Type Inference for Place-Oblivious Objects

Riyaz Haque and Jens Palsberg

University of California – Los Angeles (UCLA), USA
{rfhaque,palsberg}@cs.ucla.edu

Abstract

In a distributed system, access to local data is much faster than access to remote data. As a help to programmers, some languages require every access to be local. A program in those languages can access remote data via first a shift of the place of computation and then a local access. To enforce this discipline, researchers have presented type systems that determine whether every access is local and every place shift is appropriate. However, those type systems fall short of handling a common programming pattern that we call place-oblivious objects. Such objects safely access other objects without knowledge of their place. In response, we present the first type system for place-oblivious objects along with an efficient inference algorithm and a proof that inference is P-complete. Our example language extends the Abadi-Cardelli object calculus with place shift and existential types, and our implementation has inferred types for some microbenchmarks.

1998 ACM Subject Classification D.3.1 Formal Definitions and Theory

Keywords and phrases parallelism, locality, types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.371

1 Introduction

Places. A distributed system consists of multiple *places* of computation. At each place, a computation may store references to both local and remote data. Access to local data is much faster than access to remote data because a remote access may go across a network. Distributed languages largely agree on the syntax of local access while they differ on the syntax of remote access. Some distributed languages, such as Titanium [26, 12], use a uniform access syntax that works for both local and remote access. Such syntax is succinct yet can make run-time performance unpredictable when the programmer is uncertain about the location of data. Other languages, such as X10 [23, 6], require a remote access to be expressed as a place shift followed by a local access at the new place. The use of place shift is verbose yet enables a programmer to easily spot slow, remote data accesses, and enables a compiler to optimize local accesses. In this paper we study a core calculus with explicit place shift.

Place checks. Languages with explicit place shift require every access to be local. This can be enforced with a run-time check known as a *place check*. The place check compares the current place with the place of the accessed data. If those two places are equal, then computation proceeds normally, and otherwise the result is a run-time error. For example, if a place check fails in X10, then the X10 implementation throws a run-time exception called BadPlaceException. Place checks can degrade the overall run-time performance [5] and they defer discovery of "place bugs" until such bugs happen at run time. However, place checking can also be done statically. For example, researchers have presented static analyses [2] and type systems [16, 5, 11, 4, 3, 15, 7, 24, 13] that determine whether every access is local. This has the potential to give programmers the best of both worlds: predictable performance and

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 371–395

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

no place-check errors. Additionally, when a static technique can guarantee that a place check succeeds, an implementation can eliminate the run-time place check and thereby improve performance. Intuitively, the difference between static analysis and type checking in this context lies in their ambition levels. A static analysis tries to eliminate as many run-time place checks as possible, while a type system tries to eliminate *all* run-time place checks. In this paper we focus on type systems.

The challenge. We have identified a common programming pattern that we call *place-oblivious objects*. Such objects safely access other objects without knowledge of their place. We found uses of place-oblivious objects in 11 of the 13 X10 benchmarks that were considered by Lee and Palsberg [14].

Let us consider the following example written in a variant of Featherweight X10 [14]:

```
class Example {
   public Unit f;
   ...
   public void m() {
      final Unit x = this.f;
      async(x.location) {
           final O1 e = x.g;
      }
   }
}
```

We assume that `Unit` is a class with a field `g` of type `O1`. Objects of class `Example` are place oblivious. The reason is that the field `f` may at one time reference an object at place 1 and at another time reference an object at place 2. Still, method `m` successfully accesses the field `g` of objects in `f`, as follows. First, the body uses `final Unit x = this.f` to create an immutable reference `x`. This avoids trouble with any concurrent access that may change the contents of `f`. Second, the expression does a place shift `at(x.location)`, and finally a local access `x.g`.

In this paper, we will use an extension of the Abadi-Cardelli object calculus [1] rather than X10. We chose the Abadi-Cardelli object calculus because it has a succint semantics and a small number of type rules, which makes it a good basis for study of algorithms and for detailed proofs. In our calculus, we can write an expression similar to the body of `m` in the following way:

$$open\ x = this.f\ in\ at(x.place)\{\ x.g\ \}$$

First, the expression uses *open*, which has the same semantics as *let* but which we will give a different type rule. The *open* expression creates an immutable reference $x = this.f$ (like `final` does in the X10 code above), then does a place shift $at(x.place)$, and finally a local access $x.g$.

The type system must *(i)* assign f a type that is compatible with the types of the objects at place 1 and place 2, and *(ii)* determine that after the place shift, $x.g$ is indeed a local access. We cannot simply give f a type that says that the place of f is unknown. Such a type would imply that the type system has no knowledge of the target of the place shift, hence no basis for knowing whether $x.g$ is a local access. Place-oblivious objects occur frequently in X10 code and yet previous work falls short of type inference for such objects, mainly because of a lack of a sensible type for f.

Our results. We present the first type system for place-oblivious objects along with an efficient inference algorithm and a proof that inference is P-complete. Our type system is sound: a well-typed program is *place safe*, that is, every access is local. Our example language extends the Abadi-Cardelli object calculus [1] with place shift and existential types, and has no type annotations. Every value is an object, and every object resides permanently at a specific place. Places surface only during place shift. Our implementation has successfully inferred types for some microbenchmarks.

Our type system. We have two forms of type, namely (1) a pair of a usual object type and a place type, and (2) a *packed type*, which is just an object type. We give a packed type to a term whose evaluation potentially creates objects at different places. The idea of a packed type is to "forget" the place type. Intuitively, a packed type is a light-weight existential type [21, 22, 19]:

$$\exists\pi.(\text{object type}, \pi)$$

where the place type π cannot occur free in the object type. In contrast to most calculi with existential types, our calculus has no type annotations. In particular, we introduce a packed type via implicit *subtyping* from a usual type to a packed type, rather than with an explicit "pack" operation. Additionally, we eliminate a packed type via the an *open* construct. In the example above, we would use subtyping to give the objects at place 1 and place 2 the same packed type, and we would give f that packed type, too. Then the *open* construct assigns a fresh place Skolem constant X to x, and finally we can successfully type check $at(x.place)\{\ x.g\ \}$ because the two occurrences of x have the same place type X.

Our inference algorithm. Our type inference algorithm is the first polynomial-time inference algorithm for existential types of which we are aware. The main technical challenge for type inference is to handle the type rule for the *open* construct. That rule introduces a place Skolem constant X for the unknown place of an object with a packed type, it assigns a type B to code that uses the packed object, and finally it requires $X \notin \Delta$ and $FV(B) \subseteq \Delta$, where Δ is a list of place Skolem constants used in enclosing *open* constructs. We handle those conditions with a novel technique. Our inference algorithm has two steps.

The first step of our algorithm transforms the type inference problem to a constraint-satisfiability problem. We show that those two problems are equivalent. Each constraint is of one of these five forms:

$$u \leq_O v \quad O \subseteq_O K \quad H \leq_H H \quad H \in_H K \quad H \neq_H \mathsf{unkn}$$

where u, v are type expressions, O is an object-type variable, K is a finite set of place types, and H is a place type variable that must be assigned a place or unkn. Intuitively, $\leq_O$ denotes subtyping, $\subseteq_O$ denotes that the type denoted by O uses only places in K, $\leq_H$ denotes subtyping for place types, $\in_H$ denotes set membership, and $\neq_H$ denotes inequality. Palsberg showed in 1993 how to solve constraints of the form $u \leq_O v$ in $O(n^3)$ time [20]. The main new challenge are the constraints of the form $O \subseteq_O K$. The problem is that a solution may assign O a deeply nested type and we need to know that every level uses only places in K.

The second step of our algorithm performs a solution-preserving *closure* of the constraint set. We show that a closed constraint set is satisfiable if and only if it is *well formed* and *consistent*. We define closure with a novel set of Horn clauses, we define *well formed* to mean that each constraint of the form $u \leq_O v$ has no top-level violation of subtyping, and we define *consistent* to mean that the constraints of the forms $H \in_H K$ and $H \neq_H$ unkn have no

obvious inconsistencies. The most time-consuming steps are the closure and the consistency check, which both take $O(n^3)$ time. In total, our algorithm takes $O(n^3)$ time.

The rest of the paper. In Section 2 we present our calculus and type system, and in Section 3 we discuss sixteen examples. In Section 4 we show that type inference is equivalent to a constraint-satisfiability problem, in Section 5 we present our type inference algorithm along with an example of how type inference works, in Section 6 we give further discussion of our results, and in Section 7 we give a detailed comparison with related work. Our paper states seven theorems; we prove one of them in the main body of the paper, five of them in the appendices of the full version of the paper, which is available from our website [9], while we leave one straightforward proof to the reader.

2 Our Language

We now present the syntax, operational semantics, and type system for our calculus.

Syntax. Here is the grammar for terms:

a, b, c	$:=$	s, x, y	(variables)
	$\mid$	o	(object)
	$\mid$	$a.l$	(method call)
	$\mid$	$a.l \Leftarrow \varsigma(x)\ b$	(method update)
	$\mid$	$at(a.place)\ b$	(at a's place)
	$\mid$	$at(\rho)\ b$	(at place ρ)
	$\mid$	$open\ x = a\ in\ b$	(let-binding)
o	$:=$	$[l_i = \varsigma(x_i)\ b_i\ ^{i \in 1..n}]$	(object, l_i distinct)
v	$:=$	$at(\rho)\ o$	(value)
ρ	$\in$	Places	(place constant)

The first four productions are those of the Abadi-Cardelli object calculus [1]. Those four productions enable us to use variables, create objects, and do method call and method update. An object defines n methods named l_i, for i between 1 and n. In a method definition $\varsigma(x_i)b$, the binder ς binds a variable x_i which refers to the entire object, in analogy with *self* in Smalltalk and *this* in Java. Additionally, b is the body of the method; the method returns the value of b. In a method call $a.l$, the callee is the method l in object a. A method update $a.l \Leftarrow \varsigma(x)\ b$, replaces the method l in object a with method $\varsigma(x)b$.

Notice that methods have no parameters, aside from a name for the receiver object, so a method type is only about a return value, not parameters. Additionally, methods are written with a ς rather than a λ, and they can be updated, which means they can also work as fields, Abadi and Cardelli showed how to encode the λ-calculus into their calculus.

The last three productions provide our extension of the Abadi-Cardelli calculus. The fifth and sixth productions enable place shift, either to the place of an object a or to a place constant ρ from a finite set Places. The last production enables us to open an object, i.e., to abstract the place of an object.

Operational semantics. Figure 1 shows the small-step operational semantics. Every value is of the form $at(\rho)\ o$, which is an object at place ρ.

We use the notation $b[x := a]$ to denote the application of a substitution $[x := a]$ to b. As usual, $b[x := a]$ denotes b with every free occurrence of x replaced with a, assuming that (if needed) all local names in b have been renamed to avoid clashes with free names in a.

Term reduction judgment: $\rho \vdash a \to a'$

(O-Obj) $$\frac{}{\rho \vdash o \to at(\rho)\ o}$$

(O-Call-Cong) $$\frac{\rho \vdash a \to a'}{\rho \vdash a.l \to a'.l}$$

(O-Call-Comp) $$\frac{o \equiv [l_i = \varsigma(x_i)\ b_i\ ^{i \in 1..n}] \quad j \in 1..n \quad \rho = \rho'}{\rho \vdash (at(\rho')\ o).l_j \to b_j[x_j := at(\rho')\ o]}$$

(O-Update-Cong) $$\frac{\rho \vdash a \to a'}{\rho \vdash a.l \Leftarrow \varsigma(x)\ b \to a'.l \Leftarrow \varsigma(x)\ b}$$

(O-Update-Comp) $$\frac{\begin{array}{l} o \equiv [l_i = \varsigma(x_i)\ b_i\ ^{i \in 1..n}] \\ o' \equiv [l_i = \varsigma(x_i)\ b_i\ ^{i \in 1..n/\{j\}},\ l_j = \varsigma(x)\ b] \\ j \in 1..n \quad \rho = \rho' \end{array}}{\rho \vdash (at(\rho')\ o).l_j \Leftarrow \varsigma(x)\ b \to at(\rho')\ o'}$$

(O-AtObject-Cong) $$\frac{\rho \vdash a \to a'}{\rho \vdash at(a.place)\ b \to at(a'.place)\ b}$$

(O-AtObject-Comp) $$\frac{}{\rho \vdash at((at(\rho')\ o).place)\ b \to at(\rho')\ b}$$

(O-AtConst-Cong) $$\frac{\rho' \vdash b \to b'}{\rho \vdash at(\rho')\ b \to at(\rho')\ b'}$$

(O-AtConst-Ret) $$\frac{}{\rho \vdash at(\rho')\ v \to v}$$

(O-Open-Cong) $$\frac{\rho \vdash a \to a'}{\rho \vdash open\ x = a\ in\ b \to open\ x = a'\ in\ b}$$

(O-Open-Comp) $$\frac{o \equiv [l_i = \varsigma(x_i)\ b_i\ ^{i \in 1..n}]}{\rho \vdash open\ x = at(\rho')\ o\ in\ b \to b[x := at(\rho')\ o]}$$

Figure 1 Operational semantics.

We use the judgment $\rho \vdash a \to a'$ to denote that at place ρ, the term a takes a step to term a'. The first five rules are variants of rules for the Abadi-Cardelli calculus. The differences are the addition of a place to each judgment and the conditions $\rho = \rho'$ in rules (S-Call-Comp) and (S-Update-Comp). Those conditions express place checks that an implementation must perform at every method call and every method update. For example, in rule (S-Call-Comp), the place check says that if we want to call a method in an object at place ρ', then we need ρ' to equal the current place ρ. In other words, the access is local. If the place check fails, then the method call or method update is stuck.

Notice that we could have written (S-Call-Comp) and (S-Update-Comp) in a simpler way by replacing ρ' with ρ and omitting the explicit condition $\rho = \rho'$. We prefer the explicit style that emphasizes the place check.

The next four rules express the semantics of place shift. In particular, rule (S-AtObject-Comp) expresses that the place of $at(\rho')o$ is ρ', while rule (S-AtConst-Ret) expresses that if the body of a place shift has evaluated to a value, then we can return that value.

The final two rules express the semantics of *open*, which is the same as the semantics of standard *let*-binding. We will use a different type rule for *open* than the usual one for *let*.

We write $\rho \vdash a \to^* a'$ if either $a = a'$, or $\rho \vdash a \to^* a''$ and $\rho \vdash a'' \to a'$.

We say that a term a is *stuck* at place ρ if a is not a value and a cannot use the rules in Figure 1 to take a step at ρ. We say that a term a can *go wrong* at place ρ if for some a', we have $\rho \vdash a \to^* a'$ and a' is stuck at ρ.

Notice that our notion of going wrong embodies the dual of a notion of *place safety*, which means that every access is local. The reason is that the semantics does place checks that leave the execution stuck if a check fails. Thus, if a term a cannot go wrong at place ρ, then we can know that every place check succeeds; hence that every access is local.

Type system. The goal of our type system is to guarantee that well-typed programs cannot go wrong. In particular, we want well-typed programs to be place safe. Accordingly, our type system has a static place check for each case where the semantics has a run-time place check. If a static place check fails, the result is a type error. Here is the grammar for types:

A, B	:=	$([l_i : B_i\ ^{i\in 1..n}], \pi)$	(locality type)
π	:=	X, Y	(place Skolem constant)
	|	unkn	(packed place)
	|	ρ	(place type constant)

As shown above, a type in our system is a pair; the first part is the object type analogous to the standard Abadi-Cardelli object type and the second part is the place type π denoting the place where an object resides. Notice that an object type can be [ ] (that is, empty), which happens when $n = 0$. A place type can either be a constant (statically known), a place Skolem constant (statically unknown but immutable) or the special type unkn (unknown). Intuitively, unkn says that there exists some place where the object resides. We use the *open* construct in our calculus to convert this existential quantification into a place Skolem constant.

Give a type $A \equiv ([l_i : B_i\ ^{i\in 1..n}], \pi)$, we define its *object component* as $obj(A) = [l_i : B_i\ ^{i\in 1..n}]$ and its *place component* as $pl(A) = \pi$.

Place Skolem constants and unknown places. We assume that place Skolem constants are drawn from a countable set $\mathsf{Skolems}$. We introduce the constant unkn where $\mathsf{unkn} \notin \mathsf{Skolems}$. We will use the syntactic sugar

$$packed\ [l_i : B_i\ ^{i\in 1..n}] = ([l_i : B_i\ ^{i\in 1..n}], \mathsf{unkn})$$

which enables simpler definitions in the following.

Figure 2 shows three well-formedness rules, three subtyping rules, and nine type assignment rules. First we explain the well-formedness rules. Rule (W-Place), Rule (W-Type-Obj), and Rule (W-Env) ensure, respectively, that a place type, a locality type and the environment are well-formed with respect to a set of place Skolem constants Δ. Here, $FV(A) \subseteq \Delta$, where $A \equiv ([l_i : B_i\ ^{i\in 1..n}], \pi)$, means that

Well-formedness:

$$\text{(W-Place)}\ \frac{FV(\pi) \subseteq \Delta}{\Delta \vdash_p \pi} \qquad \text{(W-Type-Obj)}\ \frac{FV(A) \subseteq \Delta}{\Delta \vdash_T A} \qquad \text{(W-Env)}\ \frac{\Delta \vdash_T \Gamma(x) \quad \forall x \in dom(\Gamma)}{\Delta \vdash_E \Gamma}$$

Subtyping:

$$\text{(S-Ident)}\ \frac{}{A \leq A} \qquad \text{(S-Trans)}\ \frac{A \leq B \quad B \leq C}{A \leq C}$$

$$\text{(S-Obj)}\ \frac{\pi \leq \pi'}{([l_i : B_i\ ^{i \in 1..n+k}], \pi) \leq ([l_i : B_i\ ^{i \in 1..n}], \pi')}$$

Type assignment:

$$\text{(T-Sub)}\ \frac{\Delta; \Gamma; \pi_c \ \vdash\ a : A \quad A \leq B}{\Delta; \Gamma; \pi_c \ \vdash\ a : B} \qquad \text{(T-Var)}\ \frac{\Gamma(x) = B}{\Delta; \Gamma; \pi_c \ \vdash\ x : B}$$

$$\text{(T-Obj)}\ \frac{\forall j \in 1..n \ \ \Delta; (\Gamma, x_j : A); \pi_c \ \vdash\ b_j : B_j \quad \Delta \vdash_T A \quad A \equiv ([l_i : B_i\ ^{i \in 1..n}], \pi_c)}{\Delta; \Gamma; \pi_c \ \vdash\ [l_i = \varsigma(x_i)\ b_i\ ^{i \in 1..n}] : A}$$

$$\text{(T-Call)}\ \frac{\Delta; \Gamma; \pi_c \ \vdash\ a : A \quad j \in 1..n \quad A \equiv ([l_i : B_i\ ^{i \in 1..n}], \pi) \quad \pi_c = \pi}{\Delta; \Gamma; \pi_c \ \vdash\ a.l_j : B_j}$$

$$\text{(T-Update)}\ \frac{\begin{array}{c}\Delta; \Gamma; \pi_c \ \vdash\ a : A \quad \Delta; (\Gamma, x : A); \pi \ \vdash\ b : B_j \quad j \in 1..n \\ A \equiv ([l_i : B_i\ ^{i \in 1..n}], \pi) \quad \pi_c = \pi\end{array}}{\Delta; \Gamma; \pi_c \ \vdash\ a.l_j \Leftarrow \varsigma(x)\ b : A}$$

$$\text{(T-AtObject)}\ \frac{\Delta; \Gamma; \pi_c \ \vdash\ a : A \quad A \equiv ([l_i : B_i\ ^{i \in 1..n}], \pi) \quad \Delta; \Gamma; \pi \ \vdash\ b : B}{\Delta; \Gamma; \pi_c \ \vdash\ at(a.place)\ b : B}$$

$$\text{(T-AtConst)}\ \frac{\Delta; \Gamma; \rho \ \vdash\ b : B}{\Delta; \Gamma; \pi_c \ \vdash\ at(\rho)\ b : B}$$

$$\text{(T-Open)}\ \frac{\begin{array}{c}\Delta; \Gamma; \pi_c \ \vdash\ a : A \quad (\Delta, X); (\Gamma, x : (obj(A), X)); \pi_c \ \vdash\ b : B \\ X \notin \Delta \quad \Delta \vdash_T B\end{array}}{\Delta; \Gamma; \pi_c \ \vdash\ open\ x = a\ in\ b : B}$$

$$\text{(T-Prog)}\ \frac{\Delta \vdash_E \Gamma \quad \Delta \vdash_p \pi_c \quad \Delta; \Gamma; \pi_c \ \vdash\ a : A}{\vdash_P (\Delta, \Gamma, \pi_c, a, A)}$$

Figure 2 Type rules.

$\Delta \vdash_p \pi \ \ \wedge \ \ \forall i \in 1..n : FV(B_i) \subseteq \Delta$.

Note that for every Δ, we have $\Delta \vdash_p \mathsf{unkn}$.

Next we explain the subtyping rules. Those rules rely on this definition of "width" subtyping for object types:

$$[l_i : B_i\ ^{i \in 1..n+k}] \leq [l_i : B_i\ ^{i \in 1..n}]\ .$$

Additionally, the subtyping rules rely on this definition of subtyping between place types:

$$\pi \leq \pi \qquad \pi \leq \mathsf{unkn} \qquad \mathsf{unkn} \leq \mathsf{unkn}$$

Notice that $\pi \leq \mathsf{unkn}$ can help establish a subtyping relationship between a locality type and its packed form, which we use to mask the place of an object. Rule (S-Ident) and Rule (S-Trans) are standard reflexivity and transitivity rules, while Rule (S-Obj) combines "width" subtyping rule for object types with subtyping for place types.

Finally we explain the type assignment rules. We use the judgment $\Delta; \Gamma; \pi_c \vdash a : A$ to denote that under the context $\Delta; \Gamma; \pi_c$, the term a has type A. In the context $\Delta; \Gamma; \pi_c$, we use Δ to denote a list of place Skolem constants, and we use Γ to denote a finite map from variables to types. We refer to π_c as the *place context*, that is, the type of the current place of execution.

The first rule Rule (T-Sub) is the standard subtyping rule. The next four type rules are variants of the type rules for the first-order type system for the Abadi-Cardelli object calculus. The main difference is in rules (T-Call) and (T-Update) that each contains the condition $\pi_c = \pi$, which is a type-level place check. The idea is that if the type-level place check succeeds, then the term-level place-check succeeds, too. For example, in rule (T-Call) the place check $\pi_c = \pi$ says that if we want to call a method in an object at a place with type π then we need π to equal the type π_c of the current place. In other words, the access is local. If the place check fails, then the program won't type check. Also, Rule (T-Obj) contains the check $\Delta \vdash_T A$ to ensure that the resulting object type is well formed under Δ.

The next two rules type check place shift. In both cases, the type of current place is π_c yet shifts to be π in rule (T-AtObject) or ρ in rule (T-AtConst).

Rule (T-Open) is a simplified version of the corresponding rule for full-blown existential types. It says that we can substitute a fresh place Skolem constant X for the (possibly unknown) place type of a as long as we ensure that X doesn't escape the type of the body b. This allows us to treat the place of a in an abstract manner. Unlike most rules for existential types, we do not require a to have a packed type. This is merely a technical convenience; we can always use subtyping (Rule (S-Obj)) to get a packed type for a.

Finally, Rule (T-Prog) ensures that the initial Γ and π_c for a term are well-formed with respect to the initial Δ.

Type soundness. We use the standard technique of preservation and progress [21, 25] to prove type soundness. As a key step, we introduce the notion of *place independence*. A term a is place independent if and only if we have that

$$\text{if } \Delta; \Gamma; \pi_c \vdash a : A, \text{ then } \forall \pi\text{: } \Delta; \Gamma; \pi \vdash a : A.$$

We first show that values are place independent and use that to prove a standard substitution lemma, which in turn is the corner stone of the proof of preservation.

▶ **Theorem 1** (Soundness). *If $\emptyset; \emptyset; \rho \vdash a : A$, then a cannot go wrong at ρ.*

Theorem 1 says that a well-typed program cannot go wrong, hence the program is place safe: every access is local. We prove Theorem 1 in Appendix A of the full version of the paper [9].

Type inference. Let $1 \in \mathsf{Places}$ be the initial place of computation. The type inference problem is:

Given a term a, does there exist a type A such that $\emptyset; \emptyset; 1 \vdash a : A$?

We will show how to do type inference in polynomial time.

Syntactic sugar. We will use a variant of Abadi and Cardelli's encoding of let-expressions. Suppose s doesn't occur free in a, b, and define the following object o and syntactic sugar for *let*:

$$\begin{array}{rcl} o & \equiv & [f = \varsigma(s)\ a,\ r = \varsigma(s)\ b[x := at(s.place)\ s.f]] \\ let\ x = a\ in\ b & \equiv & o.r \end{array}$$

The idea of the encoding is to create an object o that facilitates the connection between a and b. We store a in field f and we store b in field r. Computation can now begin with a call to $o.r$. The substitution $b[x := at(s.place)\ s.f]$ replaces all references of x in b with accesses to $s.f$. The main novel aspect is $at(s.place)$ which ensures that $s.f$ place checks. Intuitively, we store the result of a in field f at the *current* place, yet when the expression b references x, the computation may have moved to a *different* place. The use of $at(s.place)$ makes the access happen at the place where f is.

▶ **Theorem 2** (Derived Type Rule)**.**

$$\text{(T-Let)}\quad \frac{\Delta;\Gamma;\pi_c \vdash a : A \qquad \Delta;(\Gamma, x : A);\pi_c \vdash b : B \qquad \Delta \vdash_T A, B \qquad \Delta \vdash_p \pi_c}{\Delta;\Gamma;\pi_c \vdash let\ x = a\ in\ b : B}$$

We prove Theorem 2 in Appendix B of the full version of the paper [9]. A key lemma is that $at(s.place)\ s.f$ is place independent.

3 Examples

In this section we discuss several example programs that demonstrate key properties of our calculus. In Section 3.1 we kick off with four straightforward examples. In Section 3.2 we continue with four more advanced examples that remain within what can be handled by previous work. In Section 3.3 we finally get to eight examples of place-oblivious objects. For each example, we will either show the type produced by our inference algorithm, or we will discuss why the example has no type. Later in Section 5.4, we show how type inference works for one of the advanced examples. Featherweight X10 versions of the examples are available from our website [9].

Let o_1 denote a closed value with type B_1, that is, for any Δ, Γ and π_c, we can derive $\Delta;\Gamma;\pi_c \vdash o_1 : B_1$.

3.1 Place safety with statically known places

An object can be dereferenced safely at its own (statically known) place of creation. A type system equipped with a local/non-local place analysis should be able to track this simplest form of place correlation.

▶ Example 1.

$$\begin{array}{lrcl} at(1) & [\ l & = & \varsigma(s)\ at(1)\ [r = \varsigma(s')\ o_1], \\ & m & = & \varsigma(s)\ at(1)\ s.l \\ ] & & & \end{array}$$

The example is place safe since the outermost object, s, is both allocated and always dereferenced at place 1. The place check in Rule (T-Call) for $s.l$ succeeds and the program type checks; s has the type $([l : \mathit{packed}\ [], m : \mathit{packed}\ []], 1)$.

▶ Example 2.

$$\begin{array}{lllll} at(1) & [& l & = & \varsigma(s)\ at(1)\ [r = \varsigma(s')\ o_1], \\ & & m & = & \varsigma(s)\ at(1)\ s.l.r \\ &] & & & \end{array}$$

Compared to Example 1, we have changed the body of m from $s.l$ to $s.l.r$. This is still place safe because the object returned in the body of l is also allocated at place 1. Thus for $s.l.r$, the place check in both the uses of Rule (T-Call) succeeds and hence the program type checks; s has the type $([l : ([r : \mathit{packed}\ []], 1), m : \mathit{packed}\ []], 1)$.

▶ Example 3.

$$\begin{array}{lllll} at(1) & [& l & = & \varsigma(s)\ at(1)\ [r = \varsigma(s')\ o_1], \\ & & m & = & \varsigma(s)\ at(2)\ s.l \\ &] & & & \end{array}$$

Compared to Example 1, we now change the body of m to instead execute at place 2. This fails since s, allocated at place 1, is dereferenced at place 2. Thus the place check in Rule (T-Call) for $s.l$ fails and the program does not type check.

▶ Example 4.

$$\begin{array}{lllll} at(1) & [& l & = & \varsigma(s)\ at(1)\ [r = \varsigma(s')\ o_1], \\ & & m & = & \varsigma(s)\ at(2)\ at(1)\ s.l \\ &] & & & \end{array}$$

Compared to Example 3, the body of m starts execution at place 2 but immediately switches to place 1 and then evaluates $s.l$. This is place safe and the place check in Rule (T-Call) succeeds; s gets the type $([l : \mathit{packed}\ [], m : \mathit{packed}\ []], 1)$. Intuitively, $at(2)\ at(1)\ s.l$ is semantically equivalent to $at(1)\ s.l$.

3.2 Place safety that can be checked by previous work

As long it can be inferred that two objects are created at the same place, one can be dereferenced safely while executing at the other's (possibly abstract) place. A type system with a clever locality analysis can type check such programs.

▶ Example 5.

$$\begin{array}{lllll} at(1) & [& l & = & \varsigma(s)\ [r = \varsigma(s')\ o_1], \\ & & m & = & \varsigma(s)\ at(s.place)\ s.l.r \\ &] & & & \end{array}$$

In this example, we allocate the body of l at the place of s. Also, in the body of m, we evaluate $s.l.r$ at the (abstract) place of s. It is valid to dereference s at its own place. Also, the access $l.r$ is place safe since the body of l is allocated at s's place. Hence the place checks in Rule (T-Call) for $s.l.r$ succeed and the program type checks; s gets the type $([l : ([r : \mathit{packed}\ []], 1), m : \mathit{packed}\ []], 1)$. Note that even though s's place is statically known (place 1), place safety analysis is actually independent of that; the program would still type check if $at(1)$ is changed to $at(0)$.

▶ Example 6.

$$\begin{array}{lllll} at(1) & [& l & = & \varsigma(s)\ [r = \varsigma(s')\ o_1], \\ & & m & = & \varsigma(s)\ at(s.l.place)\ s.p, \\ & & p & = & \varsigma(s)\ o_1 \\ &] & & & \end{array}$$

Here the body of field l is allocated at the same place as s. Hence it is place safe to access $s.p$ at l's place. This program type checks; s has the type $([l : ([], 1), m : packed\ [], p : packed\ []], 1)$.

▶ Example 7.

$$\begin{array}{lllll} at(1) & [& l & = & \varsigma(s)\ [r = \varsigma(s')\ o_1], \\ & & m & = & \varsigma(s)\ at(2)\ at(s.l.place)\ s.l.r \\ &] & & & \end{array}$$

Here we want to evaluate $s.l.r$ at l's place. Similar to Example 6, this seems valid since the body of l is allocated at s's place. However, this program fails to type check because $s.l$ in $at(s.l.place)$ is first evaluated at place 2. Since s is created at place 1, the place check fails. Notice that Example 3 fails for the same reason.

▶ Example 8.

$$\begin{array}{lllll} at(1) & [& l & = & \varsigma(s)\ [r = \varsigma(s')\ o_1], \\ & & m & = & \varsigma(s)\ let\ f = s.l\ in\ at(2)\ at(f.place)\ s.l.r \\ &] & & & \end{array}$$

The problem in Example 7 is solved by introducing a *let*-expression to first ensure that $s.l$ is evaluated at s's place (place 1). Since f, l and s are at the same place, $s.l.r$ can now be safely evaluated at f's place. This program type checks; the type of s is $([l : ([r : packed\ []], 1), m : packed\ []], 1)$.

3.3 Place safety for place-oblivious objects

In Examples 5–8, place safety is based upon the abstract place of an object and on that a field stays immutable once assigned. However, in Example 5, if we add the update $s.l \Leftarrow \varsigma(s')\ at(2)\ [r = \varsigma(s')\ o_1]$, then the program will fail since the contents of l is initially allocated at s's place (place 1). This is restrictive in cases where a field might be assigned objects from different places, such as a server receiving objects from multiple nodes. We use subtyping to mask an object's place and subsequently we use *open* to "reveal" it, and thereby we ensure that a field can be safely updated.

We now show eight examples of place-oblivious objects. First we show a program that embodies the main example in the introduction, and then we show two additional programs that adds the kind of update that we discussed in the previous paragraph. After that, we finish with five more advanced examples.

▶ Example 9.

$$\begin{array}{lllll} [& l & = & \varsigma(s)\ at(1)\ [r = \varsigma(s')\ o_1], \\ & m & = & \varsigma(s)\ open\ x = s.l\ in\ at(x.place)\ x.r \\ ] & & & & \end{array}$$

This example is a place-oblivious object. The body of l returns an object created at place 1. In the body of m, we first open the place of l, essentially abstracting the place of field l as

an unknown but immutable constant. We then proceed to access its field r at that place. Note that this is place safe since we are always accessing l's fields at its own place (though without actually delving into what that place is.) Hence this example type checks; s has the type $([l : \mathit{packed}\ [r : \mathit{packed}\ []], m : \mathit{packed}\ []], 1)$, x gets the type $([r : \mathit{packed}\ []], X)$ and l's body has the type $\mathit{packed}\ [r : \mathit{packed}\ []]$.

▶ Example 10.

$$\begin{array}{rcl} [\quad l & = & \varsigma(s)\ at(1)\ [r = \varsigma(s')\ o_1], \\ \quad m & = & \varsigma(s)\ \mathit{open}\ x = s.l\ \mathit{in}\ at(x.\mathit{place})\ x.r \\ ].l & \Leftarrow & \varsigma(s)\ at(2)\ [r = \varsigma(s')\ o_1] \end{array}$$

Extending Example 9, we now proceed to update the body of l with an object at place 2. This is still place safe since subtyping ensures that l is updated with a body that returns a packed object of the same type as the original method and we still access l's fields only after opening its place. Thus this example type checks; s has the type $([l : \mathit{packed}\ [r : \mathit{packed}\ []], m : \mathit{packed}\ []], 1)$, x gets the type $([r : \mathit{packed}\ []], X)$ and l's body retains the type $\mathit{packed}\ [r : \mathit{packed}\ []]$ before and after the update.

▶ Example 11.

$$\begin{array}{rcl} [\quad l & = & \varsigma(s)\ at(1)\ [r = \varsigma(s')\ o_1], \\ \quad m & = & \varsigma(s)\ at(s.l.\mathit{place})\ s.l.r \\ ].l & \Leftarrow & \varsigma(s)\ at(2)\ [r = \varsigma(s')\ o_1] \end{array}$$

This example is a variation of Example 10 in which we have inlined the definition of x. In the expression $at(s.l.\mathit{place})\ s.l.r$, the method update may change the contents of $s.l$ *between* the evaluation of $s.l.\mathit{place}$ and the evaluation of $s.l.r$. (The semantics in Section 2 is sequential but can be changed to a more general style of reduction that supports the described behavior.) The result can be a run-time place-check error. In the example, before the update, $s.l$ contains an object at place 1, while after the change, $s.l$ contains an object at place 2. The example fails to type check. The reason is that in the absence of *open* (as in Example 10), we have no sensible type for field l. Example 10 shows how our approach uses $\mathit{open}\ x = s.l$ explicitly to create an immutable reference that avoids the stated problem and helps make the example type check.

▶ Example 12.

$$\begin{array}{rcl} \mathit{open}\ x & = & [r = \varsigma(s')\ o_1]\ \mathit{in} \\ \mathit{open}\ y & = & [r = \varsigma(s')\ o_1]\ \mathit{in} \\ [l & = & \varsigma(s)\ at(x.\mathit{place})\ y.r] \end{array}$$

Here we open two packed objects as different variables x and y and then try to access y's field at x's place. This example is place safe since both the packed objects are created at the same place. However, this program does not type check because upon opening, both x and y are assigned different place types, say X and Y. While checking $y.r$, the place check in Rule (T-Call) fails since the type system assumes that $X \neq Y$.

▶ Example 13.

$$\begin{array}{rcl} \mathit{open}\ x & = & o_1\ \mathit{in} \\ \quad \mathit{let}\ y & = & \\ \quad [l & = & \varsigma(s)\ at(x.\mathit{place})\ [r = \varsigma(s')\ o_1], \\ \quad m & = & \varsigma(s)\ \mathit{let}\ f = s.l\ \mathit{in}\ at(x.\mathit{place})\ f.r \\ \quad] & \mathit{in} & y.m \end{array}$$

This program is similar to examples 9 and 11. The only difference is that we now get an extended place context that includes a new place type (X) created by unpacking an object into x. We allocate the body of l and access its fields at X. Note that we still need the *let*-expression since s is not created at X. The example type checks; assuming execution starts at place 1, s gets the type $([l : ([r : \mathit{packed}\ []], X), m : \mathit{packed}\ []], 1)$.

▶ Example 14.

$$\begin{array}{rcl} \mathit{open}\ x & = & o_1\ \mathit{in} \\ \mathit{open}\ y & = & o_1\ \mathit{in} \\ [l & = & \varsigma(s)\ \mathit{at}(y.\mathit{place})\ [q = \varsigma(s')\ o_1], \\ m & = & \varsigma(s)\ \mathit{let}\ f = s.l\ \mathit{in}\ \mathit{at}(x.\mathit{place})\ f.q \\ ] & & \end{array}$$

This program is a variation of Example 13. Here we have an extended place context with two new type variables X and Y using two *open* expressions. This example fails to type check because we try to access an object created on place Y at the place X. Note that like Example 12, this program is place safe.

▶ Example 15.

$$\begin{array}{rcl} [l & = & \varsigma(s)\ [r = \varsigma(s')\ o_1], \\ m & = & \varsigma(s)\ \mathit{open}\ x = s.l\ \mathit{in}\ x \\ ].m.r & & \end{array}$$

In this program, the body of field m opens the place of field l's object and returns the object with a new abstract place X. By Rule (T-Open), X is not visible outside the scope of the *open* expression, hence subtyping assigns a packed type to m's body. However, since an object with a packed type cannot be dereferenced, this example fails to type-check during dereferencing of field r in Rule (T-Call).

▶ Example 16.

$$\begin{array}{rcl} \mathit{open}\ x & = & o_1\ \mathit{in} \\ \mathit{at}(1)\ ([l & = & \varsigma(s)\ \mathit{at}(x.\mathit{place})\ [q = \varsigma(z)\ [r = \varsigma(s')\ o_1]], \\ m & = & \varsigma(s)\ \mathit{at}(x.\mathit{place})\ [r = \varsigma(s')\ o_1], \\ p & = & \varsigma(s)\ \mathit{let}\ f = s.l\ \mathit{in} \\ & & \quad \mathit{at}(f.\mathit{place})\ (f.q \Leftarrow \varsigma(s')\ \mathit{at}(s.\mathit{place})\ s.m) \\ w & = & \varsigma(s)\ o_1 \\ ]).w & & \end{array}$$

Finally, a slightly more complicated example. The body of l is allocated at the abstract place X obtained by opening an object in variable x. As can be seen, the bodies of q and m are also allocated at place X. Thus it is place safe to update q's body with m's body at the place of l. This example type checks. We do need a *let*-expression in the body of p for the reason mentioned earlier. Here s gets the type $([l : ([q : \mathit{packed}\ []], X), m : \mathit{packed}\ [], p : \mathit{packed}\ []], 1)$.

4 From Types to Constraints

We show how to reduce type inference to a constraint-satisfiability problem. We define $\mathcal{K} = \mathsf{Places} \cup \mathsf{Skolems} \cup \{\mathsf{unkn}\}$.

Constraint systems. An *ACD-system* is a triple $(\mathcal{V}, \mathcal{W}, \mathcal{Q})$ where $\mathcal{V}$ is a finite set of variables (typically O) that each ranges over record types of the form $[l_i : B_i \ ^{i\in 1..n}]$, where $\mathcal{W}$ is a finite set of variables (typically H) that each ranges over $\mathcal{K}$, and where $\mathcal{Q}$ is a finite set of constraints of the five forms:

$$u \leq_O v \qquad O \subseteq_O K \qquad H \leq_H H \qquad H \in_H K \qquad H \neq_H \mathsf{unkn}$$

where u, v are either O or $[l_i : (O_i, H_i) \ ^{i\in 1..n}]$, and where K ranges over finite subsets of $\mathcal{K}$. We can view a constraint $H \neq_H \mathsf{unkn}$ as a readable way to write $H \in_H \mathcal{K} \setminus \{\mathsf{unkn}\}$. We overload $\mathcal{Q}$ and use $\mathcal{Q}$ to denote $(\mathcal{V}, \mathcal{W}, \mathcal{Q})$.

Suppose h is a mapping from $\mathcal{V}$ to record types of the form $[l_i : B_i \ ^{i\in 1..n}]$, and from $\mathcal{W}$ to $\mathcal{K}$. Define $\tilde{h}$ as:

$$\tilde{h}(O) = h(O) \qquad \tilde{h}(H) = h(H) \qquad \tilde{h}([l_i : (O_i, H_i) \ ^{i\in 1..n}]) = [l_i : (h(O_i), h(H_i)) \ ^{i\in 1..n}]$$

Let $\tilde{h}(O) \subseteq K$ denote that for every subtree of the form $([l_i : B_i \ ^{i\in 1..n}], \pi)$ or $([l_i : B_i \ ^{i\in 1..n}], \mathsf{unkn})$ in the syntax tree of $\tilde{h}(O)$, $\pi \in K$ (or $\mathsf{unkn} \in K$). Given $\tilde{h}(O) \subseteq K$, it is clear that $\Delta \vdash_T \tilde{h}(O)$ where Δ is the set of all place Skolem constants X such that $X \in K$.

We say that h is a *solution* of $\mathcal{Q}$ if

$$\begin{array}{rcrcl}
u \leq_O v \text{ in } \mathcal{Q} & : & \tilde{h}(u) & \leq & \tilde{h}(v) \\
O \subseteq_O K \text{ in } \mathcal{Q} & : & \tilde{h}(O) & \subseteq & K \\
H_a \leq_H H_b \text{ in } \mathcal{Q} & : & \tilde{h}(H_a) & \leq & \tilde{h}(H_b) \\
H_a \in_H K \text{ in } \mathcal{Q} & : & \tilde{h}(H_a) & \in & K \\
H_a \neq_H \mathsf{unkn} \text{ in } \mathcal{Q} & : & \tilde{h}(H_a) & \neq & \mathsf{unkn}
\end{array}$$

Constraint generation. We now show how to map a term to a constraint system. For a term a, we define an ACD-system $(\mathcal{V}_a, \mathcal{W}_a, \mathcal{Q}_a)$. The set $\mathcal{V}_a$ consists of a variable O_c for each occurrence of a subterm c of a, a variable $\bar{O}_{c.l_j}$ for each occurrence of a subterm $c.l_j$ of a and a variable $O^\bullet_x$ for each bound variable x. The set $\mathcal{W}_a$ consists of two variables H_c, H'_c for each occurrence of a subterm c of a, variable $\bar{H}_{c.l_j}$ for each occurrence of a subterm $c.l_j$ of a and a variable $H^\bullet_x$ for each bound variable x. Intuitively, O_c and H_c denote respectively the type of the object part and place type of c "after" subtyping, $\bar{O}_{c.l_j}$ and $\bar{H}_{c.l_j}$ denote the object part and place type of $c.l_j$ "before" subtyping and H'_c is the place type of c's place of evaluation. Additionally, $O^\bullet_x$ and $H^\bullet_x$ denote the two parts of the type that one could have declared for x. We use the rules in Figure 3 to generate the set $\mathcal{Q}_a$. Specifically, we use judgments of the form $\Delta; \bar{\Gamma} \ \vdash \ a : \mathcal{Q}_a$ to denote that for a term a in the context $(\Delta; \bar{\Gamma})$, we derive the constraint set $\mathcal{Q}_a$. Here, $\bar{\Gamma}$ is the domain of Γ. For simplicity, we use $u =_O v$ to denote the two constraints $u \leq_O v$ and $v \leq_O u$. Similarly, $u =_H v$ denotes the two constraints $u \leq_H v$ and $v \leq_H u$.

Given a constraint solution h and an environment Γ, we say that h *extends* Γ, written $h \rhd \Gamma$, if and only if $\forall x \in dom(\Gamma) : \Gamma(x) = (h(O_x), h(H_x))$.

Theorem 3 shows that we can think of typability of a term c in terms of satisfiability of $\mathcal{Q}_c$.

▶ **Theorem 3** (From Types to Constraints).
$\vdash_P (\Delta, \Gamma, \pi_c, c, C)$ *if and only if* $\Delta; \bar{\Gamma} \ \vdash \ c : \mathcal{Q}_c$ *and there exists a solution* h *for*

$$\tilde{\mathcal{Q}_c} = \mathcal{Q}_c \ \cup \ \left\{ \bigcup_{y \in \bar{\Gamma}} (O_y \subseteq_O \mathcal{D} \cup \mathsf{unkn}, H_y \in_H \mathcal{D} \cup \mathsf{unkn}) \right\} \cup \{H'_c \in_H \mathcal{D}\}$$

(where $\mathcal{D} \equiv \Delta \cup \mathsf{Places}$*) such that*

$$h \rhd \Gamma \ \wedge \ \bar{\Gamma} = dom(\Gamma) \ \wedge \ \tilde{h}(H'_c) = \pi_c \ \wedge \ (\tilde{h}(O_c), \tilde{h}(H_c)) = C \ .$$

$$\text{(C-Var)}\quad \frac{x \in \bar{\Gamma}}{\Delta; \bar{\Gamma} \;\vdash\; x : \{O_x \leq_O O_x^\bullet, H_x \leq_H H_x^\bullet\}}$$

$$\text{(C-Obj)}\quad \frac{\begin{array}{l}\Delta; (\bar{\Gamma}, x_j) \;\vdash\; b_j : \mathcal{Q}_{b_j} \qquad \forall j \in 1..n \\ o \equiv [l_i = \varsigma(x_i)\ b_i\ ^{i \in 1..n}] \\ \mathcal{Q} = \mathcal{Q}_{b_1} \cup \mathcal{Q}_{b_2} \cup ... \cup \mathcal{Q}_{b_n} \cup \\ \{\ [l_i : (O_{b_i}, H_{b_i})\ ^{i \in 1..n}] \leq_O O_o, H'_o \leq_H H_o, \\ \quad \forall j \in 1..n : O_{x_j} =_O [l_i : (O_{b_i}, H_{b_i})\ ^{i \in 1..n}], \\ \quad O_{x_j} \subseteq_O \Delta \cup \mathsf{Places} \cup \{\mathsf{unkn}\},\ H'_o =_H H_{x_j},\ H'_o =_H H'_{b_j}\}\end{array}}{\Delta; \bar{\Gamma} \;\vdash\; [l_i = \varsigma(x_i)\ b_i\ ^{i \in 1..n}] : \mathcal{Q}}$$

$$\text{(C-Call)}\quad \frac{\begin{array}{l}\Delta; \bar{\Gamma} \;\vdash\; a : \mathcal{Q}_a \qquad \mathcal{Q} = \mathcal{Q}_a \cup \{\ O_a \leq_O [l_j : (\bar{O_{a.l_j}}, \bar{H_{a.l_j}})], \\ \quad \bar{O_{a.l_j}} \leq_O O_{a.l_j},\ \bar{H_{a.l_j}} \leq_H H_{a.l_j},\ H_a =_H H'_a,\ H'_{a.l_j} =_H H'_a\}\end{array}}{\Delta; \bar{\Gamma} \;\vdash\; a.l_j : \mathcal{Q}}$$

$$\text{(C-Update)}\quad \frac{\begin{array}{l}\Delta; \bar{\Gamma} \;\vdash\; a : \mathcal{Q}_a \qquad \Delta; (\bar{\Gamma}, x) \;\vdash\; b : \mathcal{Q}_b \qquad o \equiv a.l_j \Leftarrow \varsigma(x)\ b \\ \mathcal{Q} = \mathcal{Q}_a \cup \mathcal{Q}_b \cup \\ \{\ O_a \leq_O O_o,\ H_a \leq_H H_o,\ O_a \leq_O [l_j : (O_b, H_b)],\ O_a =_O O_x,\ H_a =_H H_x, \\ \quad H_a =_H H'_a,\ H'_o =_H H'_a,\ H'_o =_H H'_b\ \}\end{array}}{\Delta; \bar{\Gamma} \;\vdash\; a.l_j \Leftarrow \varsigma(x)\ b : \mathcal{Q}}$$

$$\text{(C-AtObject)}\quad \frac{\begin{array}{l}\Delta; \bar{\Gamma} \;\vdash\; a : \mathcal{Q}_a \qquad \Delta; \bar{\Gamma} \;\vdash\; b : \mathcal{Q}_b \qquad o \equiv at(a.place)\ b \\ \mathcal{Q} = \mathcal{Q}_a \cup \mathcal{Q}_b \cup \\ \{\ O_b \leq_O O_o,\ H_b \leq_H H_o,\ H'_a =_H H'_o,\ H'_b =_H H_a,\ H_a \in_H \Delta \cup \mathsf{Places}\ \}\end{array}}{\Delta; \bar{\Gamma} \;\vdash\; at(a.place)\ b : \mathcal{Q}}$$

$$\text{(C-AtConst)}\quad \frac{\begin{array}{l}\Delta; \bar{\Gamma} \;\vdash\; b : \mathcal{Q}_b \\ \mathcal{Q} = \mathcal{Q}_b \cup \{\ O_b \leq_O O_{at(\rho)\ b},\ H_b \leq_H H_{at(\rho)\ b},\ H'_b \in_H \{\rho\}\ \}\end{array}}{\Delta; \bar{\Gamma} \;\vdash\; at(\rho)\ b : \mathcal{Q}}$$

$$\text{(C-Open)}\quad \frac{\begin{array}{l}\Delta; \bar{\Gamma} \;\vdash\; a : \mathcal{Q}_a \qquad (\Delta, X); (\bar{\Gamma}, x) \;\vdash\; b : \mathcal{Q}_b \qquad o \equiv open\ x = a\ in\ b \qquad (X \notin \Delta) \\ \mathcal{Q} = \mathcal{Q}_a \cup \mathcal{Q}_b \cup \\ \{\ O_b \leq_O O_o,\ H_b \leq_H H_o,\ O_a =_O O_x,\ H_x \in_H \{X\},\ H'_o =_H H'_a, \\ \quad H'_o =_H H'_b,\ H_b \in_H \Delta \cup \mathsf{Places} \cup \{\mathsf{unkn}\},\ O_b \subseteq_O \Delta \cup \mathsf{Places} \cup \{\mathsf{unkn}\}\ \}\end{array}}{\Delta; \bar{\Gamma} \;\vdash\; open\ x = a\ in\ b : \mathcal{Q}}$$

Figure 3 Constraint generation rules.

We prove Theorem 3 in Appendix C of the full version of the paper [9]. Notice that the size of the generated constraint system is linear in the size of the program. In the following section we show how to decide in polynomial time whether an ACD-system, such as $\tilde{\mathcal{Q}}_c$, has a solution.

5 Type Inference

We first define three central notions that we use to solve ACD-systems: closure, consistency, and well-formedness. Then we state our algorithm, analyze its complexity, and give an example of how it works.

$$\text{(Closure-}\leq_O\text{-1)}\quad \frac{u \leq_O v \qquad v \leq_O w}{u \leq_O w}$$

$$\text{(Closure-}\leq_O\text{-2)}\quad \frac{\begin{array}{c} u \leq_O [l_i : (O_{b_i}, H_{b_i})\ ^{i\in 1..n}] \\ u \leq_O [l'_i : (O'_{b_i}, H'_{b_i})\ ^{i\in 1..m}] \end{array}}{\forall l_i = l'_i,\ O_{b_i} =_O O'_{b_i} \qquad H_{b_i} =_H H'_{b_i}}$$

$$\text{(Closure-}\subseteq_O\text{)}\quad \frac{O_a \leq_O [l_i : (O_{b_i}, H_{b_i})\ ^{i\in 1..n}] \qquad O_a \subseteq_O K}{\forall O_{b_i}, O_{b_i} \subseteq_O K \qquad \forall H_{b_i}, H_{b_i} \in_H K}$$

$$\text{(Closure-}\leq_H\text{)}\quad \frac{H_a \leq_H H_b \qquad H_b \leq_H H_c}{H_a \leq_H H_c}$$

$$\text{(Closure-}\neq_H\text{)}\quad \frac{H_a \in_H K \qquad \mathsf{unkn} \notin K}{H_a \neq_H \mathsf{unkn}} \qquad \text{(Closure-}\in_H\text{-1)}\quad \frac{H_a \leq_H H_b \qquad H_a \in_H K}{H_b \in_H K \cup \{\mathsf{unkn}\}}$$

$$\text{(Closure-}\in_H\text{-2)}\quad \frac{H_a \leq_H H_b \qquad H_b \in_H K \qquad H_b \neq_H \mathsf{unkn}}{H_a \in_H K}$$

Figure 4 Constraint Closure Rules.

Theorem 5 shows how closure, consistency, and well-formedness together characterize the solvability of ACD-systems. Intuitively, the closure process makes all useful facts explicit, and if none of those facts contradict well-formedness or consistency, then the constraint system is satisfiable.

5.1 Closure, consistency, and well-formedness

We use N to range over the constraint elements of the form $[l_i : (O_i, H_i)\ ^{i\in 1..n}]$.

The *closure* of an ACD-system $\mathcal{Q}$ is the union of $\mathcal{Q}$ and the constraints that can be proved from constraints in $\mathcal{Q}$ using the rules in Figure 4. In some cases, a collection of constraints may enable multiple rules to be applied; the closure contains all conclusions that can be proved. The first two rules enable straightforward reasoning about constraints on object types, while the last four rules enable straightforward reasoning about constraints on place types. The remaining Rule (Closure-$\subseteq_O$) addresses the challenge stated in Section 1, namely the constraints of the form $O \subseteq_O K$. As stated in Section 1, the challenge is that a solution to $O \subseteq_O K$ may assign O a deeply nested type and we need to know that every level uses only places in K. Rule (Closure-$\subseteq_O$) brings possible problems to the top level by (1) propagating constraints of the form $O \subseteq_O K$ "downward" when possible and (2) generating place constraints along the way. This approach to connect reasoning about constraints on object types and constraints on place types is novel and powerful.

▶ **Theorem 4** (Closure Preserves Solutions). *An ACD-system and its closure have the same set of solutions.*

We prove Theorem 4 in Appendix D of the full version of the paper [9].

We say that an ACD-system $(\mathcal{V}, \mathcal{W}, \mathcal{Q})$ is *well-formed* if and only if for every constraint in $\mathcal{Q}$ of form $s \leq_O u$, where $s \equiv [\ldots]$ and $u \equiv [l : \ldots, \ldots]$, then $s \equiv [l : \ldots, \ldots]$. Intuitively,

we require that if two record types are related by $\leq_O$ and the right-hand side has an l field, then the left-hand side does as well. We have borrowed this notion of well-formedness from Palsberg's paper [20].

We say that $\mathcal{Q}$ is *consistent* if and only if for any $s \in \mathcal{W}$ and constraints in $\mathcal{Q}$ of the form $s \in_H K_1, \ldots, s \in_H K_n$, we have $(\bigcap_{i=1}^{n} K_i) \neq \emptyset$.

Intuitively, consistency ensures that we can solve all those n constraints. This notion of consistency is novel, and while it is simple, it is just what we need to complete our characterization of satisfiability (Theorem 5).

In Appendix E of the full version of the paper [9], we show how to map an ACD-system to an automaton $\mathcal{M}_s$ that represents a type $t_{\mathcal{M}_s}$ for a type variable $s \in (\mathcal{V} \cup \mathcal{W})$. The construction of the automaton is an extension of the construction in Palsberg's paper [20]. The following section shows an example of such an automaton. The automaton may have a cycle with at least one non-ϵ transition, in which case the automaton represents a *recursive* type, rather than a finite type as defined in Section 2. Our algorithm in Section 5.2 checks whether the automaton has such a cycle. We will use the notation $\lambda s : (\mathcal{V} \cup N \cup \mathcal{W}).t_{\mathcal{M}_s}$ to denote a function that has a single argument s drawn from the set $(\mathcal{V} \cup N \cup \mathcal{W})$, and which returns $t_{\mathcal{M}_s}$.

▶ **Theorem 5** (Solvability Characterization). *A closed ACD-system $(\mathcal{V}, \mathcal{W}, \mathcal{Q})$ is solvable with recursive types if and only if it is well-formed and consistent. If it is solvable, then $\lambda s : (\mathcal{V} \cup N \cup \mathcal{W}).t_{\mathcal{M}_s}$ is a solution.*

Theorem 5 shows that for a closed ACD-system, we can check for satisfiability with recursive types by checking well-formedness and consistency. We prove Theorem 5 in Appendix E of the full version of the paper [9].

5.2 Type inference algorithm

Give a term a, we solve the type inference problem (stated in Section 2) with the following five-steps algorithm:

1. Use $\emptyset;\emptyset \vdash a : \mathcal{Q}_a$ (Figure 3) to generate $\mathcal{Q}_a$.
2. Map $\mathcal{Q}_a$ to $\tilde{\mathcal{Q}}_a$ (as defined in Theorem 3).
3. Close $\tilde{\mathcal{Q}}_a$ (Figure 4).
4. Check whether the closure of $\tilde{\mathcal{Q}}_a$ is well formed and consistent, and whether in the automata $t_{\mathcal{M}_{O_a}}$ and $t_{\mathcal{M}_{H_a}}$, every cycle has only ϵ-transitions.
5. If the checks in Step 4 all return Yes, then output $(t_{\mathcal{M}_{O_a}}, t_{\mathcal{M}_{H_a}})$ as the type of a, and otherwise output that a fails to type check.

Implementation. We have implemented our algorithm and run it on the programs in Section 3 and also some additional programs. In every case our implementation produced the expected result.

Correctness. Our algorithm is correct because (1) from Theorem 3 we have that the type inference problem is solvable if and only if the constraint set $\mathcal{Q}$ is solvable; (2) from Theorem 4 we have that $\mathcal{Q}$ is solvable if and only if the closure is solvable; (3) from Theorem 5 we have that the closure is solvable with recursive types if and only if the closure is well-formed and consistent; and (4) an automaton can represent an infinite type only via a cycle with at least one non-ϵ-transition.

5.3 Time complexity

The following observation is helpful to establish a lower bound on our type inference problem. Our calculus is a *conservative extension* of the Abadi-Cardelli calculus with finite first-order types and no subtyping [1]. To see this, let $\bar{a}$ range over terms in the Abadi-Cardelli calculus, that is, terms built from just variables, objects, method call, and method update. Additionally, let $\bar{A}, \bar{B}$ range over types of the form $[l_i : \bar{B_i}^{\,i\in 1..n}]$, and let $\bar{\Gamma}$ range over type environments that map variable names to types of the form $[l_i : \bar{B_i}^{\,i\in 1..n}]$. Finally, let judgments of the form $\bar{\Gamma} \vdash_{AC} \bar{a} : \bar{A}$ be those of the Abadi-Cardelli calculus with finite first-order types and no subtyping [1]. As a helper function, define ext to denote the function that maps a type environment $\bar{\Gamma}$ and a place type π_c to a type environment that maps a name x in the domain of $\bar{\Gamma}$ to $(\bar{\Gamma}(x), \pi_c)$.

▶ **Theorem 6** (Conservative extension). *$\bar{\Gamma} \vdash_{AC} \bar{a} : \bar{A}$ if and only if $\Delta, ext(\bar{\Gamma}, \pi_c), \pi_c \vdash \bar{a} : (\bar{A}, \pi_c)$.*

The proof of Theorem 6 is straightforward: in each direction, do induction on the structure of the type derivation. Intuitively, the proof goes through easily because Δ and π_c don't change for terms in Abadi-Cardelli calculus.

We are now ready to state the complexity of the type inference problem.

▶ **Theorem 7** (Complexity of Type Inference). *The type inference problem is P-complete and solvable in $O(n^3)$ time, where n is the size of the program.*

Proof. Let us first consider the lower bound. Palsberg [20] showed that type inference for the Abadi-Cardelli calculus with finite first-order types and no subtyping is P-hard. We can combine that result with conservative extension (Theorem 6) and get that type inference for our calculus is P-hard.

Let us then consider the upper bound. We need to consider the execution times of closure, check for well-formedness, check for consistency, and cycle detection. We will show that we can solve each of those problems in no worse than $O(n^3)$ time.

We can bound the time to compute the closure of a constraint system by application of McAllester's meta-complexity theorem [17, Theorem 1]. McAllester's theorem says that if we have a set of closure rules R, then we can map a constraint system D' to a closure D' of D in time $O(|D'| + |P_R(D')|)$, where $P_R(D')$ is the set of all *prefix firings* in D' of rules in R. We won't recall the definition of prefix firings here but merely note that for our closure rules it is straightforward to show that $|P_R(D')| = O(n^3)$. Additionally, it is straightforward to show that $|D'| = O(n^2)$. In summary, $O(|D'| + |P_R(D')|) = O(n^3)$.

We can check well-formedness with a small variation of Palsberg's algorithm [20] to check well-formedness of constraints for type inference for the Abadi-Cardelli calculus. Palsberg's algorithm runs in $O(n^2)$ time.

We can check consistency by computing $O(n)$ intersections and unions of $O(n)$ sets of $O(n)$ elements. We represent each set as a bit vector and do the check in $O(n^3)$ time.

We can check for cycles with at least one non-ϵ-transition in $O(n^2)$ time. ◀

5.4 Example

We now demonstrate our technique on a variant of Example 9 from Section 3. Consider the following program,

$$o \;\equiv\; \mathit{open}\ x = \mathit{at}(1)\ [l = \varsigma(y)\ []]\ \mathit{in}\ \mathit{at}(x.\mathit{place})\ x.l\ .$$

$$
o \left[\begin{array}{ll} O_b \leq_O O_o & H_b \leq_H H_o \\ O_x =_O O_a & H_x \in_H \{X\} \\ O_b \subseteq_O \{0,1,\mathsf{unkn}\} & H_b \in_H \{0,1,\mathsf{unkn}\} \\ & H'_o =_H H'_a \\ & H'_o =_H H'_b \end{array}\right]
\qquad
b \left[\begin{array}{ll} O_{x.l} \leq_O O_b & H_{x.l} \leq_H H_b \\ & H'_{x_1} =_H H'_b \\ & H'_{x.l} =_H H_{x_1} \\ & H_{x_1} \in_H \{0,1,X\} \end{array}\right]
$$

$$
a \left[\begin{array}{ll} O_c \leq_O O_a & H_c \leq_H H_a \\ & H'_c \in_H \{1\} \end{array}\right]
\qquad
x.l \left[\begin{array}{ll} O_{x_2} \leq_O [l : (\bar{O_{x.l}}, \bar{H_{x.l}})] & \bar{H_{x.l}} \leq_H H_{x.l} \\ \bar{O_{x.l}} \leq_O O_{x.l} & H_{x_2} =_H H'_{x_2} \\ & H'_{x_2} =_H H'_{x.l} \end{array}\right]
$$

$$
c \left[\begin{array}{ll} [l : (O_{[]}, H_{[]})] \leq_O O_c & H'_c \leq_H H_c \\ O_y =_O [l : (O_{[]}, H_{[]})] & H'_c =_H H_y \\ O_y \subseteq_O \{0,1,\mathsf{unkn}\} & H'_c =_H H'_{[]} \end{array}\right]
\qquad
x_1 \left[\begin{array}{ll} O_x \leq_O O_{x_1} & H_x \leq_H H_{x_1} \end{array}\right]
$$

$$
x_2 \left[\begin{array}{ll} O_x \leq_O O_{x_2} & H_x \leq_H H_{x_2} \end{array}\right]
$$

$$
[] \left[\begin{array}{ll} [\,] \leq_O O_{[]} & H'_{[]} \leq_H H_{[]} \end{array}\right]
$$

Figure 5 Constraints for the example program.

Figure 5 shows the generated constraint set for o and its subterms, assuming we start at place 0 and run on two places 0 and 1. We use the abbreviations $a \equiv at(1)\ [l = \varsigma(y)\ []], c \equiv [l = \varsigma(y)\ []], b \equiv at(x.place)\ x.l$. Variables generated for the two instances of x are shown with the appropriate subscript.

We get the following closed set of constraints:

- $[] \leq_O O_{[]} \leq_O O_{\bar{x.l}} \leq_O O_{x.l} \leq_O O_b \leq_O O_o$
- $O_y \leq_O [l : (O_{[]}, H_{[]})] \leq_O O_c \leq_O O_a \leq_O O_x \leq_O O_{x_1}$
- $O_y \leq_O [l : (O_{[]}, H_{[]})] \leq_O O_c \leq_O O_a \leq_O O_x \leq_O O_{x_2} \leq_O [l : (O_{x.l}, H_{x.l})]$
- $O_{\bar{x.l}} =_O O_{[]}, O_x =_O O_a, O_y =_O [l : (O_{[]}, H_{[]})]$
- $H_x \leq_H H_{x_2} \leq_H H'_{x_2} \leq_H H'_{x.l} \leq_H H_{x_1}$
- $H_y \leq_H H'_c \leq_H H'_{[]} \leq_H H_{[]} \leq_H H_{\bar{x.l}} \leq_H H_{x.l} \leq_H H_b \leq_H H_o$
- $H'_{[]} \leq_H H'_c \leq_H H_c \leq_H H_a$
- $H_y \leq_H H'_c \leq_H H_c \leq_H H_a$
- $H_{[]} =_H H_{\bar{x.l}}, H'_o =_H H'_a =_H H'_b =_H H'_{x_1}, H_{x_2} =_H H'_{x_2} =_H H'_{x.l} =_H H_{x_1}, H_y =_H H'_c =_H H'_{[]}$
- $H_y, H'_c, H'_{[]} \in_H \{1\}, \in_H \{1, \mathsf{unkn}\}$
- $H_x, H_{x_2}, H'_{x_2}, H'_{x.l}, H_{x_1} \in_H \{X\}, \in_H \{X, \mathsf{unkn}\}, \in_H \{0,1,X\}$
- $H'_o, H'_a, H'_b, H'_{x_1} \in_H \{0\}, \in_H \{0, \mathsf{unkn}\}$
- $H_{[]}, H_{\bar{x.l}}, H_{x.l}, H_b, H_o \in_H \{0,1,\mathsf{unkn}\}$
- $H_a, H_c \in_H \{1, \mathsf{unkn}\}$
- $O_y, O_{[]}, O_{\bar{x.l}}, O_{x.l}, O_b, O_o \subseteq_O \{0,1,\mathsf{unkn}\}$

It is straightforward to check that the constraint set is well-formed and consistent, hence solvable. The solution automaton is shown in Figure 6.

For a detailed understanding of the automaton, please consult the appendices of the full version of the paper [9]. Here we merely note that for readability, we omit redundant and self ϵ-transitions from the figure. Also for each node, a so-called *labeling function value* is shown in superscript. Figure 7 shows the final type derivation for o. Note that the place check succeeds for the assigned types. Also note that since a is assigned a packed type, the program will still type check if the body of a is changed to evaluate to an object at a place other than 1.

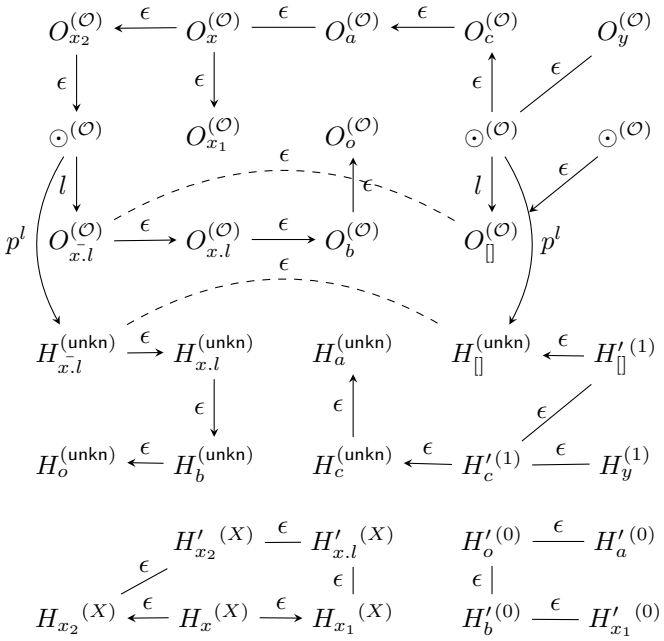

Figure 6 Automaton for the example program.

6 Discussion

In addition to the algorithm for type inference with finite types in Section 5.2, we can also do type inference with recursive types, simply by omitting the test for finiteness in Step 4 of the algorithm.

We believe that our type system is sound both for the "functional" semantics of method update in Section 2 (Theorem 1) and also for an "imperative" semantics.

For simplicity, places are not values in our calculus, yet we see no major obstacle to work with places as values.

Our algorithm can be modified to apply to the variant of Featherweight X10 that we used in the example in Section 1, as we will explain now. The only step that needs modification is constraint generation; everything else stays unchanged. The needed modifiations to the constraint generation rules in Figure 3 are rather modest. First, the rules for variables, calls, and *at* can stay essentially unchanged. Second, the rule for objects must be modified to work instead for classes and the `new` expression. Third, the rule for update must be modified to work instead for assignment and parameter passing. Fourth, the rule for open must be modified to work instead for `final`.

7 Related Work

Our work builds and improves on previous work on type systems and type inference for distributed programs. Most of the previous work comes in two flavors: object oriented or based on π-calculus. While the underlying languages are rather different, the challenges for type inference are rather similar. For object-oriented languages the challenge is about local access to fields and methods, while for π-calculus the challenge is about local access to channels.

Figure 8 gives a ten-dimensional comparison of our paper and three previous papers. The first two papers by Sewell [24] and Lhoussaine [15] are based on π-calculus [18], while the

$$\dfrac{\dfrac{\dfrac{\dfrac{\emptyset; y : (T_C, 1); 1 \ \vdash \ [\,] : ([], 1) \qquad ([], 1) \leq \mathit{packed}\ []}{\emptyset; y : (T_C, 1); 1 \ \vdash \ [\,] : \mathit{packed}\ []}}{\emptyset; \emptyset; 1 \ \vdash \ c : (T_C, 1) \quad (T_C, 1) \leq \mathit{packed}\ T_C}}{\emptyset; \emptyset; 1 \ \vdash \ c : \mathit{packed}\ T_C}}{\emptyset; \emptyset; 0 \ \vdash \ a : \mathit{packed}\ T_C}$$

$$\dfrac{X; x : (T_C, X); 0 \ \vdash \ x_1 : (T_C, X) \qquad \dfrac{X; x : (T_C, X); X \ \vdash \ x_2 : (T_C, X) \quad \boldsymbol{X = X}}{X; x : (T_C, X); X \ \vdash \ x.l : \mathit{packed}\ []}}{X; x : (T_C, X); 0 \ \vdash \ b : \mathit{packed}\ []}$$

$$\dfrac{\emptyset; \emptyset; 0 \ \vdash \ a : \mathit{packed}\ T_C \qquad X; x : (T_C, X); 0 \ \vdash \ b : \mathit{packed}\ []}{\emptyset; \emptyset; 0 \ \vdash \ o : \mathit{packed}\ []}$$

Figure 7 Type derivation for the example program. Abbreviation: $T_C \equiv [l : \mathit{packed}\ []]$.

	Sewell [24]	Lhoussaine [15]	Chandra et al. [5]	Haque & Palsberg [this paper]
Objects			√	√
Places	√	√	√	√
Place checks	√	√	√	√
at(constant place)	√	√	√	√
$at(x.place)$			√	√
Existential types		√		√
Place-oblivious objects				√
Subtyping	√	√	√	√
Type inference	√	√	√	√
Type inference in P-time				√

Figure 8 Comparison.

paper by Chandra et al. [5] and our paper are based on object-oriented languages. Each of the four papers supports places and places checks, and has a construct for place shift to a constant place, such as $at(\rho)b$ in our paper. The four papers agree on the semantics of place shift to a constant place.

The papers by Sewell [24] and Lhoussaine [15] have no notion of a place shift to a statically unknown place, such as $at(x.place)$ in our paper. Thus, one cannot express place-oblivious objects in their calculi. In contrast, the paper by Chandra et al. [5] and our paper both have notions of $at(x.place)$. One of the goals of the type system of Chandra et al. [5] is to track place correlation between objects. In particular, they provide an approach to determine whether two fields or expressions have the *same* place type. This enables successful type checking of several common programming patterns, yet leaves other open problems. In particular, the unification-based approach of Chandra et al. fails on place-oblivious objects. For example if a field f at one time references an object at place 1 and at another time references an object at place 2, and the two places 1 and 2 don't unify.

The papers by Sewell [24] and Lhoussaine [15] are part of a long line of π-calculus-based work on location-aware computation that includes [11, 4, 3]. The calculi in those papers

provide explicit language constructs for *locations* (which we call places) and *agent migration*. Notably, Hennessy and Riely [11] present an extension to the π-calculus with light-weight existential types for ensuring locality of channel communication in well-typed programs. Amadio et al. [4, 3] prove local deadlock-freedom (*receptiveness*) for a simplified version of the Hennessy-Riely calculus. Lhoussaine [15] presents an inference algorithm for a simplified version of the Hennessy-Riely calculus.

The approach of Hennessy and Riely [11] differs from ours in significant ways. In particular, the Hennessy-Riely calculus treats locations as first-class values, and a programmer has to specify explicitly the dependence between a channel name r and a location l. The resulting compound term, denoted $r@l$, serves as an explicit constructor and destructor for an existential type. In contrast, our calculus is more concise in its treatment of existentials because we use implicit subtyping to introduce a form of existential type.

The calculus of Hennessy and Riely [11] can encode a place shift. An expression such as $at(1)\ at(x.place)\ y.f$, which is place safe if x and y are at the same place, can in the Hennessy-Riely calculus be emulated by:

$$1[\![u?((x,y)@d)go\ d.y!\langle f\rangle 0]\!] \ .$$

The programmer has to use $(x,y)@d$ to specify that x and y are at the same location d. In general, the programmer has to specify any place correlation between two channels. Such place correlation can be lost if channels are quantified separately. Our calculus uses place types to track such place correlation.

The calculi in the papers [11, 4, 3, 15] use existential types in a way that is substantially different from ours. In particular, in these calculi a term can have the type

$$\exists r.\ l : loc\{r : B\}$$

our calculus quantifies the object location rather than its reference.

All four papers in Figure 8 support subtyping, though of somewhat different flavors. Sewell [24] specifies a subtyping order on channel capabilities that can distinguish local or global capabilities. Lhoussaine [15] specifies a "width" subtyping order in which a subtype defines at least the same channel names as any supertype. Finally, Chandra et al. [5] specifies a constraint-based subtype order that relates constrained types based on entailment of constraint sets. In contrast to our calculus, the calculi in the papers by Sewell [24] and Lhoussaine [15] cannot use subtyping to mask an object's location. Such subtyping is possible in the calculus of Chandra et al. but isn't fully supported by their type inference algorithm [5, Section 5].

All four papers in Figure 8 support type inference, though with completely different algorithms:

Paper	Key technique
Sewell [24]	Least upper bound of compatible types
Lhoussaine [15]	Constraint solving by unification and rewriting
Chandra et al. [5]	Constraint solving by unification
This paper	Constraint solving by closure + consistency + wellformedness

Among those four papers, only our paper has a type inference algorithm that runs in worst-case polynomial time. Sewell and Lhoussaine's papers have no discussion or results about the complexity of their inference algorithms, while Chandra et al. states that their type inference problem is undecidable (and hence that their algorithm is incomplete).

Other related work. Liblit and Aiken presented a type system and a type inference algorithm that distinguishes between local and global data [16]. The goal of their type inference algorithm is to to minimize the number of global pointers. The paper reports on an implementation for Titanium, an object-oriented language, but doesn't consider place checks, place shift, or existential types.

The idea of a type of the form $\exists\pi.([l_i : B_i \ ^{i\in 1..n}], \pi)$ stems from Jeffrey's paper on a distributed object calculus [13]. Jeffrey's calculus is a modification of the Gordon-Hankin concurrent object calculus [8] and is semantically quite different from our calculus. Intuitively, Jeffrey's calculus is about "sending the computation", while our calculus is about "sending the data." Jeffrey's calculus has no notion of place shift. Instead, it has a notion of remote spawn that superficially looks like a place shift, but more closely resembles a remote data fetch operation. Jeffrey's paper doesn't consider type inference.

The join-calculus of Fournet and Gonthier [7] has hierarchical virtual locations together with agents that can migrate between places. Their type system, however, has no notion of place checking, and the programmer has to control object distribution explicitly.

Henglein [10] presented algorithms for type inference for the Abadi-Cardelli calculus. His algorithms are faster than $O(n^3)$. However, his setting doesn't require a consistency check. Our setting requires a consistency check that with our straightforward algorithm runs in $O(n^3)$. Hence, any improvement of our bound of $O(n^3)$ must in particular improve the algorithm for consistency check.

8 Conclusion

We have presented the first type system and inference algorithm for place-oblivious objects. We believe our algorithm can be useful for programming languages such as X10. In particular, our algorithm enables a language implementation to avoid run-time place checks for place-oblivious objects.

Our calculus of distributed objects may be of independent interest. In future work we hope to extend our type system and inference algorithm with more general notions of existential types. We also hope to extend our approach to handle distributed arrays. Finally, we hope to design a calculus that does *open* implicitly.

Acknowledgments. We thank Mohsen Lesani for a meticulous review of the soundness proof, and we thank John Bender, Matt Brown, Nikolay Laptev, and the ECOOP reviewers for numerous helpful suggestions.

References

1 Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

2 Shivali Agarwal, RajKishore Barik, V. Nandivada, Rudrapatna Shyamasundar, and Pradeep Varma. Static detection of place locality and elimination of runtime checks. In G. Ramalingam, editor, *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 53–74. Springer Berlin / Heidelberg, 2008.

3 Roberto M. Amadio, Gérard Boudol, and Cédric Lhoussaine. The receptive distributed π-calculus. *ACM Trans. Program. Lang. Syst.*, 25(5):549–577, September 2003.

4 Roberto M. Amadio, Gerard Boudol, Cedric Lhoussaine, and Roberto M. Amadio. The receptive distributed π-calculus, 1999.

5 Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type inference for locality analysis of distributed data structures. In *Proceedings of the 13th ACM SIGPLAN*

Symposium on Principles and practice of parallel programming, PPoPP'08, pages 11–22, New York, NY, USA, 2008. ACM.

6 Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA'05, pages 519–538, New York, NY, USA, 2005. ACM.

7 Cedric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer Berlin Heidelberg, 2002.

8 Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing, 1998.

9 Riyaz Haque and Jens Palsberg. Type inference for place-oblivious objects. Full version of a paper with the same title in ECOOP 2015, `http://www.cs.ucla.edu/~palsberg/paper/ecoop15full.pdf`.

10 Fritz Henglein. Breaking through the n^3 barrier: Faster object type inference. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, 1997.

11 Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:2002, 1998.

12 Paul N. Hilfinger, Dan Oscar Bonachea, Kaushik Datta, David Gay, Susan L. Graham, Benjamin Robert Liblit, Geoffrey Pike, Jimmy Zhigang Su, and Katherine A. Yelick. Titanium language reference manual, version 2.19. Technical Report UCB/EECS-2005-15, EECS Department, University of California, Berkeley, Nov 2005.

13 A. S. A. Jeffrey. A distributed object calculus. In *Proc. Foundations of Object Oriented Languages*, 2000.

14 Jonathan K. Lee and Jens Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of PPOPP'10, 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, January 2010.

15 Cedric Lhoussaine. Type inference for a distributed π-calculus. In Pierpaolo Degano, editor, *Programming Languages and Systems*, volume 2618 of *Lecture Notes in Computer Science*, pages 253–268. Springer Berlin Heidelberg, 2003.

16 Ben Liblit and Alexander Aiken. Type systems for distributed data structures. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL*, pages 199–213, 2000.

17 David McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.

18 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, September 1992.

19 Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, May 1999.

20 Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995. Preliminary version in Proceedings of LICS'94, Ninth Annual IEEE Symposium on Logic in Computer Science, pages 186–195, Paris, France, July 1994.

21 Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

22 Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, Cambridge, MA, USA, 2004.

23 Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification. Technical report, IBM, January 2012.

24 Peter Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *In Proceedings of ICALP'98, LNCS 1443*, pages 695–706. Springer-Verlag, 1998.

25 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1992.

26 Katherine A. Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul N. Hilfinger, Susan L. Graham, David Gay, Phillip Colella, and Alexander Aiken. Titanium: A high-performance Java dialect. *Concurrency – Practice and Experience*, 10(11-13):825–836, 1998.