

# A Type System for Distributed Arrays

Christian Grothoff

Dept. of Computer Science  
University of Denver  
christian@grothoff.org

Jens Palsberg

UCLA Computer Science Dept.  
University of California,  
Los Angeles  
palsberg@ucla.edu

Vijay Saraswat

IBM T.J. Watson Research Center  
P.O. Box 704, Yorktown Heights,  
NY 10598, USA  
vsaraswa@us.ibm.com

## Abstract

Multicore systems with non-uniform memory are of increasing importance in desktop and server computing. Such systems represent both potentially vast computing power and significant research problems in expressing and maximizing parallelism, which has prompted researchers to design new languages such as X10, Fortress, and Chapel. Common to these languages is support for location-aware programming, which gives the programmer control over the location of both data and computations. One primary data structure in these languages is the distributed array, which elevates the location of individual array elements to a language concept. A location-aware algorithm executing on a particular core must be designed to exploit the locality of the distributed array, because access to remote data is far slower than to local data. In this paper, we solve the open problem of statically checking whether such a program has achieved the desired locality of access.

We present a statically-typed core language in which a well-typed program only accesses local parts of distributed arrays and every array access is within the array bounds. The type system integrates dependent types and set constraints, with the key operation during type checking being constraint entailment. Type checking for this system is co-NP-complete. We have integrated our type system into the object-oriented language X10 and shown that for seven benchmarks, a programmer needs insert only a few casts to make a program type check.

**Categories and Subject Descriptors** D.3 [Software]: Programming Languages

**General Terms** languages, performance, verification

**Keywords** dependent types, non-uniform memory

## 1. Introduction

### 1.1 Background

According to Moore's Law, the number of transistors on an integrated circuit doubles every 18 months. In the past few years, one of the practical manifestations of Moore's Law has been that microprocessor chip packages have begun to integrate more than one execution unit, or *core* on a single die. As the physical limits of processor clockspeeds are beginning to be reached, chip manufacturers are now turning to multicore chips to keep performance increases on track.

Multicore systems are quickly becoming mainstream. For example, Intel released its first dual-core processor, namely the Intel Pentium processor Extreme Edition ([www.intel.com](http://www.intel.com)) in 2005; AMD offers quad-core Opteron processors ([multicore.amd.com](http://multicore.amd.com)); and IBM's Cell processor has one general-purpose core and eight specialized cores ([www.ibm.com](http://www.ibm.com)). Sun Microsystems' Niagara 2 processor will debut with support for 64 hardware threads, with 4 threads per core and 8 cores per processor (<http://www.sun.com/processors/niagara>).

The full utilization of the computing power of multicore systems is a major challenge, and for that and similar purposes, researchers have designed new programming languages such as X10 [9], Fortress [2], Chapel [5], Titanium [13], Co-Array Fortran [19], and ZPL [7]. Although these languages are highly different (e.g. X10 is derived from a Java-like language, Co-Array Fortran is a Fortran-like language, etc), they also have important similarities. For example, all of the first four of those languages share the notion of *distributed arrays*, where the location of individual array elements is elevated to a language concept. Distributed arrays are a key data structure in high performance computing. We will use X10 terminology for four basic notions associated with distributed arrays:

- a *point* is an array index,
- a *region* is a set of array indices,
- a *place* is a location where data is stored and code is executed, and
- a *distribution* is a mapping from regions to places.

Language	X10	Fortress	Chapel	Titanium	Co-Array Fortran	ZPL
Array index	point	index	index	point	index	index
Set of array indices	region	n/a	domain	domain	n/a	region
Location	place	region	locale	demesne	image	n/a
Array distribution	distribution	distribution	distribution	distribution	n/a	n/a

**Table 1.** Language terminology.

The notion of regions stems from ZPL where the idea is to allow the domain of an array to be any finite set of indices, not just intervals [6, 8, 7]. In their most general form, a programmer can use regions to specify dense, sparse, multidimensional, and even hierarchical arrays.

The notion of a distribution enables a programmer to specify, for example, that the elements of an array with indices  $\{1, 2, 3, 4\}$  will be distributed such that:

elements with indices 1,2 : will be allocated at place  $P_1$   
elements with indices 3,4 : will be allocated at place  $P_2$ .

Table 1 gives an overview of the terminology used in the six languages.

Distributed arrays are linked to locality-aware algorithms in which the code on a core primarily accesses the local part of an array. A locality-aware algorithm running on a multi-core computer with nonuniform memory favors access to local data, which is much faster, over access to remote data. For example, we may have code  $e_1$  running at place  $P_1$  and code  $e_2$  running at place  $P_2$ , and  $e_1$  will try to access only data at  $P_1$ , while  $e_2$  will try to access only data at  $P_2$ . X10 enables programmers to ensure data locality by providing constructs to programmatically shift the place of execution.

This leaves us with the question: did the implementer of the locality-aware algorithm get locality right? We focus on the X10 policy that all accesses to mutable data be place-local. We would prefer to check that all accesses are local *before* running the program and thereby give the locality-aware programmer as much help as possible.

**Problem:** Can we statically check that a program only accesses local parts of distributed arrays?

This problem has remained open until now.

## 1.2 Our Results

In this paper we present a statically-typed core language for computing with distributed arrays. The core language is a dependently-typed lambda calculus [3] with distributed arrays and place-shifting operations. We prove that a well-typed program can only access local parts of distributed arrays; we also prove that every array access is within the array bounds. Our proof of type soundness uses the standard technique based on proving preservation and progress.

Our distributed arrays are defined in terms of points, regions, places, and, for simplicity, a fixed distribution for

each region. The type system uses dependent types that are parameterized by points, regions, and places. The type system integrates dependent types and set constraints. The key operation during type checking is constraint entailment; type checking itself is co-NP-complete.

For each expression-level operator on points, regions, and places, we have a similar type-level operator. A type-level constraint is a conjunction of set constraints of the forms  $r_1 \subseteq_t r_2$  and  $\sigma \in_t r$ , where  $r_1, r_2, r$  are type-level set expressions,  $\sigma$  is a type-level point expression, and  $\subseteq_t$  and  $\in_t$  denote type-level subset inclusion and set membership, respectively.

To type check an array access expression  $y_1[y_2]$ , we must (among other things) check that the access is within bounds and to local data, that is, to data in the right place. Suppose we find that  $y_1$  is an array over a region-type  $r_1$ , and that  $y_2$  is a point in region-type  $r_2$ . We can then do the bounds check by checking that  $r_2$  is a subset of  $r_1$ , and we can do the place check by checking that the data is at the current place of computation. In our type system, the bounds check is a constraint entailment problem, while the place check is a type equivalence problem. In slogan form:

check of array access = bounds check + place check  
bounds check = constraint entailment  
place check = type equivalence

We have integrated our type system into a variant of X10. In our X10 version, a programmer can write a cast to signal to the compiler that a combined subset and locality check must be performed; all array accesses without explicit casts are guaranteed to be within bounds. We have shown that for seven benchmarks, a programmer needs insert only a few casts to make a program type check. We have also measured how many dynamic checks are executed as a result of those casts; compared to doing the checks for every array access, the reduction in checks is substantial.

Our choice of point-region-place algebra and form of set constraints is sufficient to cover several fundamental examples, as we will demonstrate below. Changes to the algebra or constraints would impact the complexity of the type checker, but would likely not change the style of the type soundness proof.

In summary, this paper makes four contributions:

- a core language for computing with distributed arrays,

- a type system that ensures locality of access and memory safety for distributed arrays,
- design and implementation of a variant of X10 which embodies our type system, and
- experimental results that show for a suite of seven benchmarks, a programmer need insert only a few casts to make a program type check.

### 1.3 Related Work

Liblet and Aiken [17] presented three type systems for distributed data structures that distinguish between local and global data. Our type system goes further by supporting distributed arrays and operations that shift the place of computation.

Xi and Pfenning [24, 25] presented a type system that uses dependent types to guarantee that all array accesses are in bounds. Xi and Pfenning work with arrays defined over intervals and use a decision procedure based on Pressburger arithmetic [21] in order to show the safety of array accesses. Inspired by Xi and Pfenning, our type system goes further by supporting distributed arrays.

Instead of types, one can use static analysis to eliminate array-bounds checks. Suzuki and Ishihata [22] used theorem proving to eliminate array-bounds checks. Our work is related in that our type checker relies on using decision procedures to settle subset and locality questions. For just-in-time compiled languages such as Java where short compile time is crucial, the ABCD algorithm [4] describes a lightweight analysis based on interval constraints that is capable of eliminating on average 45% of the array bounds checks. The results range from 0 to 100% for individual benchmarks and that may make it hard for programmers to write code that achieves consistently good performance. In contrast, the type system for our core language guarantees that all array accesses (without explicit casts) are within bounds.

For parallel languages without programmer-definable distributions, researchers have developed algorithms for automatically determining distributions that can reduce or eliminate nonlocal data access [10, 12, 16, 14, 11]. An emerging point of view inherent in X10, Fortress, and Chapel is that the definition of distributions is best left to the programmer.

When speed is of utmost concern, a language designer may decide to not require any bounds checks altogether. For example, the 2005 reference manual for Titanium [13] 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 violation of locality or an access out of bounds result in the semantics getting stuck. Our type system guarantees statically that such errors cannot occur. Thus, our type system enables us to have both memory safety and high performance.

**Rest of the paper.** In Section 2 we explain several example programs; in Section 3 we present our core language and

theoretical results, and in Section 4 we present our experimental results.

## 2. Example Programs

We will give a taste of our core language and type system via six example programs. The first five example programs all type check, while the sixth program does not. The programs are written to highlight aspects of the language and type system, and not necessarily to represent the most elegant coding style.

We use functions of the form  $\lambda^{\bullet}x : t.e$  which run at the place where they are defined. Our core language also has functions  $\lambda x : t.e$  which run at the place where they are called. In both cases,  $x$  is the name of the argument,  $t$  is the type of the argument, and  $e$  is the body which evaluates to the return value of the function.

Similarly, we use dependent expressions  $\text{lam}^{\bullet}\alpha : k.e$  for which the body will be evaluated at the place where the dependent expression was defined. Additionally, our core language has dependent expressions  $\text{lam } \alpha : k.e$  which are evaluated at the place they are called. In both cases,  $\alpha$  is the name of argument,  $k$  is the kind of the argument, and  $e$  is the body.

The difference between  $\lambda^{\bullet}x : t.e$  and  $\text{lam}^{\bullet}\alpha : k.e$  is that while both take a value as argument,  $\alpha$  can only be used as a *type*, while  $x$  can only be used as a *value*. The idea of using a value as a type is what makes dependent types powerful.

While two kinds of functions (and two kinds of dependent expressions) appear a bit excessive, we have found it convenient to have both; we have been unable to define one in terms of the other.

### 2.1 init

The function `init` initializes all elements of an array to 1. This example shows how to use (1) a region as a first-class value, (2) a `for`-loop over a region, and (3) the place-changing operation  $\text{at}(e_1)\{e_2\}$ .

```
let init = lam•α : region(true).λ•a:int[α].
    for (p in a.reg) {
        at(a.reg[@p]) { a[p]=1 } }
in init<0:9>(new int[0:9])
```

```
init: Πα : region(true). int[α] → int
```

The function `init` takes two arguments, namely a region  $\alpha$  and an array `a` over region  $\alpha$ . Notice that  $\alpha$  has kind `region(true)` which means that  $\alpha$  can be any region. Notice also that  $\alpha$  is used in the type of `a` such that `a` must be an array over region  $\alpha$ .

The use of the dependent type  $\alpha$  makes `init` polymorphic: `init` can initialize any array without the need for any bounds or place checking. In the body of the `let`-expression, we have the call of `init<0:9>(new int[0:9])`. The notation `0:9` denotes the region  $\{0, 1, \dots, 9\}$ . We use `<...>` to denote an argument to a dependent expression.

The expression `a.reg` denotes the region over which the array `a` is defined. The loop variable `p` ranges over points in `a.reg`.

The body of the `for`-loop is the expression `at(a.reg[@p]) { ... }`. The expression `a.reg[@p]` denotes the *place* of the element at point `p` in the region `a.reg`. Our core language uses a fixed distribution for each region; the languages that our core language models allow user-specified distributions.

The loop body `at(a.reg[@p]) { a[p]=1 }` does the computation of `a[p]=1` at the place denoted by `a.reg[@p]`. Given that the elements of `a` may be distributed on many places, the body of the `for`-loop will run on those places.

Let us consider the type checking process for each of the two array accesses `a.reg[@p]` and `a[p]`.

In the case of `a.reg[@p]` we have that `a` has type `int[α]` and that `a.reg` has type `reg α`. The type system gives `p` the type `pt(σ, α)`, where `α` is the region to which `p` belongs, and `σ` is a fresh type variable for which we have the set constraint  $\sigma \in_t \alpha$ . We use the subscript  $t$  to denote type-level operations. The type checker then does a bounds check by checking that the type of the region of `p` is a subset of the type of the region `a.reg`, which amounts to checking that  $\alpha$  is a subset of  $\alpha$ , which is true for all  $\alpha$ .

In the case of `a[p]=1` we must do the same bounds check as we did for `a.reg[@p]` and we must also do a place check. The place check verifies that the data is at the current place of computation. In the body of `at(a.reg[@p]) { ... }`, the place of computation is given by `a.reg[@p]` which has type `pl α[@t(σ, α)]`. That place is exactly the place of the data `a[p]` so the place check succeeds. Note that we need  $\sigma$  in the type of `p` to ensure that we find the type of the place for that specific point.

## 2.2 partialinit

The function `partialinit` initializes those elements of an array that are at a given place. This example shows how to use (1) a place as a first-class value and (2) the restrict operator  $e_1 \%_s e_2$ .

```
let partialinit = lam*γ:place.λ*h:pl γ.
  lam*α:region(true).λ*a:int[α].
  at(h) { for (p in a.reg %s h) { a[p]=1 } }
in partialinit <P>(P)<0:9>(new int[0:9])
```

```
partialinit: Πγ:place.pl γ
  → (Πα:region(true).int[α] → int)
```

Compared to `init`, the function `partialinit` takes two extra arguments, namely a place  $\gamma$  and a second place  $h$ . In the body of the `let`-expression, we can see that the idea is to apply `partialinit` twice to the same place value. The idea is that we want to use the place both as a value and as a type. This idiom seems inescapable because we want to keep each construct in our core language simple. Notice that  $\gamma$  has the kind `place` which means that  $\gamma$  can be any place. We use  $\gamma$

in the type of  $h$ ; in other words, the type of  $h$  is essentially itself, which is the most accurate type possible.

The body of `partialinit` initializes those points in the array `a` which can be found at the place  $h$ . The expression `a.reg %s h` denotes those points in `a.reg` which by the fixed distribution of `a.reg` are mapped to the place  $h$ . The `for` loop iterates only over points in `a.reg %s h` and since the `for` loop is wrapped in `at(h){...}`, each access `a[p]` will happen at the place of `a[p]`.

Let us consider the type checking process for `a[p]`.

First we explain the bounds check for `a[p]`. The type of `a.reg` is `reg α` and the type of  $h$  is `pl γ`. As a result, the type of `a.reg %s h` is  $\alpha \%_t \gamma$ , which illustrates that we use a type operator to mirror the expression operator. The variable `p` then gets the type `pt(σ, α %t γ)`, where  $\sigma$  is a fresh variable for which we have the set constraint  $\sigma \in_t \alpha \%_t \gamma$ . The static bounds check will verify that the type of the region of `p` is a subset of the type of the region of `a`, which amounts to checking that  $\alpha \%_t \gamma$  is a subset of  $\alpha$ , which is true for all  $\alpha$ .

Second we explain the place check `a[p]`. This check illustrates the use of the most important *type equivalence* rule in the type system. Recall that the goal of the place check is to determine that the current place of execution is the same as the place of `a[p]`. The current place of execution is given by the enclosing `at(h)` expression, and we have that  $h$  has type `pl γ`. The type of the place of `a[p]` is given by the type expression  $\alpha[@_t(\sigma, \alpha \%_t \gamma)]$ , which says that the place is that of a point in  $\alpha$  which has its data located at a point with type `pt(σ, α %t γ)`. We can then use the type equivalence

$$\alpha[@_t(\sigma, \alpha \%_t \gamma)] \equiv \gamma$$

to conclude that the place of execution is indeed the same as the place of the data `a[p]`. We can read the type equivalence as saying, intuitively: “if we have a region  $\alpha$  and a point  $\sigma$ , where  $\sigma$  belongs to a subregion of  $\alpha$  in which all the points have their elements on the place  $\gamma$ , then indeed  $\sigma$  has its element on place  $\gamma$ .”

## 2.3 copy

The function `copy` takes two arrays over the same region and copies the elements from one array to the other. This example shows how to use (1) a function with two arguments over the same region and (2) a `forallplaces`-loop which iterates over all available places.

```
let copy = lam*α:region(true).λ*a:int[α].λ*b:int[α].
  forallplaces h { at(h) {
    for (p in (a.reg %s h)) {
      a[p] = b[p] } } }
in copy<0:7>(new int[0:7])(new int[0:7])
```

```
copy: Πα:region(true).int[α] → (int[α] → int)
```

The function `copy` takes two arrays `a` and `b`, both with region  $\alpha$ . The body of `copy` copies elements from `b` to `a`. The

body of copy uses the construct `forallplaces h { ... }` which iterates over all places available to the program and in each iterations binds the current place to `h`. For each place, the code copies elements that reside at that place. Notice that since `a` and `b` have the same region, they also have the same distribution, so for a given point `p` in that region, both `a[p]` and `b[p]` will be at the same place.

## 2.4 expand

The function `expand` takes an array and returns a new array over a bigger region; the output array will be partially initialized with values from the input array. This example shows how to use (1) constraints over region variables and (2) the intersection operator on regions.

```
let expand = lam•α:region(true).λ•a:int[α].
  lam•β:region(α ⊆t β).λ•x:reg β.
  let b = new int[x]
  in { forallplaces h { at(h) {
    for (p in a.reg ∩s (b.reg %s h)) {
      b[p] = at (a.reg[@p]) { a[p] } }
    } } ; b }
in expand<3:7>(new int[3:7])<0:10>(int[0:10])

expand: Πα:region(true).int[α]
  → (Πβ:region(α ⊆t β).reg β → int[β])
```

The function `expand` takes an array `a` over region  $\alpha$  and region `x` with region type  $\beta$ , 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. The program specifies the desired relationship between the region of `a` and `x` by giving the argument  $\beta$  a kind other than `true`. The kind of  $\beta$  is the constraint  $\alpha \subseteq_t \beta$ , which means that the region  $\alpha$  must be a subset of the region  $\beta$ . The call `expand<3:7>(new int[3:7])<0:10>(int[0:10])` 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  $3 : 7 \subseteq 0 : 10$ .

The function `expand` highlights the importance of keeping upper and lower bounds for the region of arrays during type checking. The type of `a.reg ∩s (b.reg %s h)` is `reg α ∩t (β %t π)`, where  $\pi$  is a fresh type variable about which we know that `h` has type `p1 π`.

When we type check `a[p]` we have that `p` is in region `reg α ∩t (β %t π)` and `a` has the region  $\alpha$ . So the bounds check is satisfied because `reg α ∩t (β %t π)` is a subset of  $\alpha$  for all  $\alpha$ . The place check is also satisfied because the current place of execution is exactly the place of `a[p]`.

When we type check `b[p]` we have that `b` has the region  $\beta$ , and we have  $\alpha \subseteq_t \beta$ . So the bounds check is satisfied because `reg α ∩t (β %t π)` is a subset of  $\alpha$  for all  $\alpha$  and  $\alpha \subseteq_t \beta$ . The place check is also satisfied because the current place of execution is `h` which has type `p1 π` and we know

`p` is in a region with type `reg α ∩t (β %t π)` which only includes points with elements at a place with type `p1 π`.

## 2.5 shiftleft

The function `shiftleft` shifts all elements in an array to the left, while leaving the rightmost element unchanged. This example shows how to use (1) arithmetic operators on points and (2) arithmetic operators on regions.

```
let shiftleft = lam•α:region(true).λ•a:int[α].
  let inner = (a.reg + 1) ∩s a.reg
  in { for (p in inner) { at(a.reg[@p-1]) {
    a[p-1] = at(a.reg[@p]) { a[p] } }
  } }
in shiftleft<reg 3:7>(new int[3:7])

shiftleft: Πα:region(true).int[α] → int
```

The function `shiftleft` takes an argument `a` with region  $\alpha$  and shifts all elements one position to the left, while leaving the rightmost element unchanged. In more detail, `shiftleft` first creates a region `inner` by shifting all elements of `a.reg` by one to the right and then intersecting the result with `a.reg`. If `a.reg` is simply an interval, this effectively removes the first element from `a.reg`. The type of `inner` is `reg ((α + 1) ∩t α)`. Then `shiftleft` proceeds with doing essentially `a[p-1] = a[p]` for each point `p` in the inner region.

Let us now consider the bounds checks for `a[p-1]` and `a[p]`. The expression `p` is always within the region of `a` because `p` has region  $(\alpha + 1) \cap_t \alpha$  which is a subset of  $\alpha$  for all  $\alpha$ . The expression `p-1` is always within the region of `a` because `p-1` has region  $((\alpha + 1) \cap_t \alpha) - 1$  which is a subset of  $\alpha$  for all  $\alpha$  because  $+1$  and  $-1$  cancel each other out.

## 2.6 shift

The function `shiftleft` is a buggy version of `shift`. This example shows that mistakes in the arithmetic on points can be caught by our type checker.

```
let shift = lam•α:region(true).λ•a:int[α].
  let inner = (a.reg + 1) ∩s a.reg
  in { for (p in inner) {
    at(a.reg[@p+1]) {
      a[p+1] = at(a.reg[@p]) { a[p] } }
  } }
in ...
```

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. When the type checker considers `a[p+1]` it will determine that the region of `p+1` is  $((\alpha + 1) \cap_t \alpha) + 1$  which is *not* a subset of  $\alpha$  for all  $\alpha$ .

(Kind)	$k ::= \text{point } \varphi \mid \text{region } \varphi \mid \text{place}$
(Type)	$t ::= \text{int} \mid \text{pt } (\sigma, r) \mid \text{reg } r \mid t[r]$ $\mid \text{pl } \pi \mid t \rightarrow t \mid \Pi \alpha : k.t$
(Region)	$r ::= \alpha \mid R \mid r \cup_t r \mid r \cap_t r$ $\mid r +_t c \mid r \%_t \pi$
(Point)	$\sigma ::= \alpha \mid p \mid \sigma ++_t c$
(Place)	$\pi ::= \alpha \mid P \mid r[@_t(\sigma, r)]$ $\mid \text{unknown}$
(Constraint)	$\varphi ::= r \subseteq_t r \mid \sigma \in_t r \mid \varphi \wedge \varphi$
(Value)	$v ::= c \mid p \mid R \mid l \mid P \mid \lambda x : t.e$ $\mid \text{lam } \alpha : k.e$
(ValOrVar)	$y ::= v \mid x$
(Expression)	$e ::= y \mid e_1 e_2 \mid e_1 < e_2 >$ $\mid \lambda^\bullet x : t.e \mid \text{lam}^\bullet \alpha : k.e$ $\mid \text{new } t[e] \mid y_1[y_2] \mid y_1[y_2] = e$ $\mid e.\text{reg} \mid y_1[@_s y_2]$ $\mid e_1 \cup_s e_2 \mid e_1 \cap_s e_2 \mid e +_s c$ $\mid e ++_s c \mid y_1 \%_s y_2$ $\mid \text{for } (x \text{ in } e_1)\{e_2\}$ $\mid \text{forallplaces } x\{e\}$ $\mid e_1; e_2 \mid \text{at}(y)\{e\}$
(Dep Val)	$w ::= p \mid R \mid P$

Figure 1. Syntax of the core language.

### 3. The Core Language

We now present the syntax, semantics, and type system of our core language. We prove type soundness using the standard technique of Nielson [18] and others that was popularized by Wright and Felleisen [23].

#### 3.1 Syntax

Figure 1 gives the syntax for the core language. We use  $c$  to range over integer constants,  $p$  to range over point constants,  $R$  to range over region constants (such as  $[1:4]$ , which denotes  $\{1, 2, 3, 4\}$ ),  $l$  to range over array labels drawn from a set `Label`,  $P$  to range over place constants,  $x$  to range over variable names, and  $\alpha$  to range over type-variable names. In our core language, points are integers, and we will occasionally write a point constant as  $c$ . For shifting a region by a constant we use the notation  $\{c_1, \dots, c_n\} + c = \{c_1 + c, \dots, c_n + c\}$ .

The language has seven data types, namely integers, points, regions, arrays, places, functions, and dependently-typed functions. We have deliberately avoided having distributions as values, in an effort to keep the size of the language manageable. We assume a function *distribute* which maps a region and a point in that region to a place. When we create an array over a region  $R$ , the array will be distributed according to the function *distribute*. We make no assumptions about *distribute*.

The types are defined in terms of three forms of expressions which, given an interpretation of the variables, evaluate to sets of points (regions), points, and places, respectively.

Specifically, if  $\rho$  is a mapping from region variables to regions, point variables to points, and place variables to places, then the meaning of the expressions is given as follows:

$$\begin{aligned}
\alpha\rho &= \rho(\alpha) \\
R\rho &= R \\
(r_1 \cup_t r_2)\rho &= r_1\rho \cup r_2\rho \\
(r_1 \cap_t r_2)\rho &= r_1\rho \cap r_2\rho \\
(r +_t c)\rho &= r\rho + c \\
(r \%_t \pi)\rho &= \{p \in r\rho \mid \text{distribute}(r\rho, p) = \pi\rho\} \\
p\rho &= p \\
(r ++_t c)\rho &= r\rho + c \\
P\rho &= P \\
(r_1[@_t(\sigma, r_2)])\rho &= \text{distribute}(r_1\rho, \sigma\rho).
\end{aligned}$$

The expression  $r \%_t \pi$  evaluates to a subset of  $r$  which contains those points which are mapped to  $\pi$  by *distribute*. The expression  $r[@_t(\sigma, r)]$  evaluates to the place of the point  $\sigma$  according to the distribution given by *distribute*.

The type of a point is a pair  $(\sigma, r)$  where  $\sigma$  is a type-level identity of the point and  $r$  is a region that contains the point. The type of a region is a singleton type consisting of that region itself. A dependently-typed function  $\text{lam } \alpha : k.e$  has its argument constrained by the kind  $k$ ; its type is  $\Pi \alpha : k.t$ .

The expression language contains syntax for creating and calling functions, for creating, accessing, and updating arrays, for computing with regions, for iterating over regions, for iterating over all places, and for shifting the place of execution. The expression  $e.\text{reg}$  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.

**Syntactic Sugar.** In the example programs earlier in the paper, we used the syntactic sugar  $\text{let } x = e \text{ in } \{e'\}$   $x : t$  for the core language expression  $(\lambda^\bullet x : t.e')e$ . We use `true` to denote the tautology  $\emptyset \subseteq_t \emptyset$ .

#### 3.2 Semantics

We specify the semantics of the core language using small-step operational semantics (see Figure 3.2). We use  $H$  to range over heaps:

$$H \in \text{Label} \rightarrow \text{Point} \rightarrow (\text{Value} \times \text{Place})$$

A heap maps labels to array representations. An array representation maps each point in the region of the array to its value and its place. Both uses of  $\rightarrow$  above denote a space of partial functions. We will use the notation  $(v, P)$  for elements of  $(\text{Value} \times \text{Place})$ , and we will use the operators

\_1 and \_2 to extract the first and second element of a pair, respectively. 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* at place  $P$  if we have  $H', e'$  such that  $P \vdash (H, e) \rightsquigarrow (H', e')$  using the rules below. We say that  $(H, e)$  is *stuck* at place  $P$  if  $e$  is not a value and  $(H, e)$  cannot take a step at place  $P$ . We say that  $(H, e)$  can *go wrong* at place  $P$  if we have  $H', e'$  such that  $P \vdash (H, e) \rightsquigarrow^* (H', e')$  and  $(H', e')$  is stuck at place  $P$ .

We assume that the programmer (externally to the program text) provides a function `default` which maps a closed type  $t$  to a value, for each type  $t$  used as an element type of an array in the program. The function `default` must have the property that  $\Psi; \varphi; \Gamma \vdash \text{default}(t) : t$  for a  $\Psi$  that contains suitable definitions of the labels used in `default(t)`, and for any  $\varphi$  and  $\Gamma$ . The idea is that we will use `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  $\Psi; \varphi; \Gamma \vdash \text{default}(t) : t$ .

We also assume a list `places` of the places available during the execution of the program. The only thing a program can do with `places` is to iterate over the places using the `forallplaces` construct.

In order to specify the execution order for the `for` loop construct, Rule (29) uses a function  $\text{order}(\{c_1, \dots, c_n\}) = \langle c_1, \dots, c_n \rangle$ , where  $c_1 < \dots < c_n$ .

The following rules define a call-by-value semantics and are mostly standard. Rule (7) and Rule (8) express that the body of  $\lambda^\bullet$  or  $\text{lam}^\bullet$  must execute at the place of the definition. Effectively, each of those rules creates a closure consisting of the function and the current place of execution.

The key rules (11) and (13) both have the side condition that  $l \in \mathcal{D}(H)$  and  $p \in \mathcal{D}(H(l))$  and  $P = H(l)(p)$ .<sup>2</sup> The condition  $p \in \mathcal{D}(H(l))$  is the array-bounds check;  $p$  must be in the region of the array. The condition  $P = H(l)(p)$ .<sup>2</sup> is the place check; the place of execution must equal the place of the data to be accessed. If the side condition is not met, then the semantics will get stuck.

Notice that in Rule (19) we evaluate the syntactic expression  $R_1 \cup_s R_2$  to the value  $R_1 \cup R_2$ .

Rule (29) unrolls the `for` loop and replaces the loop variable with an appropriate point in each copy of the body of the loop. Similarly Rule (30) unrolls the loop and replaces the loop variables with an appropriate place in each copy of the body of the loop. The unrolling is specified the way it is to enable the type checker to assign a type variable as the type of the loop variable and at the same time achieve that each iteration is executed using the exact value bound to the loop variable.

### 3.3 Set Constraints

We will now define satisfiability and entailment for our class of set constraints, and we will settle their complexities.

Let  $\rho$  be a mapping from region variables to regions, point variables to points, and place variables to places. We say that  $\rho$  *satisfies* a set constraint  $\varphi$  if for all  $r_1 \subseteq_t r_2$  in  $\varphi$  we have  $r_1 \rho \subseteq r_2 \rho$  and for all  $\sigma \in_t r$  in  $\varphi$  we have  $\sigma \rho \in r \rho$ . We say that a constraint  $\varphi$  is *satisfiable* if there exists a satisfying assignment for  $\varphi$ .

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

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

For our notion of constraints, the satisfiability problem is NP-complete. To understand this, first note that already for the fragment of region constraints with just variables, constants, union, and intersection, 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  $\alpha$  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  $e+_s$  or  $e++_s$  expression in the constraint. In any solution, an upper bound on the largest integer is  $n \times u \times k$ . To demonstrate, 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 a lower bound on the smallest integer used in any solution. So, for each region variable we can guess a set of polynomial size, for each point variable we can guess a point in a set of polynomial size, and for each place variable we can guess a place in the list `places`. We can then check that assignment in polynomial time.

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  $\varphi' = \text{false}$  we have that the entailment problem is the question of given  $\varphi$ , is  $\varphi$  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 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.

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e'_1)}{P \vdash (H, e_1 e_2) \rightsquigarrow (H', e'_1 e'_2)} \quad (1)$$

$$\frac{P \vdash (H, e_2) \rightsquigarrow (H', e'_2)}{P \vdash (H, v e_2) \rightsquigarrow (H', v e'_2)} \quad (2)$$

$$P \vdash (H, (\lambda x : t.e)v) \rightsquigarrow (H, e[x := v]) \quad (3)$$

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e'_1)}{P \vdash (H, e_1 \langle e_2 \rangle) \rightsquigarrow (H', e'_1 \langle e_2 \rangle)} \quad (4)$$

$$\frac{P \vdash (H, e_2) \rightsquigarrow (H', e'_2)}{P \vdash (H, v \langle e_2 \rangle) \rightsquigarrow (H', v \langle e'_2 \rangle)} \quad (5)$$

$$P \vdash (H, (\mathbf{lam} \alpha : k.e) \langle w \rangle) \rightsquigarrow (H, e[\alpha := w]) \quad (6)$$

$$P \vdash (H, \lambda^\bullet x : t.e) \rightsquigarrow (H, \lambda x : t.\mathbf{at}(P)\{e\}) \quad (7)$$

$$P \vdash (H, \mathbf{lam}^\bullet \alpha : k.e) \rightsquigarrow (H, \mathbf{lam} \alpha : k.\mathbf{at}(P)\{e\}) \quad (8)$$

$$\frac{P \vdash (H, e) \rightsquigarrow (H', e')}{P \vdash (H, \mathbf{new} t[e]) \rightsquigarrow (H', \mathbf{new} t[e'])} \quad (9)$$

$$P \vdash (H, \mathbf{new} t[R]) \rightsquigarrow (H[l \mapsto \lambda p \in R.(\mathbf{default}(t), \mathbf{distribute}(R, p))], l) \quad \text{where } l \text{ is fresh} \quad (10)$$

$$P \vdash (H, l[p]) \rightsquigarrow (H, H(l)(p).1) \quad \text{if } l \in \mathcal{D}(H) \text{ and } p \in \mathcal{D}(H(l)) \text{ and } P = H(l)(p).2 \quad (11)$$

$$\frac{P \vdash (H, e) \rightsquigarrow (H', e')}{P \vdash (H, v_1[v_2] = e) \rightsquigarrow (H', v_1[v_2] = e')} \quad (12)$$

$$P \vdash (H, l[p] = v) \rightsquigarrow (H[l \mapsto (H(l))[p \mapsto (v, H(l)(p).2)]], v) \quad \text{if } l \in \mathcal{D}(H) \text{ and } p \in \mathcal{D}(H(l)) \text{ and } P = H(l)(p).2 \quad (13)$$

$$\frac{P \vdash (H, e) \rightsquigarrow (H', e')}{P \vdash (H, e.\mathbf{reg}) \rightsquigarrow (H', e'.\mathbf{reg})} \quad (14)$$

$$P \vdash (H, l.\mathbf{reg}) \rightsquigarrow (H, \mathcal{D}(H(l))) \quad \text{if } l \in \mathcal{D}(H) \quad (15)$$

$$P \vdash (H, l[\@_s p]) \rightsquigarrow (H, H(l)(p).2) \quad \text{if } l \in \mathcal{D}(H) \text{ and } p \in \mathcal{D}(H(l)) \quad (16)$$

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e'_1)}{P \vdash (H, e_1 \cup_s e_2) \rightsquigarrow (H', e'_1 \cup_s e'_2)} \quad (17)$$

$$\frac{P \vdash (H, e_2) \rightsquigarrow (H', e'_2)}{P \vdash (H, v \cup_s e_2) \rightsquigarrow (H', v \cup_s e'_2)} \quad (18)$$

$$P \vdash (H, R_1 \cup_s R_2) \rightsquigarrow (H, R_1 \cup R_2) \quad (19)$$

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e'_1)}{P \vdash (H, e_1 \cap_s e_2) \rightsquigarrow (H', e'_1 \cap_s e'_2)} \quad (20)$$

$$\frac{P \vdash (H, e_2) \rightsquigarrow (H', e'_2)}{P \vdash (H, v \cap_s e_2) \rightsquigarrow (H', v \cap_s e'_2)} \quad (21)$$

$$P \vdash (H, R_1 \cap_s R_2) \rightsquigarrow (H, R_1 \cap R_2) \quad (22)$$

$$\frac{P \vdash (H, e) \rightsquigarrow (H', e')}{P \vdash (H, e +_s c) \rightsquigarrow (H', e' +_s c)} \quad (23)$$

$$P \vdash (H, d +_s c) \rightsquigarrow (H, d + c) \quad (24)$$

$$\frac{P \vdash (H, e) \rightsquigarrow (H', e')}{P \vdash (H, e ++_s c) \rightsquigarrow (H', e' ++_s c)} \quad (25)$$

$$P \vdash (H, p ++_s c) \rightsquigarrow (H, p + c) \quad (26)$$

$$P \vdash (H, R \%_s P') \rightsquigarrow (H, R') \quad \text{where } R' = \{p \in R \mid \mathbf{distribute}(R, p) = P'\} \quad (27)$$

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e'_1)}{P \vdash (H, \mathbf{for} (x \text{ in } e_1)\{e_2\}) \rightsquigarrow (H', \mathbf{for} (x \text{ in } e'_1)\{e_2\})} \quad (28)$$

$$P \vdash (H, \mathbf{for} (x \text{ in } R)\{e\}) \rightsquigarrow (H, ((\mathbf{lam}^\bullet \alpha : \mathbf{point}(\alpha \in_t R).\lambda^\bullet x : (\alpha, R).e) \langle c_1 \rangle) c_1; \dots; ((\mathbf{lam}^\bullet \alpha : \mathbf{point}(\alpha \in_t R).\lambda^\bullet x : (\alpha, R).e) \langle c_n \rangle) c_n; 0) \quad \text{where } \mathbf{order}(R) = \langle c_1, \dots, c_n \rangle \quad (29)$$

$$P \vdash (H, \mathbf{forallplaces} x\{e\}) \rightsquigarrow (H, ((\mathbf{lam}^\bullet \alpha : \mathbf{place}.\lambda^\bullet x : \mathbf{pl} \alpha.e) \langle P_1 \rangle) P_1; \dots; ((\mathbf{lam}^\bullet \alpha : \mathbf{place}.\lambda^\bullet x : \mathbf{pl} \alpha.e) \langle P_n \rangle) P_n; 0) \quad \text{where } \mathbf{places} = \langle P_1, \dots, P_n \rangle \quad (30)$$

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e'_1)}{P \vdash (H, e_1; e_2) \rightsquigarrow (H, e'_1; e_2)} \quad (31)$$

$$P \vdash (H, v; e) \rightsquigarrow (H, e) \quad (32)$$

$$\frac{P' \vdash (H, e) \rightsquigarrow (H', e')}{P \vdash (H, \mathbf{at}(P')\{e\}) \rightsquigarrow (H, \mathbf{at}(P')\{e'\})} \quad (33)$$

$$P \vdash (H, \mathbf{at}(P')\{v\}) \rightsquigarrow (H, v) \quad (34)$$



### 3.4 Type System

**Heap Types** We use  $\Psi$  to range over maps from array labels to types of the form  $t[R]$ . 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  $t[R] = \Psi(l)$ , then  $\mathcal{D}(H(l)) = R$  and for each  $p \in \mathcal{D}(H(l))$  we have (i)  $\Psi; \varphi; \Gamma; here \vdash H(l)(p).1 : t$  and (ii)  $distribute(R, p) = H(l)(p).2$ . We write  $\Psi \triangleleft \Psi'$  if  $\mathcal{D}(\Psi) \subseteq \mathcal{D}(\Psi')$  and  $\Psi, \Psi'$  agree on their common domain.

**Type Equivalence.** We define type equivalence via the judgments  $\varphi \vdash t \equiv t'$ ,  $\varphi \vdash r \equiv r'$ ,  $\varphi \vdash \sigma \equiv \sigma'$ , and  $\varphi \vdash \pi \equiv \pi'$ , which hold if they can be derived using the rules in Figure 2. The first three rules use a meta-variable  $q$  which ranges over  $t, r, \sigma, \pi$ .

The complexity of deciding type equivalence is dominated by the time to check constraint entailment. Given that all other aspects of type checking for our core language are in polynomial time, we conclude that type checking is co-NP-complete. In a later section, our experimental results show that the problem instances for entailment are small for our benchmarks and thus type checking is fast.

**Type Rules.** A type judgment is of the form  $\Psi; \varphi; \Gamma; here \vdash e : t$ , which holds if it is derivable using the following rules. The type *here* is the type of the current place of execution. Rule (61) for type checking a function  $\lambda x : t_1.e$  type checks the body  $e$  at the unknown place of execution, written *unknown*. The reason is that  $\lambda x : t_1.e$  will run at the place of its call site, which is statically unknown. A similar comment applies to Rule (62). Notice that the use of entailment is a condition in rules such as Rule (65). Rule (78) is a key type rule which says that to type check a loop for  $(x \text{ in } e_1)\{e_2\}$ , we check that  $e_1$  has a type  $\text{reg } r$ , and then assign  $x$  the type  $\text{pt } (\alpha, r)$  while checking  $e_2$ , where  $\alpha$  is fresh. The type rules for array lookup, Rule (69), and array update, Rule (70), ensure that (1) 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, and (2) the place of execution equals the location of the array data by requiring that the type *here* is equivalent to the type of the place of the data.

Rules for extracting constraints:

$$\text{constraint}(\text{point } \varphi) = \varphi \quad (83)$$

$$\text{constraint}(\text{region } \varphi) = \varphi \quad (84)$$

$$\text{constraint}(\text{place}) = \text{true} \quad (85)$$

We use  $W$  to range over regions  $r$  and variables  $\alpha$  of kind *place*.

Rules for kind checking:

$$\vdash \text{pt } (\sigma, r) : \text{point } \varphi \triangleright \sigma \quad (86)$$

$$\vdash \text{reg } r : \text{region } \varphi \triangleright r \quad (87)$$

$$\vdash \text{pl } \pi : \text{place } \triangleright \pi. \quad (88)$$

$$\varphi \vdash q \equiv q \quad (35)$$

$$\frac{\varphi \vdash q_1 \equiv q_2}{\varphi \vdash q_2 \equiv q_1} \quad (36)$$

$$\frac{\varphi \vdash q_1 \equiv q_2 \quad \varphi \vdash q_2 \equiv q_3}{\varphi \vdash q_1 \equiv q_3} \quad (37)$$

$$\frac{\varphi \vdash \sigma \equiv \sigma' \quad \varphi \vdash r \equiv r'}{\varphi \vdash \text{pt } (\sigma, r) \equiv \text{pt } (\sigma', r')} \quad (38)$$

$$\frac{\varphi \vdash r \equiv r'}{\varphi \vdash \text{reg } r \equiv \text{reg } r'} \quad (39)$$

$$\frac{\varphi \vdash \pi \equiv \pi'}{\varphi \vdash \text{pl } \pi \equiv \text{pl } \pi'} \quad (40)$$

$$\frac{\varphi \vdash t_1 \equiv t'_1 \quad \varphi \vdash t_2 \equiv t'_2}{\varphi \vdash t_1 \rightarrow t_2 \equiv t'_1 \rightarrow t'_2} \quad (41)$$

$$\frac{\varphi \vdash t \equiv t'}{\varphi \vdash \Pi \alpha : k.t \equiv \Pi \alpha : k.t'} \quad (42)$$

$$\varphi \vdash R_1 \cup_t R_2 \equiv R_1 \cup R_2 \quad (43)$$

$$\frac{\varphi \vdash r_1 \equiv r'_1 \quad \varphi \vdash r_2 \equiv r'_2}{\varphi \vdash r_1 \cup_t r_2 \equiv r'_1 \cup_t r'_2} \quad (44)$$

$$\varphi \vdash R_1 \cap_t R_2 \equiv R_1 \cap R_2 \quad (45)$$

$$\frac{\varphi \vdash r_1 \equiv r'_1 \quad \varphi \vdash r_2 \equiv r'_2}{\varphi \vdash r_1 \cap_t r_2 \equiv r'_1 \cap_t r'_2} \quad (46)$$

$$\varphi \vdash R +_t c \equiv R + c \quad (47)$$

$$\frac{\varphi \vdash r \equiv r'}{\varphi \vdash r +_t c \equiv r' +_t c} \quad (48)$$

$$\varphi \vdash R \%_t P \equiv \{ p \in R \mid distribute(R, p) = P \} \quad (49)$$

$$\frac{\varphi \vdash r \equiv r' \quad \varphi \vdash \pi \equiv \pi'}{\varphi \vdash r \%_t \pi \equiv r' \%_t \pi'} \quad (50)$$

$$\varphi \vdash p ++_t c \equiv p + c \quad (51)$$

$$\frac{\varphi \vdash \sigma \equiv \sigma'}{\varphi \vdash \sigma ++_t c \equiv \sigma' ++_t c} \quad (52)$$

$$\frac{\varphi \models p \in_t r \quad \varphi \models r \subseteq_t R}{\varphi \vdash R[\@_t(p, r)] \equiv distribute(R, p)} \quad (53)$$

$$\frac{\varphi \vdash r_1 \equiv r'_1 \quad \varphi \vdash \sigma \equiv \sigma' \quad \varphi \vdash r_2 \equiv r'_2}{\varphi \vdash r_1[\@_t(\sigma, r_2)] \equiv r'_1[\@_t(\sigma', r'_2)]} \quad (54)$$

$$\frac{\varphi \models \sigma \in_t r \%_t \pi}{\varphi \vdash r[\@_t(\sigma, r \%_t \pi)] = \pi} \quad (55)$$

**Figure 2.** Type equivalence rules.

$$\begin{array}{l} \Psi; \varphi; \Gamma; \text{here} \vdash c : \text{int} \quad (56) \\ \Psi; \varphi; \Gamma; \text{here} \vdash p : \text{pt}(p, R) \quad (\text{where } p \in R) \quad (57) \\ \Psi; \varphi; \Gamma; \text{here} \vdash R : \text{reg } R \quad (58) \\ \Psi; \varphi; \Gamma; \text{here} \vdash l : \Psi(l) \quad (59) \\ \Psi; \varphi; \Gamma; \text{here} \vdash P : \text{pl } P \quad (60) \\ \frac{\Psi; \varphi; \Gamma[x : t_1]; \text{unknown} \vdash e : t_2}{\Psi; \varphi; \Gamma; \text{here} \vdash \lambda x : t_1. e : t_1 \rightarrow t_2} \quad (61) \\ \frac{\Psi; \varphi \wedge \text{constraint}(k); \Gamma; \text{unknown} \vdash e : t}{\Psi; \varphi; \Gamma; \text{here} \vdash \text{lam } \alpha : k. e : \Pi \alpha : k. t} \quad (62) \\ \Psi; \varphi; \Gamma; \text{here} \vdash x : \Gamma(x) \quad (63) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash e_1 : t_1 \rightarrow t_2 \quad \Psi; \varphi; \Gamma; \text{here} \vdash e_2 : t_1}{\Psi; \varphi; \Gamma; \text{here} \vdash e_1 e_2 : t_2} \quad (64) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash e_1 : \Pi \alpha : k. t_1 \quad \Psi; \varphi; \Gamma; \text{here} \vdash e_2 : t_2 \quad \vdash t_2 : k \triangleright W \quad \varphi \models (\text{constraint}(k))[\alpha := W]}{\Psi; \varphi; \Gamma; \text{here} \vdash e_1 < e_2 > : t_1[\alpha := W]} \quad (65) \\ \frac{\Psi; \varphi; \Gamma[x : t_1]; \text{here} \vdash e : t_2 \quad \text{here} \neq \text{unknown}}{\Psi; \varphi; \Gamma; \text{here} \vdash \lambda^\bullet x : t_1. e : t_1 \rightarrow t_2} \quad (66) \\ \frac{\Psi; \varphi \wedge \text{constraint}(k); \Gamma; \text{here} \vdash e : t \quad \text{here} \neq \text{unknown}}{\Psi; \varphi; \Gamma; \text{here} \vdash \text{lam}^\bullet \alpha : k. e : \Pi \alpha : k. t} \quad (67) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash e : \text{reg } r}{\Psi; \varphi; \Gamma; \text{here} \vdash \text{new } t[e] : t[r]} \quad (68) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash y_1 : t[r_1] \quad \Psi; \varphi; \Gamma; \text{here} \vdash y_2 : \text{pt}(\sigma, r_2) \quad \varphi \models r_2 \subseteq_t r_1 \quad \varphi \models \sigma \in_t r_2 \quad \varphi \vdash \text{here} \equiv r_1[\text{@}_t(\sigma, r_2)]}{\Psi; \varphi; \Gamma; \text{here} \vdash y_1[y_2] : t} \quad (69) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash y_1 : t[r_1] \quad \Psi; \varphi; \Gamma; \text{here} \vdash y_2 : \text{pt}(\sigma, r_2) \quad \varphi \models r_2 \subseteq_t r_1 \quad \varphi \models \sigma \in_t r_2 \quad \varphi \vdash \text{here} \equiv r_1[\text{@}_t(\sigma, r_2)] \quad \Psi; \varphi; \Gamma; \text{here} \vdash e : t}{\Psi; \varphi; \Gamma; \text{here} \vdash y_1[y_2] = e : t} \quad (70) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash e : t[r]}{\Psi; \varphi; \Gamma; \text{here} \vdash e. \text{reg} : \text{reg } r} \quad (71) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash y_1 : t[r_1] \quad \Psi; \varphi; \Gamma; \text{here} \vdash y_2 : \text{pt}(\sigma, r_2) \quad \varphi \models r_2 \subseteq_t r_1 \quad \varphi \models \sigma \in_t r_2}{\Psi; \varphi; \Gamma; \text{here} \vdash y_1[\text{@}_s y_2] : \text{pl } r_1[\text{@}_t(\sigma, r_2)]} \quad (72) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash e_1 : \text{reg } r_1 \quad \Psi; \varphi; \Gamma; \text{here} \vdash e_2 : \text{reg } r_2}{\Psi; \varphi; \Gamma; \text{here} \vdash e_1 \cup_s e_2 : \text{reg } r_1 \cup_t r_2} \quad (73) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash e_1 : \text{reg } r_1 \quad \Psi; \varphi; \Gamma; \text{here} \vdash e_2 : \text{reg } r_2}{\Psi; \varphi; \Gamma; \text{here} \vdash e_1 \cap_s e_2 : \text{reg } r_1 \cap_t r_2} \quad (74) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash e : \text{reg } r}{\Psi; \varphi; \Gamma; \text{here} \vdash e +_s c : \text{reg } r +_t c} \quad (75) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash e : \text{pt}(\sigma, r)}{\Psi; \varphi; \Gamma; \text{here} \vdash e ++_s c : \text{pt}(\sigma ++_t c, r +_t c)} \quad (76) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash y_1 : \text{reg } r \quad \Psi; \varphi; \Gamma; \text{here} \vdash y_2 : \text{pl } \pi}{\Psi; \varphi; \Gamma; \text{here} \vdash y_1 \%_s y_2 : \text{reg } r \%_t \pi} \quad (77) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash e_1 : \text{reg } r \quad \Psi; \varphi \wedge (\alpha \in_t r); \Gamma[x : \text{pt}(\alpha, r)]; \text{here} \vdash e_2 : \text{int} \quad \text{here} \neq \text{unknown}}{\Psi; \varphi; \Gamma; \text{here} \vdash \text{for}(x \text{ in } e_1)\{e_2\} : \text{int}} \quad (\text{where } \alpha \text{ is fresh}) \quad (78) \\ \frac{\Psi; \varphi; \Gamma[x : \text{pl } \alpha]; \text{here} \vdash e : \text{int} \quad \text{here} \neq \text{unknown}}{\Psi; \varphi; \Gamma; \text{here} \vdash \text{forallplaces } x\{e\} : \text{int}} \quad (\text{where } \alpha \text{ is fresh}) \quad (79) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash e_1 : t_1 \quad \Psi; \varphi; \Gamma; \text{here} \vdash e_2 : t_2}{\Psi; \varphi; \Gamma; \text{here} \vdash e_1; e_2 : t_2} \quad (80) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash y : \text{pl } \pi \quad \Psi; \varphi; \Gamma; \pi \vdash e : t}{\Psi; \varphi; \Gamma; \text{here} \vdash \text{at}(y)\{e\} : t} \quad (81) \\ \frac{\Psi; \varphi; \Gamma; \text{here} \vdash e : t \quad \varphi \vdash t \equiv t'}{\Psi; \varphi; \Gamma; \text{here} \vdash e : t'} \quad (82) \end{array}$$

### 3.5 Type Soundness

We have proved the soundness of our type system. Here is a listing of the needed lemmas and theorems with brief proof sketches, concluding with the statement and proof of type soundness. The appendices give detailed proofs of Type Preservation (Theorem 1) and Progress (Theorem2).

#### LEMMA 1. (Substitution)

If  $\Psi; \varphi; \Gamma[x : t_1]; \text{here} \vdash e : t_2$  and  $\Psi; \varphi; \Gamma; \text{here} \vdash v : t_1$ , then  $\Psi; \varphi; \Gamma; \text{here} \vdash e[x := v] : t_2$ .

*Proof.* By induction on the structure of the derivation of  $\Psi; \varphi; \Gamma[x : t_1]; \text{here} \vdash e : t_2$ .  $\square$

#### LEMMA 2. (Dependent Substitution)

If  $\Psi; \varphi; \Gamma; \text{here} \vdash e : t$ , then  $\Psi; \varphi[\alpha := W]; \Gamma; \text{here}[\alpha := W] \vdash e[\alpha := W] : t[\alpha := W]$ .

*Proof.* By induction on the structure of the derivation of  $\Psi; \varphi; \Gamma; \text{here} \vdash e : t$ .  $\square$

#### LEMMA 3. (Weakening)

If  $\Psi; \varphi; \Gamma; \text{here} \vdash e : t$  and  $\varphi' \models \varphi$ , then  $\Psi; \varphi'; \Gamma; \text{here} \vdash e : t$ .

*Proof.* By induction on the structure of the derivation of  $\Psi; \varphi; \Gamma; \text{here} \vdash e : t$ .  $\square$

#### LEMMA 4. (Indifference)

If  $\Psi; \varphi; \Gamma; \text{here} \vdash v : t$ , then  $\Psi; \varphi; \Gamma; \text{here}' \vdash v : t$ .

*Proof.* Immediate from the seven type rules for values.  $\square$

#### LEMMA 5. (Canonical Forms)

- If  $\Psi; \varphi; \Gamma; \text{here} \vdash v : \text{int}$ , then  $v$  is of the form  $c$ .
- If  $\Psi; \varphi; \Gamma; \text{here} \vdash v : \text{pt}(\sigma, r)$ , then  $v$  is of the form  $p$ .
- If  $\Psi; \varphi; \Gamma; \text{here} \vdash v : \text{reg } r$ , then  $v$  is of the form  $R$ .
- If  $\Psi; \varphi; \Gamma; \text{here} \vdash v : t[r]$ , then  $v$  is of the form  $l$ , and  $l \in \mathcal{D}(\Psi)$ .
- If  $\Psi; \varphi; \Gamma; \text{here} \vdash v : \text{p1 } \alpha$ , then  $v$  is of the form  $P$ .
- If  $\Psi; \varphi; \Gamma; \text{here} \vdash v : t_1 \rightarrow t_2$ , then  $v$  is of the form  $\lambda x : t.e$ .
- If  $\Psi; \varphi; \Gamma; \text{here} \vdash v : \Pi \alpha : k.t$ , then  $v$  is of the form  $\text{lam } \alpha : k.e$ .

*Proof.* From an examination of the type rules we have that each form of type is the type of exactly one form of value, namely the one given in the lemma.  $\square$

#### THEOREM 1. (Type Preservation)

For a place  $P$ , let  $Q \in \{P, \text{unknown}\}$ . If  $\Psi; \varphi; \Gamma; Q \vdash e : t$ ,  $\models H : \Psi$ , and  $P \vdash (H, e) \rightsquigarrow (H', e')$ , then we have  $\Psi', t'$  such that  $\Psi \triangleleft \Psi'$ ,  $\Psi'; \varphi; \Gamma; Q \vdash e' : t'$ ,  $\models H' : \Psi'$ , and  $\varphi \vdash t \equiv t'$ .

*Proof.* By induction on the structure of the derivation of  $\Psi; \varphi; \Gamma; Q \vdash e : t$ .  $\square$

#### THEOREM 2. (Progress)

For a place  $P$ , let  $Q \in \{P, \text{unknown}\}$ . If  $\Psi; \text{true}; \emptyset; Q \vdash e : t$  and  $\models H : \Psi$ , then  $(H, e)$  is not stuck at place  $P$ .

*Proof.* By induction on the structure of the derivation of  $\Psi; \text{true}; \emptyset; Q \vdash e : t$ .  $\square$

#### COROLLARY 1. (Type Soundness)

For a place  $P$ , let  $Q \in \{P, \text{unknown}\}$ . If  $\Psi; \text{true}; \emptyset; Q \vdash e : t$  and  $\models H : \Psi$ , then  $(H, e)$  cannot go wrong at place  $P$ .

*Proof.* Suppose  $(H, e)$  can go wrong at place  $P$ , that is, we have  $H', e'$  such that  $P \vdash (H, e) \rightsquigarrow^n (H', e')$  and  $(H', e')$  is stuck at place  $P$ . From Theorem 1, Rule (37), and induction on  $n$ , we have  $\Psi', t'$  such that  $\Psi'; \text{true}; \emptyset; Q \vdash e' : t'$ ,  $\models H' : \Psi'$ , and  $\text{true} \models t \equiv t'$ . From Theorem 2 we have that  $(H', e')$  is not stuck at place  $P$ , a contradiction.  $\square$

## 4. Experimental Results

We have designed and implemented a variant of the object-oriented language X10 which embodies our type system. To evaluate our type system, we have created the ArrayBench benchmark suite which consists of seven programs of a total of 865 lines of code. The type system cannot completely eliminate the need for dynamic checks of array accesses. In our X10 version, a programmer can write a cast to signal to the compiler that a combined bounds and place check must be performed. All array accesses without explicit casts are guaranteed to be within bounds. The main goal of this section is to answer these two questions:

- How many casts does a programmer need to insert to make the benchmarks type check?
- How many times will those casts be executed at run time and how does that compare to dynamically checking *all* array accesses?

In the following subsections we will first explain the notion of cast in more detail and give an example. We will then present the ArrayBench benchmark suite and finally present our experimental results and answer the two questions.

Our variant of X10 is called XTC-X10 and extends X10 version 0.4 with the type system presented in this paper along with parametric types (generics) and first-order functions. The implementation is publically available at <http://grothoff.org/christian/xtc/x10/>. Our implementation can type check and execute the benchmark programs listed below along with the five type-safe example programs from Section 2.

### 4.1 Array-Access Casts

In our experience, we need three categories of array-access casts:

1. Required casts due to the fact that the type-checker is flow insensitive. The classical Java equivalent for this kind of type cast is of the form `if (a instanceof B)`

`B b = (B) a; .` Here, the cast itself is always guarded by an dominating branch that yields an assertion that the cast will succeed. These casts should be considered to be free at runtime since a reasonable compiler should be able to completely eliminate the check. They could be avoided entirely if the compiler was flow-sensitive to begin with; however, such a choice is likely to result in problems with respect to programmers' understanding of overloading resolution. In terms of language design, we believe it is better to require the programmer to put in explicit casts even if the control-flow already yields equivalent assertions.

2. Casts that are used to cover certain corner cases that could be avoided (but at the expense of using significantly more complex type constructions). For example, a function may operate on arrays of arbitrary size *as long as they are not empty*. Such a corner case might be covered by requiring the programmer to supply an additional point and have the array satisfy the condition that it must contain this point and only points larger than it. A programmer might choose to instead obtain the minimum point of the array using the build-in `min` operator and use a cast (not-null) to establish that the point exists. Our design allows the programmer to decide that the simplicity of a cast might be a better choice than a complex type construction. Typically, the cost of these casts for corner cases is minimal – programmers are likely to use them outside of loops, and often the particular checks themselves are also rather inexpensive. The reason for this is that if the cast is in a critical section of the code, the programmer has the option of using more elaborate types.
3. Casts used to produce necessary loop invariants. Some algorithms use loops which make it impossible for the type system to establish the loop invariants necessary for checking the loop or code depending on the result of the computation performed by the loop. In these cases, the programmers must add casts to produce the necessary invariants. Naturally, the compiler may still be able to use flow information to reduce the cost of these casts; however, eliminating the check completely would require a theorem-prover that is stronger than what our type system can offer.

To illustrate the use of casts, let us consider some XTC-X10 code from an implementation of the Knuth-Morris-Pratt string searching algorithm [15]. We will focus on the function `overlap` which computes the partial match table (or failure function); the code is given below in the syntax of our X10 variant. The syntax is mostly similar to Java and C++. `ValueArray` is an immutable array, which means that accesses are not required to be local – only in-bounds. The language uses “.” for the type of a local variable that the compiler is supposed to infer from the right hand side of the assignment. Type elision is an important feature since

explicitly writing sufficiently precise region type annotations for all local variables would be tedious.

The type annotation `<#1>` adds the requirement that the respective point, region or array is one-dimensional. The type annotation `<:r>` specifies that the respective point must be contained in the region `r`. In the code, a `p` is appended to numbers in order to syntactically show the difference between (one-dimensional) points and integers.

```
Array<point<#1>>
  overlap(int m, ValueArray<int:([0:m-1])#1> pat) {

    if (m <= 0)
      throw new Exception("Empty pattern!");
    . overlap = new Array<point<#1>>([0:m], 0p);
    overlap[(point<:([0:m])>>0p) = -1p; // CAST #1
    for (p : [1:m]) {
      . prev = p - 1p;
      overlap[p] = overlap[prev] + 1p;
      while ( (overlap[p] > 0p) &&
              (pat[prev] !=
               pat[(point<:pat.reg>)
                  (overlap[p]-1p)]) ) // CAST #2
              overlap[p] = 1p + overlap
                [(point<:pat.reg>)
                 (overlap[p]-1p)]; // CAST #3
    }
  }
```

The example program contains ten locations where array accesses occur. The first array accesses in the example occur during the initialization of the `overlap` array to zero. The initialization is done in the array constructor (not shown) which corresponds closely to the `init` function from Section 2. The compiler requires no further information to avoid any bounds-checks on the constructor.

The second array access, initializing `overlap[0p]` to `-1` requires a bounds-check as indicated by the cast in the code. Cast #1 (identified by comments in the example) in `overlap` falls into both category 1 and 2. The fact that `m` was checked to be positive in the first line of the function establishes that `0p` is in the (now non-empty) interval  $[0 : m]$ . However, because the type checker is flow-insensitive, a cast is needed. The programmer might have chosen to declare `m` to be strictly positive – a minimal and sane restriction of the API – and avoided both the cast and the sanity check in the first line. Capturing such corner cases with types is often possible, but programmers are likely to use such “dirty” casts wherever they fail to find appropriate types.

The type system is able to prove the safety of the five accesses to `overlap[p]` and `overlap[prev]` inside of the `for` loop. The access to `overlap[p]` is safe because  $p \in [1 : m] \subseteq [0 : m]$ , and  $[0 : m]$  is the domain of `overlap` which is immediate from the creation of `overlap`. Similarly, `overlap[prev]` is safe because  $prev \in [0 : m - 1] \subseteq [0 : m]$ . Note that these proofs are a variation on the `partialinit` and `shiftleft` examples from Section 2,

except that here the subset relationships are completely inferred – region type declarations are usually only necessary to describe constraints on arguments and return values, type inference is sufficient for reasoning within methods.

The array access to `pat[prev]` is an example where region types must be provided. Because the `pat` array is known to be defined over the interval  $[0:m-1]$  and `prev` iterates over exactly the same interval ( $prev = p - 1p$  and  $p \in [1 : m]$ ), the access `pat[prev]` is safe. Ensuring that the type checker can verify the safety of this kind of access is the main difficulty for programming with region types: it is not always obvious which region type should be used for a particular argument.

When working with region types, programmers also need to be aware of the limitations of the type checker. In the example, the access to `pat` at the index `overlap[p]-1p` requires a cast since it cannot be shown to be safe by the type checker. Cast #2 highlights the problem that the type system may not always be able to establish proper loop invariants (category 3). For the points in the `overlap` array, the type system does verify that all points are one-dimensional. However, it cannot establish a loop invariant that would show that the assignment of the form `overlap[p] = overlap[q] + 1p` never produces points with a value larger than  $m + 1$ .

The situation is similar for the array access to `overlap` at index `overlap[p]-1p`. Again, the type checker fails to establish the loop invariant (`overlap[p]` is always a valid index into `overlap`). However, the type checker is able to deduce that `pat.reg`  $\subseteq [0 : m]$ , allowing the programmer to simply repeat cast #2. Cast #3 could thus be considered falling into both categories 1 and 3.

Overall, the example has 10 array accesses. The type system is able to show that 7 of those (including 5 in the innermost loop) are statically safe. Given that two of the remaining casts are identical, only a single bounds-check remains within any of the loops of the example.

## 4.2 The ArrayBench Benchmark Suite

The ArrayBench Benchmark Suite consists of seven benchmark programs. We adapted the benchmarks from code written in X10, mainly by making the code use regions. This section briefly explains the functionality of each benchmark, the style of parallelism (if any) and the overall amount of communication. Table 2 gives some fundamental benchmark statistics. LOC denotes the number of lines of code; # IR denotes the number of nodes in the intermediate representation.

The X10 language model features two levels of parallelism: parallel execution on different places and parallel execution at the same place. Consequently, for each benchmark program we will give three figures: *PP*, *SP* and *SW*. The figure *PP* is the amount of place-parallelism (for a maximum of *P* places available) and describes how many places compute in parallel. A value of 1 indicates that a computation is not distributed, a value of *P* is used for a computation that uses all available places in parallel. The figure *SP* is the amount of

name	LOC	# IR	PP	SP	SW	$O_M$	$O_S$
Series	87	2018	<i>P</i>	1	$n/P$	0	0
KMP	74	2407	1	1	$m + n$	0	0
Reverse	96	3659	<i>P</i>	1	$n/P$	$P^2$	$n$
Crypt	250	5759	1	<i>P</i>	$n/P$	0	0
Crypt-2	220	5873	1	<i>P</i>	$n/P$	0	0
SOR	70	1702	1	$n$	$n$	0	0
DSOR	68	1742	<i>P</i>	$n/P$	$n$	$n$	$n^2$

**Table 2.** Size (in lines of code (LOC) and number of nodes in the intermediate representation (# IR) of the compiler) and classification of parallelism for the benchmarks.

single-place parallelism, in other words, how many activities are running in parallel at the same place. In particular, these places will be able to access the same share of the global partitioned address space. A value of 1 indicates that there is only one activity per place involved in the computation. Finally, the figure *SW* is the amount of sequential work that each parallel activity performs. The product of *PP*, *SP* and *SW* gives the total amount of work required for the benchmark (for example,  $O(n^2)$  for SOR and DSOR).

For communication, we give two figures.  $O_M$  is the number of messages exchanged.  $O_S$  is the sum of the size of these messages. The figures for communication do not include initial distribution of the computation and data (which for all parallel benchmarks can be done with  $O_M(P)$  messages transmitting  $O_S(n/P)$  data with *P* being the number of places).

The ArrayBench benchmarks implement the following algorithms:

**Series:** Calculates the first  $n$  fourier coefficients of the function  $(x + 1)^x$  defined on the interval  $[0, 2]$ . Uses one array-access cast in source code.

**KMP:** Sequential implementation of Knuth-Morris-Pratt string searching algorithm (with pattern of size  $m$  and string of size  $n$ ). Uses six array-access casts in source code.

**Reverse:** Given an array distributed across places, reverses the order of the elements. Uses two array-access casts in source code.

**Crypt:** Implements the IDEA symmetric blockcipher (encrypt and decrypt) using integer increment operations to iterate over a stream. Uses nine array-access casts in source code.

**Crypt-2:** Implements the IDEA symmetric blockcipher (encrypt and decrypt) using region iterators to iterate over a stream. Uses three array-access casts in source code.

**SOR:** Given a 2D array, performs successive over-relaxation [20] of an  $n \times n$  matrix. Uses two array-access casts in source code.

**DSOR:** Given a 2D array, performs distributed successive over-relaxation of an  $n \times n$  matrix. Uses no array-access casts.

name	entailment checks		dynamic array-access casts			
	total number	max. size	without types		with types	
			S	L	S	L
Series	7324	24	12	23	2	2
KMP	11705	42	150	618	124	496
Reverse	48138	46	114	240	12	48
Crypt	24898	24	2684	9980	2591	9887
Crypt-2	65316	31	2684	9980	15	15
SOR	62488	95	192	1200	2	2
DSOR	105374	115	192	1200	0	0

**Table 3.** Numbers of dynamic checks required for the benchmarks.

In summary, the seven benchmarks need 23 array-access casts in a total of 865 lines of array-intensive code; that is about one cast per 37 lines of code. Our compiler helps with finding the places where casts are needed. We conclude that a programmer needs insert only a few array-access casts to make a program type check. Because the casts are infrequent, the effort required from the programmer to investigate possible restructuring of the code to eliminate such casts—should they be in performance-critical sections of the code—is acceptable.

### 4.3 Measurements and Assessment

We collected our run-time measurements by instrumenting the implementation of our X10 variant.

Table 3 table shows the number of dynamic checks required for the various benchmarks. We ran each benchmark on two input sizes (marked as “S” for small input, and as “L” for large input).

Using the classification scheme described earlier, the majority of the static type casts required for the ArrayBench suite falls into the category (3), followed by casts in category (1). Casts in category (1) are usually obvious to the programmer and have no runtime overhead. Determining that a cast falls into category (2) or (3) is less obvious – the reason for this is that there might be non-obvious ways to change the structure or typing of the code which would allow the cast to be eliminated.

Using the types, the compiler will verify that all array accesses are in bounds and local using a decision procedure that tries to determine subset relationships between symbolic expressions. Note that the XTC-X10 compiler allows overloading of methods based on dependent typing, resulting in many more invocations of the decision procedure than there are static array accesses in the code. The heuristic used to determine subset relationships that is implemented in our prototype has exponential complexity. However, the problem sizes are relatively small (up to 115 nodes in the symbolic expression tree for ArrayBench). We expect this to continue to be true even for larger benchmarks than the ones studied since type checking can be done per method, and individ-

ual methods are unlikely to become extremely large. For the size of the expressions studied in our experiments, the execution time of our heuristic is so fast that it cannot be properly measured, especially given that the implementation is currently in Java where noise from the garbage collector and JIT compiler interfere with measurements on that scale. The total compile time of the ArrayBench benchmarks, including parsing and compilation of 3.000 lines of core X10 runtime libraries, is about 5s on a modern machine for a cold run of the JVM.

Our prototype does not allow us to gather meaningful runtime performance data for the generated code. XTC-X10 compiles the benchmarks into SSA-form which is currently interpreted using a multi-threaded interpreter which is written in Java and simulates a distributed runtime. While this does not allow us to give specific speed-up data, it is possible to count the number of bounds and place checks that a language without region types would have to perform and compare it to the number of dynamic region and place casts (which are equivalent to those bounds and place checks) in the typed language. We do not distinguish between bounds checks and place checks because for array locality, any place check is effectively a bounds check for place-adjusted bounds. Consequently, for some particular checks, the distinction would often not be possible.

As expected, the typed language always outperforms the untyped language in terms of the total number of dynamic checks required. For some benchmarks (KMP, Crypt), the reduction that can be achieved is rather small – here, most accesses had to be converted into casts of category (3). For other benchmarks, only a handful of casts remain, and these are often in code that is run only once. This is illustrated by running the benchmarks with two different input sizes. For Series, Crypt2, SOR and DSOR, the total number of dynamic checks does not change if the problem size is increased. The reason for this is that the casts here deal with corner cases, such as initialization. Note that the particular problem sizes chosen for the benchmarks are tiny – for example, the smaller version of Crypt uses a stream of 128 bytes, SOR uses a 6x6 array, and Series computes 3 Fourier coefficients. For larger benchmark sizes, the reduction in the number of dynamic checks will clearly be more dramatic, as shown by the respective second dynamic values.

The Crypt-2 benchmark deserves some further discussion. The difference between Crypt and Crypt-2 is that most casts were eliminated by replacing the integer-arithmetic that was used to walk over the stream (`i++`) with iterators over regions. These iterators are equivalent to the generators of the ordered point list in the operational semantics of the `for` statement in the core language. In particular, they are guaranteed to yield only points that are inside of the region (unlike the `i++` statement which, if used in a loop, does not have an obvious bound). Permitting the programmer to use the (region-typed) iterators directly instead of a `for` loop al-

lows preservation of the original structure of the code. Iterators do have the disadvantage that there is an implicit check – as part of the iterator logic, the iterator verifies that a next element actually exists. This check is a range check that could be seen as a bounds check; however, the check of the iterator is also similar to the bounds check performed by any `for` loop. The numbers given for Crypt-2 do not include the test performed by the iterator, just as the numbers in all benchmarks do not include tests performed for the execution of `for` loops.

In summary, when we run seven statically-typed benchmarks with large inputs, the run-time system will in one case execute no array-access casts at all, and in another case execute almost as many array-access casts as an execution that checks all array accesses. The good news is that in most of the cases, rather few dynamic array-access casts are executed. We conclude that the type system is successful in achieving both better safety and better performance.

## 5. Future Work and Conclusion

In future work we will investigate how to type-check user-defined distributions. We also plan to study richer constraint systems that can represent the particularities of specific programming idioms. Our existing X10 prototype already supports an extended constraint algebra beyond that used in the core language. In particular, the algebra includes support for arithmetic constraints. The extended algebra is needed in order to type check common constructs in actual applications. The underlying principles of the type system presented in this paper are independent of the particular choice of constraint algebra, which we expect to evolve in step with the power of constraint solvers and the needs of application developers.

Our core language may be a good starting point for other foundational work on programming languages for multi-core systems. Our type system contains an interesting mix of dependent types and set constraints which may be more broadly applicable. We have proved our type system sound and shown via experiments that seven benchmarks can be type checked after the insertion of a few array-access casts.

## Acknowledgments

We thank Neal Glew for finding a bug in a previous version of the type rules. We thank Christopher Donawa and Rajkishore Barik for sharing with us with performance data on the cost of bounds-checking in Java and X10. We thank Krista Grothoff for editing. The research work reported here is supported in part by DARPA under contract number NBCH30390004.

## 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] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The fortress language specification version 0.618. Technical report, Sun Microsystems, Inc., 2005.
- [3] David Aspinall and Martin Hofmann. Dependent types. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter Dependent Types, pages 45–86. The MIT Press, 2005.
- [4] 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.
- [5] Brad Chamberlain, Steve Deitz, Mary Beth Hribar, and Wayne Wong. Chapel. Technical report, Cray Inc, <http://chapel.cs.washington.edu>, 2007.
- [6] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. The case for high level parallel programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–86, July–September 1998.
- [7] 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.
- [8] Bradford L. Chamberlain, E Christopher Lewis, Calvin Lin, and Lawrence Snyder. Regions: An abstraction for expressing array computation. In *Proceedings of the ACM International Conference on Array Programming Languages*, 1999.
- [9] 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 programming, systems, languages, and applications*, pages 519–538. ACM SIGPLAN, 2005.
- [10] Daniel Chavarría-Miranda and John Mellor-Crummey. Effective communication coalescing for data-parallel applications. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 14–25, New York, NY, USA, 2005. ACM Press.
- [11] Paul Feautrier. Toward automatic partitioning of arrays on distributed memory computers. In *ICS '93: Proceedings of the 7th international conference on Supercomputing*, pages 175–184, New York, NY, USA, 1993. ACM Press.
- [12] Manish Gupta and Prithviraj Banerjee. Paradigm: a compiler for automatic data distribution on multicomputers. In *ICS '93: Proceedings of the 7th international conference on Supercomputing*, pages 87–96, New York, NY, USA, 1993. ACM Press.

- [13] 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.
- [14] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *ACM Trans. Program. Lang. Syst.*, 20(4):869–916, 1998.
- [15] Donald Knuth, Jr James H. Morris, and Vaughan Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
- [16] Peizong Lee and Zvi Meir Kedem. Automatic data and computation decomposition on distributed memory parallel computers. *ACM Trans. Program. Lang. Syst.*, 24(1):1–50, 2002.
- [17] Ben Liblit and Alexander Aiken. Type systems for distributed data structures. In *Proceedings of POPL'00, 27th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 199–213, 2000.
- [18] Flemming Nielson. The typed lambda-calculus with first-class processes. In *Proceedings of PARLE'89*, pages 357–373, 1989.
- [19] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *ACM SIGPLAN Fortran Forum Archive*, 17:1–31, August 1998.
- [20] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. Successive overrelaxation (sor). In *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, pages 866–869. Cambridge University Press, 1992.
- [21] 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.
- [22] 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.
- [23] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
- [24] 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.
- [25] 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.

## A. Proof of Type Preservation

Here is the statement of Type Preservation (Theorem 1):

For a place  $P$ , let  $Q \in \{P, \text{unknown}\}$ . If  $\Psi; \varphi; \Gamma; Q \vdash e : t, \models H : \Psi$ , and  $P \vdash (H, e) \rightsquigarrow (H', e')$ , then we have  $\Psi', t'$  such that  $\Psi \triangleleft \Psi', \Psi'; \varphi; \Gamma; Q \vdash e' : t', \models H' : \Psi'$ , and  $\varphi \vdash t \equiv t'$ .

*Proof.* We proceed by induction on the structure of the derivation of  $\Psi; \varphi; \Gamma; Q \vdash e : t$ . There are now twenty-five subcases depending on which one of the type rules was the last one used in the derivation of  $\Psi; \varphi; \Gamma; Q \vdash e : t$ .

In eight of those cases,  $e$  is either a value or a variable  $x$ , and hence  $(H, e)$  cannot take a step. We will now consider each of the remaining seventeen cases.

- Rule (64): the derivation is of the form:

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

We now have three subcases depending on which rule was used to make  $(H, e_1 e_2)$  take a step.

If Rule (1), that is,

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e'_1)}{P \vdash (H, e_1 e_2) \rightsquigarrow (H', e'_1 e_2)}$$

was used to take a step, then we have from the induction hypothesis that we have  $\Psi'$  such that  $\Psi \triangleleft \Psi', \Psi'; \varphi; \Gamma; Q \vdash e'_1 : t_1 \rightarrow t_2$ , and  $\models H' : \Psi'$ . From  $\Psi \triangleleft \Psi'$  and  $\Psi; \varphi; \Gamma; Q \vdash e_2 : t_1$  we have  $\Psi'; \varphi; \Gamma; Q \vdash e_2 : t_1$ . From  $\Psi'; \varphi; \Gamma; Q \vdash e'_1 : t_1 \rightarrow t_2$  and  $\Psi'; \varphi; \Gamma; Q \vdash e_2 : t_1$ , and Rule (64), we conclude  $\Psi'; \varphi; \Gamma; Q \vdash e'_1 e_2 : t_2$ .

If Rule (2), that is,

$$\frac{P \vdash (H, e_2) \rightsquigarrow (H', e'_2)}{P \vdash (H, v e_2) \rightsquigarrow (H', v e'_2)}$$

was used to take a step, then we have from the induction hypothesis that we have  $\Psi'$  such that  $\Psi \triangleleft \Psi', \Psi'; \varphi; \Gamma; Q \vdash e'_2 : t_1$ , and  $\models H' : \Psi'$ . From  $\Psi \triangleleft \Psi'$  and  $\Psi; \varphi; \Gamma; Q \vdash e_1 : t_1 \rightarrow t_2$  we have  $\Psi'; \varphi; \Gamma; Q \vdash e_1 : t_1 \rightarrow t_2$ . From  $\Psi'; \varphi; \Gamma; Q \vdash e_1 : t_1 \rightarrow t_2$  and  $\Psi'; \varphi; \Gamma; Q \vdash e'_2 : t_1$ , and Rule (64), we conclude  $\Psi'; \varphi; \Gamma; Q \vdash e_1 e'_2 : t_2$ .

If Rule (3), that is,

$$P \vdash (H, (\lambda x : t.e)v) \rightsquigarrow (H, e[x := v])$$

was used to take a step, then we have from Rule (61) that  $\Psi; \varphi; \Gamma[x : t_1]; Q \vdash e : t_2$ , so we pick  $\Psi' = \Psi$  and we have from Lemma 1 that  $\Psi; \varphi; \Gamma; Q \vdash e[x := v] : t_2$ .

- Rule (65): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash e_1 : \Pi \alpha : k.t_1 \quad \Psi; \varphi; \Gamma; Q \vdash e_2 : t_2 \quad \vdash t_2 : k \triangleright W \quad \varphi \models (\text{constraint}(k))[\alpha := W]}{\Psi; \varphi; \Gamma; Q \vdash e_1 \langle e_2 \rangle : t_1[\alpha := W]}$$



We now have three subcases depending on which rule was used to make  $(H, e_1 < e_2 >)$  take a step.

If Rule (4) or Rule (5) was used to take a step, then the proof is similar to that given above for the case of function application (Rule (1)); we omit the details.

If Rule (6), that is,

$$P \vdash (H, (\mathbf{1am} \alpha : k.e) < w >) \rightsquigarrow (H, e[\alpha := w])$$

was used to take a step, then we have from Rule (62) that  $\Psi; \varphi \wedge \mathit{constraint}(k); \Gamma; \mathit{unknown} \vdash e : t$ . We pick  $\Psi' = \Psi$ . We pick  $\alpha$  such that  $\alpha$  does not occur free in  $\varphi$ . Let  $\varphi' = \mathit{constraint}(k)$ . From  $\Psi; \varphi \wedge \varphi'; \Gamma; Q \vdash e : t$  and Lemma 2, we have  $\Psi; (\varphi \wedge \varphi')[\alpha := W]; \Gamma; Q \vdash e[\alpha := W] : t[\alpha := W]$ , which is the same as  $\Psi; \varphi \wedge (\varphi'[\alpha := W]); \Gamma; Q \vdash e[\alpha := W] : t[\alpha := W]$ . From  $\Psi; \varphi \wedge (\varphi'[\alpha := W]); \Gamma; Q \vdash e[\alpha := W] : t[\alpha := W]$ ,  $\varphi \models \varphi'[\alpha := W]$ , and Lemma 3, we have  $\Psi; \varphi; \Gamma; Q \vdash e[\alpha := W] : t[\alpha := W]$ .

- Rule (66): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma[x : t_1]; Q \vdash e : t_2 \quad Q \neq \mathit{unknown}}{\Psi; \varphi; \Gamma; Q \vdash \lambda^{\bullet} x : t_1. e : t_1 \rightarrow t_2}$$

If Rule (7), that is,

$$P \vdash (H, \lambda^{\bullet} x : t_1. e) \rightsquigarrow (H, \lambda x : t_1. \mathbf{at}(P)\{e\})$$

was used to take a step, then from  $Q \in \{P, \mathit{unknown}\}$  and  $Q \neq \mathit{unknown}$ , we have  $Q = P$ . From Rule (60) we have  $\Psi; \varphi; \Gamma[x : t_1]; \mathit{unknown} \vdash P : \mathbf{p1} P$ . From  $\Psi; \varphi; \Gamma[x : t_1]; \mathit{unknown} \vdash P : \mathbf{p1} P$  and  $\Psi; \varphi; \Gamma[x : t_1]; P \vdash e : t_2$  and Rule (81), we have  $\Psi; \varphi; \Gamma[x : t_1]; \mathit{unknown} \vdash \mathbf{at}(P)\{e\} : t_2$ . From  $\Psi; \varphi; \Gamma[x : t_1]; \mathit{unknown} \vdash \mathbf{at}(P)\{e\} : t_2$  and Rule (61) we have  $\Psi; \varphi; \Gamma; Q \vdash \lambda x : t_1. \mathbf{at}(P)\{e\} : t_1 \rightarrow t_2$ .

- Rule (67): the derivation is of the form:

$$\frac{\Psi; \varphi \wedge \mathit{constraint}(k); \Gamma; \mathit{here} \vdash e : t \quad \mathit{here} \neq \mathit{unknown}}{\Psi; \varphi; \Gamma; \mathit{here} \vdash \mathbf{lam}^{\bullet} \alpha : k. e : \Pi \alpha : k. t}$$

If Rule (8), that is,

$$P \vdash (H, \mathbf{lam}^{\bullet} \alpha : k. e) \rightsquigarrow (H, \mathbf{1am} \alpha : k. \mathbf{at}(P)\{e\})$$

was used to take a step, then we can prove that  $\Psi; \varphi; \Gamma; \mathit{here} \vdash \mathbf{1am} \alpha : k. \mathbf{at}(P)\{e\} : \Pi \alpha : k. t$  in a manner similar to the previous case of Rule (66); we omit the details.

- Rule (68): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash e : \mathbf{reg} r}{\Psi; \varphi; \Gamma; Q \vdash \mathbf{new} t[e] : t[r]}$$

We now have two subcases depending on which rule was used to make  $(H, \mathbf{new} t[e])$  take a step.

If Rule (9) was used to take a step, then the proof is similar to that given above for the case of function application (Rule (1)); we omit the details.

If Rule (10), that is,

$$P \vdash (H, \mathbf{new} t[R]) \rightsquigarrow (H[l \mapsto \lambda p \in R. (\mathbf{default}(t), \mathit{distribute}(R, p))], l) \text{ where } l \text{ is fresh}$$

was used to take a step, then we have  $e = R$ , so from Rule (58) we have  $r = R$ . We define  $\Psi'$  to be an extension of  $\Psi[l \mapsto t[R]]$  such that  $\Psi \triangleleft \Psi'$  and  $\Psi'$  contains suitable definitions for the labels used in  $\mathbf{default}(t)$ ; we omit the details. Let  $H'$  be an extension of  $H[l \mapsto \lambda p \in R. (\mathbf{default}(t), \mathit{distribute}(R, p))]$  such that  $H'$  contains suitable definitions for the labels used in  $\mathbf{default}(t)$ ; we omit the details. From Rule (59) we have  $\Psi'; \varphi; \Gamma; Q \vdash l : \Psi'(l)$ . We finally need to show  $\models H' : \Psi'$ . From the construction of  $\Psi'$  and  $H'$  we have that they extend the domains of  $\Psi$  and  $H$ , respectively, with the same labels. From  $\models H : \Psi$  we have  $\mathcal{D}(H) = \mathcal{D}(\Psi)$ , so we conclude  $\mathcal{D}(H') = \mathcal{D}(\Psi')$ . Moreover, we have  $R = \mathcal{D}(\lambda p \in R. (\mathbf{default}(t), \mathit{distribute}(R, p)))$  and we have  $\Psi'; \varphi; \Gamma; Q \vdash \mathbf{default}(t) : t$ . Finally, for each  $p \in R$  we have  $H(l)(p).2 = \mathit{distribute}(R, p)$ .

- Rule (69): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash y_1 : t[r_1] \quad \Psi; \varphi; \Gamma; Q \vdash y_2 : \mathbf{pt}(\sigma, r_2) \quad \varphi \models r_2 \subseteq_t r_1 \quad \varphi \models \sigma \in_t r_2 \quad \varphi \vdash Q \equiv r_1[\@_t(\sigma, r_2)]}{\Psi; \varphi; \Gamma; Q \vdash y_1[y_2] : t}$$

If Rule (11), that is,

$$P \vdash (H, l[p]) \rightsquigarrow (H, H(l)(p).1) \text{ if } l \in \mathcal{D}(H) \text{ and } p \in \mathcal{D}(H(l)) \text{ and } P = H(l)(p).2$$

was used to take a step, then we have  $y_1 = l$  and  $y_2 = p$ . From  $\Psi; \varphi; \Gamma; Q \vdash l : t[r_1]$  and Rule (59) we have that  $r_1 = R$  and  $\Psi(l) = t[R]$ . We pick  $\Psi' = \Psi$  and from  $\models H : \Psi$  we have  $\Psi; \varphi; \Gamma; Q \vdash H(l)(p).1 : t$ .

- Rule (70): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash y_1 : t[r_1] \quad \Psi; \varphi; \Gamma; Q \vdash y_2 : \mathbf{pt}(\sigma, r_2) \quad \varphi \models r_2 \subseteq_t r_1 \quad \varphi \models \sigma \in_t r_2 \quad \varphi \vdash Q \equiv r_1[\@_t(\sigma, r_2)]}{\Psi; \varphi; \Gamma; Q \vdash y_1[y_2] = e : t}$$

We now have two subcases depending on which rule was used to make  $(H, y_1[y_2] = e_3)$  take a step.

If Rule (12) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (1); we omit the details.

If Rule (13), that is,

$$P \vdash (H, l[p] = v) \rightsquigarrow (H[l \mapsto (H(l))[p \mapsto (v, H(l)(p).2)]], v) \text{ if } l \in \mathcal{D}(H) \text{ and } p \in \mathcal{D}(H(l)) \text{ and } P = H(l)(p).2$$

was used to take a step, then we have  $y_1 = l$ ,  $y_2 = p$ ,  $e = v$ . From  $\Psi; \varphi; \Gamma; Q \vdash l : t[r_1]$  and Rule (59) we have that  $r_1 = R$  and  $\Psi(l) = t[R]$ . We have  $\Psi; \varphi; \Gamma; Q \vdash$

$v : t$  so we need to prove  $\models H[l \mapsto (H(l))[p \mapsto (v, H(l)(p).2)]] : \Psi$ . From  $l \in \mathcal{D}(H)$  we have  $\mathcal{D}(H[l \mapsto (H(l))[p \mapsto v]]) = \mathcal{D}(H)$ . Notice that  $H(l)(p).2 = H[l \mapsto (H(l))[p \mapsto (v, H(l)(p).2)](l)(p).2$ . The remaining thing to prove is

$$\Psi; \varphi; \Gamma; Q \vdash (H[l \mapsto (H(l))[p \mapsto (v, H(l)(p).2)])(l)(p).1 : t.$$

We have  $(H[l \mapsto (H(l))[p \mapsto (v, H(l)(p).2)])(l)(p).1 = v$  and we have  $\Psi; \varphi; \Gamma; Q \vdash v : t$ .

- Rule (71): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash e : t[r]}{\Psi; \varphi; \Gamma; Q \vdash e.\text{reg} : \text{reg } r}$$

We now have two subcases depending on which rule was used to make  $(H, e.\text{reg})$  take a step.

If Rule (14) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (1); we omit the details.

If Rule (15), that is,

$$P \vdash (H, l.\text{reg}) \rightsquigarrow (H, \mathcal{D}(H(l))) \quad \text{if } l \in \mathcal{D}(H)$$

was used to take a step, then we have from  $\Psi; \varphi; \Gamma; Q \vdash l : t[r]$  and Rule (59) that  $\Psi(l) = t[r]$ . Moreover we have that  $r$  is of the form  $R$ . From  $\models H : \Psi$  and  $\Psi(l) = t[R]$ , we have  $\mathcal{D}(H(l)) = R$ . We pick  $\Psi' = \Psi$  and from Rule (58) we conclude  $\Psi; \varphi; \Gamma; Q \vdash \mathcal{D}(H(l)) : \text{reg } R$ .

- Rule (72): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash y_1 : t[r_1] \quad \Psi; \varphi; \Gamma; Q \vdash y_2 : \text{pt}(\sigma, r_2) \quad \varphi \models r_2 \subseteq_t r_1 \quad \varphi \models \sigma \in_t r_2}{\Psi; \varphi; \Gamma; Q \vdash y_1[\text{@}_s y_2] : \text{p1 } r_1[\text{@}_t(\sigma, r_2)]}$$

If Rule (16), that is,

$$P \vdash (H, l[\text{@}_s p]) \rightsquigarrow (H, H(l)(p).2) \quad \text{if } l \in \mathcal{D}(H) \text{ and } p \in \mathcal{D}(H(l))$$

was used to take a step, then we have  $y_1 = l$  and  $y_2 = p$ . From  $\Psi; \varphi; \Gamma; Q \vdash l : t[r_1]$  and Rule (59) we have that  $r_1 = R$  and  $\Psi(l) = t[R]$ . From  $\Psi; \varphi; \Gamma; Q \vdash p : \text{pt}(\sigma, r_2)$  and Rule (57) we have that  $\sigma = p$ . We have  $H' = H$  and we pick  $\Psi' = \Psi$ . From  $\models H : \Psi$  we have  $H(l)(p).2 = \text{distribute}(R, p)$  and  $\mathcal{D}(H(l)) = R$ . From Rule (60) we have that we must show  $H(l)(p).2 \equiv r_1[\text{@}_t(\sigma, r_2)]$ . We have  $r_1[\text{@}_t(\sigma, r_2)] = R[\text{@}_t(p, r_2)]$ . We have  $H(l)(p).2 = \text{distribute}(R, p)$ . We also have  $\varphi \models r_2 \subseteq_t R$  and  $\varphi \vdash p \in_t r_2$  so from Rule (53) we have  $\varphi \vdash R[\text{@}_t(p, r_2)] \equiv \text{distribute}(R, p)$ . We conclude  $H(l)(p).2 = \text{distribute}(R, p) \equiv R[\text{@}_t(p, r_2)] = r_1[\text{@}_t(\sigma, r_2)]$ , as desired.

- Rule (73): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash e_1 : \text{reg } r_1 \quad \Psi; \varphi; \Gamma; Q \vdash e_2 : \text{reg } r_2}{\Psi; \varphi; \Gamma; Q \vdash e_1 \cup_s e_2 : \text{reg } r_1 \cup_t r_2}$$

We now have three subcases depending on which rule was used to make  $(H, e_1 \cup_s e_2)$  take a step.

If Rule (17) or Rule (18) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (1); we omit the details.

If Rule (19), that is,

$$P \vdash (H, R_1 \cup_s R_2) \rightsquigarrow (H, R_1 \cup R_2)$$

was used to take a step, then we have from Rule (58) that we must show  $\varphi \vdash R_1 \cup_t R_2 \equiv R_1 \cup R_2$ , which is Rule (43).

- Rule (74): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash e_1 : \text{reg } r_1 \quad \Psi; \varphi; \Gamma; Q \vdash e_2 : \text{reg } r_2}{\Psi; \varphi; \Gamma; Q \vdash e_1 \cap_s e_2 : \text{reg } r_1 \cap_t r_2}$$

We now have three subcases depending on which rule was used to make  $(H, e_1 \cap_s e_2)$  take a step.

If Rule (20) or Rule (21) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (1); we omit the details.

If Rule (22), that is,

$$P \vdash (H, R_1 \cap_s R_2) \rightsquigarrow (H, R_1 \cap R_2)$$

was used to take a step, then we have from Rule (58) that we must show  $\varphi \vdash R_1 \cap_t R_2 \equiv R_1 \cap R_2$ , which is Rule (45).

- Rule (75): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash e : \text{reg } r}{\Psi; \varphi; \Gamma; Q \vdash e +_s c : \text{reg } r +_t c}$$

We now have two subcases depending on which rule was used to make  $(H, e +_s c)$  take a step.

If Rule (23) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (1); we omit the details.

If Rule (24), that is,

$$P \vdash (H, R +_s c) \rightsquigarrow (H, R + c)$$

was used to take a step, then we have from Rule (58) that we must show  $\varphi \vdash R +_t c \equiv R + c$ , which is Rule (47).

- Rule (76): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash e : \text{pt}(\sigma, r)}{\Psi; \varphi; \Gamma; Q \vdash e ++_s c : \text{pt}(\sigma ++_t c, r +_t c)}$$

We now have two subcases depending on which rule was used to make  $(H, e ++_s c)$  take a step.

If Rule (25) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (1); we omit the details.

If Rule (26), that is,

$$P \vdash (H, p ++_s c) \rightsquigarrow (H, p + c)$$

was used to take a step, then we have from Rule (57) that we must show  $\varphi \vdash p ++_t c \equiv p + c$ , which is Rule (51).

- Rule (77): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash y_1 : \text{reg } r \quad \Psi; \varphi; \Gamma; Q \vdash y_2 : \text{pl } \pi}{\Psi; \varphi; \Gamma; Q \vdash y_1 \%_s y_2 : \text{reg } r \%_t \pi}$$

If Rule (27), that is,

$$P \vdash (H, R \%_s P') \rightsquigarrow (H, R')$$

$$\text{where } R' = \{ p \in R \mid \text{distribute}(R, p) = P' \}$$

was used to take a step, then we have from Rule (58) that we must show  $\varphi \vdash r \%_s \pi \equiv R'$ , which is Rule (49).

- Rule (78): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash e_1 : \text{reg } r \quad Q \neq \text{unknown} \quad \Psi; \varphi \wedge (\alpha \in_t r); \Gamma[x : \text{pt}(\alpha, r)]; Q \vdash e_2 : \text{int}}{\Psi; \varphi; \Gamma; Q \vdash \text{for}(x \text{ in } e_1)\{e_2\} : \text{int}} \quad (\alpha \text{ fresh})$$

We now have two subcases depending on which rule was used to make  $(H, \text{for}(x \text{ in } e_1)\{e_2\})$  take a step.

If Rule (28) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (1); we omit the details.

If Rule (29), that is,

$$P \vdash (H, \text{for}(x \text{ in } R)\{e_2\}) \rightsquigarrow$$

$$(H, ((\text{lam}^\bullet \alpha : \text{point}(\alpha \in_t R). \lambda^\bullet x : (\alpha, R).e_2) \langle c_1 \rangle) c_1; \dots; ((\text{lam}^\bullet \alpha : \text{point}(\alpha \in_t R). \lambda^\bullet x : (\alpha, R).e_2) \langle c_n \rangle) c_n; 0))$$

$$\text{where } \text{order}(R) = \langle c_1, \dots, c_n \rangle$$

was used to take a step, then we have  $r = R$ . From  $\Psi; \varphi; \Gamma[x : \text{pt}(\alpha, R)]; Q \vdash e_2 : t$  and  $Q \neq \text{unknown}$  and Rule (66) we have  $\Psi; \varphi; \Gamma; Q \vdash \lambda^\bullet x : (\alpha, R).e_2 : \text{pt}(\alpha, R) \rightarrow t$ . From  $\Psi; \varphi; \Gamma; Q \vdash \lambda^\bullet x : (\alpha, R).e_2 : \text{pt}(\alpha, R) \rightarrow t$  and Rule (67) we have  $\Psi; \varphi; \Gamma; Q \vdash \text{lam}^\bullet \alpha. \lambda^\bullet x : (\alpha, R).e_2 : \Pi \alpha : \text{point}(\alpha \in_t R). \text{pt}(\alpha, R) \rightarrow t$ . From Rule (57) and the definition of  $\text{order}(R)$  we have  $\Psi; \varphi; \Gamma; Q \vdash c_i : \text{pt}(c_i, R)$ . From  $\Psi; \varphi; \Gamma; Q \vdash c_i : \text{pt}(c_i, R)$  and  $\Psi; \varphi; \Gamma; Q \vdash \text{lam}^\bullet \alpha. \lambda^\bullet x : (\alpha, R).e_2 : \Pi \alpha : \text{point}(\alpha \in_t R). \text{pt}(\alpha, R) \rightarrow t$  and  $\vdash (c_i, R) : \text{point}(\alpha \in_t R) \triangleright c_i$  and  $\varphi \models \text{constraint}(\text{point})[\alpha := c_i]$  and Rule (65) we have  $\Psi; \varphi; \Gamma; Q \vdash (\text{lam}^\bullet \alpha. \lambda^\bullet x : (\alpha, R).e_2) \langle c_i \rangle : \text{pt}(c_i, R) \rightarrow t[\alpha := c_i]$ . From  $\Psi; \varphi; \Gamma; Q \vdash (\text{lam}^\bullet \alpha. \lambda^\bullet x : (\alpha, R).e_2) \langle c_i \rangle : \text{pt}(c_i, R) \rightarrow t[\alpha := c_i]$  and  $\Psi; \varphi; \Gamma; Q \vdash c_i : \text{pt}(c_i, R)$  and Rule (64) we have  $\Psi; \varphi; \Gamma; Q \vdash ((\text{lam}^\bullet \alpha. \lambda^\bullet x : (\alpha, R).e_2) \langle c_i \rangle) c_i : t[\alpha := c_i]$ . From Rule (80) and Rule (56) we conclude

$$\Psi; \varphi; \Gamma; Q \vdash ((\text{lam}^\bullet \alpha. \lambda^\bullet x : (\alpha, R).e_2) \langle c_1 \rangle) c_1; \dots; ((\text{lam}^\bullet \alpha. \lambda^\bullet x : (\alpha, R).e_2) \langle c_n \rangle) c_n; 0 : \text{int}.$$

- Rule (79): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma[x : \text{pl } \alpha]; Q \vdash e : \text{int} \quad Q \neq \text{unknown}}{\Psi; \varphi; \Gamma; Q \vdash \text{forallplaces } x \{e\} : \text{int}} \quad (\alpha \text{ fresh})$$

If Rule (30), that is,

$$P \vdash (H, \text{forallplaces } x \{e\}) \rightsquigarrow (H, ((\text{lam}^\bullet \alpha : \text{place}. \lambda^\bullet x : \text{pl } \alpha. e) \langle P_1 \rangle) P_1; \dots; ((\text{lam}^\bullet \alpha : \text{place}. \lambda^\bullet x : \text{pl } \alpha. e) \langle P_n \rangle) P_n; 0))$$

where  $\text{places} = \langle P_1, \dots, P_n \rangle$

was used to take a step, then we can prove that  $\Psi; \varphi; \Gamma; \text{here} \vdash ((\text{lam}^\bullet \alpha : \text{place}. \lambda^\bullet x : \text{pl } \alpha. e) \langle P_1 \rangle) P_1; \dots; ((\text{lam}^\bullet \alpha : \text{place}. \lambda^\bullet x : \text{pl } \alpha. e) \langle P_n \rangle) P_n; 0 : \text{int}$  in a fashion similar to the case for for-loops and Rule (78) and Rule(29); we omit the details.

- Rule (80): the derivation is of the form:

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

We now have two subcases depending on which rule was used to make  $(H, e_1; e_2)$  take a step.

If Rule (31) was used to take a step, then the proof is similar to that given above for the case of function application and Rule (1); we omit the details.

If Rule (32), that is,

$$P \vdash (H, v; e) \rightsquigarrow (H, e)$$

was used to take a step, then we pick  $\Psi' = \Psi$  and we have  $\Psi; \varphi; \Gamma; Q \vdash e : t_2$ .

- Rule (81): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma; Q \vdash y : \text{pl } \pi \quad \Psi; \varphi; \Gamma; \pi \vdash e : t}{\Psi; \varphi; \Gamma; Q \vdash \text{at}(y)\{e\} : t}$$

We now have two subcases depending on which rule was used to make  $(H, \text{at}(x)\{e\})$  take a step.

If Rule (33), that is,

$$\frac{P' \vdash (H, e) \rightsquigarrow (H', e')}{P \vdash (H, \text{at}(P')\{e\}) \rightsquigarrow (H, \text{at}(P')\{e'\})}$$

was used to take a step, then we have that  $y = P'$ . From Rule (60) we have that  $\pi = P'$ . So, we can apply the induction hypothesis to  $\Psi; \varphi; \Gamma; \pi \vdash e : t$  and get that  $\Psi; \varphi; \Gamma; \pi \vdash e' : t$ . From Rule (81) we conclude that  $\Psi; \varphi; \Gamma; Q \vdash \text{at}(y)\{e'\} : t$ .

If Rule (34), that is,

$$P \vdash (H, \text{at}(P')\{v\}) \rightsquigarrow (H, v)$$

was used to take a step, then we have  $H' = H$  and we pick  $\Psi' = \Psi$ . We also have  $e = v$ . From  $\Psi; \varphi; \Gamma; \pi \vdash v : t$  and Lemma 4, we have  $\Psi; \varphi; \Gamma; \text{here} \vdash v : t$ .

- Rule (82): the derivation is of the form

$$\frac{\Psi; \varphi; \Gamma; \text{here} \vdash e : t \quad \varphi \vdash t \equiv t'}{\Psi; \varphi; \Gamma; \text{here} \vdash e : t'}$$

From the induction hypothesis we have  $\Psi', t''$  such that  $\Psi \triangleleft \Psi', \Psi'; \varphi; \Gamma; Q \vdash e' : t'', \models H' : \Psi'$ , and

$\varphi \vdash t \equiv t''$ . From  $\varphi \vdash t \equiv t'$  and  $\varphi \vdash t \equiv t''$  and Rule (37) and Rule (36), we have  $\varphi \vdash t' \equiv t''$ . From Rule (82) we conclude that  $\Psi; \varphi; \Gamma; here \vdash e' : t'$ .  $\square$

## B. Proof of Progress

Here is the statement of Progress (Theorem 2):

For a place  $P$ , let  $Q \in \{P, unknown\}$ . If  $\Psi; true; \emptyset; Q \vdash e : t$  and  $\models H : \Psi$ , then  $(H, e)$  is not stuck at place  $P$ .

*Proof.* We proceed by induction on the structure of the derivation of  $\Psi; true; \emptyset; Q \vdash e : t$ . There are now twenty-five subcases depending on which one of the type rules was the last one used in the derivation of  $\Psi; true; \emptyset; Q \vdash e : t$ .

In seven of those cases, the derivation is of the form:  $\Psi; true; \emptyset; Q \vdash v : t$ . where  $v$  is a value, hence  $(H, v)$  is not stuck at place  $P$ . The derivation cannot be of the form:  $\Psi; true; \emptyset; Q \vdash x : t$  because Rule (63) cannot apply. We will now consider each of the remaining seventeen cases.

- Rule (64): the derivation is of the form:

$$\frac{\Psi; true; \emptyset; Q \vdash e_1 : t_1 \rightarrow t_2 \quad \Psi; true; \emptyset; Q \vdash e_2 : t_1}{\Psi; true; \emptyset; Q \vdash e_1 e_2 : t_2}$$

From the induction hypothesis, we have that  $(H, e_1)$ ,  $(H, e_2)$  are not stuck at place  $P$ . If  $(H, e_1)$  can take a step at place  $P$ , then  $(H, e_1 e_2)$  can also take a step at place  $P$  using Rule (1). If  $e_1$  is a value and  $(H, e_2)$  can take a step at place  $P$ , then also  $(H, e_1 e_2)$  can take a step at place  $P$  using Rule (2). If  $e_1, e_2$  are both values, then we have from Lemma 5 that  $e_1$  is of the form  $\lambda x : t.e$ , so  $(H, e_1 e_2)$  can take a step at place  $P$  using Rule (3).

- Rule (65): the derivation is of the form:

$$\frac{\Psi; true; \emptyset; Q \vdash e_1 : \Pi \alpha : k.t_1 \quad \Psi; true; \emptyset; Q \vdash e_2 : t_2 \vdash t_2 : k \triangleright W \quad true \models (constraint(k))[\alpha := W]}{\Psi; true; \emptyset; Q \vdash e_1 \langle e_2 \rangle : t_1[\alpha := W]}$$

From the induction hypothesis, we have that  $(H, e_1)$  is not stuck at place  $P$ . If  $(H, e_1)$  can take a step at place  $P$ , then  $(H, e_1 \langle e_2 \rangle)$  can take also a step at place  $P$  using Rule (4). If  $e_1$  is a value and  $(H, e_2)$  can take a step at place  $P$ , then  $(H, e_1 \langle e_2 \rangle)$  can also take a step at place  $P$  using Rule (5). If  $e_1, e_2$  are both values, then we have from Lemma 5 that  $e_1$  is of the form  $\lambda \text{am } \alpha : k.e$ , and we have from  $\vdash t_2 : k \triangleright W$  and Lemma 5 that  $e_2$  is of the form  $w$ , so  $(H, e_1 \langle e_2 \rangle)$  can take a step using Rule (6).

- Rule (66): the derivation is of the form:

$$\frac{\Psi; true; \emptyset[x : t_1]; Q \vdash e : t_2 \quad Q \neq unknown}{\Psi; true; \emptyset; Q \vdash \lambda^\bullet x : t_1.e : t_1 \rightarrow t_2}$$

From Rule (7) we have that  $\lambda^\bullet x : t_1.e$  can take a step.

- Rule (67): the derivation is of the form:

$$\frac{\Psi; \varphi \wedge constraint(k); \Gamma; Q \vdash e : t \quad Q \neq unknown}{\Psi; \varphi; \Gamma; here \vdash \text{lam}^\bullet \alpha : k.e : \Pi \alpha : k.t}$$

From Rule (8) we have that  $\text{lam}^\bullet \alpha : k.e$  can take a step.

- Rule (68): the derivation is of the form:

$$\frac{\Psi; true; \emptyset; Q \vdash e : \text{reg } r}{\Psi; true; \emptyset; Q \vdash \text{new } t[e] : t[r]}$$

From the induction hypothesis we have that  $(H, e)$  is not stuck at place  $P$ . If  $(H, e)$  can take a step at place  $P$ , then  $(H, \text{new } t[e])$  can also take a step at place  $P$  using Rule (9). If  $e$  is a value, then we have from Lemma 5 that  $e$  is of the form  $R$ , so  $(H, \text{new } t[e])$  can take a step using Rule (10).

- Rule (69): the derivation is of the form:

$$\frac{\Psi; true; \emptyset; Q \vdash y_1 : t[r_1] \quad \Psi; true; \emptyset; Q \vdash y_2 : \text{pt } (\sigma, r_2) \quad true \models r_2 \subseteq_t r_1 \quad \varphi \models \sigma \in_t r_2 \quad \varphi \vdash Q \equiv r_1[\text{@}_t(\sigma, r_2)]}{\Psi; true; \emptyset; Q \vdash y_1[y_2] : t}$$

We have that  $y_1, y_2$  must be values and we have from Lemma 5 that  $y_1$  is of the form  $l$ ,  $l \in \mathcal{D}(\Psi)$  and  $y_2$  is of the form  $p$ . Further we have that  $Q = P$ , since *unknown* is not equivalent to anything other than itself. Let  $t[R]$  denote  $\Psi(l)$ . From  $l \in \mathcal{D}(\Psi)$  and  $\models H : \Psi$ , we have that  $l \in \mathcal{D}(H)$  and  $R = \mathcal{D}(H(l))$ . We have  $r_1 = R$ . From the type rule for point constants, we have that  $r_2$  is of the form  $R'$  and that  $p \in R'$ . We have  $true \models R' \subseteq_t R$ . From  $true \models R' \subseteq_t R$ , we have  $R' \subseteq R$ , hence  $p \in R' \subseteq R$ . From  $H \models \Psi$  we have  $distribute(R, p) = H(l)(p)$ . From Rule (57) we have  $\sigma = p$ . We conclude  $P = r_1[\text{@}_t(\sigma, r_2)] = distribute(R, p) = H(l)(p)$ . So,  $(H, e_1[e_2])$  can take a step using Rule (11).

- Rule (70): the derivation is of the form:

$$\frac{\Psi; true; \emptyset; Q \vdash y_1 : t[r_1] \quad \Psi; true; \emptyset; Q \vdash y_2 : \text{pt } (\sigma, r_2) \quad true \models r_2 \subseteq_t r_1 \quad true \models \sigma \in_t r_2 \quad true \vdash Q \equiv r_1[\text{@}_t(\sigma, r_2)] \quad \Psi; true; \emptyset; Q \vdash e : t}{\Psi; true; \emptyset; Q \vdash y_1[y_2] = e : t}$$

We have that  $y_1, y_2$  must be values and we have from Lemma 5 that  $y_1$  is of the form  $l$ ,  $l \in \mathcal{D}(\Psi)$  and  $y_2$  is of the form  $p$ . Further we have that  $Q = P$ , since *unknown* is not equivalent to anything other than itself. From the induction hypothesis we have that  $(H, e)$  is not stuck at place  $P$ . If  $(H, e)$  can take a step at place  $P$ , then  $(H, y_1[y_2] = e)$  can also take a step at place  $P$  using Rule (12). Suppose now that  $e$  is a value. The proof that  $y_1[y_2] = e$  can take a step at place  $P$  using Rule (13) is similar to that given above for the case of array lookup (Rule (69)), because Rule (11) has the same side condition; we omit the details.

- Rule (71): the derivation is of the form:

$$\frac{\Psi; true; \emptyset; Q \vdash e : t[r]}{\Psi; true; \emptyset; Q \vdash e.\text{reg} : \text{reg } r}$$

From the induction hypothesis we have that  $(H, e)$  is not stuck. If  $(H, e)$  can take a step at place  $P$ , then  $(H, e.\text{reg})$  can also take a step using Rule (14). If  $e$  is a value, then  $(H, e.\text{reg})$  can take a step using Rule (15).

- Rule (72): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash y_1 : t[r_1] \quad \Psi; \text{true}; \emptyset; Q \vdash y_2 : \text{pt}(\sigma, r_2) \quad \text{true} \models r_2 \subseteq_t r_1 \quad \text{true} \models \sigma \in_t r_2}{\Psi; \text{true}; \emptyset; Q \vdash y_1[\text{@}_s y_2] : \text{pl } r_1[\text{@}_t(\sigma, r_2)]}$$

We have that  $y_1, y_2$  must be values and we have from Lemma 5 that  $y_1$  is of the form  $l$ ,  $l \in \mathcal{D}(\Psi)$  and  $y_2$  is of the form  $p$ . The proof that  $y_1[\text{@}_s y_2]$  can take a step at place  $P$  using Rule (16) is similar to that given above for the case of array lookup and Rule (69), because Rules (11) and (13) have a stronger side condition; we omit the details.

- Rule (73): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e_1 : \text{reg } r_1 \quad \Psi; \text{true}; \emptyset; Q \vdash e_2 : \text{reg } r_2}{\Psi; \text{true}; \emptyset; Q \vdash e_1 \cup_s e_2 : \text{reg } r_1 \cup_t r_2}$$

From the induction hypothesis we have that  $(H, e_1), (H, e_2)$  are not stuck at place  $P$ . If  $(H, e_1)$  can take a step at place  $P$ , then  $(H, e_1 \cup_s e_2)$  can also take a step at place  $P$  using Rule (17). If  $e_1$  is a value and  $(H, e_2)$  can take a step at place  $P$ , then  $(H, e_1 \cup_s e_2)$  can also take a step at place  $P$  using Rule (18). If  $e_1, e_2$  are both values, then we have from Lemma 5 that  $e_1$  is of the form  $R_1$  and that  $e_2$  is of the form  $R_2$ , so  $(H, e_1 \cup_s e_2)$  can take a step at place  $P$  using Rule (19).

- Rule (74): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e_1 : \text{reg } r_1 \quad \Psi; \text{true}; \emptyset; Q \vdash e_2 : \text{reg } r_2}{\Psi; \text{true}; \emptyset; Q \vdash e_1 \cap_s e_2 : \text{reg } r_1 \cap_t r_2}$$

From the induction hypothesis we have that  $(H, e_1), (H, e_2)$  are not stuck at place  $P$ . If  $(H, e_1)$  can take a step at place  $P$ , then  $(H, e_1 \cap_s e_2)$  can also take a step at place  $P$  using Rule (20). If  $e_1$  is a value and  $(H, e_2)$  can take a step at place  $P$ , then  $(H, e_1 \cap_s e_2)$  can also take a step at place  $P$  using Rule (21). If  $e_1, e_2$  are both values, then we have from Lemma 5 that  $e_1$  is of the form  $R_1$  and that  $e_2$  is of the form  $R_2$ , so  $(H, e_1 \cap_s e_2)$  can take a step at place  $P$  using Rule (22).

- Rule (75): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e : \text{reg } r}{\Psi; \text{true}; \emptyset; Q \vdash e +_s c : \text{reg } r +_t c}$$

From the induction hypothesis we have that  $(H, e)$  is not stuck at place  $P$ . If  $(H, e)$  can take a step at place  $P$ , then  $(H, e +_s c)$  can also take a step at place  $P$  using Rule (23). If  $e$  is a value, then we have from Lemma 5 that  $e$  is of the form  $R$ , so  $(H, e +_s c)$  can take a step at place  $P$  using Rule (24).

- Rule (76): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e : \text{pt}(\sigma, r)}{\Psi; \text{true}; \emptyset; Q \vdash e ++_s c : \text{pt}(\sigma ++_t c, r +_t c)}$$

From the induction hypothesis we have that  $(H, e)$  is not stuck at place  $P$ . If  $(H, e)$  can take a step at place  $P$ ,

then  $(H, e ++_s c)$  can also take a step at place  $P$  using Rule (25). If  $e$  is a value, then we have from Lemma 5 that  $e$  is of the form  $p$ , so  $(H, e ++_s c)$  can take a step at place  $P$  using Rule (26).

- Rule (77): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash y_1 : \text{reg } r \quad \Psi; \text{true}; \emptyset; Q \vdash y_2 : \text{pl } \pi}{\Psi; \text{true}; \emptyset; Q \vdash y_1 \%_s y_2 : \text{reg } r \%_t \pi}$$

We have that  $y_1, y_2$  must be values and we have from Lemma 5 that  $y_1$  is of the form  $R$  and  $y_2$  is of the form  $P'$ . So,  $(H, y_1 \%_s y_2)$  can take a step at place  $P$  using Rule (27).

- Rule (78): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e_1 : \text{reg } r \quad Q \neq \text{unknown} \quad \Psi; (\alpha \in_t r); \emptyset[x : \text{pt}(\alpha, r)]; Q \vdash e_2 : \text{int}}{\Psi; \text{true}; \emptyset; Q \vdash \text{for}(x \text{ in } e_1)\{e_2\} : \text{int}} \quad (\alpha \text{ fresh})$$

From the induction hypothesis we have that  $(H, e_1)$  is not stuck at place  $P$ . If  $(H, e_1)$  can take a step at place  $P$ , then  $(H, \text{for}(x \text{ in } e_1)\{e_2\})$  can also take a step at place  $P$  using Rule (28). If  $e_1$  is a value, then we have from Lemma 5 that  $e_1$  is of the form  $R$ , so  $(H, \text{for}(x \text{ in } e_1)\{e_2\})$  can take a step at place  $P$  using Rule (29).

- Rule (79): the derivation is of the form:

$$\frac{\Psi; \varphi; \Gamma[x : \text{pl } \alpha]; Q \vdash e : \text{int} \quad Q \neq \text{unknown}}{\Psi; \varphi; \Gamma; \text{here} \vdash \text{forallplaces } x\{e\} : \text{int}} \quad (\alpha \text{ fresh})$$

We have that  $\text{forallplaces } x\{e\}$  can take a step using Rule (30).

- Rule (80): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e_1 : t_1 \quad \Psi; \text{true}; \emptyset; Q \vdash e_2 : t_2}{\Psi; \text{true}; \emptyset; Q \vdash e_1; e_2 : t_2}$$

From the induction hypothesis we have that  $(H, e_1)$  is not stuck at place  $P$ . If  $(H, e_1)$  can take a step at place  $P$ , then  $(H, e_1; e_2)$  can also take a step at place  $P$  using Rule (31). If  $e_1$  is a value, then  $(H, e_1; e_2)$  can take a step at place  $P$  using Rule (32).

- Rule (81): the derivation is of the form:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash y : \text{pl } \pi \quad \Psi; \text{true}; \emptyset; \pi \vdash e : t}{\Psi; \text{true}; \emptyset; Q \vdash \text{at}(y)\{e\} : t}$$

We have that  $y$  must be a value and we have from Lemma 5 that  $y$  must be of the form  $P'$ . From the induction hypothesis we have that  $(H, e)$  is not stuck at place  $P'$ . If  $(H, e)$  can take a step at place  $P'$ , then  $(H, \text{at}(y)\{e\})$  can take a step at place  $P$  using Rule (33). If  $e$  is a value, then  $(H, \text{at}(y)\{e\})$  can take a step at place  $P$  using Rule (34).

- Rule (82): the derivation is of the form

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e : t \quad \text{true} \vdash t \equiv t'}{\Psi; \text{true}; \emptyset; Q \vdash e : t'}$$

From the induction hypothesis we have that  $e$  is not stuck at place  $P$ .

□