Constrained Types for Object-Oriented Languages

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

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

Radha Jagadeesan
Depaul University
rjagadeesan@cs.depaul.edu

Jens Palsberg
University of California–Los Angeles
palsberg@cs.ucla.edu

Christian Grothoff
University of Denver
christian@grothoff.org

Abstract

We formalize the basic ideas of dependent-types for Java-like languages, in the context of FX10, a Featherweight version of X10.

1. Introduction

X10 is a modern statically typed object-oriented (OO) language designed for high productivity in the high performance computing (HPC) domain [23]. X10, like most OO languages is designed around the notion of objects, as instances of *classes*. However, X10 places equal emphasis on *arrays*, a central data-structure in high performance computing. In particular, X10 supports dense, distributed multi-dimensional arrays of value and reference types, built over index sets known as *regions*, and mappings from index sets to places, known as *distributions*. X10 supports a rich algebra of operations over regions, distributions and arrays.

In designing a static type system for X10 a central problem arises. It becomes necessary to permit types, such as `region(2)`, the type of all two-dimensional regions, `int[5]`, the type of all arrays of `int` of length `5`, and `int[region(2)]`, the type of all `int` arrays over two dimensional regions. The underlying general idea is that of *dependent types*: types parametrized by *values* [28].

XXX emph pluggable nature of the constraint system.

In this paper we develop a general syntactic and semantic framework for *constrained types*, user-defined dependent types in the context of modern class-based OO languages such as Java, C♯ and X10. Our central insight is that a rich, user-extensible type system can be developed on top of predicates over the *immutable* state of objects. Such types statically capture many common invariants naturally arising in code. Given a single dependently typed class or interface `C`, a user may construct a potentially infinite family of types `C(:c)` where `c` is a predicate on the immutable state.

In designing this framework, we had the following criteria:

- **Ease of use.** The framework must be easy to use for practicing programmers.
- **Flexibility.** The framework must permit the development of concrete, specific type systems tailored to the application area at hand, enabling a kind of pluggable type system [2]. Hence, the framework must be parametric in the kinds of expressions used in the type system.
- **Modularity.** The rules for type-checking must be specified once in a way that is independent of the particular vocabulary of operations used in the dependent type system.
- **Integration with OO languages.** The framework must work smoothly with nominal type systems found in Java-like languages, and must permit separate compilation.
- **Static checking.** The framework must permit mostly static type-checking.

1.1 Overview

XXX Overview of our design.

Our basic approach to introducing dependent types into class-based statically typed OO languages is to follow the spirit of generic types, but use values instead of types.

We permit the definition of a class `C` to specify a list of typed parameters, or *properties*, `(T1 x1, ..., Tk xk)` similar in syntactic structure to a method argument list. Each property in this list is treated as a `public final` instance field of the class. We also permit the specification of a *class invariant*, a *where clause* [6] in the class definition. A where clause is a boolean expression on the parameters separated from the parameter list with a "`:`". The compiler ensures that all instances of the class created at runtime satisfy the where clause associated with the class. Thus, for instance, we may specify a class `List` with an `int length` property as follows:

```
class List(int length: length >= 0) {...}
```

Given such a definition for a class `C`, types can be constructed by *constraining* the properties of `C`. In principle, *any* boolean expression over the properties specifies a type: the type of all instances of the class which satisfy the boolean expression. Thus, `List(:length == 3)` is a permissible type, as are `List(:length <= 41)` and even `List(:length * f() >= g())`. If `C` has no properties, the only type that can be constructed is the type `C`.

Accordingly, a *constrained type* is of the form `C(:e)`, the name of a class or interface `C`, called the *base class*, followed by a where clause `e`, called the *condition*, a boolean expression on the properties of the base class. The denotation, or semantic interpretation, of such a type is the set of all instances of subclasses of the base class whose properties satisfy the condition. Clearly, for the denotation of a constrained type t to be non-empty the condition of t must be consistent with the class invariant, if any, of the base class of t. The compiler is required to ensure that the type of any variable declaration is non-empty.

For brevity, we write `C` as a type as well; it corresponds to the (vacuously) constrained type `C(:true)`. We also permit the syntax `C(t1,..., tk)` for the type `C (:x1 == t1 && ...&& xk == tk)` (assuming that the property list for `C` specifies the `k` properties

`x1,..., xk`, and each term `ti` is of the correct type). Thus, using the definition above, `List(n)` is the type of all lists of length `n`.

1.2 An example

Figure 1 shows a `List` class. Intuitively, this definition states that a `List` has a `int` property `length`, which must be non-negative. The class has two fields that hold the head and tail of the list.

The first constructor `List()` has a postcondition that signifies that it returns a list of length 0. The constructor body contains a `property` statement that initializes the `length` property to `0`. The compiler verifies that the constructor postcondition and the class invariant are implied by the `property` statement and any `super` calls in the constructor body imply

The second constructor returns a singleton list of length `1`. The third constructor returns a list of length `m+1`, where `m` is the length of the second argument. If an argument appears in the return type then the argument must be declared `final`. Thus the argument will point to the same object throughout the evaluation of the constructor body.

Functions that append one list to another or that reverse a list can be defined quite naturally. In both cases the size of the list returned is known statically (as a function of the size of the list and the argument to the method). The example also illustrates a method `filter` which returns a list whose length cannot be known statically since it depends on the argument predicate `f`; the list length can be bounded, however.

1.3 Claims

XXX Claims of this paper.

As in staged languages [14, 26], the design distinguishes between compile-time and run-time evaluation. Constrained types are checked (mostly) at compile-time. The compiler uses a constraint solver to perform universal reasoning ("for all possible values of method parameters") for dependent type-checking. There is no run-time constraint-solving. However, run-time casts to dependent types are permitted; these casts involve arithmetic, not algebra—the values of all parameters are known.

We outline the design for a compiler that can use an extensible constraint-solver for type-checking. This design is implemented in the X10 compiler. No extension of an underlying virtual machine is necessary, except as may be useful in improving efficiency (for example, eliminating array bounds checks). The compiler translates source programs into target programs without dependent types but with `assume` and `assert` statements. A general constraint propagator that depends only on the operational semantics of the language and is constrained on the underlying constraint solver may be run on the program in order to eliminate branches and `asserts` forced by the assumptions. If all `asserts` cannot be eliminated at compile-time, some residual constraint-checking (*not* solving) may need to be performed at runtime. XXX contrast with hybrid type checking.

Rest of this paper. The next section reviews related work. Section 3 fleshes out the syntactic and semantic details of the proposal. Formal semantics and a soundness theorem are presented in Section 4. Section 5 works through a number of examples using a variety of constraint systems. Sections 7 and 8 conclude the paper with a discussion of future work.

2. Related work

Constraint-based type systems:

[13]
[9]
[5]
[1]
[11]
[24]
[17]
[27]

Types of the form $T\backslash C$, where C is a set of subtyping constraints.

Not dependent.

[7]

Pottier [18, 20] presents a constraint-based type system for an ML-like language with subtyping.

HM(X) [25, 19, 21] is a constraint-based framework for Hindley–Milner style type systems. The framework is parameterized on the specific constraint system X; instantiating X yields extensions of the HM type system.

- Pottier
- HM(X)
- Xi and Pfenning (POPL99)

  Dependent types in practical programming. Dependent ML. DML(C). Objects drawn from constraint domain C.

  Index objects must be pure. Singleton types int(n).

  ML_0^{Π}: Refinement of the ML type system: does not affect the operational semantics. Can erase to ML_0.
- Ada dependent types

  Ada has constrained array definitions. A constraint [**?**]. Not clear if their dependent. Are there other dependent types?
- singleton kinds (Chris Stone)
- Nested types, vObj, Scala, gbeta, J&

  Nested types: witness types, p vObj, Scala: p.type. J&: p.class gbeta: p.C
- Where clauses for F-bounded polymorphism (Theta and PolyJ)

  Bounded quantification: Cardelli and Wegner. Bound T with T' F-bounds: Canning, Cook, Hill, Olthoff, Mitchell. Bound T with F(T).

  Not dependent types.
- Hybrid type checking (Flanagan, POPL06)

  Refinement types. Types can be arbitrary predicates:

  z : Int — z ¿= 0 equivalent to the X10 type: Int(:self ¿= 0)

  Subtyping is undecidable. Type-checker can report "yes", "no", or "don't know". If the latter, dynamic checks inserted where subsumption occurs.

  Also Hybrid Types, Invariants and Refinements for Imperative Objects (Flanagan, Freund, Tomb, FOOL06).

  Constraints must be pure. Update is not pure. Calls must have pure actuals + receiver. What about alising?
- Soft typing with conditional types (Aiken, POPL94)

  Aiken, Wimmers, and T.K. Lakshman.
- Cayenne (Augustsson, ICFP98)

  Result type of function can depend on argument value. Any expression is a type. Haskell-like. Undecidable type checking.
- Cardelli, type checking dependent types and subtypes
- Esterel
- Findler, Felleisen, Contracts for higher-order functions (ICFP02)

  example: int[¿ 9]

  contracts are either simple predicates or function contracts. defined by (define/contract ...)

```
public value class List(int length : length >= 0) {
  Object head = null;
  List(length-1) tail = null;

  /** Returns the empty list. */
  public List(:length==0)() {
    property(0);
  }

  /** Returns a singleton list. */
  public List(:length==1)(Object head) {
    this(head, new List());
  }
  /** Returns the cons of head and tail. */
  public List(:length==tail.length+1)(Object head, final List tail) {
    property(tail.length+1);
    this.head = head;
    this.tail = tail;
  }
  public List(length+arg.length) append(final List arg) {
    return (length == 0)
          ? arg
          : new List(head, tail.append(arg));
  }
  public List(length) rev() {
    return rev(new List());
  }
  public List(length+acc.length) rev(final List acc) {
    return (length == 0)
          ? acc
          : tail.rev(new List(head, acc));
  }
  /** Return a list of compile-time unknown length, obtained by filtering
      this with f. */
  public List(:self.length <= this.length) filter(Predicate f) {
    if (length==0) return this;
    if (f.isTrue(head)) {
      List<Object> l = tail.filter(f);
      return new List(l+1)(head, l);
    } else {
      return tail.filter(f);
    }
  }
}
```

Figure 1. List example

enforced at run-time.

- Jif (final access paths in security labels)
- FX-90
- ESCJava [8], Spec#
- JSR 308, Javari
- Freeman, Pfenning, Refinement types for ML (PLDI91)
- Holt, Cordy, the Turing programming language
- Mandelbaum, Walker, Harper, effective thy of type refinements
- Ou, Tan, Mandelbaum, Walker, Dynamic typing with dependent types

  Separate dependent and simple parts of the program. Statically type the dependent parts. Dynamic checks when passing values into dependent part.
- Dependently typed data structurse (Xi)
- Dead code elimination through dependent types (Xi)

3. Constrained FJ

Our basic approach to introducing dependent types into class-based statically typed OO languages is to follow the spirit of generic types, but use values instead of types.

We permit the definition of a class `C` to specify a list of typed parameters, or *properties*, `(T1 x1, ..., Tk xk)` similar in syntactic structure to a method argument list. Each property in this list is treated as a `public final` instance field of the class. We also permit the specification of a *class invariant*, a *where clause* [6] in the class definition. A where clause is a boolean expression on the parameters separated from the parameter list with a "`:`". The compiler ensures that all instances of the class created at runtime satisfy the where clause associated with the class. Thus, for instance, we may specify a class `List` with an `int length` property as follows:

```
class List(int length: length >= 0) {...}
```

Given such a definition for a class `C`, types can be constructed by *constraining* the properties of `C`. In principle, *any* boolean expression over the properties specifies a type: the type of all instances of the class which satisfy the boolean expression. Thus, `List(:length == 3)` is a permissible type, as are `List(:length`

<= 41) and even List(:length * f() >= g()). If C has no properties, the only type that can be constructed is the type C.

Accordingly, a *constrained type* is of the form C(:e), the name of a class or interface C, called the *base class*, followed by a where clause e, called the *condition*, a boolean expression on the properties of the base class. The denotation, or semantic interpretation, of such a type is the set of all instances of subclasses of the base class whose properties satisfy the condition. Clearly, for the denotation of a constrained type t to be non-empty the condition of t must be consistent with the class invariant, if any, of the base class of t. The compiler is required to ensure that the type of any variable declaration is non-empty.

For brevity, we write C as a type as well; it corresponds to the (vacuously) constrained type C(:true). We also permit the syntax C(t1,..., tk) for the type C (:x1 == t1 && ...&& xk == tk) (assuming that the property list for C specifies the k properties x1,..., xk, and each term ti is of the correct type). Thus, using the definition above, List(n) is the type of all lists of length n.

Parametric types naturally come equipped with a *subtyping structure*: type t_1 is a subtype of t_2 if the denotation of t_1 is a subset of t_2. This definition satisfies Liskov's Substitution Principle [12]), and implies that C(:e1) is a subtype of C(:e2) if e1 implies e2. In particular, for all conditions e, C(:e) is a subtype of C. C(:e) is empty exactly when e conjoined with C's class invariant is inconsistent.

Two constrained types C1(:e1) and C2(:e2) are considered equivalent if C1 and C2 are the same base type and e1 and e2 are equivalent when considered as logical expressions.

3.1 Method and constructor preconditions

Methods and constructors may specify preconditions on their parameters as where clauses. For an invocation of a method (or constructor) to be type-correct, the associated where clause must be statically known to be satisfied. The return type of a method may also contain expressions involving the arguments to the method. However, we will require that any argument used in this way must be declared final, ensuring it is not mutated by the method body. For instance:

```
List(arg.length-1)
  tail(final List arg : arg.length > 0) {...}
```

will be a valid method declaration. It says that tail must be passed a non-empty list, and it returns a list whose length is one less than its argument.

3.1.1 Constructors for dependent classes

Like a method definition, a constructor may specify preconditions on its arguments and a postcondition on the value produced by the constructor.

Postconditions may be specified in a constructor declaration between the name of the class and the argument list of the constructor using a where clause. The where clause can reference only the properties of the class.

For instance, the nullary constructor for List ensures that the property length has the value 0:

```
/** Returns the empty list. */
public List(:length==0)() {
  property(0);
}
```

The property statement is used to set all the properties of the new object simultaneously. Capturing this assignment in a single statement simplifies checking that the constructor postcondition and class invariant are established. In a class has properties, every path through the constructor must contain exactly one property statement.

3.2 Constraints

In this framework, types may be constrained by any boolean expression over the properties. For practical reasons, restrictions need to be imposed to ensure constraint checking is decidable.

The condition of a constrained type must be a pure function only of the properties of the base class. Because properties are final instance fields of the object, this requirement ensures that whether or not an object belongs to a constrained type does not depend on the *mutable* state of the object. That is, the status of the predicate "this object belongs to this type" does not change over the lifetime of the object. Second, by insisting that each property be a *field* of the object, the question of whether an object is of a given type can be determined merely by examining the state of the object and evaluating a boolean expression. Of course, an implementation is free to not *explicitly* allocate memory in the object for such fields. For instance, it may use some scheme of colored pointers to implicitly encode the values of these fields [**?**].

Further, by requiring that the programmer distinguish certain final fields of a class as properties, we ensure that the programmer consciously controls *which* final fields should be available for constructing constrained types. A field that is "accidentally" final may not be used in the construction of a constrained type. It must be declared as a property.

Java-like languages permit constructors to throw exceptions. This is necessary to deal with the situation in which the arguments to a constructor for a class C are such that no object can be constructed which satisfies the invariants for C. Dependent types make it possible to perform some of these checks at compile-time. The class invariant of a class explicitly captures conditions on the properties of the class that must be satisfied by any instance of the class. Constructor preconditions capture conditions on the constructor arguments. The compiler's static check for non-emptiness of the type of any variable captures these invariant violations at compile-time.

3.3 Extending dependent classes

A class may extend a constrained class.

*MetaNote: This should be standard. A class definition may extend a dependent super class, e.g. class Foo(int i) extends Fum(i*i)* { ... }. *The expressions in the actual parameter list for the super class may involve only the parameters of the class being defined. The intuition is that these parameters are analogous to explicit arguments that must be passed in every super-constructor invocation.*

3.4 Dependent interfaces

Java does not allow interfaces to specify instance fields. Rather all fields in an interface are final static fields (constants).

X10 supports rich user-definable extensions to the type system by allowing the user of a type to construct new constrained types: new types that are predicates on the immutable state of the base type. For interfaces to support this extension, they must support user-definable properties, so that constrained types can be built over interfaces.

As with classes, an interface definition may specify properties in a list after the name of the interface. Similarly, an interface definition may specify a where clause in its property list. Methods in the body of an interface may have where clauses as well.

All classes implementing an interface must have a property with the same name and type (either declared in the class or inherited from the superclass) for each property in the interface. If a class implements multiple interfaces and more than one of them specify a property with the same name, then they must all agree on the type of the property. The class must have a single property with the given name and type.

The general form of a class declaration is now:

```
class C(T1 x1, ..., Tk xk)
      extends B(:e)
      implements I1(:e1), ..., In(:en) {...}
```

For such a declaration to type-check, it must be that the class invariant implies `inv(I) && e`, where `inv(I)` is the invariant associated with interface `I`. Again, a parameteric class or interface `I` is taken as shorthand for `I(:true)`. Further, every method specified in the interface must have a corresponding method in the class with the same signature whose precondition, if any, is implied by the precondition of the method in the interface.

3.5 XXX more stuff

CFJ with field assignments.

Discussion of language design issues

– how should method resolution be done in the presence of constrained types?

– conditional fields. – recursive definitions of predicates in the constraint language through the use of CLP.

Constraint system (generic presentation). Design is constraint-system agnostic.

Principal clause

4. Semantics

Syntax. The syntax for the language is specified in Figure 3.

We assume a fixed constraint system, $\mathcal{C}$, with inference relation $\vdash_{\mathcal{C}}$. All constraint systems are required to support the trivial constraint `true`, conjunction, existential quantification and equality on constraint terms. Constraint terms include (final) variables, the special variable `self` (which may occur only in constraints `c` which occur in a constrained type `C(:c)`), and field selections `t.f`. Finally, we assume that constraints are closed under variable substitution. We denote the application of the substitution $[\overline{t}/\overline{x}]$ to a constraint c by $c[\overline{t}/\overline{x}]$.

A type declaration `C(:c) x` constrains `x` to contain references to only those objects `o` that are instances of (subclasses of) `C` and for which the constraint `c` is true provided that occurrences of `self` in `c` are replaced by `o`. Thus in the constraint `c` of a constrained type `C(:c)`, `self` may be used to reference the object whose type is being specified. Note that `self` is distinct from `this` – `this` is permitted to occur in the clause of a type `T` only if `T` occurs in an instance field declaration or instance method declaration of a class; as usual, `this` is considered bound to the instance of the class to which the field or method declaration applies.

A class declaration `class C(`$\overline{\mathtt{T}}\ \overline{\mathtt{f}}$`: c) extends D(: d)` $\{\overline{\mathtt{M}}\}$ is thought of as declaring a class `C` with the fields $\overline{\mathtt{f}}$ (of type $\overline{\mathtt{T}}$), a *declared class invariant* `c`, a *super-class invariant* `d` and a collection of methods $\overline{\mathtt{M}}$. The constraints `c` and `d` are true for all instances of the class `C` (this is verified in the rule for type-checking constructors, T-New). In these constraints, `this` may be used to reference the current object; `self` does not have any meaning and must not be used.

A method declaration $\mathtt{T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}} : \mathtt{c})\{\ldots\}$ specifies the type of the arguments and the result, as usual. The method arguments $\overline{\mathtt{x}}$ may occur in the argument types $\overline{\mathtt{T}}$ and the return type $\mathtt{T}_0$. The constraint `c` specifies additional constraints on the arguments $\overline{\mathtt{x}}$ and `this` that must hold for a method invocation to be legal. Note that `self` does not make sense in `c` (no type is being defined), and must not occur in `c`.

Type judgements. $\sigma(\Gamma)$ is the set of constraints on the variables whose type assertions are specified by Γ, generated by replacing each type assertion `C(:c) x` in Γ with `c`$[\mathtt{x}/\mathtt{self}]$. The rule T-Constr permits a type `C(:c)` for a variable `x` to be strengthened with information entailed per $\mathcal{C}$ from the information about `x` and other variables specified in Γ.

The rule T-Inv permits information about the class invariant `c` of a class `C` to be used when determining the constrained clause for a variable of type `C`.

In T-Field, we postulate the existence of a receiver object `o` of the given static type ($\mathtt{T}_0$). $fields(\mathtt{T}_0, o)$ is the set of typed fields for $\mathtt{T}_0$ with all occurrences of `this` replaced by `o`. We record in the resulting constraint that $\mathtt{o.f}_i = \mathtt{self.f}_i$.[1] This permits transfer of information that may have been recorded in $\mathtt{T}_0$ about the field $\mathtt{f}_i$.

Similarly, in T-Invk we postulate the existence of a receiver object `o` of the given static type. For any type T, object `o` of type T and method name `m`, let $mtype(\mathtt{T}, \mathtt{m}, \mathtt{o})$ be a copy of the signature of the method with `this` replaced by `o`. We establish (under the assumption that the formals ($\overline{\mathtt{z}}$) have the static type of the actuals)[2] that actual types are subtypes of the formal types, and the method constraint is satisfied. This permits us to record the constraint `d` on the return type, with the formal variables $\overline{\mathtt{z}}$ existentially quantified.[3]

In T-New, similarly, we establish that the static types of the actual arguments to the constructor are subtypes of the declared types of the field, and contain enough information to satisfy the class invariant, `c`. The declared types (and `c`) contain references to `this.`$\overline{\mathtt{f}}$; these must be replaced by the formals $\overline{\mathtt{f}}$, which carry information about the static type of the actuals. Note that the object `o` we hypothesized in an analogous situation in T-Invk does not exist; it will exist on successful invocation of the constructor. The constrained clause of the `new` expression contains all the information that can be gleaned from the static types of the actuals by assigning them to the corresponding fields of the object being created.

THEOREM 4.1 (Subject Reduction). *If* $\Gamma \vdash T$ `e` *and* $\mathtt{e} \longrightarrow \mathtt{e}'$, *then for some type* S, $\Gamma \vdash S$ $\mathtt{e}'$ *and* $\Gamma \vdash S \sqsubseteq T$.

Let the normal form of expressions be given by *values*, i.e. expressions

(Values) `v` ::= `new C(`$\overline{\mathtt{v}}$`)`

THEOREM 4.2 (Type Soundness). *If* $\vdash T$ `e` *and* $\mathtt{e} \longrightarrow^* \mathtt{e}'$, *then* $\mathtt{e}'$ *is either (1) a value with* $\vdash$ `S v` *and* $\vdash$ `S` $\sqsubseteq$ `T`, *for some type* `S`, *or, (2) an expression containing a subexpression* `(T)new C(`$\overline{\mathtt{v}}$`)` *where* $\not\vdash$ `C` $\sqsubseteq$ `T`$[\mathtt{new\ C}(\overline{\mathtt{v}})/\mathtt{self}]$.

5. Examples

Specific constraint systems used when developing specific examples.

In the following we will use the shorthand `C(`$\overline{\mathtt{t}}$`: c)` for the type `C(:`$\overline{\mathtt{f}} = \overline{\mathtt{t}}$`,c)` where the declaration of the class `C` is `class C(`$\overline{\mathtt{T}}\ \overline{\mathtt{f}}$`: c)`.... Also, we abbreviate `C(`$\overline{\mathtt{t}}$` : true)` as `C(`$\overline{\mathtt{t}}$`)`. Finally, we will also have need to use the shorthand $\mathtt{C}_1(\overline{t}{:}\mathtt{c}_1)\&\ldots\mathtt{C}_k(\overline{\mathtt{t}}_k : \mathtt{c}_k)$ for the type $\mathtt{C}_1(:\overline{\mathtt{f}}_1 = \overline{\mathtt{t}}_1, \ldots, \overline{\mathtt{f}}_k = \overline{\mathtt{t}}_k, \mathtt{c}_1, \ldots, \mathtt{c}_k$ provided that the $\mathtt{C}_i$ form a subtype chain and the declared fields of $\mathtt{C}_i$ are $\mathtt{f}_i$.

Constraints naturally allow for partial specification (e.g. inequalities) or incomplete specification (no constraint on a variable) with the same simple syntax. In the example below, the type of `a` does not place any constraint on the second dimension of `a`, but this dimension can be used in other types (e.g. the return type).

[1] A new name `o` is necessary to name this object since `e` cannot be used. Arbitrary term expressions `e` are not permitted in constraints; the functions used in `e` may not be known to the constraint system, and `e` may have side-effects.

[2] This is stronger than assuming $\overline{\mathtt{z}}$.

[3] Recall that the $\overline{\mathtt{z}}$ may occur in `d` but most not occur in a type in the calling environment; hence they must be existentially quantified in the resulting constraint.

Syntax. The rules for CFJ are:

$$
\begin{array}{rrcl}
\text{(Class)} & \mathtt{L} & ::= & \mathtt{class\ C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}} : \mathtt{c})\ \mathtt{extends\ T}\ \{\overline{\mathtt{M}}\} \\
\text{(Method)} & \mathtt{M} & ::= & \mathtt{T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}} : \mathtt{c})\{\mathtt{return\ e;}\} \\
\text{(Expr)} & \mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{new\ C}(\overline{\mathtt{e}}) \mid (\mathtt{T})\mathtt{e} \\
 & & & \\
\text{(C Terms)} & \mathtt{t} & ::= & \mathtt{x} \mid \mathtt{self} \mid \mathtt{this} \mid \mathtt{t.f} \\
\text{(Constraint)} & \mathtt{c,d} & ::= & \mathtt{true} \mid \mathtt{a} \mid \mathtt{t} = \mathtt{t} \mid \mathtt{c,c} \mid \mathtt{T\,x;c} \\
\text{(Type)} & \mathtt{S,T,U} & ::= & \mathtt{C}(:\mathtt{d})
\end{array}
$$

Subtyping Judgements. We let Γ stand for multisets of type assertions, of the form $T\,x$, and constraints. We define $\sigma(\Gamma)$ to be the set of constraints obtained from Γ by replacing each type assertion $\mathtt{C}(:\mathtt{d})\ \mathtt{x}$ with $\mathtt{d}[\mathtt{x}/\mathtt{self}]$.

$$\Gamma \vdash \mathtt{T} \sqsubseteq \mathtt{T} \qquad \frac{\mathtt{class\ C}(\ldots)\ \mathtt{extends\ D}(\ldots)\{\ldots\}}{\vdash \mathtt{C} \sqsubseteq \mathtt{D}}$$

$$\frac{\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{D} \quad \sigma(\Gamma), \mathtt{c} \vdash_{\mathcal{C}} \mathtt{d}}{\Gamma \vdash \mathtt{C}(:\mathtt{c}) \sqsubseteq \mathtt{D}(:\mathtt{d})} \qquad \frac{\Gamma \vdash \mathtt{S} \sqsubseteq \mathtt{T} \quad \Gamma \vdash \mathtt{T} \sqsubseteq \mathtt{U}}{\Gamma \vdash \mathtt{S} \sqsubseteq \mathtt{U}}$$

Type Judgements. Let $\mathtt{C}$ be a class declared as $\mathtt{class\ C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}} : \mathtt{c})\ \mathtt{extends\ D}(:\mathtt{d})\{\overline{\mathtt{M}}\}$. We let $inv(\mathtt{C})$ stand for the conjunction $\mathtt{c,d}$ and (recursively) $inv(\mathtt{D})$. We bottom out with $inv(\mathtt{Object}) = \mathtt{true}$.

$$\Gamma, \mathtt{T\ x} \vdash \mathtt{T\ x}\ \text{(T-Var)}$$

$$\frac{\Gamma \vdash \mathtt{C}(:\mathtt{c})\ \mathtt{x} \quad \sigma(\Gamma), \mathtt{c}[\mathtt{x}/\mathtt{self}] \vdash_{\mathcal{C}} \mathtt{d}[\mathtt{x}/\mathtt{self}]}{\Gamma \vdash \mathtt{C}(:\mathtt{d})\ \mathtt{x}}\ \text{(T-Constr)} \qquad \frac{\Gamma, \mathtt{c}[\mathtt{x}/\mathtt{this}] \vdash \mathtt{C}(:\mathtt{d})\ \mathtt{x} \quad \mathtt{c} = inv(C)}{\Gamma \vdash \mathtt{C}(:\mathtt{d})\ \mathtt{x}}\ \text{(T-Inv)}$$

$$\frac{\Gamma \vdash \mathtt{T}_0\ \mathtt{e} \quad fields(\mathtt{T}_0, \mathtt{o}) = \overline{\mathtt{U}}(:\overline{\mathtt{d}})\overline{\mathtt{f}} \quad (\mathtt{o}\text{ fresh for }\Gamma, \mathtt{d}_i)}{\Gamma \vdash \mathtt{U}_i(:\mathtt{T}_0\ \mathtt{o}; \mathtt{o.f}_i = \mathtt{self}, \mathtt{d}_i)\,\mathtt{e.f}_i}\ \text{(T-Field)} \qquad \Gamma \vdash \mathtt{T}\ (\mathtt{T})\mathtt{e}\ \text{(T-Cast)}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{T}_0\ \mathtt{e}_0, \overline{\mathtt{T}}\ \overline{\mathtt{e}} \quad mtype(\mathtt{T}_0, \mathtt{m}, \mathtt{o}) = \overline{\mathtt{Z}}\ \overline{\mathtt{z}} : \mathtt{c} \to \mathtt{S}(:\mathtt{d}) \\ \Gamma, \mathtt{T}_0\ \mathtt{o}, \overline{\mathtt{T}}\ \overline{\mathtt{z}} \vdash \mathtt{c}, \overline{\mathtt{T}} \sqsubseteq \overline{\mathtt{Z}}\ \ (\mathtt{o}\text{ fresh for }\Gamma, \mathtt{c}, \mathtt{d})\end{array}}{\Gamma \vdash \mathtt{S}(:\mathtt{T}_0\ \mathtt{o}; \overline{\mathtt{T}}\ \overline{\mathtt{z}}; \mathtt{d})\ \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})}\ \text{(T-Invk)} \qquad \frac{\begin{array}{l}\Gamma \vdash \overline{\mathtt{T}}\ \overline{\mathtt{e}} \quad fields(C) = \overline{\mathtt{Z}}\ \overline{\mathtt{f}} \quad \mathtt{c} = inv(\mathtt{C}) \\ \Gamma, \overline{\mathtt{T}}\ \overline{\mathtt{f}} \vdash \mathtt{c}[\overline{\mathtt{f}}/\mathtt{this.f}], \overline{\mathtt{T}} \sqsubseteq \overline{\mathtt{Z}}[\overline{\mathtt{f}}/\mathtt{this.f}]\end{array}}{\Gamma \vdash \mathtt{C}(:\overline{\mathtt{T}}\ \overline{\mathtt{f}}; \mathtt{self.f} = \overline{\mathtt{f}})\ \mathtt{new\ C}(\overline{\mathtt{e}})}\ \text{(T-New)}$$

Method and Class Typing.

$$\frac{\overline{\mathtt{T}}\ \overline{\mathtt{x}}, \mathtt{C\ this}, \mathtt{c} \vdash \mathtt{S\ e}, \mathtt{S} \sqsubseteq \mathtt{T}}{\mathtt{T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}} : \mathtt{c})\{\mathtt{return\ e;}\}\ \text{OK in C}} \qquad \frac{\overline{M}\ \text{OK in } C}{\mathtt{class\ C}(\overline{\mathtt{T}}\ \overline{\mathtt{f}} : \mathtt{c})\ \mathtt{extends\ D}(:\mathtt{d})\ \{\overline{\mathtt{M}}\}\ \text{OK}}$$

Figure 2. Constrained FJ

Computation:

$$\frac{fields(C) = \overline{C}\ \overline{f}}{(\mathtt{new\ C}(\overline{\mathtt{e}})).\mathtt{f}_i \longrightarrow \mathtt{e}_i}\ \text{(R-FIELD)}$$

$$\frac{mbody(\mathtt{m}, \mathtt{C}) = \overline{x}.\overline{e}_0}{(\mathtt{new\ C}(\overline{\mathtt{e}})).\mathtt{m}(\overline{\mathtt{d}}) \longrightarrow [\overline{d}/\overline{x}, \mathtt{new}\ C(\overline{e})/\mathtt{this}]\mathtt{e}_0}\ \text{(R-INVK)}$$

$$\frac{\vdash C \sqsubseteq D[\mathtt{new}\ C(\overline{\mathtt{d}})/\mathtt{self}]}{(\mathtt{D})(\mathtt{new\ C}(\overline{\mathtt{d}})) \longrightarrow \mathtt{new}\ C(\overline{\mathtt{d}})}\ \text{(R-CAST)}$$

Congruence:

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{e}_0.\mathtt{f} \longrightarrow \mathtt{e}_0{}'.\mathtt{f}}\ \text{(RC-FIELD)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{e.m}(\overline{\mathtt{e}}) \longrightarrow \mathtt{e}_0{}'.\mathtt{m}(\overline{\mathtt{e}})}\ \text{(RC-INVK-RECV)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{e.m}(\ldots, \mathtt{e}_i, \ldots) \longrightarrow \mathtt{e}_0.\mathtt{m}(\ldots, \mathtt{e}_i', \ldots)}\ \text{(RC-INVK-ARG)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{new\ C}(\ldots, \mathtt{e}_i, \ldots) \longrightarrow \mathtt{new\ C}(\ldots, \mathtt{e}_i', \ldots)}\ \text{(RC-NEW-ARG)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{(\mathtt{C})\mathtt{e}_0 \longrightarrow (\mathtt{C})\mathtt{e}_0{}'}\ \text{(RC-CAST)}$$

Figure 3. Reduction rules for Constrained FJ

```
class Matrix(int m, int n) {
  Matrix(m,a.n) mul(Matrix(:m=this.n) a) {...}
  ...
}
```

Constraints naturally permit the expression of existential types:

```
class List(int length) {
List(:self.length < length) filter(Comparator k) { ...}
...
}
```

5.1 Binary methods

The binary method problem [3].

```
interface Set {
    Set(:class == this.class)
        union(Set(:class == this.class) s);
    boolean superSetOf(Set(:class == this.class) s);
}

class IntSet implements Set {
    long bits;

    Set(:class == this.class)
        union(Set(:class == this.class) s) {

        Set(:class == this.class) r = new IntSet();
        // need to be able to conclud that r : IntSet
        r.bits = this.bits | s.bits;
        return r;
    }
    boolean superSetOf(Set(:self.class == this.class) s) {
        return (s.bits & ~bits) == 0;
    }
}
```

5.2 AVL trees

```
class AVLList(int(:self >= 0) height) {...}
class Leaf(Object key) extends AVLList(0) {...}
class Node(Object key, AVLList l, AVLList r
           : int d=l.height-r.height; -1 <= d, d <= 1)
    extends AVLList(max(l.height,r.height)+1){..}
```

5.3 Red–black trees

Red/black trees may be modeled similarly. Such trees have the invariant that (a) all leaves are black, (b) each non-leaf node has the same number of black nodes on every path to a leaf (the black height), (c) the immediate children of every red node are black.

```
class Tree(int blackHeight) {...}
class Leaf extends Tree(0) { int value; ...}
class Node(boolean isBlack,
    Tree(:this.isBlack || isBlack) l,
    Tree(:this.isBlack || isBlack,
         blackHeight=l.blackHeight) r
) extends Tree(l.blackHeight+1) {
   int value; ...}
```

5.4 Bounds checks

```
class Array {
    T[] a;
    T get(int(:0 <= self && self < a.length) i) { return a[i]; }
    void set(int(:0 <= self && self < a.length) i, T v) { a[i] = v; }
}
```

5.5 Nullable types

Nullable types (T(:self != null))

5.6 A distributed binary tree

5.7 Places

5.8 *k*-dimensional regions

5.9 Point

5.10 Distribution

5.11 Arrays

[10]

Finally we can now define arrays. An array is built over a distribution and a base type.

5.12 Clocks

Clock types

5.13 Capabilities

Capablities (from Radha and Vijay's paper on neighborhoods)

5.14 Ownership types

Figure 5 demonstrates how ownership types [4] can be encoded in CFJ. The example is from XXX [**?**] and shows the code for a simple map implemented using a list of key–value pairs.

Cyclic: this.nodes.o == this

5.15 Discussion

Dependent types are of use in annotations [16].

6. Implementation

The dependent type system is implemented in the X10 compiler [23], which is implemented as an extension of Java using the Polyglot compiler framework [15].

Polyglot implements a source-to-source base Java compiler that is extended to translate X10 to Java. For purposes of this paper, we ignore the additional statement and expression types introduced in X10 and treat the language as simply Java extended with constrained types.

Type-checking is implemented as two passes. The first pass performs type-checking using the base compiler implementation augmented with additional code for new statement and expression types introduced in X10. In this pass, expressions used in dependent types are type-checked using the non-dependent type system, however, no constraint solving is performed. The second pass generates and solves constraints via an ask–tell interface [22]. If constraints cannot be solved, an error is reported.

After constraint checking, the X10 code is translated to Java. The basic idea behind the translation is simple. Each dependent class is translated into a single class of the same name without dependent types). The explicit properties of the dependent class are translated into `public final` (instance) fields of the target class. A `property` statement in a constructor is translated to a sequence of assignments to initialize the property fields.

For each property, there is also a getter method in the class. Properties declared in interfaces are translated into getter method signatures. Subclasses implementing these interfaces thus provide the required properties by implementing the generated interfaces.

For the most part, constraints are simply erased from the generated code. However, dependent types may be used in casts and `instanceof` expressions. These are translated to Java in straightforward manner by evaluating the constraint with `self` bound to the expression being tested.

```
1  /**
2     A distributed binary tree.
3     @author Satish Chandra 4/6/2006
4     @author vj
5   */
6  //                              ____P0
7  //                             |     |
8  //                             |     |
9  //                           _P2  __P0
10 //                          |  | |   |
11 //                          |  | |   |
12 //                         P3 P2 P1 P0
13 //                          *  *  *  *
14 // Right child is always on the same place as its parent;
15 // left child is at a different place at the top few levels of the tree,
16 // but at the same place as its parent at the lower levels.
17
18 class Tree(localLeft: boolean,
19            left: nullable Tree(& localLeft => loc=here),
20            right: nullable Tree(& loc=here),
21            next: nullable Tree) extends Object {
22     def postOrder:Tree = {
23         val result:Tree = this;
24         if (right != null) {
25             val result:Tree = right.postOrder();
26             right.next = this;
27             if (left != null) return left.postOrder(tt);
28         } else if (left != null) return left.postOrder(tt);
29         this
30     }
31     def postOrder(rest: Tree):Tree = {
32         this.next = rest;
33         postOrder
34     }
35     def sum:int = size + (right==null => 0 : right.sum()) + (left==null => 0 : left.sum)
36 }
37 value TreeMaker {
38     // Create a binary tree on span places.
39     def build(count:int, span:int): nullable Tree(& localLeft==(span/2==0)) = {
40         if (count == 0) return null;
41         {val ll:boolean = (span/2==0);
42          new Tree(ll,  eval(ll => here : place.places(here.id+span/2)){build(count/2, span/2)},
43            build(count/2, span/2),count)}
44     }
45 }
```

Figure 4. A distributed binary tree

```
⟦e instanceof C(:c)⟧ =
  new Object() {
    boolean check(Object o) {
      if (o instanceof C) {
        C self = (C) o;
        return ⟦c⟧;
      }
      return false;
    }
  }.check(⟦e⟧)
```

```
⟦(C(:c) e⟧ =
  new Object() {
    C cast(C self) {
      if (⟦c⟧)
        return self;
      throw new ClassCastException();
    }
  }.cast((C) ⟦e⟧)
```

7. Future work

State-dependent constrained types
Use of dependent types for optimization
Constraints on control-flow
Type inference

```
1  class Owned(Object o) {
2    Owned(:self.o == o)(final Object o) {
3      property(o);
4    }
5  }
6
7  class Map(Object ko, Object vo) extends Owned {
8    final Vector(:self.o == this && self.vo == this) nodes;
9
10   Map(:self.o == o && self.vo == vo && self.ko == ko)
11      (final Object o, final Object ko, final Object vo) {
12     super(o);
13     property(ko, vo);
14   }
15
16   void put(final Owned(:self.o == this.ko) key,
17            final Owned(:self.o == this.vo) value) {
18     nodes.add(new Node(this, key, value));
19   }
20
21   nullable<Owned(:self.o == this.vo)> get(final Owned(:self.o == this.ko) key) {
22     final Iterator(:self.o == this && self.vo == this) i = nodes.iterator();
23     while (i.hasNext()) {
24       Node(:self.o == this && self.map == this) n =
25         (Node(:self.o == this && self.map == this)) i.next();
26
27       if (n.key.equals(key)) {
28         return n.value;
29       }
30     }
31     return null;
32   }
33 }
34
35 class Node(Map map) extends Owned {
36   Owned(:self.o == this.map.ko) key;
37   Owned(:self.o == this.map.vo) value;
38
39   public Node(:self.o == self.map && self.map == m)
40     (final Map m, final Owned(:self.o == m.ko) key, final Owned(:self.o == m.vo) value) {
41     super(m);
42     property(m);
43     this.key = key;
44     this.value = value;
45   }
46 }
47
48 class Vector(java.lang.Object vo) extends Owned {
49   Vector(:self.o == o && self.vo == vo)(final Object o, final Object vo) {
50     super(o);
51     property(vo);
52   }
53
54   void add(final Owned(:self.o == this.vo) x) { ... }
55
56   Iterator(:self.o == this.o && self.vo == this.vo) iterator() {
57     return new Iterator(o, vo);
58   }
59 }
60
61
62 class Iterator(Object vo) extends Owned {
63   Iterator(:self.o == o && self.vo == vo)(final Object o, final Object vo) {
64     super(o);
65     property(vo);
66   }
67
68   boolean hasNext() { ... }
69   Owned(:self.o == this.vo) next() { ... }
70 }
```


Figure 5. Ownership types

8. Conclusions

We have presented a simple design for dependent types in Java-like languages. The design considerably enriches the space of (mostly) statically checkable types expressible in the language. This is particularly important for data-structurs such as lists and arrays. We have shown a simple translation scheme for dependent types into an underlying language with `assert` and `assume` statements. The assert and assume statements generated by this translation have the important property of state invariance. This enables a very simple notion of simplification for such programs. A general constraint propagator can simplify programs by using ask and tell operations on the underlying constraint system. Assert statements are removed if they are entailed by the conjunction of `assume`s on each path to the statement.

Our treatment is parametric in that the underlying constraint system can vary. Indeed the constraint system is not required to be complete; any incompleteness results merely in certain asserts being relegated to runtime. Some of these asserts may throw runtime exceptions if they are violated.

In future work we plan to investigate optimizations (such as array bounds check elimination) enabled by dependent types. We also plan to pursue much richer constraint systems, e.g., those necessary to deal with regions, cyclic and block-cyclic distributions etc.

Acknowledgments

Igor Peshansky, Lex Spoon, Vincent Cave.

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

[1] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 31–41, June 1993.

[2] Gilad Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, October 2004.

[3] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, Fall 1995.

[4] D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *ECOOP*, 2001.

[5] Pavel Curtis. *Constrained Quantification in Polymorphic Type Analysis*. PhD thesis, Cornell University, February 1990.

[6] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 156–168, Austin, TX, October 1995.

[7] Manuel Fähndrich. *Bane: A library for scalable constraint-based program analysis*. PhD thesis, University of California Berkeley, 1999.

[8] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java, 2002.

[9] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In *European Symposium on Programming (ESOP)*, number 300 in LNCS, pages 94–114, March 1988.

[10] Christian Grothoff, Jens Palsberg, and Vijay Saraswat. Safe arrays via regions and dependent types. Technical Report RC23911, IBM T.J. Watson Research Center, 2006.

[11] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[12] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(1):1811–1841, November 1994.

[13] John C. Mitchell. Coercion and type inference. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages (POPL'84)*, pages 174–185, 1984.

[14] Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages*. Cambridge University Press, 1992.

[15] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, number 2622, pages 138–152, Warsaw, Poland, April 2003.

[16] Nathaniel Nystrom and Vijay Saraswat. An annotation and copmiler plugin system for X10. Technical Report RC24198, IBM T.J. Watson Research Center, 2007.

[17] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995.

[18] François Pottier. Simplifying subtyping constraints. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, volume 31, pages 122–133, 1996.

[19] François Pottier. A semi-syntactic soundness proof for HM(X). Technical Report RR 4150, INRIA, March 2001.

[20] François Pottier. Simplifying subtyping constraints, a theory. *Information and Computation*, 170(2):153–183, November 2001.

[21] François Pottier and Didier Rémy. *Advanced Topics in Types and Programming Languages*, chapter 10, The Essence of ML Type Inference. MIT Press, 2004.

[22] V. Saraswat. *Concurrent Constraint Programming*. Doctoral Dissertation Award and Logic Programming. MIT Press, 1993.

[23] V. Saraswat et al. Report on the programming language X10. Technical report, IBM T.J. Watson Research Center, 2006.

[24] Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2–3):197–226, December 1994.

[25] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)*, 1997.

[26] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *ACM/SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, 1997.

[27] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Third International Static Analysis Symposium (SAS)*, number 1145 in LNCS, pages 349–365, 1996.

[28] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227, San Antonio, TX, January 1999.