Refinement Type Inference via Abstract Interpretation

Ranjit Jhala
UCSD
jhala@cs.ucsd.edu

Rupak Majumdar
UCLA
rupak@cs.ucla.edu

Andrey Rybalchenko
TUM
rybal@in.tum.de

Abstract

Refinement Types are a promising approach for checking behavioral properties of programs written using advanced language features like higher-order functions, parametric polymorphism and recursive datatypes. The main limitation of refinement type systems to date is the requirement that the programmer provides the types of all functions, after which the type system can *check* the types and hence, verify the program.

In this paper, we show how to automatically *infer* refinement types, using existing abstract interpretation tools for imperative programs. In particular, we demonstrate that the problem of refinement type inference can be reduced to that of computing invariants of simple, first-order imperative programs without recursive datatypes. As a result, our reduction shows that any of the wide variety of abstract interpretation techniques developed for imperative programs, such as polyhedra, counterexample guided predicate abstraction and refinement, or Craig interpolation, can be directly applied to verify behavioral properties of modern software in a fully automatic manner.

1. Introduction

Automatic verification of semantic properties of modern programming languages is an important step toward reliable software systems. For higher-order programming languages with inductive datatypes or polymorphic instantiation, the main verification tool has been type systems, which traditionally capture only coarse data-type properties (such as `int`s are only added to `int`s), and require the programmer to explicitly annotate program invariants if more precise invariants about program computations are required.

For example, *refinement* type systems [33] associate data types with refinement predicates that capture richer properties of program computation. Using refinement types, one can state, for instance, that a program variable `xs` has the refinement type "non-zero integer," or that the integer division function has the refinement type $\text{int} \rightarrow \{\nu : \text{int} \mid \nu \neq 0\} \rightarrow \text{int}$ which states that the second argument must be non-zero. Then if a program with refinement type type-checks, one can assert that there is no division-by-zero error in the program. The idea of refinement types to express precise program invariants is well-known [3, 10, 12, 13, 27, 33]. However, in each of the above systems, the programmer must provide refinements for each program type, and the type system *checks* the provided type refinements for consistency. We believe that this burden of annotations has limited the widespread adoption of refinement type systems.

For *imperative* programming languages, algorithms based on abstract interpretation can be used to *automatically infer* many program invariants [2,6,16], thereby proving many semantic properties of practical interest. However, these tools do not precisely model modern programming features such as closures and higher-order functions or inductive datatypes, and so in practice, they are too imprecise when applied to higher-order programs.
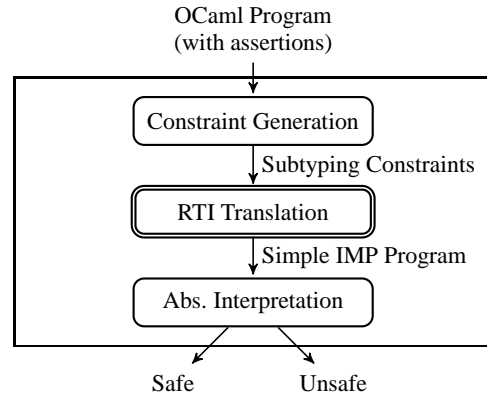

Figure 1. RTI algorithm.

In this paper, we present an algorithm to *automatically* verify properties of higher-order programs through refinement type inference (RTI) by combining refinement type systems for higher-order programs with invariant synthesis techniques for first-order programs. Our main technical contribution is a translation from type constraints derived from a refinement type system for higher-order programs to a first-order imperative program with assertions, such that the assertions hold in the first-order program iff there is a refinement type that makes the higher-order program type-check. Moreover, a suitable type refinement for the higher-order program can be constructed from the invariants of the first-order program. Thus, our algorithm replaces the manual annotation burden for refinement types with automatically constructed program invariants on the translated program, thus enabling fully automatic verification of programs written in modern languages.

The RTI algorithm (Figure 1) proceeds in three steps.

Step 1: Type-Constraint Generation. First, it performs Hindley-Milner type inference [11] to construct ML types for the program, and uses these types to generate *refinement templates*, *i.e.*, types in which *refinement variables* κ are used to represent the unknown refinement predicates. Then, the algorithm uses a standard syntax-directed procedure to generate subtyping constraints over the templates such that the program type checks (*i.e.*, is safe) if the subtyping constraints are satisfiable [3, 19, 29, 33].

Step 2: Translation. Second, it translates the set of type constraints to a *first-order, imperative program over base values* such that the type constraints are satisfiable if and only if the imperative program does not violate any assertions.

Step 3: Abstract Interpretation. Finally, an abstract interpretation technique for first order imperative programs is used to prove that the first order program is safe. The proof of safety produced by this analysis automatically translates to solutions to the refinement

type variables, thus generating refinement types for the original ML program.

The main contribution of this paper is the RTI translation algorithm. The advantage of the translation is that it allows one to apply any of the well-developed semantic imperative program analyses based on abstract interpretation (*e.g.,* polyhedra [9] and octagons [6], counterexample-guided predicate abstraction refinement (CEGAR) [2, 16], Craig interpolation [16, 22], constraint-based invariant generation [4, 30] random interpretation [15], *etc.*) to the verification of modern software with polymorphism, inductive datatypes, and higher-order functions. Instead of painstakingly reworking each semantic analysis for imperative programs to the higher order setting, possibly re-implementing them in the process, one can use our translation, and apply any existing analysis as is. In fact, using the translation, our implementation *directly* uses a CEGAR and interpolation based safety verification tool to verify properties of OCAML programs.

In essence, our algorithm separates syntactic reasoning about function calls and inductive data types (handled well by typing constraints) from semantic reasoning about data invariants (handled well by abstract domains). The translation from refinement type constraints to imperative programs in Step 2 is the key enabler. The translation, and the proof that the satisfiability of type constraints and safety of the translated program are equivalent, are based on the following observations.

The first observation is that refinement type variables κ define *relations* over the value being defined by the refinement type and the finitely many variables that are in-scope at the point where the type is defined. In the imperative program, each finite-arity relation can be encoded with a variable that encodes a relation. Each refinement type constraint can be encoded as a straight-line sequence that reads tuples from and writes tuples to the relation variables, and the set of constraints can be encoded as a non-terminating while-loop that in each iteration, non-deterministically executes one of the blocks. Thus, the problem of determining the existence of appropriate relations reduces to that of computing (overapproximations) of the set of tuples in each relation variable in the translated program (Theorem 1).

Our second observation is that if the translated program is in a special *read-write-once* form, where within each straight-line block a relation variable is read and written *at most once*, then one can replace all relation-valued variables with variables whose values range over tuples (Theorem 2). Moreover, we prove that we can, without affecting satisfiability, preprocess the refinement typing constraints so that the translated program is a read-write-once program (Theorem 3). Together, the observations yield a simple and direct translation from refinement type inference to simple imperative programs.

We have instantiated our algorithm in a verification tool for OCAML programs. Our implementation generates refinement type constraints using the algorithm of [29], and uses the ARMC [28] software model checker to verify the translated programs. This allows fully automatic verification of a set of OCAML benchmarks for which previous approaches either required manual annotations (either the refinement types [33] or their constituent predicates [29]), or an elaborate customization and adaptation of the counterexample-guided abstraction refinement paradigm [31]. Thus, we show, for the first time, how abstract interpretation can be lifted "as-is" to the practical refinement type inference for modern, higher-order languages.

While we have focused on the verification of functional programs, our approach is language independent, and requires only an appropriate refinement type system for the source language.

```
let rec iteri i xs f =
  match xs with
  | []      -> ()
  | x::xs' -> f i x;
              iteri (i+1) xs' f

let mask a xs =
  let g j y = a.(j) <- y && a.(j) in
  if Array.length a = List.length xs then
    iteri 0 xs g
```

Figure 2. ML Example

2. Overview

We begin with an example that illustrates how our refinement type inference (RTI) algorithm combines type constraints and abstract interpretation to automatically verify safety properties of *functional* ML programs with higher-order functions and recursive structures. We show that the combination of syntactic type constraints and semantic abstract interpretation enables the automatic verification of properties that are currently beyond the scope of either technique in isolation.

An ML Example. Figure 2(a) shows a simple ML program that updates an array a using the elements of the list xs. The program comprises two functions. The first is a higher-order list *indexed-iterator*, iteri, that takes as arguments a starting index i, a (polymorphic) list xs, and an iteration function f. The iterator goes over the elements of the list and invokes f on each element and the index corresponding to the element's position in the list. The second is a client, mask, of the iterator iteri that takes as input a boolean array a and a list of boolean values xs, and if the lengths match, calls the indexed iterator with an iteration function g that masks the jth element of the array.

Suppose that we wish to statically verify the safety of the array reads and writes in function g; that is to prove that whenever g is invoked, $0 \leq j < \text{len}(a)$. As this example combines higher-order functions, recursion, data-structures, and arithmetic constraints on array indices, it is difficult to analyze automatically using either existing type systems or abstract interpretation implementations in isolation. The former do not precisely handle arithmetic on indices, and the latter do not precisely handle higher-order functions and are often imprecise on data structures. We show how our RTI technique can automatically prove the correctness of this program.

Refinement Types. To verify the program, we compute program invariants that are expressed as *refinements* of ML types with predicates over program values [3, 19, 29]. The predicates are additional constraints that must be satisfied by every value of the type. A base value, say of type int, can be described by the refinement type $\{\nu : \text{int} \mid p\}$ where ν is a special *value variable* representing the type being defined, and p is a *refinement predicate* which constrains the range of ν to a subset of integers. For example, the type $\{\nu : \text{int} \mid 0 \leq \nu < \text{len}(a)\}$ denotes the set of integers c that are between 0 and the value of the expression $\text{len}(a)$. Thus, the unrefined type int abbreviates $\{\nu : \text{int} \mid true\}$, which does not constrain the set of integers. Base types can be combined to construct *dependent function types*, written $x : T_1 \rightarrow T_2$, where T_1 is the type of the domain, T_2 the type of the range, and where the name x for the formal parameter can appear in the refinement predicates in T_2. For example, the type

$$x : \{\nu : \text{int} \mid \nu \geq 0\} \rightarrow \{\nu : \text{int} \mid \nu = x + 1\}$$

is the type of a function which takes a non-negative integer parameter and returns an output which is one more than the input. In the following, we write τ for the type $\{\nu : \tau \mid true\}$. When ν and τ are clear from the context, we write $\{p\}$ for $\{\nu : \tau \mid p\}$.

Safety Specification. Refinement types can be used to *specify* safety properties by encoding pre-conditions into primitive operations of the language. For example, consider the array read $\mathtt{a.(j)}$ (resp. write $\mathtt{a.(j) \leftarrow e}$) in $\mathtt{g}$ which is an abbreviation for $\mathtt{get\ a\ j}$ (resp. $\mathtt{set\ a\ j\ e}$). By giving $\mathtt{get}$ and $\mathtt{set}$ the refinement types

$$\mathtt{a}\!:\!\alpha\,\mathtt{array} \to \{\nu\!:\!\mathtt{int} \mid 0 \leq \nu < \mathtt{len(a)}\} \to \alpha\,,$$
$$\mathtt{a}\!:\!\alpha\,\mathtt{array} \to \{\nu\!:\!\mathtt{int} \mid 0 \leq \nu < \mathtt{len(a)}\} \to \alpha \to \mathtt{unit}\,,$$

we can specify that in any program the array accesses must be within bounds. More generally, arbitrary safety properties can be specified by giving $\mathtt{assert}$ the appropriate refinement type [29].

Safety Verification. The ML type system is too imprecise to prove the safety of the array accesses in our example as it infers that $\mathtt{g}$ has type $\mathtt{j}\!:\!\mathtt{int} \to \mathtt{y}\!:\!\mathtt{bool} \to \mathtt{unit}$, *i.e.,* that $\mathtt{g}$ can be called with *any* integer $\mathtt{j}$. If the programmer manually provides the refinement types for all functions and polymorphic type instantiations, refinement-type checking [3, 12, 33] can be used to verify that the provided types were consistent and strong enough to prove safety. This is analogous to providing pre- and post-conditions and loop-invariants for verifying imperative programs. For our example, the refinement type system could check the program if the programmer provided the types:

$$
\begin{aligned}
\mathtt{iteri} ::\ & \mathtt{i}\!:\!\mathtt{int} \to \mathtt{xs}\!:\!\{\nu\!:\!\alpha\ \mathtt{list} \mid 0 \leq \mathtt{len}(\nu)\} \to \\
& (\mathtt{j}\!:\!\{\mathtt{i} \leq \nu < \mathtt{len(xs)}\} \to \alpha \to \mathtt{unit}) \to \mathtt{unit} \\
\mathtt{g} ::\ & \mathtt{j}\!:\!\{0 \leq \nu < \mathtt{len(a)}\} \to \mathtt{bool} \to \mathtt{unit}
\end{aligned}
$$

Here, we omitted refinement predicates that are equal to true, e.g., for $\mathtt{i}$ in the type of $\mathtt{iteri}$.

Automatic Verification via RTI. As even this simple example illustrates, the type annotation burden for verification is extremely high. Instead, we would like to verify the program without requiring the programmer to provide every refinement type. The RTI algorithm proceeds in three steps. First, we syntactically analyze the *source* program to generate subtyping constraints over refinement templates. Second, we translate the constraints into an equivalent simple imperative *target* program. Third, we semantically analyze the target program to determine whether it is safe, from which we conclude that the constraints are satisfiable and hence, the source program is safe. Next, we illustrate these steps using Figure 2 as the source program.

2.1 Step 1: Constraint Generation

In the first step, we generate a system of refinement type constraints for the source program [19, 29]. To do so, we (a) build templates that refine the ML types with refinement variables that stand for the unknown refinements, and (b) make a syntax-directed pass over the program to generate subtyping constraints that capture the flow of values. For the functions $\mathtt{iteri}$ and $\mathtt{g}$ from Figure 2, with the respective ML types

$$\mathtt{i}\!:\!\mathtt{int} \to \mathtt{xs}\!:\!\alpha\ \mathtt{list} \to (\mathtt{j}\!:\!\mathtt{int} \to \alpha \to \mathtt{unit}) \to \mathtt{unit}$$
$$\mathtt{j}\!:\!\mathtt{int} \to \mathtt{bool} \to \mathtt{unit}$$

we would generate the respective templates

$$\mathtt{i}\!:\!\mathtt{int} \to \mathtt{xs}\!:\!\{0 \leq \mathtt{len}(\nu)\} \to (\mathtt{j}\!:\!\{\kappa_1\} \to \alpha \to \mathtt{unit}) \to \mathtt{unit}$$
$$\mathtt{j}\!:\!\{\kappa_2\} \to \mathtt{bool} \to \mathtt{unit}$$

Notice that these templates simply refine the ML types with refinement variables κ_1, κ_2 that stand for the unknown refinements. For clarity of exposition, we have added the refinement *true* for some variables (*e.g.,* for the type α and $\mathtt{bool}$); our system would automatically infer the unknown refinements. We model the length of lists (resp. arrays) with an uninterpreted function $\mathtt{len}$ from the lists (resp. arrays) to integers, and (again, for brevity) add the refinement stating $\mathtt{xs}$ has a non-negative length in the type of $\mathtt{iteri}$.

After creating the templates, we make a syntax-directed pass over the program to generate constraints that capture relationships between refinement variables. There are two kinds of type constraints – *well-formedness* and *subtyping*.

Well-formedness Constraints capture scoping rules, and ensure that the refinement predicate for a type can only refer to variables that are in scope. Our example has two constraints:

$$\mathtt{i}\!:\!\mathtt{int}; \mathtt{xs}\!:\!\alpha\ \mathtt{list} \vdash \{\nu\!:\!\mathtt{int} \mid \kappa_1\} \qquad \text{(w1)}$$
$$\mathtt{a}\!:\!\mathtt{bool}\ \mathtt{array}; \mathtt{xs}\!:\!\alpha\ \mathtt{list} \vdash \{\nu\!:\!\mathtt{int} \mid \kappa_2\} \qquad \text{(w2)}$$

The first constraint states that κ_1, which represents the unknown refinement for the first parameter passsed to the higher-order iterator $\mathtt{iteri}$, can only refer to the two program variables that are in-scope at that point, namely $\mathtt{i}$ and $\mathtt{xs}$. Similarly, the second constraint states that κ_2, which refines the first argument of $\mathtt{g}$, can only refer to $\mathtt{a}$ and $\mathtt{xs}$, which are in scope where $\mathtt{g}$ is defined.

Subtyping Constraints reduce the flow of values within the program into subtyping relationships that must hold between the source and target of the flow. Each constraint is of the form

$$G \vdash T_1 <: T_2$$

where G is an *environment* comprising a sequence of type bindings, and T_1 and T_2 are refinement templates. The constraint intuitively states that under the environment G, the type T_1 must be a subtype of T_2. The subtyping constraints are generated syntactically from the code. First consider the function $\mathtt{iteri}$. The call to $\mathtt{f}$ generates

$$G \vdash \{\nu = \mathtt{i}\} <: \{\kappa_1\} \qquad \text{(c1)}$$

where the environment G comprises the bindings

$$
\begin{aligned}
G \doteq\ & \mathtt{i}\!:\!\{true\};\ \mathtt{xs}\!:\!\{0 \leq \mathtt{len}(\nu)\}; \\
& \mathtt{x}\!:\!\{true\};\ \mathtt{xs'}\!:\!\{0 \leq \mathtt{len}(\nu) = \mathtt{len(xs)} - 1\}
\end{aligned}
$$

the constraint ensures that at the callsite, the type of the actual is a subtype of the formal. The bindings in the environment are simply the refinement templates for the variables in scope at the point the value flow occurs. The type system yields the information that the length of $\mathtt{xs'}$ is one less than $\mathtt{xs}$ as the former is the tail of the latter [18, 33]. Similarly, the recursive call to $\mathtt{iteri}$ generates

$$
\begin{aligned}
G \vdash\ & \{\mathtt{j} : \kappa_1 :\to \alpha \to \mathtt{unit}\} <: \\
& \{(\mathtt{j}\!:\!\kappa_1 \to \alpha \to \mathtt{unit})[\mathtt{i} + 1/\mathtt{i}][\mathtt{xs'}/\mathtt{xs}]\}
\end{aligned}
$$

which states that type of the actual $\mathtt{f}$ is a subtype of the third formal parameter of $\mathtt{iteri}$ after applying substitutions $[\mathtt{i} + 1/\mathtt{i}]$ and $[\mathtt{xs'}/\mathtt{xs}]$ that capture the passing in of the actuals $\mathtt{i} + 1$ and $\mathtt{xs'}$ for the first two parameters respectively. By pushing the substitutions inside and applying the standard rules for function subtyping, this constraint simplifies to

$$G \vdash \{\kappa_1[\mathtt{i}/\mathtt{i} + 1][\mathtt{xs}/\mathtt{xs'}]\} <: \{\kappa_1\} \qquad \text{(c2)}$$

Next, consider the function $\mathtt{mask}$. The array accesses inside $\mathtt{g}$ generate the "bounds-check" constraint

$$G'; \mathtt{j}\!:\!\{\kappa_2\}; \mathtt{y}\!:\!\{true\} \vdash \{\nu = \mathtt{j}\} <: \{0 \leq \nu < \mathtt{len(a)}\} \quad \text{(c3)}$$

where $G' \doteq \mathtt{a}\!:\!\mathtt{bool}\ \mathtt{array}; \mathtt{xs}\!:\!\{0 \leq \mathtt{len}(\nu)\}$ has bindings for the other variables in scope. Finally, the flow due to the third parameter for the call to $\mathtt{iteri}$ yields

$$G'; \mathtt{len(a)} = \mathtt{len(xs)} \vdash \{\mathtt{j}\!:\!\kappa_2 \to \tau\} <: \{(\mathtt{j}\!:\!\kappa_1 \to \tau)[0/\mathtt{i}]\}$$

where for brevity we write τ for $\texttt{bool} \to \texttt{unit}$, and omit the trivial substitution $[\texttt{xs}/\texttt{xs}]$ due to the second parameter. The last conjunct in the environment captures the guard from the $\texttt{if}$ under whose auspices the call occurs. By pushing the substitutions inside and applying standard function subtyping, the above reduces to

$$G'; \texttt{len(a)} = \texttt{len(xs)} \vdash \{\kappa_1[0/\texttt{i}]\} <: \{\kappa_2\} \qquad \text{(c4)}$$

For brevity we omit trivial constraints like $\cdot \vdash \texttt{int} <: \texttt{int}$. If the set of constraints constructed above is satisfiable, then there is a valid refinement typing of the program [29], and hence the program is safe.

2.2 Step 2: Translation to Imperative Program

Determining the satisfiability of the constraints requires semantic analysis about program computations. In the second step, our key technical contribution, we show a translation that reduces the constraint satisfiability problem to checking the safety of a simple, imperative program. Our translation is based on two observations.

Refinements are Relations. The first observation is that type refinements are defined through *relations*: the set of values denoted by a refinement type $\{\nu : \tau \mid p\}$ where p refers to the program variables $\texttt{x}_1, \dots, \texttt{x}_n$ of the respective base types $\tau_1, \dots, \tau_n$ is equivalent to the set

$$\{t_0 \mid \exists(t_1, \dots, t_n) \text{ s.t. } (t_0, t_1, \dots, t_n) \in R_p \land \quad \} \\ t_1 = \texttt{x}_1 \land \dots t_n = \texttt{x}_n$$

where R_p is an $(n+1)$-ary relation in $\tau \times \tau_1 \times \dots \times \tau_n$ defined by p. For example, the set of values denoted by $\{\nu : \texttt{int} \mid \nu \le \texttt{i}\}$ is equivalent to the set:

$$\{t_0 \mid \exists t_1 \text{ s.t. } (t_0, t_1) \in R_\le \land t_1 = \texttt{i}\},$$

where $R_\le$ is the standard $\le$-ordering relation over the integers. In other words, each refinement variable κ can be seen as the projection on the first co-ordinate of a $(n+1)$-relation over the variables $(\nu, x_1, \dots, x_n)$, where $x_1, \dots, x_n$ are the variables in the well-formedness constraint for κ (*i.e.,* the variables in scope of κ). Thus, the problem of determining the satisfiability of the constraints is analogous to the problem of determining the existence of appropriate relations.

Relations are Records. The second observation is that the problem of finding appropriate relations can be reduced to the problem of analyzing a simple imperative program with variables ranging over relations. In the imperative program, each refinement variable, standing for an n-ary relation, is translated into a record variable with n-fields. Each subtyping constraint can be translated into a block of reads-from and writes-to the corresponding records. The set of all tuples that can be written into a given record on some execution of the program defines the corresponding relation. The entire program is an infinite loop, which in each iteration non-deterministically chooses a block of reads and writes defined by a constraint.

The arity of a relation, and hence the number of fields of the corresponding record, is determined by the well-formedness constraints. For example, the constraint (w1) specifies that κ_1 corresponds to a ternary relation, that is, a set of triples where the 0th element (corresponding to ν) is an integer, the 1st element (corresponding to $\texttt{i}$) is an integer, and the 2nd element (corresponding to $\texttt{xs}$) is a list. We encode this in the imperative program via a record variable κ_1 with three fields $\kappa_1.0$, $\kappa_1.1$ and $\kappa_1.2$.

Figure 3 shows the imperative program translated from the constraints for our running example. We use the subtyping constraints to define the flow of tuples into records. For example, consider constraint (c2) which is translated to the block marked $\texttt{/*c2*/}$. Each variable in the type environment is translated to a correspond-

ing variable in the program. The block has a sequence of assignments that define the environment variables. For example, we know $\texttt{i}$ has type $\texttt{int}$, so there is an assignment of an arbitrary integer to $\texttt{i}$. When there is a known refinement in the binding, the non-deterministic assignment is followed by an assume operation (a conditional) that establishes that the value assigned satisfied the given refinement. For example $\texttt{xs}$ gets assigned an arbitrary value, but then the assume establishes the fact that the length of $\texttt{xs}$ is non-negative. Similarly $\texttt{xs}'$ gets assigned an arbitrary value, that has non-negative length and whose length is 1 less than that of $\texttt{xs}$. The LHS of (c2) reads a tuple from κ_1 whose first and second fields are assumed to equal the $\texttt{i} + 1$ and $\texttt{xs}'$ respectively. Finally, the triple $(\nu, \texttt{i}, \texttt{xs})$ is written into the record κ_1 which is the RHS of (c2).

Next, consider the translated block for the bounds-check constraint (c3). Here, the translation is as before but the RHS is a known refinement predicate (that stipulates the integer be within bounds). In this case, instead of writing into the record that defines the RHS, the translation contains an assertion over the corresponding variables that ensures that the refinement predicate holds.

Relational vs. Imperative Semantics. There is a direct correspondence between the refinement-relations and the record variables when the translated program is interpreted under a Relational semantics, where (1) the records range over (initially empty) *sets of tuples*, (2) each write adds a new tuple to the record's set, and, (3) each read non-deterministically selects some tuple from the record's set. Under these semantics, we can show that the constraints are satisfiable iff the imperative program is safe (*i.e.,* no assert fails on any execution) (Theorem 1).

Unfortunately, these semantics preclude the direct application of mature invariant generation and safety verification techniques *e.g.,* those based on abstract interpretation or CEGAR-based software model checking, as those techniques do not deal well with set-valued variables. We would like to have an imperative semantics where each record contains a single value, the last tuple written to it. We show that there is a syntactic subclass of programs for which the two semantics coincide. That is, a program in the subclass is safe under the imperative semantics if and only if it is safe under the set-based semantics (Theorem 2). Furthermore, we show a technique that ensures that the translated program belongs to the subclass (Theorem 3).

The attractiveness of the translation is that the resulting programs fall in a particularly pleasant subclass of programs which do not have any advanced language features like higher-order functions, polymorphism, and recursive data structures, or variables over complex types such as sets, that are the bane of semantic analyses. Thus, the translation yields simple imperative programs to which a wide variety of semantic analyses directly apply.

2.3 Step 3: Invariant Generation.

Together these results imply that we can run off-the-shelf abstract interpretation and invariant generation tools on the translated program, and use the result of the analysis to determine whether the original ML program is typable.

For the translated program shown in Figure 3, the CEGAR-based software model checker ARMC [28] finds that the assertion is never violated, and computes the invariants:

$$\kappa_1.1 \le \kappa_1.0 \land \kappa_1.0 < \texttt{len}(\kappa_1.2)$$
$$0 \le \kappa_2.0 < \texttt{len}(\kappa_2.1)$$

which, when plugging in ν, $\texttt{i}$ and $\texttt{xs}$ for the $0^{th}, 1^{st}, 2^{nd}$ fields of κ_1 and ν, $\texttt{a}$ for the $0^{th}, 1^{st}$ fields of κ_2 respectively, yields the refinements

$$\kappa_1 \doteq \texttt{i} \le \nu < \texttt{len}(\texttt{xs}) \quad \kappa_2 \doteq 0 \le \nu < \texttt{len}(\texttt{a})$$

```
loop{   /*c1*/
        i ← nondet();
        xs ← nondet(); assume(0 ≤ len(xs));
        xs' ← nondet(); assume(0 ≤ len(xs') = len(xs) − 1);
        ν ← nondet(); assume(ν = i);
        κ₁ ← (ν, i, xs)

    ‖   /*c2*/
        i ← nondet();
        xs ← nondet(); assume(0 ≤ len(xs));
        xs' ← nondet(); assume(0 ≤ len(xs') = len(xs) − 1);
        (t₀, t₁, t₂) ← κ₁;
        assume(t₁ = i + 1);
        assume(t₂ = xs');
        ν ← t₀;
        κ₁ ← (ν, i, xs)

    ‖   /*c3*/
        a ← nondet();
        xs ← nondet(); assume(0 ≤ len(xs));
        (t₀, t₁, t₂) ← κ₂;
        j ← t₀;
        assert(0 ≤ j < len(a))

    ‖   /*c4*/
        a ← nondet();
        xs ← nondet(); assume(0 ≤ len(xs));
        assume(len(a) = len(xs));
        (t₀, t₁, t₂) ← κ₁;
        assume(t₁ = 0);
        assume(t₂ = xs);
        ν ← t₀;
        κ₂ ← (ν, a, xs)
}
```

Figure 3. Translated Program

which suffice to typecheck the original ML. Indeed, these predicates for κ_1 and κ_2 are easily shown to satisfy the constraints (c1), (c2), (c3), and (c4).

3. Constraints

We start by formalizing constraints over types refined with predicates. To this end, we make precise the notions of refinement predicates (Section 3.1), refinement types (Section 3.2), constraints over refinement types and the notion of satisfaction (Section 3.3).

A discussion of how such constraints can be generated in a syntax-guided manner from program source is outside the scope of this paper; we refer the reader to the large body of prior research that addresses this issue [3, 19, 29, 33].

Notation. We use uppercase (Z) to denote sets, lowercase z to denote elements, and $\langle Z \rangle$ for a sequence of elements in Z.

3.1 Refinement Logic

Figure 4 shows the syntax of refinement predicates. In our discussion, we restrict the predicate language to the typed quantifier-free logic of linear integer arithmetic and uninterpreted functions. However, it is straightforward to extend the logic to include other domains equipped with effective decision procedures and abstract interpreters.

Types and Environments. Our logic is equipped with a fixed set of *types* denoted τ, comprising the basic types `int` for *integer* values, `bool` for *boolean* values, and `ui`, a family of *uninterpreted types* that are used to encode complex source language types such as products, sums, polymorphic type variables, recursive types *etc.*. We assume there is a fixed set of uninterpreted functions. Each uninterpreted function `f` has a fixed type $\tau_{\mathtt{f}} \doteq \langle \tau_{\mathtt{f}}^{i} \rangle \to \tau_{\mathtt{f}}^{o}$. An *environment* is a sequence of variable-type bindings.

Expressions and Predicates. In our logic, *expressions* e comprise variables, linear arithmetic (*i.e.,* addition and multiplication by constants), and applications of uninterpreted functions `f`. Note that as is standard in semantic program analyses, complex operations like division or non-linear multiplication be modelled using uninterpreted functions. Finally, *predicates* comprise atomic comparisons of expressions, or boolean combinations of sub-predicates. We write *true* (resp. *false*) as abbreviations for $0 = 0$ (resp. $0 = 1$).

Well-formedness. We say that a predicate p is *well-formed* in an environment Γ if every variable appearing in p is bound in Γ and p is "type correct" in the environment Γ.

Validity. For each type τ, we write $\mathcal{U}(\tau)$ to denote the set of concrete values of τ. An *interpretation* σ is a map from variables x to concrete values, and functions `f` to maps from $\mathcal{U}(\langle \tau_{\mathtt{f}}^{i} \rangle)$ to $\mathcal{U}(\tau_{\mathtt{f}}^{o})$. We say that σ is *valid* under Γ if for each $x : \tau \in \Gamma$, we have $\sigma(x) \in \mathcal{U}(\tau)$. We say that a predicate p is *valid* in an environment Γ, if $\sigma(p)$ evaluates to *true* for every σ valid under Γ.

3.2 Refinement Types

Figure 4 shows the syntax of refinement types and environments.

Refinements. A *refinement* r is either a predicate p drawn from our logic, or a *refinement variable with pending substitutions* $\kappa[y_1/x_1] \ldots [y_n/x_n]$. Intuitively, the former represent *known* refinements (or invariants), while the latter represent the *unknown* invariants that hold of different program values. The notion of pending substitutions [1, 19] offers a flexible way of capturing the value flow that arises in the context of function parameter passing (in the functional setting), or assignment (in the imperative setting), even when the underlying invariants are unknown.

Refinement Types and Environments. A *refinement type* $\{\nu : \tau \mid r\}$ is a triple consisting of a *value variable* ν denoting the value being described by the refinement type, a type τ describing the underlying type of the value, and a refinement r. A *refinement environment* G is a sequence of refinement type bindings.

The value variables are special variables distinct from the program variables, and can occur inside the refinement predicates. Thus, intuitively, the refinement type describes the set of concrete values of the underlying type τ which additionally satisfy the refinement predicate. For example, the refinement type: $\{\nu : \mathtt{int} \mid \nu \neq 0\}$ describes the set of non-zero integers and, $\{\nu : \mathtt{int} \mid \nu = \mathtt{x} + \mathtt{y}\}$ describes the set of integers whose value equals the sum of the values of the (program) variables `x` and `y`.

Note that path-sensitive branch information can be captured by adding suitable bindings to the refinement environment. For example, the fact that some expression is only evaluated under the if-condition that $\mathtt{x} > 100$ can be captured in the environment via a refinement type binding $\mathtt{x}_b : \{\nu : \mathtt{bool} \mid \mathtt{x} > 100\}$.

3.3 Refinement Constraints and Solutions

Figure 4 shows the syntax of refinement constraints. Our refinement type system has two kinds of constraints.

Subtyping Constraints are of the form

$$G \vdash \{\nu : \tau \mid r_1\} <: \{\nu : \tau \mid r_2\}$$

Intuitively, a subtyping constraint states that when the program variables satisfy the invariants described in G, the set of values described by the refinement r_1 must be *subsumed by* the set of values described by the refinement type r_2.

Well-formedness Constraints are of the form $\Gamma \vdash \{\nu : \tau \mid r\}$. Intuitively, a well-formedness constraints states that the refinement r must be a well-typed predicate in the environment G extended with the binding $\nu : \tau$ for the value variable.

Embedding. To formalize the notions of constraint validity and satisfaction, we embed subtyping constraints into our logic. We define

the function $\mathsf{Emb}(\cdot)$ that maps refinement types, environments and subtyping constraints to predicates in our logic.

$$
\begin{aligned}
\mathsf{Emb}(\{\nu{:}\tau \mid p\}) &\doteq p \\
\mathsf{Emb}(x{:}T;G) &\doteq \mathsf{Emb}(T)[\nu/x] \wedge \mathsf{Emb}(G) \\
\mathsf{Emb}(\emptyset) &\doteq true \\
\mathsf{Emb}(G \vdash T_1 <: T_2) &\doteq \mathsf{Emb}(G) \Rightarrow \mathsf{Emb}(T_1) \Rightarrow \mathsf{Emb}(T_2)
\end{aligned}
$$

Similarly, we define the function $\mathsf{Shape}(\cdot)$ that maps refinement types and environments to types and environments in our logic.

$$
\begin{aligned}
\mathsf{Shape}(\{\nu{:}\tau \mid p\}) &\doteq \tau \\
\mathsf{Shape}(x{:}T;G) &\doteq x{:}\mathsf{Shape}(T);\mathsf{Shape}(G) \\
\mathsf{Shape}(\emptyset) &\doteq \emptyset
\end{aligned}
$$

Validity. A subtyping constraint $G \vdash T_1 <: T_2$ that does not contain refinement variables is *valid* if the predicate $\mathsf{Emb}(G \vdash T_1 <: T_2)$ is valid under environment $\mathsf{Shape}(G)$. A well-formedness constraint $\Gamma \vdash \{\nu{:}\tau \mid p\}$ that does not contain refinement variables is *valid* if the predicate p is well-formed in the environment Γ.

Relational Interpretations. We assume, without loss of generality, that each refinement variable κ is associated with a unique well-formedness constraint $x_1{:}\tau_1;\ldots;x_n{:}\tau_n \ \vdash \ \{\nu{:}\tau_0 \mid \kappa\}$ called the well-formedness constraint for κ. In this case, we say κ has *arity* $n+1$. Furthermore, we assume that wherever a κ of arity $n+1$ appears in a subtyping constraint, it appears with a sequence of n pending substitutions $[y_1/x_1]\ldots[y_n/x_n]$. This assumption is without loss of generality, as we can enforce it with trivial substitutions of the form $[x_i/x_i]$. A *relational interpretation* for κ of arity $n+1$, is an $(n+1)$-ary relation in $\mathcal{U}(\tau_0) \times \ldots \times \mathcal{U}(\tau_n)$. A *relational model* is a map from refinement variables κ to relational interpretations.

Constraint Satisfaction. A set of constraints C is *satisfiable* if for all interpretations for uninterpreted functions $\mathtt{f}$, there exists a relational model S such that, when each occurrence of a refinement type $\{\nu{:}\tau \mid \kappa[y_1/x_1]\ldots[y_n/x_n]\}$ in C is substituted with

$$\{\nu{:}\tau \mid \exists t_1,\ldots,t_n.S(\kappa)(\nu,t_1,\ldots,t_n) \wedge t_1 = y_1 \wedge \ldots t_n = y_n)\}$$

every subtyping constraint after the substitution is valid. In this case, we say that S is a *solution* for C.

4. Imperative Programs

RTI translates the satisfiability problem for refinement type constraints to the question of checking the safety of an imperative program in a simple imperative language IMP. In this section, we formalize the syntax of IMP programs and define the Relational semantics and the Imperative semantics.

4.1 Syntax

Figure 5 shows the syntax of IMP programs. An *instruction* (I) is a sequence of assignments, assumptions and assertions. A *program* (P) is an infinite loop over a block, whose body is a non-deterministic choice between a finite number of instructions $\mathtt{I}_1,\ldots,\mathtt{I}_n$. Next, we describe the different kinds of instructions. For ease of notation, we assume that there is only one base type τ, and let V denote the set of values of type τ.

Variables. IMP programs have two kinds of variables. (1) *base* variables, denoted by ν, x, y and t (and subscripted versions thereof), which range over values of type τ. (2) *relation* variables, denoted by κ, each of which have a fixed arity n and range over tuples of values or sets of n-tuples of values depending on the semantics.

Base Assignments. IMP programs have two kinds of assignments to base variables. Either (1) an expression over base variables (cf. Figure 4) is evaluated and assigned to the base variable, or,

τ	$::=$		**Types:**
	$\mid$	$\mathtt{int}$	base type of integers
	$\mid$	$\mathtt{bool}$	base type of booleans
	$\mid$	$\mathtt{ui}$	complex uninterpreted type
Γ	$::=$		**Environments:**
	$\mid$	$x{:}\tau;\Gamma$	binding
	$\mid$	$\emptyset$	empty
e	$::=$		**Expressions:**
	$\mid$	x	variable
	$\mid$	n	integer
	$\mid$	$e_1 + e_2$	addition
	$\mid$	$n \times e$	affine multiplication
	$\mid$	$\mathtt{f}(\langle e \rangle)$	function application
p	$::=$		**Predicates:**
	$\mid$	$e_1 \bowtie e_2$	comparison
	$\mid$	$\neg p$	negation
	$\mid$	$p_1 \wedge p_2$	conjunction
	$\mid$	$p_1 \Rightarrow p_2$	implication
r	$::=$		**Refinements:**
	$\mid$	p	predicate
	$\mid$	$\kappa[y_1/x_1]\ldots[y_n/x_n]$	ref. var. with substitutions
T	$::=$	$\{\nu{:}\tau \mid r\}$	**Refinement Types**
G	$::=$		**Refinement Environments:**
	$\mid$	$x{:}T;G$	binding
	$\mid$	$\emptyset$	empty
c	$::=$	$G \vdash T_1 <: T_2$	**Subtype Constraints**
w	$::=$	$\Gamma \vdash T$	**WF Constraints**

Figure 4. Predicates, Refinements and Constraints.

$\mathtt{I}$	$::=$		**Instructions:**
	$\mid$	$x \leftarrow e$	assign expr
	$\mid$	$x \leftarrow \mathtt{nondet}()$	havoc
	$\mid$	$(t_0,\ldots,t_n) \leftarrow \kappa$	get tuple
	$\mid$	$\kappa \leftarrow (x_0,\ldots,x_n)$	set tuple
	$\mid$	$\mathtt{assume}(p)$	assume
	$\mid$	$\mathtt{assert}(p)$	assert
	$\mid$	$\mathtt{I}_1; \mathtt{I}_2$	sequence
$\mathtt{P}$	$::=$	$\mathtt{loop}\{\mathtt{I}_1 [\![\ldots [\![\mathtt{I}_n\}$	Program

Figure 5. Imperative Programs: Syntax

(2) an arbitrary value of the appropriate base type is assigned to the base variable, *i.e.,* the variable is "havoc-ed" with a non-deterministically chosen value.

Tuple Assignments. The operations *get tuple* and *set tuple* respectively read a tuple from and write a tuple to a relation variable.

Assumes and Asserts. IMP programs have the standard assume and assert instructions using predicates over the base variables (cf. Figure 4). We write $\mathtt{skip}$ as an abbreviation for $\mathtt{assume}(0 = 0)$.

4.2 Relational Semantics

We define the Relational semantics as a state transition system. In this semantics, κ variables range over *sets of* tuples over V.

Relational States. A state $s^\sharp$ in the Relational semantics is either the special *error state* $\mathcal{E}$ or a map from program variables to values such that every base variable is mapped to a value in V, and every relation variable of arity n is mapped to a (possibly empty) set of tuples in V^n. Let $\Sigma^\sharp$ be the set of all Relational-program states.

For a state $s^\sharp$ which is not $\mathcal{E}$, variable x and value v we write $s^\sharp[x \mapsto v]$ for the map which maps x to v and every other key x' to $s^\sharp(x')$. We lift maps $s^\sharp$ from base variables to values to maps from expressions (and predicates) to values in in the natural way.

Initial State. The initial state $s_0^\sharp$ of an IMP program in the Relational semantics is a map in which every base variable is mapped to a fixed value from V, and every relation variable is mapped to the empty set.

Transition Relation. The transition relation is defined through a $\mathsf{Post}^\sharp$ operator, shown in Figure 6, which maps a state $s^\sharp$ and an instruction $\mathtt{I}$ to the *set* of states that the program can be in *after* executing the instruction from the state $s^\sharp$. We lift $\mathsf{Post}^\sharp$ to a set of states $\hat{\Sigma}^\sharp \subseteq \Sigma^\sharp$ in the natural way:

$$\mathsf{Post}^\sharp(\hat{\Sigma}^\sharp, \mathtt{I}) \doteq \bigcup \{\mathsf{Post}^\sharp(s^\sharp, \mathtt{I}) \mid s^\sharp \in \hat{\Sigma}^\sharp\}$$

Notice that the program halts if a get instruction is executed with an empty relation variable, or an $\mathtt{assume}(p)$ is executed in a state that does not satisfy p.

Safety. Let P be the program $\mathtt{loop}\{\mathtt{I}_1 \| \ldots \| \mathtt{I}_n\}$. The set of *Relational-reachable states* of P, denoted $\mathsf{Reach}^\sharp(\mathtt{P})$ is defined by induction as:

$$
\begin{aligned}
\mathsf{Reach}^\sharp(\mathtt{P}, 0) &\doteq \{s_0^\sharp\} \\
\mathsf{Reach}^\sharp(\mathtt{P}, m+1) &\doteq \bigcup \{\mathsf{Post}^\sharp(\mathsf{Reach}^\sharp(\mathtt{P}, m), \mathtt{I}_j) \mid 1 \leq j \leq n\} \\
\mathsf{Reach}^\sharp(\mathtt{P}) &\doteq \bigcup \{\mathsf{Reach}^\sharp(\mathtt{P}, m) \mid 0 \leq m\}
\end{aligned}
$$

A program P is *Relational-safe* if $\mathcal{E} \notin \mathsf{Reach}^\sharp(\mathtt{P})$.

4.3 Imperative Semantics

Next, we define the Imperative semantics, as a state transition system. In this semantics, κ variables κ range over tuples over V.

Imperative States. In the Imperative semantics, each state s is either the special *error state* $\mathcal{E}$ or a map from program variables to values such that every base variable is mapped to a value in V, and every relation variable of arity n is mapped either to a tuple in V^n or to the special *undefined* value $\bot$. Let Σ denote the set of all a Imperative-program states.

Initial State. The initial state s_0 of an IMP program in the Imperative semantics is a map in which every base variable is mapped to a fixed value from V, and every relation variable is mapped to $\bot$.

Transition Relation. The transition relation is defined using a Post operator, which is identical to $\mathsf{Post}^\sharp$ in the Relational semantics except for the tuple-get and tuple-set instructions. Figure 6 shows the operator Post for get and set operations. Again, Post is lifted to a set of states in the natural way. Notice that the program halts if a get instruction is executed with an *undefined* relation variable, or an $\mathtt{assume}(p)$ is executed in a state that does not satisfy p.

Safety. Let P be the program $\mathtt{loop}\{\mathtt{I}_1 \| \ldots \| \mathtt{I}_n\}$. The set of *Imperative-reachable states* of P, denoted $\mathsf{Reach}(\mathtt{P})$ is defined by induction as:

$$
\begin{aligned}
\mathsf{Reach}(\mathtt{P}, 0) &\doteq \{s_0\} \\
\mathsf{Reach}(\mathtt{P}, m+1) &\doteq \bigcup \{\mathsf{Post}(\mathsf{Reach}(\mathtt{P}, m), \mathtt{I}_j) \mid 1 \leq j \leq n\} \\
\mathsf{Reach}(\mathtt{P}) &\doteq \bigcup \{\mathsf{Reach}(\mathtt{P}, m) \mid 0 \leq m\}
\end{aligned}
$$

A program P is *Imperative-safe* if $\mathcal{E} \notin \mathsf{Reach}(\mathtt{P})$.

5. From Type Constraints to IMP Programs

In this section we formalize the translation from type constraints into IMP programs and prove that the constraints are satisfiable if and only if the translated program is safe.

Refinement Type Translation

$$
\begin{aligned}
[\![\{\nu \colon \tau \mid p\}]\!]_{get} &\doteq \quad \nu \leftarrow \mathtt{nondet}(); \\
& \qquad\quad \mathtt{assume}(p) \\[4pt]
[\![\{\nu \colon \tau \mid p\}]\!]_{set} &\doteq \quad \mathtt{assert}(p) \\[4pt]
[\![\{\nu \colon \tau \mid \kappa[y_1 \ldots y_n / x_1 \ldots x_n]\}]\!]_{get} &\doteq \quad (t_0, \ldots, t_n) \leftarrow \kappa; \\
& \qquad\quad \mathtt{assume}(y_1 = t_1); \\
& \qquad\qquad \vdots \\
& \qquad\quad \mathtt{assume}(y_n = t_n); \\
& \qquad\quad \nu \leftarrow t_0 \\[4pt]
[\![\{\nu \colon \tau \mid \kappa[y_1 \ldots y_n / x_1 \ldots x_n]\}]\!]_{set} &\doteq \quad \kappa \leftarrow (\nu, y_1, \ldots, y_n)
\end{aligned}
$$

Binding Translation

$$
\begin{aligned}
[\![x \colon T; G]\!] &\doteq \quad [\![\tau]\!]_{get};\ x \leftarrow \nu;\ [\![G]\!] \\[4pt]
[\![\cdot]\!] &\doteq \quad \mathtt{skip}
\end{aligned}
$$

Constraint Translation

$$[\![G \vdash T_1 <: T_2]\!] \quad \doteq \quad [\![G]\!];\ [\![T_1]\!]_{get};\ [\![T_2]\!]_{set}$$

Constraint Set Translation

$$[\![\{c_1, \ldots, c_n\}]\!] \quad \doteq \quad \mathtt{loop}\{[\![c_1]\!] \| \ldots \| [\![c_n]\!]\}$$

Figure 7. Translating Constraints to IMP Programs

5.1 Translation

Figure 7 formalizes the translation from (a set of) refinement type constraints C to an IMP program $[\![C]\!]$. We use the WF constraints to translate each relation variable κ of arity $n+1$ into a corresponding tuple variable κ of arity $n+1$.

The translation is syntax-driven. We translate each subtyping constraint $G \vdash T_1 <: T_2$ into a straight-line block of instructions with three parts: a sequence of instructions that establishes the environment bindings ($[\![G]\!]$), a sequence of instructions that "gets" the values corresponding to the LHS ($[\![T_1]\!]_{get}$) and a sequence of instructions that "sets" the (LHS) values into the appropriate RHS ($[\![T_2]\!]_{set}$). The translation for a set of constraints is an infinite loop that non-deterministically chooses among the blocks for each constraint.

Each environment binding gets translated as a "get". Bindings with unknown refinements are translated into tuple-get operations, followed by $\mathtt{assume}$ statements that establish the equalities corresponding to the pending substitutions. Bindings with known refinements are translated into non-deterministic assignments followed by a $\mathtt{assume}$ that enforces that the refinement holds on the non-deterministic value.

Each "set" operation to an unknown refinement is translated into a tuple-set instruction that writes the tuple corresponding to the pending substitutions into the translated tuple variable. Finally, each "set" operation corresponding to a known refinement is translated to an $\mathtt{assert}$ instruction; intuitively, in such constraints the RHS defines an upper bound on the set of values populating the type, and the $\mathtt{assert}$ serves to enforce the upper bound requirement in the translated program.

The correctness of the procedure is stated by the following theorem.

THEOREM 1. *C is satisfiable iff $[\![C]\!]$ is* Relational-safe.

The proof of this theorem follows from the properties of the following function α that maps a set $\hat{\Sigma}^\sharp \subseteq \Sigma^\sharp$ of Relational-states

Common Operations

$$\mathsf{Post}^\sharp(\mathcal{E}, \mathtt{I}) \doteq \{\mathcal{E}\}$$
$$\mathsf{Post}^\sharp(s^\sharp, \mathtt{I}_1; \mathtt{I}_2) \doteq \mathsf{Post}^\sharp(\mathsf{Post}^\sharp(s^\sharp, \mathtt{I}_1), \mathtt{I}_2)$$
$$\mathsf{Post}^\sharp(s^\sharp, x \leftarrow e) \doteq \{s^\sharp[x \mapsto s^\sharp(e)]\}$$
$$\mathsf{Post}^\sharp(s^\sharp, x \leftarrow \mathtt{nondet}()) \doteq \{s^\sharp[x \mapsto c] \mid c \in V\}$$
$$\mathsf{Post}^\sharp(s^\sharp, \mathtt{assume}(p)) \doteq \begin{cases} \{s^\sharp\} & \text{if } s^\sharp(p) = true \\ \emptyset & \text{otherwise} \end{cases}$$
$$\mathsf{Post}^\sharp(s^\sharp, \mathtt{assert}(p)) \doteq \begin{cases} \{s^\sharp\} & \text{if } s^\sharp(p) = true \\ \{\mathcal{E}\} & \text{otherwise} \end{cases}$$

Tuple Operations: Relational Semantics

$$\mathsf{Post}^\sharp(s^\sharp, (t_0, \ldots, t_n) \leftarrow \kappa) \doteq \{s^\sharp[t_0 \mapsto v_0] \ldots [t_n \mapsto v_n] \mid (v_0, \ldots, v_n) \in s^\sharp(\kappa)\}$$
$$\mathsf{Post}^\sharp(s^\sharp, \kappa \leftarrow (x_0, \ldots, x_n)) \doteq \{s^\sharp[\kappa \mapsto s^\sharp(\kappa) \cup \{(s^\sharp(x_0), \ldots, s^\sharp(x_n))\}]\}$$

Tuple Operations: Imperative Semantics

$$\mathsf{Post}(s, (t_0, \ldots, t_n) \leftarrow \kappa) \doteq \begin{cases} \{s[t_0 \mapsto v_0] \ldots [t_n \mapsto v_n]\} & \text{if } s(\kappa) = (v_0, \ldots, v_n) \\ \emptyset & \text{if } s(\kappa) = \bot \end{cases}$$
$$\mathsf{Post}(s, \kappa \leftarrow (x_0, \ldots, x_n)) \doteq \{s[\kappa \mapsto (s(x_0), \ldots, s(x_n))]\}$$

Figure 6. Relational and Imperative Semantics: Other cases of Post **identical to** $\mathsf{Post}^\sharp$

to constraint solutions:

$$\alpha(\hat{\Sigma}^\sharp) \doteq \lambda\kappa. \bigcup \{s^\sharp(\kappa) \mid s^\sharp \in \hat{\Sigma}^\sharp\}$$

The function α enjoys the following property, which can be proven by induction on the construction of $\mathsf{Reach}^\sharp$, that relates the satisfying solutions of the constraints to the Relational-reachable states of the translated program. Theorem 1 follows from the following observations. If S satisfies C then $\alpha(\mathsf{Reach}^\sharp(\llbracket C \rrbracket))(\kappa) \subseteq S(\kappa)$ for all κ. If $\mathcal{E} \notin \mathsf{Reach}^\sharp(\llbracket C \rrbracket)$ then $\alpha(\mathsf{Reach}^\sharp(\llbracket C \rrbracket))$ satisfies C.

5.2 Read-Write-Once Programs

At this point, via Theorem 1, we have reduced checking satisfiability of type constraints to the problem of verifying assertions of IMP programs under the (non-standard) Relational semantics. Unfortunately, under these semantics, the program contains variables (κ) which range over *sets* of tuples. This makes it inconvenient to directly apply abstract-interpretation based techniques for imperative programs which typically assume the (standard) Imperative semantics; each technique has to be painstakingly adapted to the non-standard semantics.

We would be home and dry if we could prove the equivalence of the Relational and Imperative semantics; that is, if we could show that an IMP program was Relational-safe if and only if it was Imperative safe. Unfortunately, this is not true.

Example. Consider the IMP program:

$$\mathtt{loop}\{ \quad \begin{matrix} \nu \leftarrow \mathtt{nondet}(); \\ \kappa \leftarrow (\nu) \end{matrix} \quad \middle\| \quad \begin{matrix} (t_0) \leftarrow \kappa; \\ \nu \leftarrow t_0; x \leftarrow \nu; \\ (t_0) \leftarrow \kappa; \\ \nu \leftarrow t_0; y \leftarrow \nu; \\ \mathtt{assert}(x = y) \end{matrix} \quad \}$$

This program *is not* Relational-safe as the set-operation in the first instruction populates κ with the set of all integers, and the get-operation in the second instruction can assign different values to integer values to x and y. However the program *is* Imperative-safe as whenever the second instruction executes, κ will be undefined or contain some arbitrary integer that is assigned to both x and y, which causes the assert to succeed.

This example pinpoints exactly why the two semantics differ. In the Relational semantics, in any given loop iteration, different gets on the same κ can return *different* tuples, while in the Imperative semantics the gets are correlated and return the same tuple.

Read-Write-Once Programs. An IMP instruction is a *read-write-once* instruction if any relation variable κ is read from and written to at most once in the instruction. That is, read-write-once means at most one write and at most one read (and not at most one read or write). An IMP program is a *read-write-once* program if each instruction in its loop is a read-write-once instruction. We can show that for Read-Write-Once IMP programs the Relational and Imperative semantics are equivalent.

THEOREM 2. *If* P *is a* read-write-once IMP *program then* P *is* Relational-safe *iff* P *is* Imperative-safe.

To prove this theorem, we formalize the connection between the reachable states under the two different semantics, using the function Expand, which maps a Relational-state to a set of Imperative states:

$$\mathsf{Expand}(s^\sharp) \doteq \left\{ s \;\middle|\; \begin{matrix} s(x) = s^\sharp(x) & \text{for base variables} \\ s(\kappa) = \langle v \rangle & \text{if } \langle v \rangle \in s^\sharp(\kappa) \\ s(\kappa) = \bot & \text{if } s^\sharp(\kappa) = \emptyset \\ s = \mathcal{E} & \text{if } s^\sharp = \mathcal{E} \end{matrix} \right\}$$

We lift the function to sets of Relational states in the natural way:

$$\mathsf{Expand}(\hat{\Sigma}^\sharp) \doteq \bigcup \{\mathsf{Expand}(s^\sharp) \mid s^\sharp \in \hat{\Sigma}^\sharp\}$$

Next, we can show that read-write-once instructions enjoy the following property, by case splitting on the form of I.

LEMMA 1. *[Step] If* I *is a read-write-once instruction then* $\mathsf{Expand}(\mathsf{Post}^\sharp(s^\sharp, \mathtt{I})) = \mathsf{Post}(\mathsf{Expand}(s^\sharp), \mathtt{I})$.

We use this property to show that the reachable states under the different semantics are equivalent.

LEMMA 2. *If* P $= \mathtt{loop}\{\mathtt{I}_1 \| \ldots \| \mathtt{I}_n\}$ *is a read-write-once program, then* $\mathsf{Expand}(\mathsf{Reach}^\sharp(\mathtt{P})) = \mathsf{Reach}(\mathtt{P})$.

PROOF. To prove that $\mathsf{Reach}(\mathtt{P}) \subseteq \mathsf{Expand}(\mathsf{Reach}^\sharp(\mathtt{P}))$, we show

$$\forall m : \mathsf{Reach}(\mathtt{P}, m) \subseteq \mathsf{Expand}(\mathsf{Reach}^\sharp(\mathtt{P}))$$

by straightforward induction on m, noting that $s_0 \in \mathsf{Expand}(s_0^\sharp)$, and $\mathsf{Post}(\mathsf{Expand}(s^\sharp), \mathtt{I}) \subseteq \mathsf{Post}^\sharp(s^\sharp, \mathtt{I})$ for any Relational-state $s^\sharp \in \Sigma^\sharp$, instruction I, and any program P (not necessarily read-write-once).

To show inclusion in the other direction, we prove

$$\forall m : \mathsf{Expand}(\mathsf{Reach}^{\sharp}(\mathsf{P}, m)) \subseteq \mathsf{Reach}(\mathsf{P})$$

by induction on m. For the base case,

$$\mathsf{Expand}(\mathsf{Reach}^{\sharp}(\mathsf{P}, 0)) = \mathsf{Reach}(\mathsf{P}, 0) \subseteq \mathsf{Reach}(\mathsf{P})$$

by the definition of the initial states. By induction, assume that

$$\mathsf{Expand}(\mathsf{Reach}^{\sharp}(\mathsf{P}, m)) \subseteq \mathsf{Reach}(\mathsf{P})$$

Let $s' \in \mathsf{Expand}(\mathsf{Reach}^{\sharp}(\mathsf{P}, m + 1))$. By Lemma 1, either s' is already in $\mathsf{Reach}^{\sharp}(\mathsf{P}, m)$, in which case the inductive hypothesis applies and hence $s' \in \mathsf{Reach}(\mathsf{P})$, or

$$s' \in \mathsf{Post}(\mathsf{Expand}(\mathsf{Reach}^{\sharp}(\mathsf{P}, m), \mathtt{I}_j)$$

for some j. That is, there is a $s \in \mathsf{Expand}(\mathsf{Reach}^{\sharp}(\mathsf{P}, m)$ such that $s' \in \mathsf{Post}(s, \mathtt{I}_j)$. From the induction hypothesis $s \in \mathsf{Reach}(\mathsf{P})$. As $\mathsf{Reach}(\mathsf{P})$ is closed under Post, we conclude $s' \in \mathsf{Reach}(\mathsf{P})$. $\square$

5.3 Cloning

At this point, we have shown that the Imperative semantics of read-write-once programs are equivalent to the Relational semantics. All that remains is to show that the translation procedure of Figure 7 produces read-write-once programs. Unfortunately, this is not true.

Example. Consider the following constraints:

$$\emptyset \vdash \{\kappa\}, \emptyset \vdash \{true\} <: \{\kappa\}, x{:}\kappa; y{:}\kappa \vdash \{true\} <: \{x = y\}$$

It is easy to check that on the above constraints, the translation procedure yields the IMP program from the previous example, which is not read-write-once.

The reason the translated program is not a read-write-once program is that there can be constraints $G \vdash T_1 <: T_2$ in which κ occurs in multiple places within G and T_1.

To solve this problem, we can simply *clone* the κ variables that occur multiple times inside a constraint, and use different clones at each occurrence! We formalize this as a procedure Clone that maps a finite set of constraints to another finite set. The procedure works as follows. For each κ that is read upto n times in some constraint, we make n clones, $\kappa^1, \ldots, \kappa^n$, and

1. for the ith occurence of κ within any constraint, we use the ith clone κ^i (instead of κ), and,
2. for each constraint where κ appears on the right hand side, we make n clones of the constraints where in the ith cloned constraint, we use κ^i (instead of κ).

The first step ensures that each κ is read-once in any constraint, and the second step ensures that the clones correspond to exactly the same set of tuples as the original variable κ. We can prove that Clone enjoys the following properties.

THEOREM 3. *Let C be a finite set of constraints.*

1. $[\![\mathsf{Clone}(C)]\!]$ *is a read-write-once program.*
2. $\mathsf{Clone}(C)$ *is satisfiable iff C is satisfiable.*

It is easy to verify that $[\![\mathsf{Clone}(C)]\!]$ is a read-write-once program. Furthermore, any satisfying solution for the original constraints can be mapped directly to a solution for the cloned constraints. To go in the other direction, we must map a solution that satisfies the cloned constraints to one that satisfies the original constraints. This is trivial if the solution for the cloned constraints maps each clone κ^i to the same set of tuples. We show that if the cloned constraints have a satisfying solution, they have a solution that satisfies the above property. To this end, we prove the following lemma that states that for *any* set of constraints, the satisfying solutions are closed under intersection.

Program	Time (sec)	Invariant
		Refinement Types
max	0.091	$\kappa_1.1 \leq \kappa_1.0 \wedge \kappa_1.2 \leq \kappa_1.0$
		$\kappa_x \doteq true, \kappa_y \doteq true, \kappa_1 \doteq x \leq v \wedge y \leq v$
sum	0.071	$0 \leq \kappa_2.0 \wedge \kappa_2.1 \leq \kappa_2.0$
		$\kappa_k \doteq true, \kappa_2 \doteq 0 \leq v \wedge k \leq v$
foldn	0.060	$0 \leq \kappa_i.0 \wedge 0 \leq \kappa_3.0 \wedge \kappa_3.0 < \kappa_3.2$
		$\kappa_i \doteq 0 \leq v, \kappa_3 \doteq 0 \leq v \wedge v < n$
arraymax	0.135	$0 \leq \kappa_4.0 \wedge 0 \leq \kappa_5.0 \wedge$ $0 \leq \kappa_6.0 \wedge \kappa_g.0 < \mathtt{len}(\kappa_g.1)$
		$\kappa_4 0 \doteq v, \kappa_5 \doteq 0 \leq v,$ $\kappa_6 \doteq 0 \leq v, \kappa_g \doteq v < \mathtt{len}(\mathtt{a})$
mask	0.098	$\kappa_1.0 < \mathtt{len}(\kappa_1.4) \wedge \kappa_1.1 \leq \kappa_1.0 \wedge$ $0 \leq \kappa_2.0 \wedge \kappa_2.0 < \mathtt{len}(\kappa_2.3)$
		$\kappa_1 v < \mathtt{len}(\mathtt{xs}) \wedge i \leq v,$ $\kappa_2 \doteq 0 \leq v \wedge v < \mathtt{len}(\mathtt{a})$
samples	0.117	$0 \leq \kappa_2.0 \wedge \kappa_2.0 < \mathtt{len}(\kappa_2.4) \wedge$ $0 \leq \kappa_3.0 \wedge \kappa_3.0 < \mathtt{len}(\kappa_3.3) \wedge 0 \leq \kappa_6.0$
		$\kappa_2 \doteq 0 \leq v \wedge v < \mathtt{len}(\mathtt{b}),$ $\kappa_3 \doteq 0 \leq v \wedge v < \mathtt{len}(\mathtt{a}), \kappa_6 \doteq 0 \leq v$

Table 1. Experimental evaluation using a predicate abstraction-based verification tool on examples from [29]. The third column presents the invariant for the translated program, and the resulting refinement types.

LEMMA 3. *If S_1 and S_2 are solutions that satisfy C then $S_1 \cap S_2 \doteq \lambda\kappa.S_1(\kappa) \cap S_2(\kappa)$ satisfies C.*

Thus if S satisfies the cloned constraints then by symmetry and Lemma 3 the solution that maps *each* cloned variable to $\cap_{i=1}^{n} S(\kappa^i)$ also satisfies the cloned constraints, and hence, directly yields a solution to the original constraints.

Finally, as a corollary of Theorems 1,2,3 we get our main result that reduces the question of refinement type constraint satisfaction, to that of safety verification.

THEOREM 4. *C is satisfiable iff $[\![\mathsf{Clone}(C)]\!]$ is Imperative-safe.*

While we state Theorems 1 and 3 as preserving satisfiability, the proof shows how the solutions can be effectively mapped between C and $[\![C]\!]$ (or $[\![\mathsf{Clone}(C)]\!]$). In particular, while the intersection of two non-trivial solutions can be a trivial solution, it would be guaranteed that in that case, the trivial solution satisfies C. Stated in terms of invariants, Lemma 3 states the observation that that there may be several non-comparable inductive invariants to prove a safety property, but in that case, the intersection of all the inductive invariants is also an inductive invariant.

6. Experiments

We have implemented a verification tool for OCAML programs based on RTI. We use the liquid types infrastructure implemented in DSOLVE [29] to generate refinement type constraints from OCAML programs. We use ARMC [28], a software model checker using predicate abstraction and interpolation-based refinement, as the verifier for the translated imperative program.

Table 1 shows the results of running our tool on a suite of small OCAML examples from [29]. For array manipulating programs, the safety objective is to prove array accesses are within bounds. For MAX we prove that the output is larger than input values. For SUM we prove that the sum is larger than the largest summation term.

Table 2 presents the running time of our tool on the benchmark programs for the Depcegar verifier [31]. We observe that despite of our blackbox treatment of ARMC as a constraint solver we obtain competitive running times compared to Depcegar on most of the examples (Depcegar uses a customized procedure for unfolding

Program	Time	# iterations	# predicates
boolflip.ml	2.17s	7	21
sum.ml	0.24s	5	14
sum-acm.ml	0.11s	1	3
sum-all.ml	3.51s	10	26
mult.ml	4.67s	10	25
mult-cps.ml	780.24s	11	27
mult-all.ml	18.44s	9	24
boolflip-e.ml	0.65s		
sum-e.ml	0.01s		
sum-acm-e.ml	0.02s		
sum-all-e.ml	0.79s		
mult-e.ml	0.01s		
mult-cps-e.ml	7.69s		
mult-all-e.ml	144.93s		

Table 2. Experimental evaluation of our tool on Depcegar benchmarks [31]. The third column presents the number of abstraction refinement iterations required by ARMC. The last column gives the number of predicates discovered by ARMC. For the programs with suffix "-e", which are incorrect, we omit the number of iterations and predicates and only show the time required by ARMC to find a counterexample.

constraints and creating interpolation queries that yield refinement types).

Most of the predicates discovered by the interpolation-based abstraction refinement procedure implemented in ARMC fall into the fragment "two variables per inequality." The example MASK required a predicate that refers to three variables, see κ_1. While our initial experiments used a CEGAR-based tool, we expect optimized abstract interpreters for numerical domains to also work well for this class of properties.

7. Extensions and Related Work

7.1 Completeness

The soundness of safety verification for higher-order programs for any domain follows from the soundness of constraint generation (*e.g.,* Theorem 1 in [29]) and Theorem 4. Since the safety verification problem for higher-order programs is undecidable, the technique cannot be complete in general. Even in the finite-state case, in which each base type has a finite domain (*e.g.,* booleans), completeness depends on the generation of type constraints. For example, in our examples and in our implementation, we have assumed a *context insensitive* constraint generation from program syntax, *i.e.,* we have not distinguished the types of the same function at different call points. This entails a loss of information, as the following example demonstrates. Consider

```
let check f x y = assert (f x = y) in
check (fun a -> a) false false ;
check (fun a -> not a) false true
```

where the builtin function `assert` has the type $\{\nu:\texttt{bool} \mid \nu\} \to \texttt{unit}$. The refinement template for `check` generated by our constraint generation process is

$$(\texttt{x} : \{\nu:\texttt{bool} \mid \kappa_1\} \to \{\kappa_2\}) \to \{\kappa_3\} \to \{\kappa_4\} \to \texttt{unit}$$

which is too weak to show that the program is safe. This is because the template "merges" the two call sites for `check`.

One way to get context sensitivity is through *intersection types* [12, 14, 20, 25]. For the above example, we can show type safety using the following refined type for `check`:

$$\bigwedge \begin{array}{l} (\texttt{x} : \texttt{bool} \to \{\nu = \texttt{x}\}) \to \{\neg\nu\} \to \{\neg\nu\} \to \texttt{unit} \\ (\texttt{x} : \texttt{bool} \to \{\nu = \neg\texttt{x}\}) \to \{\neg\nu\} \to \{\nu\} \to \texttt{unit} \end{array}$$

It is important to note that Theorems 1 and 2 hold for *any* set of constraints. Thus, one way to get completeness in the finite state case is to generate refinement templates using intersection types, perform the translation to IMP programs, and then using a complete invariant generation technique for finite state systems. The key observation (made in [20]) that ensures a finite number of constraints, is that there is at most a finite number of "contexts" in the finte state case, and hence a finite number of terms in the intersection types. The bad news is that the bound on the number of contexts is $\exp_n(k)$, where n is the highest order of any function in the program, k is the maximum arity of any function in the program, and $\exp_n(k)$ is a stack of n exponentials, defined by $\exp_0(k) = k$, and $\exp_{n+1}(k) = 2^{\exp_n(k)}$.

Fully context-sensitive constraints are used in [20] to show completeness in the finite case, at the price of $\exp_n(k)$ in *every case*, not just the worst case. In our exposition and our implementation, we have traded off precision for scalability: while we lose precision by generating context-insensitive constraints, we avoid the $\exp_n$ blow-up that comes with full context sensitivity. However, it has been shown through practical benchmarks that since the types themselves capture relations between the inputs and outputs, the context-insensitive constraint generation suffices to prove a variety of complex programs safe [3, 18, 29].

When considering completeness properties in special cases, we point out completeness wrt. the discovery of refinement predicates in octagons/difference bounds abstract domains [24] and template-based invariant generation for linear arithmetic [7] and extensions with uninterpreted function symbols [5], which carries over from respective verification approaches.

7.2 Related Work

Higher-Order Programs. Kobayashi [20, 21] gives an algorithm for model checking arbitrary μ-calculus properties of finite-data programs with higher order functions by a reduction to model checking for higher-order recursion schemes (HORS) [26]. For safety verification, RTI shows a promising alternative.

First, the reduction to HORS critically depends on a finite-state abstraction of the data. In contrast, our reduction defers the data abstraction to the abstract interpreter working on the imperative program, thus enabling the direct application of abstract interpreters working over infinite domains. Since abstract interpreters over infinite abstract domains are strictly more powerful than (infinite families of) finite ones [8], our approach can be strictly more powerful for infinite-state programs.

Second, in the translation of an abstracted program to a HORS, this algorithm eliminates Boolean variables by enumerating all possible assignments to them, giving an exponential blow-up from the program to the HORS. In contrast, our technique preserves the Boolean state *symbolically*, enabling the use of efficient symbolic algorithms for verification. For example, for the simple example:

```
let f b1 ... bn x =
  if (b1 || ... || bn) then lock x;
  if (b1 || ... || bn) then unlock x
in let f (*) ... (*) (newlock ())
```

where we wish to prove that lock and unlock alternate. Kobayashi's translation [20] gives an *exponential* sized HORS, with a version of f for each assignment to b1,...,bn. In contrast, our reduction preserves the source-level expressions and is linear, and amenable to symbolic verification techniques (e.g., BDDs). Previous experience with software model checking [2, 16, 17] shows that the number of reachable states is often drastically smaller than 2^p where p is the number of Booleans. Thus, the pre-processing step that enumerates Booleans may not lead to a scalable implementation.

Might [23] describes *logic-flow analysis*, a general safety verification algorithm for higher-order languages, which is the product of a k-CFA like call-strings analysis and a form of SMT-based predicate abstraction (together with widening). In contrast, our work shows how higher-order languages can be analyzed directly via abstract analyses designed for first-order imperative languages.

Inference of refinement types using conterexample-guided techniques was recentrly identified as a promising direction [31, 32]. In contrast, our approach is not limited to CEGAR and facilitates the applicability of a wide range abstract interpretation techniques for precise reasoning about program data.

Software Verification. This work was motivated by the recent success in software model checking for first-order imperative programs [2, 6, 16, 22], and the desire to apply similar techniques to modern programming languages with higher order functions. Our starting point was refinement types [14, 19], implemented in dependent ML [33] to give strong static guarantees, and the work on liquid types [18, 29] that applied predicate abstraction to infer refinement types. By enabling the application of automatic invariant generation from software model checking, RTI reduces the need for programmer annotations in refinement type systems.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1:375–416, 1991.

[2] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3. ACM, 2002.

[3] J. Bengtson, K. Bhargavan, C. Fournet, A.D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, 2008.

[4] D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis in combination theories. In *VMCAI*, 2007.

[5] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*. Springer, 2007.

[6] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.

[7] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*. Springer, 2003.

[8] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP*, LNCS 631, pages 269–295. Springer-Verlag, 1992.

[9] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*. ACM Press, 1978.

[10] S. Cui, K. Donnelly, and H. Xi. Ats: A language that combines programming with theorem proving. In *FroCos*, 2005.

[11] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, 1982.

[12] J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2007.

[13] C. Flanagan. Hybrid type checking. In *POPL*. ACM, 2006.

[14] T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, 1991.

[15] S. Gulwani and G.C. Necula. Discovering affine equalities using random interpretation. In *POPL*, pages 74–84, 2003.

[16] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04*. ACM, 2004.

[17] H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *CAV*, pages 137–151, 2006.

[18] M. Kawaguchi, P. Rondon, , and R. Jhala. Type-based data structure verification. In *PLDI*, pages 304–315, 2009.

[19] K. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *ESOP*, 2007.

[20] N. Kobelshashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL*, 2009.

[21] N. Kobayashi and C.-H.L. Ong. A type system equivalent to modal μ-calculus model checking of recursion schemes. In *LICS*, 2009.

[22] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*. 2006.

[23] Matthew Might. Logic-flow analysis of higher-order programs. In *POPL*, pages 185–198, 2007.

[24] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[25] M. Naik and J. Palsberg. A type system equivalent to a model checker. *ACM Trans. Program. Lang. Syst.*, 30(5), 2008.

[26] C.-H.L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, 2006.

[27] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP TCS*, pages 437–450, 2004.

[28] A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *PADL*, 2007.

[29] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[30] S. Sankaranarayanan, H.B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, 2005.

[31] Tachio Terauchi. Dependent types from counterexamples. In *POPL*. ACM, 2010.

[32] Hiroshi Unno and Naoki Kobayashi. Dependent type inference with interpolants. In *PPDP*. ACM, 2009.

[33] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.