Constrained Types – Future Directions

Vijay Saraswat[1], David Cunningham[1], Liana Hadarean[2], Louis Mandel[3,4], Avraham Shinnar[1], and Olivier Tardieu[1]

[1] IBM TJ Watson Research Center
[2] New York University
[3] Laboratoire de Recherche en Informatique, Universite Paris-Sud 11
[4] Laboratoire dInformatique de l'Ecole Normale Superieure, INRIA

Abstract. The use of constraints in types is quite natural. Yet, integrating constraint based types into the heart of a modern, statically typed, object-oriented programming language is quite tricky. Over the last five years we have designed and implemented the *constrained types* framework [16, 23] in the programming language X10 [5]. In this paper we review the conceptual design, the practical implementation issues, and the many new questions that are raised. We expect the pursuit of these questions to be a profitable area of future work.

Keywords: Constraints, Programming Languages, Types, Constraint Programming, Constrained Types

1 Introduction

The use of constraints in types is quite natural.

Consider a high performance programming language (such as X10 [5]) intended for use in scale-out parallel computations. Given the centrality of arrays to parallel programming, it makes sense for such a language to support arrays of different ranks. For instance an array of rank 3 needs three indices to access an underlying member – hence it is clearly an error to use one index. It makes sense, therefore, to capture information about the rank of the array in the *type* of the variable, so it can be statically checked. Thus we should be able to write a type of the form `Array[Int]{self.rank==3}` to isolate those arrays of integers that are such that their rank takes on the value 3. Clearly, to write reusable code, we should be able to assert that arrray has rank `m` where `m` is some immutable variable in the environment. Thus we would be able to assert that a `copy` method takes an argument `a` of type `Array[Int]` and another argument `b` of the same shape, i.e. of type `Array[Int]{self.rank==a.rank}`. Thus types may be associated with *constraints* – boolean valued expressions in a fixed vocabulary of operations that refer to variables whose value is fixed but unknown at the time of processing. Type checking then involves checking entailment relations between some constraints: an expression of type `T=Array[Int]{c}` should be assignable to a variable of type `S=Array[Int]{d}` if and only if every value of type `T` is also of type `S`, i.e. if and only if `d` entails `c` as a constraint. Thus type-checking reduces to constraint-solving.

These simple intuitions lay behind the introduction of constrained types in X10 [16, 23]. However, modern statically type-checked, object-oriented languages such as X10 have sophisticated type systems at the heart of their design. Extending such a system with constraints to yield a language that is very powerful and at the same time usable has turned out to be a complicated task.

In this paper we relate our practical experience in designing and implementing constrained types in X10, and discuss many directions for future work that have opened up.

The rest of this paper is as follows. Section 2 discusses the basic design of constrained types. Section 3 highlights some of the power of constrained types by showing how ownership types can be built on top. Section 4 discusses several directions for work for constrained types.

In each section, as appropriate, we highlight research questions for further study.

2 Constrained types

The simplest example of an an X10 type is a class, struct or interface name, e.g. `String`. X10 also permits *generic* types, i.e. types such as `Array[Int]` that take types as arguments.

A *constrained type* is of the form `T{c}` where `T` is a type, and `c` is a *constraint*, defined over an underlying constraint system $\mathcal{C}$. For now we focus on the constraint system implemented in X10 2.2.3 [5]. This permits the expression of constraints of the form `t==t` and `t !=t` (and their conjunctions), where the constraint terms `t` are built from (immutable) variables `v` or through the selection of (immutable) fields, `t.f`. In `c` the special variable `self` may be used to represent the object whose type is being defined[5] For example:

- `Int{self==0}` is the type of the constant `0` (one reads it as: the type of all `Int`s that are equal to `0`)
- `Score{self !=null}` is the type of all `Score` (non-null) objects
- `Matrix[T]{self.I==self.J}` is the type of all square matrices over `T`
- `Matrix[T]{self.I==a.I,self.J==b.J}` is the type of all matrices whose `I`th dimension is the same as the `I`th dimension of `a`, and whose `J`th dimension is the same as the `j`th dimension of `b`. Here, `a` and `b` must name (mmutable) local variables visible at this point in the code.

Constrained types can be used wherever types can be used. In a language such as X10, a strongly typed, object-oriented programming language in the tradition of Java, this means constrained types are integrated in an absolutely fundamental place in the language design. Constrained types can be used as types of method parameters, local variables, fields, as return types of methods, as supertypes in an `extends` clause, and as an interface type in an `implements` clause, in `instanceof` tests, and in casts (in `as` operations).

For instance, one can declare the signature of a matrix multiply method so that it captures invariants about the shape of the matrices involved:

[5] Thus one should think of `c` as implicitly defining the function `(self:T):Boolean=>c` that picks out a subset of `T`.

```
class Matrix[T](I:Int,J:Int) {
  ...
  def mult(a:Matrix[T]{self.I==J}):Matrix[T]{self.I==I,self.J==a.J)
  {
    ...
  }
```

Additionally, X10 permits class constructors to specify a return type. Thus a particular constructor may specify that it produces values whose return type is stronger than just the type associated with the name of the class/struct of the constructor.

X10 also permits a constraint, the *class invariant*, to be associated with a class or struct. It is an error for the return type of a constructor to not entail the class invariant. Since a type can only refer to immutable fields, and all immutable fields have a value at the time the constructor returns, this is enough to guarantee that an object that is an instance of a class `C` satisfies the invariant for `C`.

Similary, X10 permits a constraint, the *method guard*, to be associated with a method definition. Invocations of this method can only succeed if the constraint is entailed at the site of the invocation (with the actuals substituted for the formals).

Constrained types can be verbose. X10 programmers may therefore define and use *typedefs*. For instance:

```
type Rail[T] = Array[T]{self.rank==1,self.zeroBased,self.rect};
```

permits the programmer to simply write `Rail[Int]` and have it stand for the type `array[Int]` with the restriction that its rank must be `1`, that the array must be defined over a region that starts with the index `0`, that the region must be rectangular (i.e. of the form `m..n`, for some integers `m` and `n`). Typedefs such as `Rail[T]` are used extensively in X10 code.

2.1 Type checking

In principle, types are checked at compile-time. This means the compiler must verify that if an expression `e` is to be assigned to a variable `v`, then the type of `e` is a *subtype* of the type of `v`.

The subtype relation on constrained types translates to an entailment relation on constraints: `T{c1}` is a subtype of `T{c2}` if `c1` entails `c2`. This means that to statically check programs involving constrained type a compiler needs access to a solver that can answer questions of entailment and consistency involving constraints generated from the program text. In these queries any program variable is represented as a symbolic variable with an unkown value (but of the type of the variable).

Only certain fields called *properties* can be accessed through the variable `self`. Properties are immutable instance fields of classes and structs that are declared in a special way. The X10 2.2 type system has the restriction that the types of properties have to be "simpler" than the class/struct on which they are defined, so that no cycles are allowed in the graph with classes/structs as nodes and an edge from `S` to `T` if `S` has a property of type `T`. This permits constraints to be solved efficiently.

Localized type inference In order to avoid having the programmer write explicit types everywhere, X10 supports localized type-inference. In an immutable variable declaration, `val x=e;`, type of the variable `x` may be elided, it is inferred to be the type of the initializer `e`. Types must always be provided for parameters of methods or functions, and for mutable variables. Similarly, the return type of methods may be elided. It is taken to be the computed upper bound of the types of expressions in `return` statements in the body of the method. The computed upper bound of a set of types `T{c1}`, . . . , `T{cn}` is the strongest type that entailed by each of `c1`, . . . , `cn`.

Localized type inference often infers surprisingly precise types for variables. For instance given the code

```
class List(n:Int) {
    ...
    def rev():List{self.n==this.n}=...;
}

def m() {
  val l = new List(10);
   val x = l.rev().n;
   Console.OUT.println("x=" + x);
}
```

the compiler will actually infer that `x` is of type `Intself==l.n`. Therefore, it is advisable for programmers to omit the types of initialized immutable variables, and let the compiler propagate the information.

However, we found that surprisingly programmers sometimes wanted to specify explicit type information (e.g. to help with readability of the code). Therefore we have introduced a *partial type specification* construct in X10. The programmer can write the code as:

```
val x <: Int = l.rev().n;
```

The type of `x` will be inferred as usual by the compiler. However the compiler now also has the obligation to check that this type is at least as strong as the bound provided by the programmer (viz, `Int`).

Thus the programmer can get both the benefits of precise type tracking in the compiler and the benefit of actual textual representation of (an approximation of) the type.

Dynamic type-checking There are occasions in which it makes sense for the compiler to generate dynamic tests for constrained types instead of static checks. It may be the case that the programmer is simply prototyping code and does not wish to provide extra book-keeping in terms of constraints on all types.

Consider for example:

```
public static def main(args:Array[String]) {
    val N = args.size > 1 ? Int.parseInt(args(0)): 10;
    Console.OUT.println("fib("+N+ "=  " + fib(N));
}
```

Here the programmer implicitly assumes that `args` is an array of rank `1`, and would therefore be surprised to find a compiler error complaining that `args(0)` is not well typed.[6]

We have found that programmers new to X10 (e.g. Java programmers) are constantly surprised by this. This experience encouraged us to introduce a *dynamic checking* feature, and turn it on by default. Under dynamic checking, if a compilation were to fail because a value of type `T{c}` could not be established to be of type `T{d}`, but it could possibly be of this type (i.e. `c` and `d` are mutually consistent) then code is generated to check dynamically that this value is of type `T{d}`.[7] At the end of compilation, the compiler prints out how many such dynamic checks it inserted. If the programmer turns on a verbose check compiler flag, then the compiler prints out what condition it was not able to establish. The programmer can set a compiler flag to turn static checking on.

Thus the methodology we suggest to new X10 programmers is: compile your code using dynamic checking (the default settings on the compiler support this) and fix errors to get it running. When you are ready to performance tune your program, ensure that you remove the dynamic checks by strengthening types in the program as necessary. Verify by turning static checking on.

3 Application: Ownership Types as Constrained Types

Contrained types are a very powerful extension of the Java type system. Several *ad hoc* type systems introduced in the literature for object oriented languages can be described as variants of constrained types.

Here we ilustrate the basic methodology with *ownership types* [6, 7]. These type were introduced to capture some reasoning about relationships between objects on the heap.

Our basic approach is to change the root of the object hierarchy to include an `owner` property. At runtime the `owner` property can either hold the value `null` meaning that there is no owner, or some other object. Since properties are immutable, the owner must have existed before the object in question was constructed, which forces the graph of object ownership to be acyclic. Since each object has at most one owner, the shape of the graph is a forest (with each root having `owner==null`. Using a "ghost" property allows the owner property to be elided at runtime, but this means dynamic casts may not involve ownership types since the required information is no longer available.

Statically, various forms of ownership types can be written. Table 1 compares some X10 types with equivalents in other system (blank means the type is not expressible).

In addition, guards on method declarations can mention object owners, e.g., to say that two parameters `p1` and `p2` have the same owner, while not specifying what the owner actually is. This would be written with the following guard annotation: `{p1.owner == p2.owner}`

[6] The array library is written with so that the array index operation that takes one argument can be called only on an array whose static type asserts that it has rank `1`. This way no code needs to be generated to dynamically check this property.

[7] If it is known that the check will always fail at run-time, the compiler will still fail the code at compile-time.

X10	Ownership Types	Universe Types	informal description
`C`	`C<_>`	`any C`	Owner is unknown
`C{owner==this}`	`C<this>`	`rep C`	Owner is me
`C{owner==this.owner}`	`C<x>`	`peer C`	Same owner as me
`C{owner==tmp}`	`C<tmp>`		Owner is some final variable in scope
`C{owner!=null}`			`Object` is not the root of a hierarchy
`List[C{owner==this}]`	`List<C<this>>`	`List<rep C>`	List of objects owned by me

Table 1. X10 ownership types and their equivalents

Here is an example of a simple linked list using constraint-based ownership types in X10:

```
class OwnedObject(owner:Any) {
    public def this (o:Any) : OwnedObject{self.owner==o} {property(o);}
}

type Node[T](o:Any)=Node[T]{self.owner==o};
class Node[T] extends OwnedObject {
    // All nodes in the list have the same owner
    public var nxt: Node[T] (owner);
    cargo:T;
    // Creating a node in this
    public def this (o:Any, cargo:T) :Node[T](o) {
        super(o);
        this.cargo = cargo;
    }
}

class List[T] extends OwnedObject {
    public var begin: Node[T](this);
    public def this (owner:Any) {super(owner);}
    public def prepend(cargo: T) {
        val old_begin = begin;
        begin = new Node[T](this, cargo);
        begin.nxt = old_begin;
    }
    public def iterateOver (func:(T) => void) {
        for (var n:Node[T](this)=begin ; n!=null ; n=n.nxt) {
            func(n.cargo);
        }
    }
}

val mylist = new List[String](null);
mylist.prepend("foo");
mylist.prepend("bar");
mylist.iterateOver((o:String)=>{Console.OUT.println(o);});
```

Note that so far this is a "pure" ownership type system, in that the only static enforcement is that the ownership annotations are correct. This can be useful by itself but it is more useful to further restrict the program using extra restrictions on the types. This allows the guarantee of owners-as-dominators, a restriction on object aliasing that can help with static verification, program analysis, and accelerated garbage collection. In X10 one would require that all types are sufficiently constrained that they entail `self.owner` is reachable by chasing the `owner` properties from `this`. This rule would be implemented with a compiler plugin, but would benefit considerably through expressing owners with constraints, and using the compilers existing ability to reason about constrained types to implement its logic.

Another possible extension is for static reasoning about locks, e.g. static race safety [4, 8] or the implementation of atomic sections using locks. In such a system, the ownership relationship is assumed to be the same as the guarded-by relationship. I.e. the owner of a given object is locked to protect the shared mutable state of that object. Thus, the locking discipline is known by the type checker, which can then ensure that shared memory accesses are protected by appropriate locks (or insert appropriate locking).

Note also that some ownership types consider the owner field to be too heavy an overhead to pay on every object. Thus they erase ownership types at runtime. Using a "ghost" property, a constrained type system can also expose this same tradeoff.

Finally, multiple owner properties in an X10 class can be used to expose multiple simultaneous ownership hierarchies.

4 Directions for work on constrained types

With the basic type-checker integration framework in place in the compiler, the crucial next step is to integrate richer constraint systems into the type-checker.

4.1 Integration of richer constraint systems

The X10 compiler can be extended by plugging in a richer constraint systems. Constraint domains of practical interest include Boolean Algebras, Presburger Arithmetic, Tuples, Bit-vectors etc. Recall that verifying that `T{c1}` is a subtype of `T{c2}` translates into checking that `c1` entails `c2`. Therefore type-checking types with richer constraints requires the compiler to rely on solvers specialized for all of the above mentioned domains and their combinations.

The recent progress over the past several years in the field of Satisfiability Modulo Theories (SMT) makes the use of off-the-shelf SMT solvers an attractive option. The growing community around SMT solvers has lead to standardization around a common input format [2]. SMT solvers can efficiently decide the satisfiabilty of formulas over fragments of first-order logic plus certain standard theories, such as equality with uninterpreted functions, integer and real arithmetic, bit-vectors, arrays, inductive data-types and others. While SMT solvers rely on very effcient, usually complete decision procedures for deciding ground formulas, most SMT solvers are incomplete and do not guarantee termination in the quantified case.

We instrumented the X10 compiler to discharge the subtyping entailment checks in SMT-LIB v2.0 [2] format and we used various external SMT solvers (CVC4[1], CVC3[3], Z3[10]). We started with simple constraints involving only conjunctions of equalities and disequalities and field and method access. We modeled field and method access using uninterpreted function symbols, and object references as `Integer`s. For example the field dereference `a.f` where the field `f` is of `Boolean` type would be represented by the term `f(a)`, where `f` is an uninterpreted function symbol of type `Integer => Boolean`. We chose to model object references as `Integer`s to respect X10's language semantics with (Java-like) reference based equality. Using integers also guarantees that there can be an unbounded number of objects of each class. An alternative we considered was to model references as uninterpreted sorts. However, this requires adding axioms to ensure that the sort has an unbounded number of elements. Consider the following formula: $\forall x : S \forall y : S . x = y$. The formula is satisfiable in an interpretation where the sort S has only one element.

We found that the SMT solvers we tried (CVC4, CVC3, Z3) struggled with handling seemingly simple constraints involving quantified formulas over equality and uninterpred functions. The difficulty seems to stem from the nature of the type inference process in the X10 programming language which results in the generation of quantified constraints. Consider for example the typing rules for field instantation and method invocation (for a full set of typing rules see [16]):

$$\frac{\Gamma \vdash \mathtt{e:S} \qquad \Gamma, \mathtt{z:S} \vdash \mathtt{z\ has\ f:T}}{\Gamma \vdash \mathtt{e.f:T}\{\exists \mathtt{z:S.self == z.f}\}}\ (T-FIELD)$$

The T-FIELD rule uses existential quantification to project out the receiver `e` whose value is not known at runtime by replacing it by existentially quantified variable `z` of type `S`. The rule infers the type of `e.f` to be `T {∃ z:S. self == z.f }`. The T-INVK rule works similarly by projecting out the receiver as well as the arguments to the method call. Infering the most specific type requires existentially projecting out local variables. Thus the resulting constraints are of the form $\forall \bar{x} \exists \bar{y}$ `c`.

Assuming that the field `f` is of type `Integer`, checking that the resulting constraint is valid yields the following SMT formula $\forall self : Integer \exists z : Integer. self = f(z)$. The constraint essentially says that `f` is a surjective function on the `Integer`s. All the SMT solvers we have tried return unknown for this query. We believe this is due to the unbounded nature of the query.

Quantifier reasoning in SMT solvers has been a long-standing challenge. Most modern solvers employ one of the two techniques to reason about quantifiers. To prove the formula unsatisfiable they use heuristic instantiation (instantiating universal quantifiers to find a counter-example [9]). To prove the formula satifisable solvers attempt to construct an interpretation that satisifes the formula [11] (in a manner akin to model checking). The first technique would not apply to the above formula as it is satisfiable. The second technique fails because the interpretation of f required to satisfy the formula requires knowing the value of an infinite sets of points of f.

Note that in the special case when `c` is a conjunction of equalities and disequalities, the validity of the formula $\forall \bar{x} \exists \bar{y}$ `c` can easily be checked using a congruence closure algorithm (this is exactly what the current implementation of X10 does).

Although SMT solvers have been used as backends to extended static checking tools, this issue did not seem to arise before. We believe this is due to the fact the constraints we generate come from type inference and not type checking. Future research opportunities include: (i) optimizing SMT solvers for these type of constraints generated as part of the type inference process, perhaps leading to new quantifier decision procedures optimized to handle such constraints; (ii) using domain specific knowledge to simplify the constraints and eliminate the existential quantifiers (if possible) before discharging them to an external solver.

Question: Can SMT solvers be used effectively in X10 type checking? If not, how should they be extended to deal with X10 type checking?

4.2 Type inference

The X10 type checker even extended to do return type inference, or inference of type parameters implements a form of "forward" reasoning. For instance, the validity of a method call depends on the argument types not the other way around. But often the opposite information flow would make sense as well, which we now discuss.

Let's start with a flawed method declaration.

```
def arrayRead(a:Array, i:Int) = a(i);
```

In this example, the array access `a(i)` does not type check because the operator `(Int)` on array is only applicable to arrays of rank `1`. The following is correct.

```
def arrayRead(a:Array, i:Int){a.rank==1} = a(i);
```

While the X10 compiler is perfectly capable of rejecting the first method declaration, from a practical standpoint, this might not be the right decision. Indeed, the programmer may have intended to call this method only with arrays of rank `1`, even if s/he failed to capture this intent with the right guard. To support such use, the X10 compiler supports a compilation mode where, instead of rejecting the program up front, it inserts a runtime check (cast) to ensure the array has rank `1`, thus preserving type soundness (see Section 2.1)

However an ever better fix would be to infer the missing guard `a.rank==1` and check that the program augmented with this constraint type checks (possibly inferring futher constraints in cascade). In essence rather than checking entailment relations—such as those generated by a method call—we can take advantage of these entailment relations to generate extra constraints for the enclosing method —when an entailment relation does not hold for the source program as is.

The logical foundation for such "backward" information flow is *constraint abduction* [15, 14, 12, 13]. Given formulas A and C, find formula B such that $A, B \vdash C$. Here A represents what we already know from the context, C what we need for the program artifact to type check and B is the additional constraint we are looking for.

In the above example A is the empty formula and C is `a.rank==1` so there is an obvious ideal solution where B is identical to C. Abduction in general is a subtle operation, and such constraint inference has limits. For instance, consider program:

```
def assertIsZero(v:Int){v==0} {}

def m(b:Boolean, x:Int, y:Int) {
  val z = b ? x : y;
  assertIsZero(z);
}
```

Here we could infer the guard `b==true,x==0` or `b==false,y==0` or else `x==0,y==0` hence probably should do neither (in the absence of additional information).

An interesting special case of constraint inference is when we can infer, say, the constraint `c` in type `T{c}` such that `T{c}` is a singleton type. Consider for example:

```
def dup(a:Rail[Int], b:Rail[Int]) {
   copy(a,b);
}
abstract def copy(a:Rail[Int], b:Rail[Int]{self.length==a.length});
```

In this example we can infer the guard `v==0` hence we can derive not only the type of `v` from the code, but also its value. It therefore makes sense to extend the language with *holes* and use constraint inference to infer the missing values.

```
type List(n:Int)=List{self.n==n};
class List(n:Int) {
    def dup(L:List(n)):void {
        ...
        }
}
def m(l:List) {
   val x <:Int = ?;
   val y = new List(x); //2
   l.dup(y); //3
   return y;
}
```

From line 2 the compiler will infer the type of `y` is `List{self.n==x}`. From line 3 the compiler will realize that for the program to type-check, it must be the case that the constraint `y.n==l.n` must be entailed. In order to ensure that the program type-checks, the compiler will then abduce the constraint codex==l.n.[8] Thus the compiler has actually inferred an initializer for `x`, *viz.* `l.len`.

Thus the constraint inference engine may be used as a synthesis tool to generate fragments of code left unspecified, that is, support program *sketching* [21, 22, 20]. Note that unlike sketching, the programmer does not have to provide a separate specification for the hole. The only information used by the value synthesis algorithm is that the program is intended to correctly type-check.

It is straightforward to add *implicit* annotations on top of such a mechanism (cf [17]).

Question: How does abductive constraint inference for constrained types relate to Scala's support for implicit types?

[8] This in conjunction with `y.n==x` entails `y.n==l.n`.

4.3 Additional questions

Declarative debugging of type checking failures X10 standard libraries (e.g. for arrays, and their associated data-structures, regions, points, distributions) extensively use constrained types. In case of a type-check error induced by a failure of constraint entailment, the X10 compiler needs to provide some indication of why the type-check failed, but not necessarily overwhelm the programmer with unrelated constraint information.

This problem is complicated by the pervasive use of type inference in X10. The reason that a type check fails at a particular point in the body of the method may have to do with incomplete information associated with the type of any of the variables visible at that point. Each of these variables may have had their type inferred based on initialization expressions whose type may be erroneous because the types of the method calls they contain may be errorneous (because they were explicitly specified or inferred to be less precise than desired).

Question: Is it possible to devise a declarative debugging methodology [19] to help the programmer debug a type checking problem?

4.4 Flow sensitive types

The X10 compiler does not perform any flow sensitive computation of the type context. That is, the compiler is unaware of the semantics of conditionals. If the test of an if then else statement implies some constraint about the immutable variables visible at that point in the code, then it would be sound for the compiler to assume this constraint when inferring types of variables in the then part of the code (and conversely for the else part).

Liquid Types ([18]) offers an interesting approach to combine flow sensitive dependent types with predicate abstraction.

Question: How does one develop the ideas of Liquid Types within the X10 type system?

4.5 Constraint-based type-state

Strom and Yemini developed the idea of *type-state*, i.e. heap dependent types. Consider a file for instance. Certain operations on a file are available only when the file has been opened. Many operations are not available when the file has been closed. Thus changes to the heap can affect the operations available on an object.

Question: How does one develop heap-sensitive constrained types?

5 Conclusion

Acknowledgements. We gratefully acknowledge extensive discussions with past and current collaborators on constraint-based types – Nathaniel Nystrom, Igor Peshansky, David Grove, Joel Galenson, Vijay Ganesh.

References

1. C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification*, pages 171–177. Springer, 2011.
2. C. Barrett, A. Stump, and C. Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, 2010.
3. C. Barrett and C. Tinelli. Cvc3. In *Computer Aided Verification*, pages 298–302. Springer, 2007.
4. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2002. ACM Press.
5. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, Oct. 2005.
6. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33 of *ACM SIGPLAN Notices*, pages 48–64, New York, Oct. 1998. ACM Press.
7. D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. Summers. Universe Types for Topology and Encapsulation. 5382:72–112, 2008.
8. D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *VAMP 07*, pages 20–51, September 2007.
9. L. De Moura and N. Bjørner. Efficient e-matching for smt solvers. *Automated Deduction–CADE-21*, pages 183–198, 2007.
10. L. De Moura and N. Bjrner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, page 337340, 2008.
11. Y. Ge and L. De Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification*, pages 306–320. Springer, 2009.
12. M. Maher. Heyting domains for constraint abduction. In *Proceedings of the 19th Australian joint conference on Artificial Intelligence: advances in Artificial Intelligence*, AI'06, pages 9–18, Berlin, Heidelberg, 2006. Springer-Verlag.
13. M. Maher and G. Huang. On computing constraint abduction answers. In *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR '08, pages 421–435, Berlin, Heidelberg, 2008. Springer-Verlag.
14. M. J. Maher. Abduction of linear arithmetic constraints. In *Proceedings of the 21st international conference on Logic Programming*, ICLP'05, pages 174–188, Berlin, Heidelberg, 2005. Springer-Verlag.
15. M. J. Maher. Herbrand constraint abduction. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*, pages 397–406. IEEE Computer Society, 2005.
16. N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. Constrained types for object-oriented languages. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 457–474, New York, NY, USA, 2008. ACM.
17. B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. *SIGPLAN Not.*, 45(10):341–360, 2010.
18. P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.

19. E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA, 1983.
20. A. Solar Lezama. Program synthesis by sketching. Technical Report UCB/EECS-2008-176, EECS Department, University of California, Berkeley, Dec 2008.
21. A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 281–294, New York, NY, USA, 2005. ACM.
22. A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 404–415, New York, NY, USA, 2006. ACM.
23. O. Tardieu, N. Nystrom, I. Peshansky, and V. Saraswat. Constrained Kinds. In *Proceedings of the 27th ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '12, New York, NY, USA, 2012. ACM.