is in NP we must first argue that we only need to consider sets of polynomial size; we can then guess a satisfying assignment and check that assignment in polynomial time. Let us first *flatten* the constraint by, for each subexpression e, replacing e with a variable α and adding an extra conjunct $\alpha = e$. In the flattened constraint, let n be the number of variables

in the constraint, let u be the largest integer mentioned in any region constant in the constraint, and let k be the largest c used in any $e +_s +_s$ or $e ++_s ++_s$ expression in the constraint. In any solution, an upper bound on the largest integer is $n \times u \times k$. To see that notice that either the constraint system is not satisfiable or else the biggest integer we can construct is by a sequence of +k operations, each involving a different variable. Similarly, we have have a lower bound on the smallest integer used in any solution.

For our notion of set constraints, the entailment problem is co-NP-complete. To see that, first note that $\varphi \models \varphi'$ if and only if $\varphi \wedge \neg\varphi'$ is unsatisfiable. For the fragment of cases where $\varphi' = \texttt{false}$ we have that the entailment problem is the question of given φ, is φ unsatisfiable, which is co-NP-complete. So, the full entailment problem is co-NP-hard. Second, note that the entailment problem is in co-NP; we can we can easily collect the set of all points mentioned in the constraints, then guess an assignment, and finally check that the assignment is not a satisfying assignment, in polynomial time.

Heap Types. We use Ψ to range over maps from array labels to types of the form $t[d]$. We use the judgment $\models H : \Psi$ which holds if (1) $\mathcal{D}(H) = \mathcal{D}(\Psi)$ and (2) if for each $l \in \mathcal{D}(H)$ we let $t[d] = \Psi(l)$, then $\mathcal{D}(H(l)) = d$ and for each $p \in \mathcal{D}(H(l))$ we have $\Psi; \varphi; \Gamma \vdash H(l)(p) : \ t$. We write $\Psi \lhd \Psi'$ if $\mathcal{D}(\Psi) \subseteq \mathcal{D}(\Psi')$ and Ψ, Ψ' agree on their common domain.

Type Rules. A type judgment is of the form $\Psi; \varphi; \Gamma \vdash e : t$, which holds if it is derivable the rules in Figure 2. The type rules ensure that φ is satisfiable at every point in a type derivation of a type-correct program. In particular, Rule (22) contains the explicit conditions that $\varphi \wedge \varphi'[\alpha := \rho]$ are satisfiable. Rule 32 is a key type rule which says that to type check a loop $\texttt{for}\ (x\ \texttt{in}\ e_1)\{e_2\}$, we check that e_1 has a type $\texttt{reg}\ r$, and then assign x the type $\texttt{pt}\ r$ while checking e_2. The type rules for array lookup, Rule (24), and array update, Rule (25), ensure that the point is in bounds by requiring that the type of the point is a region which is a subset of the region of the array.

4 Experiments

We have implemented the presented type system in XTC-X10, a prototype implementation of an X10 variant that is publically available on our webpage[3]. The implementation is able to type-check all of the example programs from Section 2. Our prototype has some syntactic differences compared to IBM's reference implementation of X10, mostly because it adds additional features such as region types, operator overloading and generics. We have converted some of the benchmarks from the IBM reference implementation of X10 to work with XTC-X10. Since those benchmarks were originally written for languages without regions, most of the work is put into making the code use regions.

IBM's reference implementation of X10 contains a parallel implementation of the SOR benchmark, which illustrates the necessary conversion to regions. The

[3] http://grothoff.org/christian/xtc/x10/

$$\Psi;\varphi;\Gamma \vdash c:\ \texttt{int} \tag{14}$$

$$\Psi;\varphi;\Gamma \vdash p:\ \texttt{pt}\ d \quad (\text{where } p \in d) \tag{15}$$

$$\Psi;\varphi;\Gamma \vdash d:\ \texttt{reg}\ d \tag{16}$$

$$\Psi;\varphi;\Gamma \vdash l:\ \Psi(l) \tag{17}$$

$$\frac{\Psi;\varphi;\Gamma[x:t_1] \vdash e:\ t_2}{\Psi;\varphi;\Gamma \vdash \lambda x:t_1.e:\ t_1 \to t_2} \tag{18}$$

$$\frac{\Psi;\varphi \wedge \varphi';\Gamma \vdash e:\ t}{\Psi;\varphi;\Gamma \vdash \texttt{lam}\ \alpha:\varphi'.e:\ \Pi\alpha:\varphi'.t} \tag{19}$$

$$\Psi;\varphi;\Gamma \vdash x:\ \Gamma(x) \tag{20}$$

$$\frac{\Psi;\varphi;\Gamma \vdash e_1:\ t_1 \to t_2 \quad \Psi;\varphi;\Gamma \vdash e_2:\ t_1}{\Psi;\varphi;\Gamma \vdash e_1\ e_2:\ t_2} \tag{21}$$

$$\frac{\Psi;\varphi;\Gamma \vdash e:\ \Pi\alpha:\varphi'.t \quad \varphi \wedge \varphi'[\alpha := \rho] \text{ is satisfiable}}{\Psi;\varphi;\Gamma \vdash e\texttt{<}\rho\texttt{>}\ :\ t[\alpha := \rho]} \tag{22}$$

$$\frac{\Psi;\varphi;\Gamma \vdash e:\ \texttt{reg}\ r}{\Psi;\varphi;\Gamma \vdash \texttt{new}\ t[e]:\ t[r]} \tag{23}$$

$$\frac{\Psi;\varphi;\Gamma \vdash e_1:\ t[r] \quad \Psi;\varphi;\Gamma \vdash e_2:\ \texttt{pt}\ r' \quad \varphi \models r' \subseteq_t r}{\Psi;\varphi;\Gamma \vdash e_1[e_2]:\ t} \tag{24}$$

$$\frac{\Psi;\varphi;\Gamma \vdash e_1:\ t[r] \quad \Psi;\varphi;\Gamma \vdash e_2:\ \texttt{pt}\ r' \quad \varphi \models r' \subseteq_t r \quad \Psi;\varphi;\Gamma \vdash e_3:\ t}{\Psi;\varphi;\Gamma \vdash e_1[e_2] = e_3:\ t} \tag{25}$$

$$\frac{\Psi;\varphi;\Gamma \vdash e:\ t[r]}{\Psi;\varphi;\Gamma \vdash e.\texttt{reg}\ :\ \texttt{reg}\ r} \tag{26}$$

$$\frac{\Psi;\varphi;\Gamma \vdash e_1:\ \texttt{reg}\ r_1 \quad \Psi;\varphi;\Gamma \vdash e_2:\ \texttt{reg}\ r_2}{\Psi;\varphi;\Gamma \vdash e_1 \cup_s e_2:\ \texttt{reg}\ r_1 \cup_t r_2} \tag{27}$$

$$\frac{\Psi;\varphi;\Gamma \vdash e_1:\ \texttt{reg}\ r_1 \quad \Psi;\varphi;\Gamma \vdash e_2:\ \texttt{reg}\ r_2}{\Psi;\varphi;\Gamma \vdash e_1 \cap_s e_2:\ \texttt{reg}\ r_1 \cap_t r_2} \tag{28}$$

$$\frac{\Psi;\varphi;\Gamma \vdash e_1:\ \texttt{reg}\ r_1 \quad \Psi;\varphi;\Gamma \vdash e_2:\ \texttt{reg}\ r_2}{\Psi;\varphi;\Gamma \vdash e_1 \setminus_s e_2:\ \texttt{reg}\ r_1 \setminus_t r_2} \tag{29}$$

$$\frac{\Psi;\varphi;\Gamma \vdash e:\ \texttt{reg}\ r}{\Psi;\varphi;\Gamma \vdash e +_s c:\ \texttt{reg}\ r +_t c} \tag{30}$$

$$\frac{\Psi;\varphi;\Gamma \vdash e:\ \texttt{pt}\ r}{\Psi;\varphi;\Gamma \vdash e +\!+_s c:\ \texttt{pt}\ r +_t c} \tag{31}$$

$$\frac{\Psi;\varphi;\Gamma \vdash e_1:\ \texttt{reg}\ r \quad \Psi;\varphi;\Gamma[x:\ \texttt{pt}\ r] \vdash e_2:\ t}{\Psi;\varphi;\Gamma \vdash \texttt{for}\ (x\ \texttt{in}\ e_1)\{e_2\}:\ t} \tag{32}$$

$$\frac{\Psi;\varphi;\Gamma \vdash e_1:\ t_1 \quad \Psi;\varphi;\Gamma \vdash e_2:\ t_2}{\Psi;\varphi;\Gamma \vdash e_1;e_2:\ t_2} \tag{33}$$

Fig. 2. Type rules

SOR code iterates over the inner region of a two dimensional array and updates each point with a weighted average over the point and its immediate neighbors. The parallel version interleaves processing odd and even rows. The original X10 code that works with the IBM reference implementation is shown below.

```
void sor(double omega, double[.] G, int iter) {
  int M = G.distribution.region.rank(0).size();
  int N = G.distribution.region.rank(1).size();
  final double omega_over_four = omega * 0.25;
  final double one_minus_omega = 1.0 - omega;
  int Mm1 = M-1; int Nm1 = N-1;
  for (int p=0; p<iter; p++) {
    for (int o = 0; o <= 1 ; o++) {
      final int offset = o;
      finish foreach (point[ii] : 0: ((Mm1-1)-(1+offset))/2) {
        int i = 2*ii + 1 + offset;
        for (int j=1; j<Nm1; j++)
          G[i,j] = omega_over_four * (G[i-1,j] + G[i+1,j] + G[i,j-1]
           + G[i,j+1]) + one_minus_omega * G[i,j];
      } } } }
```

We have converted the code above to work with our prototype and to make better use of regions. Aside from being statically checkable, the rewritten SOR benchmark is also more generic. Where the original code required a rectangular array to be used, the new code works with two dimensional arrays of any shape. It uses the `project` operator on regions to compute the extent of the shape in each dimension. This is necessary since the outer `foreach` needs to parallelize processing of rows but not of columns and thus cannot simply iterate over the entire array. Note that the inner `for` loop iterates over a single row by keeping the first dimension fixed to `i`. Intersecting the selected row with `inner` allows the type system to verify that the accesses are in bounds – and ensures that the program works correctly for non-rectangular arrays. Note that the implementation automatically infers region types for local variables (particularly the type of `ij`) and therefore no region type annotations are required for this benchmark.

```
const point NORTH = new point(1,0); const point WEST  = new point(0,1);
void run(double omega, Array<double> G, int iter) {
  region outer = G.dist.reg;
  region inner = outer & (outer +  WEST) & (outer -  WEST)
                       & (outer + NORTH) & (outer - NORTH);
  double omega_over_four = omega * 0.25d;
  double one_minus_omega = double.ONE - omega;
  region d0 = inner.project(0); region d1 = inner.project(1);
  if (d1.size() == 0) return;
  int d1MIN = ((point)d1.min())[0]; int d1MAX = ((point)d1.max())[0];
  for (it[off] : [1:iter*2]) {
    int om2 = off % 2;
    finish foreach (pnt[i] : d0) {
      if (i % 2 == om2)
        for (ij : inner & [i,d1MIN:i,d1MAX])
          G[ij] = omega_over_four * (G[ij-NORTH] +
                    G[ij+NORTH] + G[ij-WEST] + G[ij+WEST])
                + one_minus_omega * G[ij];
    } } }
```

The current XTC-X10 prototype uses an interpreter written in Java to execute the generated code. As a result, giving meaningful performance numbers based on XTC-X10 is not possible at this time. However, the performance performance impact of eliminating bounds checks can be estimated by crudely disabling all checks in an existing implementation. Disabling dynamic checks in IBM's X10 reference implementation can improve performance by a factor of 3 for some benchmarks. Similarly, disabling bounds checks in IBM's Java Virtual Machine can improve performance of benchmarks with intensive array access patterns by a factor of 1.75 (Personal communication with Christopher Donawa.)

The number of bounds checks usually differs dramatically between applications. The following table lists total execution times and number of bounds checks (total and per second of execution time) for various X10 benchmarks (adapted from the Java Grande Benchmark Suite).

Benchmark	Total checks	runtime (s)	checks/s
Crypt	3800520	15.6	245194
Moldyn	308416	12.9	23908
RayTracer	900	3.8	236
CG	830899147	355.0	2340560

5 Future Work and Conclusion

Many of the benchmarks available for X10 cannot be checked within the core language presented in this paper. The dominant reason is X10's partitioned global address space. In X10, arrays can be distributed across places; for those arrays accesses must not only be in bounds but also local to the current place. The presented type system can be made to handle those cases by extending the operator language to cover operations that manipulate distributions (mappings of regions to places). Other benchmarks cannot be checked because their index operations involve intricate integer arithmetic which cannot be mapped to regions in obvious ways.

We are investigating the use of additional region operators that might enable us to check more of these benchmarks. However, adding additional operators further complicates the decision procedures for satisfiability and/or entailment.

Out-of-bounds array accesses remain a leading cause of security problems and, according to the National Vulnerability Database [10], buffer overflows are responsible for 233 out of 863 CERT technical alerts or vulnerability notes in the years 2004 and 2005. Our type system can guarantee that no out-of-bounds array accesses will happen, thereby also obviating the need for doing dynamic checks of such accesses.

Acknowledgments. We thank Christopher Donawa and Rajkishore Barik for providing us with performance data on the cost of bounds-checking in Java and X10. Vivek Sarkar wrote the original X10 version of the SOR benchmark.

References

1. Alexander Aiken, Dexter Kozen, Moshe Y. Vardi, and Edward L. Wimmers. The complexity of set constraints. In *CSL*, pages 1–17, 1993.
2. Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, 2000.
3. Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, November 2001.
4. Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL: A machine independent programming language for parallel computers. *IEEE Transactions on Software Engineering*, 26(3):197–211, March 2000.
5. Bradford L. Chamberlain and Lawrence Snyder. Array language support for parallel sparse computation. In *Proceedings of the ACM International Conference on Supercomputing*, 2001.
6. Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Vijay A. Saraswat, Vivek Sarkar, and Christoph Von Praun. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 519–538. ACM SIGPLAN, 2005.
7. Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. High-level language support for user-defined reductions. *Journal of Supercomputing*, 23(1), 2002.
8. Necula G.. Proof-Carrying Code. *Proceedings of POPL'97, 24th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, 106–119, 1997.
9. P. N. Hilfinger, Dan Bonachea, Kaushik Datta, David Gay, Susan Graham, Ben Liblit, Geoff Pike, Jimmy Su, and Katherine Yelick. Titanium language reference manual. Technical report, U.C. Berkeley, 2005.
10. Peter Mell, Karen Kent, Ashish Goel, Ellery Horton, Tanyette Miller, Robert Chang, Michael Reilly, and Kathy Ton-Nu. National vulnerability database. http://nvd.nist.gov/, 2006.
11. Flemming Nielson. The typed lambda-calculus with first-class processes. In *Proceedings of PARLE'89*, pages 357–373, 1989.
12. William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, New York, NY, USA, 1991. ACM Press.
13. B. Pierce. Types and Programming Languages. MIT Press, 2002.
14. Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 132–143, New York, NY, USA, 1977. ACM Press.
15. Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
16. Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of PLDI'98, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, 1998.
17. Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of POPL'99, 26th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 214–227, 1999.