Preprint.

A Type System for Distributed Arrays

Christian Grothoff
UCLA Computer Science Department
University of California, Los Angeles
christian@grothoff.org

Jens Palsberg
UCLA Computer Science Department
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

Data locality and memory safety are important goals for high-performance programming languages targeting multi-core systems with non-uniform memory. The data locality aspect is important for distributed algorithms running on high-performance systems where remote data access can both be costly and require the language system to generate different and expensive code. This paper presents a core language and a type system that achieves memory safety including locality of access for distributed arrays. A well-typed program cannot violate the in-bounds or locality requirements when accessing an array.

The type system integrates dependent types with a new class of constraints over points, regions of points, and places. The key operation during type checking is constraint entailment; type checking is co-NP-complete.

We have implemented a prototype compiler for a variant of X10, a new language for high-performance computing, that extends the standard object-oriented type system of X10 with the dependent types of the core language. The paper presents encouraging experimental results that show that the type system can be used to effectively eliminate dynamic checks and statically ensure memory safety and locality of access.

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

General Terms languages, performance, theory, verification

Keywords dependent types, distributed memory, high performance computing, X10

1. Introduction

Current object-oriented language facilities for concurrent and distributed programming are inadequate for addressing the needs of modern parallel machines (SMPs and clusters) that feature a complex memory hierarchy and possibly non-uniform memory access. Given the likelihood that the majority of desktop systems in the future will be multi-core SMP designs and that the majority of server systems will be configured as tightly-coupled clusters, there is a need for new language systems that simplify the deployment of computations spanning multiple nodes. In order to be able to give a sufficiently simple and adequate cost model to programmers, a new language should require that basic read and write operations are local with respect to the place of execution. If all basic read and write operations are guaranteed to be local by the language, programmers will continue to be able to reason about the performance of their algorithms with respect to memory accesses.

Our work is heavily influenced by X10 [11]. X10 is a new language for high performance computing (HPC). X10 uses the model of a partitioned global address space. The language features the abstract notion of a *place* to denote the location at which computations are executed. Each partition of the global address space is associated with a particular place, and X10 mandates that all accesses to mutable data be place-local. Furthermore, X10 features language constructs for explicit parallel and distributed computations. Naturally, supporting parallel distributed computations requires support for distributed data structures – for arrays, in particular. X10's requirement that all accesses to mutable data must be place-local includes distributed arrays. Consequently, accesses to X10 arrays must not only in-bounds (X10 is memory-safe) but also be executed locally (X10 is place-safe) at the respective place associated with the index by the distribution of the array.

X10 is different from previous HPC languages in that it requires the programmer to write code that ensures data locality and enables programmers to ensure data locality by providing constructs to programatically shift the place of execution. programatically shift the place of execution. Having a general place-shifting mechanism is good news for application programmers, since it affords them a great deal of flexibility. However, it does make it more difficult for the language system to generate fast, safe code for these general distributed parallel computations.

A central problem in this context is to statically check that memory accesses are local in the presence of distributed arrays and place-shifting computations. This paper shows how to solve this problem using a new type system which employs dependent types over a particular vocabulary of constraints. There is a variety of constraint systems that can be chosen for this purpose; for the core language presented in this paper, we chose a small operational algebra for illustration which is sufficient to cover several fundamental examples. Changes to the constraint system would impact the complexity of the type checker, but do not change the core ideas necessary for the type soundness proof.

The results of this paper are the establishment of an applied dependently-typed lambda calculus [3] that can be used to establish locality of access for distributed arrays in a computation with place-shifting operations. We have proven type soundness and settled the complexity of the key decision problems. The paper illustrates the type system with various examples. Our results substantially generalize the pioneering work of Pfenning and Xi [28, 29] on dependently typed programming languages (by applying them to a clustered setting and extending the constraint domain to include

regions) and the work of Aiken and Liblit [20] (by covering place-shifting operations).

We have implemented the type system in an experimental compiler for X10 called XTC-X10. We have measured how many dynamic checks can be eliminated with this extension as a result of the static proof for several programs. The reduction in checks is substantial. In future work we plan to explicitly study richer constraint systems that can represent the particularities of specific programming idioms. Our existing prototype already supports an extended constraint algebra beyond that used in the core calculus. 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.

2. Related Work

Modern high performance computing (HPC) languages must contain a rich language for arrays, a dominant data-structure in the HPC space. Chamberlain et al. [8, 6, 10, 13] proposed regions as a construct for specifying array operations in languages for parallel programming. A region is a set of points, rather than an interval or a product of intervals, and enables the programmer to define an array over a region and to write a loop that iterates over a region. Regions were later adopted in Titanium [16] and X10 [11]; in X10, regions are a first-class values. All of ZPL, Titanium, and X10 provide the programmer with a rich algebra of region operators to manipulate arrays. A programmer can use regions to specify computations on dense or sparse, multidimensional and hierarchical arrays. While convenient, regions do not eliminate the risk of array bounds violations. Until now, language implementations have resorted to checking array accesses dynamically or to warning the programmer that bounds violations lead to undefined behavior. For performance and productivity, it is preferable that array computations are statically checked to be safe.

Our type system is inspired by that of Xi and Pfenning [28, 29]. Like Xi and Pfenning, we use dependent types to avoid array-bounds checks. Xi and Pfenning use a decision procedure based on Pressburger arithmetic [25] in order to show the safety of array accesses. In contrast to Xi and Pfenning's language and type system, we study a programming model and type system based on regions. Our type system uses types that are parameterized over regions. Operations on region values are mapped to corresponding operations on region types. The mapping is defined such that subset relations for regions values corresponds to subtyping of their respective types. Establishing that an index is in-bounds for a particular array is equivalent to establishing that region over which the index may range is a subset of the region over which the array is defined. A subtyping relationship between the respective region types implies the desired subset relationship and can thus be used to statically prove the safety of the access.

Furthermore, our type system needs to take data locality into account. For this purpose the type system supports the notion of distributions, functions from array indices to places. Distributions are used to determine how data stored in arrays is distributed among the places of the computation. The core language features various operations on distributions that are used to ensure locality of access. In this respect, our work generalizes the work of Aiken and Liblit [20] which introduces a type-system for distributed objects. In contrast to their work, our type-system covers place-shifting operations and includes distributed arrays.

An alternative to using types to eliminate bounds checks is the use of static analysis. Early work on using static analysis to eliminate bounds checks investigated the use of theorem proving [26] to eliminate checks. Our work is related in that we use types to guide a decision procedure, a technique which is also used in proof carrying code [22]. For just-in-time compiled languages such as Java where compile time is crucial, the ABCD algorithm [4] describes a light-weight analysis based on interval constraints that is capable of eliminating on average 45% of the array bounds checks. However, the results range from 0 to 100% for the various individual benchmarks, which may make it hard for programmers to write code that achieves consistently good performance.

When speed is of utmost concern, a language designer may decide to not require any bounds checks altogether. For example, the reference manual for Titanium [16], a modern language for high-performance computing, defines that operations which cause bounds violations result in the behavior of the rest of the program being *undefined*. The semantics of our core language is similar: a violation of locality or accesses out of bounds result in the semantics getting stuck. The contribution of our paper is a type system which guarantees statically that these violations cannot occur. Thus, our type system enables us to have both memory safety and high performance.

2.1 The core of six languages for high-performance computing

Computations with arrays using distributed memory are fundamental to high-performance computing (HPC). Recent proposals for languages for HPC, such as X10 [11], Chapel [5], and Titanium [16] are increasingly adopting ZPL-style regions [7, 9, 8] to achieve better performance and programmer productivity for array computations. Similarly, there is a trend towards exposing the memory hierarchy of modern systems to the programmer. Languages like X10 and Fortress [2] allow or even force the programmer to specify data distributions. Even HPC languages that present the programmer with a flat, uniform memory model must deal with data distribution and communication issues in their implementation. For these languages, research into algorithms that try to determine good data distributions to eliminate or reduce communication is important [12]. While the distribution aspect of our core language is clearly not useful for source-level type-checking of these languages, the core language does represent a potential type-system for a typed intermediate representation to be used by compilers for those languages. Given the challenges inherent in automatic data distribution [15, 19, 17, 14], another possible use would be the use of automated algorithms to suggest data distributions which could then be manually tuned for improved performance. Regardless of the level at which the language system captures information about arrays and data distributions, a type-system modeled around the core concepts presented in this paper can be used to give static correctness guarantees.

Table 2 gives an overview of the terminology used by different HPC language designs. Often these developing languages use different terms for the same or very similar abstractions. In the rest of this paper, we will use the terminology used by X10. Table 1 compares the features of various HPC languages with respect to arrays and memory. We say that the model of distributed memory is uniform if from the programmers point of view there is only one kind of memory. The model is flat if different partitions of the memory exist, but they have no structured relationship between them. The model is hierarchical if the partitions represent a memory hierarchy with multiple levels. Access is uniform if the the programmer does not need to distinguish between accesses to different partitions. With global/local access, the programmer is able to access remote memory using specific constructs. In X10, programmers cannot directly access mutable remote memory – the computation must move. In other words, in X10, non-value access must be local

with respect to the computation. Note that some of the listed HPC languages are still experimental. We base our statements on the limited documentation available at this time, which often reflects the state of early prototypes. When the language description does not seem to use a term or if it does not seem to specify the behavior, the respective entry in the table lists *unknown*. If the concept does not apply to the language, the table lists *n/a*.

3. Example Programs

We will present our core language and type system via six example programs which we show in Figure 1. The first five example programs all type check, while the sixth program (`shift`) does not. We use functions of the form $\lambda^\bullet$x.e which run at the place where they are defined. Our core language also has functions $\lambda x.e$ which run at the place where they are called. Similarly, we use dependent expressions lam$^\bullet \alpha.e$ for which the body will be evaluated at the place the dependent expression was defined. Additionally, our core language has dependent expressions `lam` $x.e$ which are evaluated at the place they are called.

The function `init` initializes all points in an array to 1. The function `init` takes two arguments, namely a region α and an array over region α. The use of the dependent type α makes `init` polymorphic: `init` can initialize any array without the need for any bounds checking. The expression `a.reg` has type `reg` α, and `p` which ranges over `a.reg` has type `pt` (σ, α), where α is the important part and σ can be ignored here. At the time of the assignment to `a[p]`, we have that the type of `p` matches the type of the region of `a`. The loop body uses `at(a.reg[@p]) { a[p]-1 }` to do the computation of `a[p]-1` at the place of `a[p]`, a common idiom in the benchmarks we have studied.

The function `partialinit` allows partial initialization of an array. It takes two extra arguments, namely a place value `h` and a corresponding place type variable γ which represents the same place as a type. The body of `partialinit` initializes those points in the argument array which can be found at the place `h`. In our core language, every region comes with a predefined mapping, called *distribution*, of points to places. The expression `a.reg` $\%_s$ `h` denotes those points in `a.reg` which by the 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`]. 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 α $\%_t$ γ, which illustrates that we use a type operator to mirror the expression operator. The variable `p` then gets the type `pt` $(\beta, \alpha \ \%_t \ \gamma)$, where β is a fresh variable that denotes the type-level identity of p. When we type check the access `a[p]`, the region check determines that the region of `p`, namely α, is a subset of the region of `a`, which is also α. For `a[p]`, the place check determines that the current place of execution is the same as the place of `a[p]`. The 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 $\alpha[@_t(\beta, \alpha \ \%_t \ \gamma)]$, which says that the place is that of a point in α which has its data locate at place γ. So, we can use the type equivalence $\alpha[@_t(\beta, \alpha \ \%_t \ \gamma)] \equiv \gamma$ to conclude that the place of execution is indeed the same as the place of the data `a[p]`.

The function `copy` takes two arrays `a` and `b`, both with region α. 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. 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.

The function `expand` takes an array `a` and region `x`, where `x` must be a superset of the region of `a`, and creates and returns a new array `b` over the region `x`. The function `expand` partially initializes the new array `b` with values from `a` at overlapping points. `expand` is interesting in that it highlights the importance of keeping upper and lower bounds for the region of arrays during type checking. Notice that argument β comes with the constraint $\alpha \subseteq_t \beta$, which means that the region α must be a subset of the region β. The call `partialinit <P>(P)<reg 0:9>(new int[0:9])<reg 1:8>(reg 1:8)` is a good example of the kind of reasoning that the programmer has to do when programming directly in the core language; the call satisfies the constraint $\beta \subseteq_t \alpha$ because $[0:8] \subseteq [0:9]$.

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

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

4. The Core Language

We now present the syntax, semantics, and type system of our core language. In the Appendix, we prove type soundness using the standard technique of Nielson [23] and others that was popularized by Wright and Felleisen [27].

Syntax. Figure 2 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 α 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 $\{c_1, \ldots, c_n\} + c = \{c_1 + c, \ldots, c_n + c\}$.

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 managable. 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 ρ is a mapping from region variables to regions, point variables to points, and place variables to places, then the meaning of the

Language	X10	Fortress	Chapel	Titanium	Co-Array Fortran	ZPL
Distributed memory	flat	hierarchical	flat	flat	flat	uniform
Access model	non-value access local	uniform	uniform	global/local	global/local	uniform
Array shapes	arbitrary	rectangular	arbitrary	rectangular	rectangular	arbitrary
Array size	finite	finite	finite or inf.	finite	finite	finite
Region algebra	rich	none	small	rich	none	rich
Distribution algebra	rich	small	build-ins	none	SPMD only	n/a
Memory safety	yes	unknown	yes	optional	no	unknown

Table 1. Features of HPC languages.

Language	X10	Fortress	Chapel	Titanium	Co-Array Fortran	ZPL
Locality	Place	Region	locale	demesnes	image	n/a
set of array indices	region	n/a	domain	domain	n/a	region
array index	point	index	index	point	index	index/direction
immutable data	value	value	unknown	immutable	n/a	n/a
dimensionality	rank	unknown	unknown	arity	rank, co-rank	rank

Table 2. HPC language terminology overview.

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 by *distribute* are mapped to π. The expression $r[@_t(\sigma, r)]$ evaluates to the place of the point σ according the distribution given by *distribute*.

The type of a point is a pair (σ, r) where σ 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 $\texttt{lam}\ \alpha : k.e$ 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 $e\texttt{.reg}$ returns the region of an array. The expression $e +\!+_s c$ adds a constant c to the point to which e evaluates. The expression $e +_s c$ adds a constant to each of the points in the region to which e evaluates.

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

In the example programs earlier in the paper, we used the syntactic sugar $\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ \{\ e'\ \}$ in order to represent $(\lambda^\bullet x.e')e$. We also used a few other constructs such as `p+1` which are not part of the core language but which could be added easily. We will use `true` to denote the tautology $\emptyset \subseteq_t \emptyset$.

Semantics. We specify the semantics of the core language using small-step operational semantics (see page 6). We use H to range over heaps:

$$H \in \texttt{Label} \to \texttt{Point} \to (\texttt{Value} \times \texttt{Place})$$

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

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 $\text{order}(\{c_1, \ldots, c_n\}) = \langle c_1, \ldots, c_n \rangle$, where $c_1 < \ldots < c_n$.

The following rules define a call-by-value semantics and are mostly standard. 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 $P = H(l)(p).2$. The condition $p \in \mathcal{D}(H(l))$ is the array-bounds check; p must be in the region of the array. The condition $P = H(l)(p).2$ is the

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


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


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


let expand = lam•α:region.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<reg 3:7>(new int[3:7])
            <reg 0:10>(int[0:10])
expand: Πα:region.int[α]
      → (Πβ:region(α ⊆t β).reg β → int[β])


let shiftleft = lam•α:region.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.int[α] → int


let shift = lam•α:region.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 ...
```

Figure 1. Example programs

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

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

Figure 2. Syntax of the core language.

to assign a type variable as first/only part of 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.

Rule (7) and Rule (8) express that the body of $\lambda^\bullet$ or 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.

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

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

The *satisfiability problem* is this: given a constraint φ, is φ satisfiable? The *entailment problem* is as follows: given two constraints φ, φ', 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 α and adding an extra conjunct $\alpha = e$. In the flattened constraint, let n be the number of variables in the constraint, let u be the largest integer mentioned in any region constant in the constraint, and let k be the largest c used in any $e +_s +_s$ or $e +\!+_s +\!+_s$ expression in the constraint. In any solution, an upper bound on the largest integer is $n \times u \times k$. To 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 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

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e_1')}{P \vdash (H, e_1\ e_2) \rightsquigarrow (H', e_1'\ e_2)} \tag{1}$$

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

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

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e_1')}{P \vdash (H, e_1\texttt{<}e_2\texttt{>}) \rightsquigarrow (H', e_1'\texttt{<}e_2\texttt{>})} \tag{4}$$

$$\frac{P \vdash (H, e_2) \rightsquigarrow (H', e_2')}{P \vdash (H, v\texttt{<}e_2\texttt{>}) \rightsquigarrow (H', v\texttt{<}e_2'\texttt{>})} \tag{5}$$

$$P \vdash (H, (\texttt{lam}\ \alpha : k.e)\texttt{<}w\texttt{>}) \rightsquigarrow (H, e[\alpha := w]) \tag{6}$$

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

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

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

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

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

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

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

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

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

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

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e_1')}{P \vdash (H, e_1 \cup_s e_2) \rightsquigarrow (H', e_1' \cup_s e_2)} \tag{17}$$

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

$$P \vdash (H, R_1 \cup_s R_2) \rightsquigarrow (H, R_1 \cup R_2) \tag{19}$$

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e_1')}{P \vdash (H, e_1 \cap_s e_2) \rightsquigarrow (H', e_1' \cap_s e_2)} \tag{20}$$

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

$$P \vdash (H, R_1 \cap_s R_2) \rightsquigarrow (H, R_1 \cap R_2) \tag{22}$$

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

$$P \vdash (H, d +_s c) \rightsquigarrow (H, d + c) \tag{24}$$

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

$$P \vdash (H, p +\!\!+_s c) \rightsquigarrow (H, p + c) \tag{26}$$

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

$$\frac{P \vdash (H, e_1) \rightsquigarrow (H', e_1')}{P \vdash (H, \texttt{for}\ (x\ \texttt{in}\ e_1)\texttt{\{}e_2\texttt{\}}) \rightsquigarrow (H', \texttt{for}\ (x\ \texttt{in}\ e_1')\texttt{\{}e_2\texttt{\}})} \tag{28}$$

$$\begin{array}{c} P \vdash (H, \texttt{for}\ (x\ \texttt{in}\ R)\texttt{\{}e\texttt{\}}) \rightsquigarrow (H, ((\text{lam}^\bullet \alpha : \texttt{point}(\alpha \in_t R).\lambda^\bullet x : (\alpha, R).e)\texttt{<}c_1\texttt{>})c_1; \ldots; \\ ((\text{lam}^\bullet \alpha : \texttt{point}(\alpha \in_t R).\lambda^\bullet x : (\alpha, R).e)\texttt{<}c_n\texttt{>})c_n; 0) \qquad \text{where order}(R) = \langle c_1, \ldots, c_n \rangle \end{array} \tag{29}$$

$$\begin{array}{c} P \vdash (H, \texttt{forallplaces}\ x\texttt{\{}e\texttt{\}}) \rightsquigarrow (H, ((\text{lam}^\bullet \alpha : \texttt{place}.\lambda^\bullet x : \texttt{pl}\ \alpha.e)\texttt{<}P_1\texttt{>})P_1; \ldots; \\ ((\text{lam}^\bullet \alpha : \texttt{place}.\lambda^\bullet x : \texttt{pl}\ \alpha.e)\texttt{<}P_n\texttt{>})P_n; 0) \qquad \text{where } \mathit{places} = \langle P_1, \ldots, P_n \rangle \end{array} \tag{30}$$

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

$$P \vdash (H, v; e) \rightsquigarrow (H, e) \tag{32}$$

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

$$P \vdash (H, \texttt{at(}P'\texttt{)\{}v\texttt{\}}) \rightsquigarrow (H, v) \tag{34}$$

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 $\varphi' = \texttt{false}$ we have that the entailment problem is the question of given φ, is φ unsatisfiable, which is co-NP-complete. So, the full entailment problem is co-NP-hard. Second, note that the entailment problem is in co-NP; we can easily collect the set of all points mentioned in the constraints, then guess an assignment, and finally check that the assignment is not a satisfying assignment, in polynomial time.

Heap Types. We use Ψ to range over maps from array labels to types of the form $t[R]$. 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) $\Psi; \varphi; \Gamma; \mathit{here} \vdash H(l)(p).1 : t$ and (ii) $\mathit{distribute}(R, p) = H(l)(p).2$. We write $\Psi \lhd \Psi'$ if $\mathcal{D}(\Psi) \subseteq \mathcal{D}(\Psi')$ and Ψ, Ψ' 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 3 The first three rules use a meta-variable q which ranges over t, r, σ, π.

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 $\Psi; \varphi; \Gamma; \mathit{here} \vdash e : t$, which holds if it is derivable using the following rules. The type *here* is the type of the current place of execution. 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 $\texttt{for}\ (x\ \texttt{in}\ e_1)\{e_2\}$, we check that e_1 has a type $\texttt{reg}\ r$, and then assign x the type $\texttt{pt}\ (\alpha, r)$ while checking e_2, where α 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:

$$\mathit{constraint}(\texttt{point}\ \varphi) = \varphi \tag{83}$$
$$\mathit{constraint}(\texttt{region}\ \varphi) = \varphi \tag{84}$$
$$\mathit{constraint}(\texttt{place}) = \texttt{true} \tag{85}$$

We use W to range over regions r and variables α of kind `place`.

Rules for kind checking:

$$\vdash \texttt{pt}\ (\sigma, r) : \texttt{point}\ \varphi \rhd \sigma \tag{86}$$
$$\vdash \texttt{reg}\ r : \texttt{region}\ \varphi \rhd r \tag{87}$$
$$\vdash \texttt{pl}\ \pi : \texttt{place} \rhd \pi. \tag{88}$$

5. Experimental Results

We have implemented our type system in XTC-X10, a prototype implementation of an X10 variant. The prototype extends X10 v0.4 with parametric types (generics), first-order functions and an extended form of the dependent type system presented in this paper. The implementation is publically available at `http://grothoff.org/christian/xtc/x10/`. Our implementation can type check and execute the benchmark programs

$$\varphi \vdash q \equiv q \tag{35}$$
$$\frac{\varphi \vdash q_1 \equiv q_2}{\varphi \vdash q_2 \equiv q_1} \tag{36}$$
$$\frac{\varphi \vdash q_1 \equiv q_2 \qquad \varphi \vdash q_2 \equiv q_3}{\varphi \vdash q_1 \equiv q_3} \tag{37}$$
$$\frac{\varphi \vdash \sigma \equiv \sigma' \qquad \varphi \vdash r \equiv r'}{\varphi \vdash \texttt{pt}\ (\sigma, r) \equiv \texttt{pt}\ (\sigma', r')} \tag{38}$$
$$\frac{\varphi \vdash r \equiv r'}{\varphi \vdash \texttt{reg}\ r \equiv \texttt{reg}\ r'} \tag{39}$$
$$\frac{\varphi \vdash \pi \equiv \pi'}{\varphi \vdash \texttt{pl}\ \pi \equiv \texttt{pl}\ \pi'} \tag{40}$$
$$\frac{\varphi \vdash t_1 \equiv t_1' \qquad \varphi \vdash t_2 \equiv t_2'}{\varphi \vdash t_1 \to t_2 \equiv t_1' \to t_2} \tag{41}$$
$$\frac{\varphi \vdash t \equiv t'}{\varphi \vdash \Pi\alpha : k.t \equiv \Pi\alpha : k.t'} \tag{42}$$
$$\varphi \vdash R_1 \cup_t R_2 \equiv R_1 \cup R_2 \tag{43}$$
$$\frac{\varphi \vdash r_1 \equiv r_1' \qquad \varphi \vdash r_2 \equiv r_2'}{\varphi \vdash r_1 \cup_t r_2 \equiv r_1' \cup_t r_2'} \tag{44}$$
$$\varphi \vdash R_1 \cap_t R_2 \equiv R_1 \cap R_2 \tag{45}$$
$$\frac{\varphi \vdash r_1 \equiv r_1' \qquad \varphi \vdash r_2 \equiv r_2'}{\varphi \vdash r_1 \cap_t r_2 \equiv r_1' \cap_t r_2'} \tag{46}$$
$$\varphi \vdash R +_t c \equiv R + c \tag{47}$$
$$\frac{\varphi \vdash r \equiv r'}{\varphi \vdash r +_t c \equiv r' +_t c} \tag{48}$$
$$\varphi \vdash R \,\%_t\, P \equiv \{\, p \in R \mid \mathit{distribute}(R, p) = P \,\} \tag{49}$$
$$\frac{\varphi \vdash r \equiv r' \qquad \varphi \vdash \pi \equiv \pi'}{\varphi \vdash r \,\%_t\, \pi \equiv r' \,\%_t\, \pi'} \tag{50}$$
$$\varphi \vdash p +\!+_t c \equiv p + c \tag{51}$$
$$\frac{\varphi \vdash \sigma \equiv \sigma'}{\varphi \vdash \sigma +\!+_t c \equiv \sigma' +\!+_t c} \tag{52}$$
$$\frac{\varphi \models p \in_t r \qquad \varphi \models r \subseteq_t R}{\varphi \vdash R[@_t(p, r)] \equiv \mathit{distribute}(R, p)} \tag{53}$$
$$\frac{\varphi \vdash r_1 \equiv r_1' \qquad \varphi \vdash \sigma \equiv \sigma' \qquad \varphi \vdash r_2 \equiv r_2'}{\varphi \vdash r_1[@_t(\sigma, r_2)] \equiv r_1'[@_t(\sigma', r_2')]} \tag{54}$$
$$\frac{\varphi \models \sigma \in_t r \,\%_t\, \pi}{\varphi \vdash r[@_t(\sigma, r \,\%_t\, \pi)] = \pi} \tag{55}$$

Figure 3. Type equivalence rules.

listed below along with the five type-safe example programs from Section 3. We adapted the benchmarks from code written in X10, mainly by making the code use regions. We collected our measurements by instrumenting the implementation of our X10 variant.

5.1 The ArrayBench Benchmark Suite

The ArrayBench Benchmark Suite consists of seven benchmark programs. This section briefly explains the functionality of each benchmark, the style of parallelism (if any) and the overall amount of communication.

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

$$\Psi;\varphi;\Gamma;\mathit{here} \vdash c :\ \texttt{int} \tag{56}$$

$$\Psi;\varphi;\Gamma;\mathit{here} \vdash p :\ \texttt{pt}\ (p,R) \quad (\text{where } p \in R) \tag{57}$$

$$\Psi;\varphi;\Gamma;\mathit{here} \vdash R :\ \texttt{reg}\ R \tag{58}$$

$$\Psi;\varphi;\Gamma;\mathit{here} \vdash l :\ \Psi(l) \tag{59}$$

$$\Psi;\varphi;\Gamma;\mathit{here} \vdash P :\ \texttt{pl}\ P \tag{60}$$

$$\frac{\Psi;\varphi;\Gamma[x:t_1];\mathit{unknown} \vdash e :\ t_2}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \lambda x:t_1.e :\ t_1 \to t_2} \tag{61}$$

$$\frac{\Psi;\varphi \wedge \mathit{constraint}(k);\Gamma;\mathit{unknown} \vdash e :\ t}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \texttt{lam}\ \alpha:k.e :\ \Pi\alpha:k.t} \tag{62}$$

$$\Psi;\varphi;\Gamma;\mathit{here} \vdash x :\ \Gamma(x) \tag{63}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 :\ t_1 \to t_2 \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash e_2 :\ t_1}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1\ e_2 :\ t_2} \tag{64}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 :\ \Pi\alpha:k.t_1 \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash e_2 : t_2 \qquad \vdash t_2 :\ k \rhd W \qquad \varphi \models (\mathit{constraint}(k))[\alpha := W]}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1\texttt{<}e_2\texttt{>} :\ t_1[\alpha := W]} \tag{65}$$

$$\frac{\Psi;\varphi;\Gamma[x:t_1];\mathit{here} \vdash e :\ t_2 \qquad \mathit{here} \neq \mathit{unknown}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \lambda^{\bullet} x:t_1.e :\ t_1 \to t_2} \tag{66}$$

$$\frac{\Psi;\varphi \wedge \mathit{constraint}(k);\Gamma;\mathit{here} \vdash e :\ t \qquad \mathit{here} \neq \mathit{unknown}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \text{lam}^{\bullet}\alpha:k.e :\ \Pi\alpha:k.t} \tag{67}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e :\ \texttt{reg}\ r}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \texttt{new}\ t[e] :\ t[r]} \tag{68}$$

$$\frac{\begin{array}{c}\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1 :\ t[r_1] \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash y_2 :\ \texttt{pt}\ (\sigma,r_2) \\ \varphi \models r_2 \subseteq_t r_1 \qquad \varphi \models \sigma \in_t r_2 \qquad \varphi \vdash \mathit{here} \equiv r_1[@_t(\sigma,r_2)]\end{array}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1[y_2] :\ t} \tag{69}$$

$$\frac{\begin{array}{c}\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1 :\ t[r_1] \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash y_2 :\ \texttt{pt}\ (\sigma,r_2) \qquad \varphi \models r_2 \subseteq_t r_1 \\ \varphi \models \sigma \in_t r_2 \qquad \varphi \vdash \mathit{here} \equiv r_1[@_t(\sigma,r_2)] \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash e :\ t\end{array}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1[y_2] = e :\ t} \tag{70}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e :\ t[r]}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e.\texttt{reg} :\ \texttt{reg}\ r} \tag{71}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1 :\ t[r_1] \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash y_2 :\ \texttt{pt}\ (\sigma,r_2) \qquad \varphi \models r_2 \subseteq_t r_1 \qquad \varphi \models \sigma \in_t r_2}{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1[@_s y_2]\ :\ \texttt{pl}\ r_1[@_t(\sigma,r_2)]} \tag{72}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 :\ \texttt{reg}\ r_1 \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash e_2 :\ \texttt{reg}\ r_2}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 \cup_s e_2 :\ \texttt{reg}\ r_1 \cup_t r_2} \tag{73}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 :\ \texttt{reg}\ r_1 \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash e_2 :\ \texttt{reg}\ r_2}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 \cap_s e_2 :\ \texttt{reg}\ r_1 \cap_t r_2} \tag{74}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e :\ \texttt{reg}\ r}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e +_s c :\ \texttt{reg}\ r +_t c} \tag{75}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e :\ \texttt{pt}\ (\sigma,r)}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e +\!\!+_s c :\ \texttt{pt}\ (\sigma +\!\!+_t c, r +_t c)} \tag{76}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1 :\ \texttt{reg}\ r \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash y_2 :\ \texttt{pl}\ \pi}{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1\ \%_s\ y_2 :\ \texttt{reg}\ r\ \%_t\ \pi} \tag{77}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 :\ \texttt{reg}\ r \qquad \Psi;\varphi \wedge (\alpha \in_t r);\Gamma[x :\ \texttt{pt}\ (\alpha,r)];\mathit{here} \vdash e_2 :\ t \qquad \mathit{here} \neq \mathit{unknown}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \texttt{for}\ (x\ \texttt{in}\ e_1)\{e_2\} :\ \texttt{int}} \quad (\text{where } \alpha \text{ is fresh}) \tag{78}$$

$$\frac{\Psi;\varphi;\Gamma[x:\texttt{pl}\ \alpha];\mathit{here} \vdash e :\ t \qquad \mathit{here} \neq \mathit{unknown}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \texttt{forallplaces}\ x\{e\} :\ \texttt{int}} \quad (\text{where } \alpha \text{ is fresh}) \tag{79}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 :\ t_1 \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash e_2 :\ t_2}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1; e_2 :\ t_2} \tag{80}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash y :\ \texttt{pl}\ \pi \qquad \Psi;\varphi;\Gamma;\pi \vdash e :\ t}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \texttt{at}(y)\{e\} :\ t} \tag{81}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e :\ t \qquad \varphi \vdash t \equiv t'}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e :\ t'} \tag{82}$$

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

Table 3. 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.

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 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).

Table 3 gives some fundamental benchmark statistics. 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 dependent 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 dependent casts in source code.

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

Crypt: Implements the IDEA symmetric blockcipher (encrypt and decrypt) using integer increment operations to iterate over a stream. Uses 9 dependent casts in source code.

Crypt: Implements the IDEA symmetric blockcipher (encrypt and decrypt) using region iterators to iterate over a stream. Uses 3 dependent casts in source code.

SOR: Given a 2D array, performs successive over-relaxation [24] of an $n \times n$ matrix. Uses two dependent casts in source code.

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

5.2 Region and Place Casts in the Benchmarks

The region casts and place casts in the benchmarks roughly fall into three categories:

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 necesary 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.

5.3 Example

For example, function `overlap` computes the partial match table (or failure function) of the Knuth-Morris-Pratt string searching algorithm [18]. The syntax is 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. 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`.

```
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
  }
}
```

name	entailment checks		dynamic dependent casts			
	total number	max. size	without types S	without types L	with types S	with types 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 4. Numbers of dynamic checks required for the benchmarks.

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.

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 $m+1$. Cast `#3` is simply repeating the same cast as cast `#2` and could thus be considered falling into both categories 1 and 3.

5.4 Measurements and Assessment

Table 4 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. For ArrayBench, there is on average one such cast in 50 lines of code. Because these 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.

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 individual 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 interfer 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 allows 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.

6. Future Work and Conclusion

Out-of-bounds array accesses remain a leading cause of security problems and, according to the National Vulnerability Database [21], buffer overflows are responsible for 233 out of 863 CERT technical alerts or vulnerability notes in the years 2004 and 2005. Our type system can guarantee that no out-of-bounds array accesses will happen, thereby also obviating the need for doing dynamic checks of such accesses.

Acknowledgments

We thank 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, Shannon Hoffswell, and John Plevyak. Chapel specification (version 0.4). Technical report, Cray Inc, http://chapel.cs.washington.edu/, 2005.

[6] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, November 2001.

[7] 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.

[8] 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.

[9] 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.

[10] Bradford L. Chamberlain and Lawrence Snyder. Array language support for parallel sparse computation. In *Proceedings of the ACM International Conference on Supercomputing*, 2001.

[11] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Vijay A. Saraswat, Vivek Sarkar, and Christoph Von Praun. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 519–538. ACM SIGPLAN, 2005.

[12] 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.

[13] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. High-level language support for user-defined reductions. *Journal of Supercomputing*, 23(1), 2002.

[14] 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.

[15] 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.

[16] 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.

[17] Ken Kennedy and Ulrich Kremer. Automatic data layout for

distributed-memory machines. *ACM Trans. Program. Lang. Syst.*, 20(4):869–916, 1998.

[18] Donald Knuth, Jr James H. Morris, and Vaughan Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[19] 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.

[20] Ben Liblit and Alexander Aiken. Type systems for distributed data structures. In *Proceedings of POPL'00, 27nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 199–213, 2000.

[21] Peter Mell, Karen Kent, Ashish Goel, Ellery Horton, Tanyette Miller, Robert Chang, Michael Reilly, and Kathy Ton-Nu. National vulnerability database. http://nvd.nist.gov/, 2006.

[22] George Necula. Proof-carrying code. In *Proceedings of POPL'97, 24th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[23] Flemming Nielson. The typed lambda-calculus with first-class processes. In *Proceedings of PARLE'89*, pages 357–373, 1989.

[24] 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.

[25] 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.

[26] 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.

[27] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[28] 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.

[29] 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 Soundness

LEMMA 1. **(Substitution)**
If $\Psi;\varphi;\Gamma[x:t_1];here \vdash e:\ t_2$ *and* $\Psi;\varphi;\Gamma;here \vdash v:\ t_1$*, then* $\Psi;\varphi;\Gamma;here \vdash e[x:=v]:t_2$.

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

LEMMA 2. **(Dependent Substitution)**
If $\Psi;\varphi;\Gamma;here \vdash e:\ t$*, then* $\Psi;\varphi[\alpha:=W];\Gamma;here[\alpha:=W] \vdash e[\alpha:=W]:\ t[\alpha:=W]$.

Proof. By induction on the structure of the derivation of $\Psi;\varphi;\Gamma;here \vdash e:\ t$. □

LEMMA 3. **(Weakening)**
If $\Psi;\varphi;\Gamma;here \vdash e:\ t$ *and* $\varphi' \models \varphi$*, then* $\Psi;\varphi';\Gamma;here \vdash e:\ t$.

Proof. By induction on the structure of the derivation of $\Psi;\varphi;\Gamma;here \vdash e:\ t$. □

LEMMA 4. **(Indifference)**
If $\Psi;\varphi;\Gamma;here \vdash v:\ t$*, then* $\Psi;\varphi;\Gamma;here' \vdash v:\ t$.

Proof. Immediate from the seven type rules for values. □

LEMMA 5. **(Canonical Forms)**

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

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. □

THEOREM 1. **(Type Preservation)**
For a place P*, let* $Q \in \{P, unknown\}$*. If* $\Psi;\varphi;\Gamma;Q \vdash e:\ t$*,* $\models H:\Psi$*, and* $P \vdash (H,e) \rightsquigarrow (H',e')$*, then we have* Ψ', t' *such that* $\Psi \lhd \Psi'$*,* $\Psi';\varphi;\Gamma;Q \vdash e':\ t'$*,* $\models H':\Psi'$*, 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 a 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:

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

  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 Ψ' such that $\Psi \lhd \Psi'$, $\Psi';\varphi;\Gamma;Q \vdash e_1':\ t_1 \to t_2$, and $\models H':\Psi'$. From $\Psi \lhd \Psi'$ and $\Psi;\varphi;\Gamma;Q \vdash e_2:\ t_1$ we have $\Psi';\varphi;\Gamma;Q \vdash e_2:\ t_1$. From $\Psi';\varphi;\Gamma;Q \vdash e_1':\ t_1 \to t_2$

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 Ψ' such that $\Psi \lhd \Psi'$, $\Psi'; \varphi; \Gamma; Q \vdash e_2' : t_1$, and $\models H' : \Psi'$. From $\Psi \lhd \Psi'$ and $\Psi; \varphi; \Gamma; Q \vdash e_1 : t_1 \to t_2$ we have $\Psi'; \varphi; \Gamma; Q \vdash e_1 : t_1 \to t_2$. From $\Psi'; \varphi; \Gamma; Q \vdash e_1 : t_1 \to t_2$ 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:

$$\frac{\begin{array}{c}\Psi; \varphi; \Gamma; Q \vdash e_1 : \Pi\alpha : k.t_1 \qquad \Psi; \varphi; \Gamma; Q \vdash e_2 : t_2 \\ \vdash t_2 : k \rhd W \qquad \varphi \models (\mathit{constraint}(k))[\alpha := W]\end{array}}{\Psi; \varphi; \Gamma; Q \vdash e_1\texttt{<}e_2\texttt{>} : t_1[\alpha := W]}$$

We now have three subcases depending on which rule was used to make $(H, e_1\texttt{<}e_2\texttt{>})$ 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,

$$P \vdash (H, (\texttt{lam}\; \alpha : k.e)\texttt{<}w\texttt{>}) \rightsquigarrow (H, e[\alpha := w])$$

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 α such that α does not occur free in φ. 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:

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

If Rule (7), that is,

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

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 $\Psi; \varphi; \Gamma[x : t_1]; \mathit{unknown} \vdash P : \texttt{pl}\; P$. From $\Psi; \varphi; \Gamma[x : t_1]; \mathit{unknown} \vdash P : \texttt{pl}\; P$ and $\Psi; \varphi; \Gamma[x : t_1]; P \vdash e : t_2$ and Rule (81), we have $\Psi; \varphi; \Gamma[x : t_1]; \mathit{unknown} \vdash \texttt{at(}P\texttt{)\{}e\texttt{\}} : t_2$. From $\Psi; \varphi; \Gamma[x : t_1]; \mathit{unknown} \vdash \texttt{at(}P\texttt{)\{}e\texttt{\}} : t_2$ and Rule (61) we have $\Psi; \varphi; \Gamma; Q \vdash \lambda x : t_1.\texttt{at(}P\texttt{)\{}e\texttt{\}} : t_1 \to t_2$.

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

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

If Rule (8), that is,

$$P \vdash (H, \text{lam}^\bullet \alpha : k.e) \rightsquigarrow (H, \texttt{lam}\; \alpha : k.\texttt{at(}P\texttt{)\{}e\texttt{\}})$$

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

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

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

We now have two subcases depending on which rule was used to make $(H, \texttt{new}\; t[e])$ 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,

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

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

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

$$\frac{\begin{array}{ll}\Psi; \varphi; \Gamma; Q \vdash y_1 : t[r_1] & \Psi; \varphi; \Gamma; Q \vdash y_2 : \texttt{pt}\; (\sigma, r_2) \\ \varphi \models r_2 \subseteq_t r_1 & \varphi \models \sigma \in_t r_2 \qquad \varphi \vdash Q \equiv r_1[@_t(\sigma, r_2)]\end{array}}{\Psi; \varphi; \Gamma; Q \vdash y_1[y_2] : t}$$

If Rule (11), that is,

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

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:

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

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,

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

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:

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

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

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

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 $\Psi; \varphi; \Gamma; Q \vdash \mathcal{D}(H(l)) : \mathtt{reg}\ R$.

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

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

If Rule (16), that is,

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

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 $\Psi; \varphi; \Gamma; Q \vdash p : \mathtt{pt}\ (\sigma, r_2)$ and Rule (57) we have that $\sigma = p$. We have $H' = H$ and we pick $\Psi' = \Psi$. From $\models H : \Psi$ we have $H(l)(p).2 = \mathit{distribute}(R, p)$ and $\mathcal{D}(H(l)) = R$. From Rule (60) we have that we must show $H(l)(p).2 \equiv r_1[@_t(\sigma, r_2)]$. We have $r_1[@_t(\sigma, r_2)] = R[@_t(p, r_2)]$. We have $H(l)(p).2 = \mathit{distribute}(R, p)$. We also have $\varphi \models r_2 \subseteq_t R$ and $\varphi \vdash p \in_t r_2$ so from Rule (53) we have $\varphi \vdash R[@_t(p, r_2)] \equiv \mathit{distribute}(R, p)$. We conclude $H(l)(p).2 = \mathit{distribute}(R, p) \equiv R[@_t(p, r_2)] = r_1[@_t(\sigma, r_2)]$, as desired.

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

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

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:

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

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:

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

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:

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

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:

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

If Rule (27), that is,

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

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:

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

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

$$P \vdash (H, \mathtt{for}\ (x\ \mathtt{in}\ R)\{e_2\}) \rightsquigarrow (H, ((\mathrm{lam}^\bullet \alpha : \mathtt{point}(\alpha \in_t R).\lambda^\bullet x : (\alpha, R).e_2)\texttt{<}c_1\texttt{>})c_1; \ldots; ((\mathrm{lam}^\bullet \alpha : \mathtt{point}(\alpha \in_t R).\lambda^\bullet x : (\alpha, R).e_2)\texttt{<}c_n\texttt{>})c_n; 0) \quad \text{where order}(R) = \langle c_1, \ldots, c_n \rangle$$

was used to take a step, then we have $r = R$. From $\Psi; \varphi; \Gamma[x : \mathtt{pt}\ (\alpha, R)]; Q \vdash e_2 : t$ and $Q \neq \mathit{unknown}$ and Rule (66) we have $\Psi; \varphi; \Gamma; Q \vdash \lambda^\bullet x : (\alpha, R).e_2 : \mathtt{pt}\ (\alpha, R) \to t$. From $\Psi; \varphi; \Gamma; Q \vdash \lambda^\bullet x : (\alpha, R).e_2 : \mathtt{pt}\ (\alpha, R) \to t$ and Rule (67) we have $\Psi; \varphi; \Gamma; Q \vdash \mathrm{lam}^\bullet \alpha.\lambda^\bullet x : (\alpha, R).e_2 : \Pi\alpha : \mathtt{point}(\alpha \in_t R).\mathtt{pt}\ (\alpha, R) \to t$. From Rule (57) and the definition of order(R) we have $\Psi; \varphi; \Gamma; Q \vdash c_i : \mathtt{pt}\ (c_i, R)$. From $\Psi; \varphi; \Gamma; Q \vdash c_i : \mathtt{pt}\ (c_i, R)$ and $\Psi; \varphi; \Gamma; Q \vdash \mathrm{lam}^\bullet \alpha.\lambda^\bullet x :$

$(\alpha, R).e_2 : \Pi\alpha : \texttt{point}(\alpha \in_t R).\texttt{pt}\ (\alpha, R) \to t$ and $\vdash (c_i, R) : \texttt{point}(\alpha \in_t R) \triangleright c_i$ and $\varphi \models \mathit{constraint}(\texttt{point})[\alpha := c_i]$ and Rule (65) we have $\Psi; \varphi; \Gamma; Q \vdash (\text{lam}^\bullet \alpha.\lambda^\bullet x : (\alpha, R).e_2)\texttt{<}c_i\texttt{>} : \texttt{pt}\ (c_i, R) \to t[\alpha := c_i]$. From $\Psi; \varphi; \Gamma; Q \vdash (\text{lam}^\bullet \alpha.\lambda^\bullet x : (\alpha, R).e_2)\texttt{<}c_i\texttt{>} : \texttt{pt}\ (c_i, R) \to t[\alpha := c_i]$ and $\Psi; \varphi; \Gamma; Q \vdash c_i : \texttt{pt}\ (c_i, R)$ and Rule (64) we have $\Psi; \varphi; \Gamma; Q \vdash ((\text{lam}^\bullet \alpha.\lambda^\bullet x : (\alpha, R).e_2)\texttt{<}c_i\texttt{>})c_i : t[\alpha := c_i]$. From Rule (80) and Rule (56) we conclude

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

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

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

If Rule (30), that is,

$$P \vdash (H, \texttt{forallplaces}\ x\{e\}) \rightsquigarrow (H, ((\text{lam}^\bullet \alpha : \texttt{place}.\lambda^\bullet x : \texttt{pl}\ \alpha.e)\texttt{<}P_1\texttt{>})P_1; \ldots; ((\text{lam}^\bullet \alpha : \texttt{place}.\lambda^\bullet x : \texttt{pl}\ \alpha.e)\texttt{<}P_n\texttt{>})P_n; 0) \text{ where } \mathit{places} = \langle P_1, \ldots, P_n \rangle$$

was used to take a step, then we can prove that $\Psi; \varphi; \Gamma; \mathit{here} \vdash ((\text{lam}^\bullet \alpha : \texttt{place}.\lambda^\bullet x : \texttt{pl}\ \alpha.e)\texttt{<}P_1\texttt{>})P_1; \ldots; ((\text{lam}^\bullet \alpha : \texttt{place}.\lambda^\bullet x : \texttt{pl}\ \alpha.e)\texttt{<}P_n\texttt{>})P_n; 0) : \texttt{int}$ 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:

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

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:

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

We now have two subcases depending on which rule was used to make $(H, \texttt{at}(x)\{e\})$ take a step.
If Rule (33), that is,

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

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 $\Psi; \varphi; \Gamma; Q \vdash \texttt{at}(y)\{e'\} : t$.
If Rule (34), that is,

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

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 $\Psi; \varphi; \Gamma; \mathit{here} \vdash v : t$.

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

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

From the induction hypothesis we have Ψ', 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 $\Psi; \varphi; \Gamma; \mathit{here} \vdash e' : t'$.

□

THEOREM 2. **(Progress)**
For a place P, let $Q \in \{P, \mathit{unknown}\}$. If $\Psi; \mathit{true}; \emptyset; Q \vdash e : t$ 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 $\Psi; \text{true}; \emptyset; Q \vdash e : t$. There are now twenty-five subcases depending on which one of the type rules was the last one used in the derivation of $\Psi; \text{true}; \emptyset; Q \vdash e : t$.

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

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

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

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:

$$\frac{\Psi; \text{true}; \emptyset; Q \vdash e_1 : \Pi\alpha : k.t_1 \qquad \Psi; \text{true}; \emptyset; Q \vdash e_2 : t_2 \quad \vdash t_2 : k \triangleright W \qquad \text{true} \models (\mathit{constraint}(k))[\alpha := W]}{\Psi; \text{true}; \emptyset; Q \vdash e_1\texttt{<}e_2\texttt{>} : t_1[\alpha := W]}$$

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 $(H, e_1\texttt{<}e_2\texttt{>})$ 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 $(H, e_1\texttt{<}e_2\texttt{>})$ 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 $\texttt{lam}\ \alpha : k.e$, and we have from $\vdash t_2 : k \triangleright W$ and Lemma 5 that e_2 is of the form w, so $(H, e_1\texttt{<}e_2\texttt{>})$ can take a step using Rule (6).

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

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

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

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

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

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

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

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

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 $(H, \texttt{new}\ t[e])$ 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 $(H, \texttt{new}\ t[e])$ can take a step using Rule (10).

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

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

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 $\mathit{distribute}(R, p) = H(l)(p).2$. From Rule (57) we have $\sigma = p$. We conclude $P = r_1[@_t(\sigma, r_2)] = \mathit{distribute}(R, p) = H(l)(p).2$. So, $(H, e_1[e_2])$ can take a step using Rule (11).
- Rule (70): the derivation is of the form:

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

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:

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

From the induction hypothesis we have that (H, e) is not stuck. If (H, e) can take a step at place P, then $(H, e.\texttt{reg})$ can also take a step using Rule (14). If e is a value, then $(H, e.\texttt{reg})$ can take a step using Rule (15).
- Rule (72): the derivation is of the form:

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

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 $y_1[@_s y_2]$ 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:

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

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:

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

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:

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

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:

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

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:

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

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:

$$\frac{\begin{array}{l}\Psi;\text{true};\emptyset;Q\vdash e_1:\ \texttt{reg}\ r \qquad Q\neq \mathit{unknown}\\ \Psi;(\alpha\in_t r);\emptyset[x:\ \texttt{pt}\ (\alpha,r)];Q\vdash e_2:\ t\end{array}}{\Psi;\text{true};\emptyset;Q\vdash \texttt{for}\ (x\ \texttt{in}\ e_1)\{e_2\}:\ \texttt{int}}\ (\alpha\text{ fresh})$$

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 $(H, \texttt{for}\ (x\ \texttt{in}\ e_1)\{e_2\})$ 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 $(H, \texttt{for}\ (x\ \texttt{in}\ e_1)\{e_2\})$ can take a step at place P using Rule (29).
- Rule (79): the derivation is of the form:

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

We have that $\texttt{forallplaces}\ x\{e\}$ can take a step using Rule (30).
- Rule (80): the derivation is of the form:

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

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:

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

  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 $(H, \texttt{at}(y)\{e\})$ can take a step at place P using Rule (33). If e is a value, then $(H, \texttt{at}(y)\{e\})$ can take a step at place P using Rule (34).
- Rule (82): the derivation is of the form

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

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

□

COROLLARY 1. **(Type Soundness)**
For a place P, let $Q \in \{P, unknown\}$. If $\Psi; true; \emptyset; Q \vdash e : t$ 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 Ψ', 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. □