

HMC: Verifying Functional Programs Using Abstract Interpreters

Ranjit Jhala¹, Rupak Majumdar², and Andrey Rybalchenko³

¹ UC San Diego

² MPI-SWS and UC Los Angeles

³ TU München

Abstract. We present Hindley-Milner-Cousots (HMC), an algorithm that reduces verification of safety properties of typed higher-order functional programs to interprocedural analysis for first-order imperative programs. HMC works as follows. First, it uses the type structure of the functional program to generate a set of logical refinement constraints whose satisfaction implies the safety of the source program. Next, it transforms the logical refinement constraints into a simple first-order imperative program and an invariant that holds iff the constraints are satisfiable. Finally, it uses an invariant generator for first-order imperative programs to discharge the invariant. We have implemented HMC and describe preliminary experimental results using two imperative checkers – ARMC and IN-TERPROC – to verify OCAML programs. By composing type-based reasoning grounded in program syntax and state-based reasoning grounded in abstract interpretation, HMC enables the *fully automatic* verification of programs written in modern programming languages.

1 Introduction

Automatic verification of semantic properties of modern programming languages is an important step toward reliable software systems. For higher-order functional programming languages with inductive and polymorphic datatypes, the main verification tool has been type systems. These type systems traditionally capture only coarse data-type properties (*e.g.*, functions expecting `ints` are only passed `ints`), but not more precise, semantic properties (*e.g.*, that an array index is within bounds). For first-order imperative programming languages, automatic tools based on abstract interpretation, such as ASTREE [3], SLAM [1], BLAST [7], *etc.*, can infer program invariants and prove many semantic properties of practical interest. While these tools faithfully model the semantics of base values like `ints`, they are overly imprecise on modern programming features such as closures, higher-order functions, inductive datatypes or polymorphism.

We present Hindley-Milner-Cousots (HMC), an algorithm that combines type-based reasoning for higher-order languages with invariant generation for first-order languages to prove semantic properties of programs fully automatically and without additional programmer annotations.

The link between types and invariants is the notion of *refinement type checking* [2, 14, 20, 26], a type-based analogue of Floyd-Hoare logic. A refinement type is a type whose “leaves” are base types decorated with *refinement predicates*. For example, the refinement type $\{x:\text{int} \mid x < 100\}$ *list* describes a list of integers, each

of which is smaller than 100 and $\text{int} \rightarrow \{\mathbf{x} : \text{int} \mid \mathbf{x} \neq 0\} \rightarrow \text{int}$ describes a function (*e.g.*, integer division) whose second (curried) argument must be non-zero. By riding atop type-structure, refinements can express sophisticated data structure invariants as well [4, 5, 12]. While refinement type checking can be used to verify functional programs [2], the programmer must manually provide the refinements which is analogous to the burden of writing loop-invariants together with pre- and post-conditions in the imperative setting. HMC eliminates the need for programmer annotations and thereby enables automatic checking via a three-step process.

Step 1: Constraint Generation. HMC generates a set of refinement constraints whose satisfaction implies the safety of the source program. To verify safety of a functional program, we need to compute safe overapproximations of the sets of values that various expressions can evaluate to (*i.e.*, the functional analogue of “reachable states” in the imperative setting). With refinement types, overapproximation is formalized via subtyping. Thus, in the first phase, HMC makes a syntax directed pass over the functional program to generate a set of subtyping constraints over *refinement templates* that represent the unknown refinement types for various program expressions. The templates employ *refinement variables* κ as placeholders for the unknown refinement predicates, which are analogous to program invariants, that decorate the leaves of the complex types. Crucially, as the overapproximation is structured via types, we can use the standard rules for subtyping complex types to reduce the complex subtyping constraints to a set of simple implication constraints [13, 22], whose satisfaction implies program safety.

Step 2: Constraint Translation. Next, HMC transforms the implication constraints into a first-order imperative program that is safe if and only if the constraints are satisfiable. This translation – our main technical contribution – is founded upon two key insights. First, the refinement variables κ , normally viewed as placeholders for (unknown) refinement predicates, semantically represent (unknown) n -ary relations over the value being defined by the refinement type and the $n - 1$ variables that are in scope at the point where the type is defined. Second, the constraints on each κ can be used to encode a simple *first-order imperative* function F_κ whose input-output semantics corresponds to an n -ary relation that satisfies the constraints on κ . The $n - 1$ components of the relation are treated as input parameters of the function F_κ , and the value component of the relation becomes the output of the function. Using these insights we design an algorithm that translates type-bindings into function calls and implications into assignments/assumes, respectively. Thus, we obtain a first-order imperative program that is safe if and only if the constraints are satisfiable, *i.e.*, whose safety implies the safety of the source functional program.

Thus, the two-step HMC algorithm uses type-structure to reduce the safety of a higher-order functional program to the safety of a first-order imperative program.

Step 3: Verification of Obtained First-Order Program. In a third step, we use existing invariant generation tools for first-order imperative programs to automatically verify the safety of the imperative program produced during the second step.

The most immediate dividend of our approach is that HMC allows one to readily apply any of the well-developed semantic imperative program analyses to the

verification of modern software with polymorphism, inductive datatypes, and higher-order functions. More importantly, HMC provides a “separation-of-concerns” that can open the door to a suite of precise model checkers and abstract interpreters capable of handling languages with advanced features. Using HMC, the analysis designer can factor the analysis into two parts: a syntactic, type-system based component that analyzes macroscopic language concerns like collections, inductive types, polymorphism, closures, *etc.*, and a semantic, abstract interpretation-based component that analyzes microscopic language concerns like invariant relationships between primitive integers or booleans. Thus, HMC provides a simple way to incorporate independent progress in type systems for specifying complex control as well as dataflow and in invariant generation techniques into the verification flow. For example, one can tune the precision and scalability of an analysis either by changing the amount of context-sensitivity in the type system (*e.g.*, via intersection types) or by using more/less precise abstract domains (*e.g.*, using polyhedra instead of octagons).

To demonstrate the feasibility of our approach, we have developed two safety verifiers for ML programs, HMC(ARMC) and HMC(INTERPROC), which use the CEGAR-based ARMC [21] software model checker and the Polyhedra-based INTERPROC [17] analyzer, respectively, 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 [26] or their constituent predicates [22]), or an elaborate customization and adaptation of the counterexample-guided abstraction refinement paradigm [24].

Related Work. Our starting point was the vast body of work in the verification of imperative programs (see, *e.g.*, [10] for a survey), including tools such as SLAM [1], BLAST [8], and ASTREE [3], and to “lift” the techniques to higher-order programming languages. We were influenced by work on refinement types [6, 13] implemented in dependent ML [26] and, more recently, combined with predicate abstraction [12, 22], but wanted to eliminate the need for explicit annotations (or predicates).

Kobayashi [15, 16] gives an algorithm for model checking higher order programs by a reduction to model checking for higher-order recursion schemes (HORS) [19], which has been augmented to perform CEGAR [24, 25]. For safety verification, HMC shows a promising alternative, enabling us to reuse the vast literature on invariant generation for first order programs (using abstract interpreters or model checkers).

While we have implemented our tool for ML, our constraint language is generic and can express refinement constraints arising out of other expressive source languages, such as F7 [2] or C [23], which include module signatures, recursive and contextual types, mutable state, *etc.* Thus, through the collaboration of types and invariants, HMC opens the door to the automatic safety verification of complex properties of programs in modern languages.

2 Overview

We outline the steps of the HMC algorithm. For lack of space, we omit the full formalization and correctness proofs and provide the main insights through an example. The formalization can be found in [11].

```

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

```

Fig. 1. ML Example

An ML Example. Figure 1 shows a simple ML program that updates an array *a* using the elements of the list *xs*. The program comprises two functions. The first function 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 function 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 infer handle arithmetic constraints on indices, and the latter do not precisely handle higher-order functions and are often imprecise on data structures. We show how our 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 [2, 13, 22]. The predicates are additional constraints that must be satisfied by every value of the type. We work with a fixed set of *base types* β , comprising *int* for *integer* values, *bool* for *boolean* values, and *ui*, a family of *uninterpreted types* that encode complex source language types such as products, sums, recursive types *etc.* A base value, say of type β , can be described by the refinement type $\{\nu:\beta \mid p\}$ where ν is the value variable of the refinement type that names the value being defined, and *p* is a *refinement predicate* which constrains the range of ν to a subset of β . For example, $\{\nu:\text{int} \mid 0 \leq \nu < \text{len}(a)\}$ denotes the set of integers that are between 0 and the value of the expression *len(a)*. Thus, the (unrefined) type *int* abbreviates $\{\nu:\text{int} \mid \text{true}\}$. Base types can be combined to construct *dependent function types*, where the value variable for the input type, *i.e.*, the name of the formal parameter, can appear in the refinement predicates in the output type, thereby expressing a “post-condition” that relates the function’s outputs with its inputs. For example, $\{x:\text{int} \mid 0 \leq x\} \rightarrow \{\nu:\text{int} \mid \nu = x + 1\}$ is the type of a function which takes a non-negative integer parameter and returns an output which is one more than the input. Thus, the input and output types describe pre- and post-conditions

for the function. In the following, we write $x:\beta$ for the type $\{x:\beta \mid true\}$, and $x:r$ for $\{x:\beta \mid r\}$, when β is clear from the context,

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 $a.(j)$ (resp. write $a.(j) \leftarrow e$) in g which is an abbreviation for $\text{get } a \ j$ (resp. $\text{set } a \ j \ e$). By giving get and set the types

$$\begin{aligned} a:\alpha \text{ array} &\rightarrow \{i:\text{int} \mid 0 \leq i < \text{len}(a)\} \rightarrow \alpha, \\ a:\alpha \text{ array} &\rightarrow \{i:\text{int} \mid 0 \leq i < \text{len}(a)\} \rightarrow \alpha \rightarrow \text{unit}, \end{aligned}$$

we can specify that in any program the array accesses must be within bounds. More generally, arbitrary safety properties can be specified [22] by giving `assert` the refinement type $\{p:\text{bool} \mid p = \text{true}\} \rightarrow \text{unit}$.

We assume that the call-by-value dynamic semantics of ML programs is formalized using a standard small-step (contextual) operational semantics.¹ We write \rightsquigarrow for the single evaluation step relation for expressions, and write \rightsquigarrow^* to describe the reflexive, transitive closure of \rightsquigarrow . To capture program errors, we assume there is a special `Err` value, and if a constant is applied to a value that is not in the domain of the constant (*e.g.*, accessing an array out of bounds, or calling `assert` with `false`), then the application reduces to `Err`. For a program e , we say that e *safe* if there is no derivation of the form $e \rightsquigarrow^* \text{Err}$. In other words, a program is safe if it never reduces to `Err`.

Safety Verification. The ML type system is too imprecise to prove the safety of the array accesses in our example as it infers that g has type $j:\text{int} \rightarrow y:\text{bool} \rightarrow \text{unit}$, *i.e.*, that g can be called with *any* integer j . If the programmer manually provides the refinement types for all functions and polymorphic type instantiations, refinement-type checking [2, 5, 26] 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, a refinement type system could check the program if the programmer provided the types:

$$\begin{aligned} \text{iteri} :: i:\text{int} &\rightarrow \{xs:\alpha \text{ list} \mid 0 \leq \text{len}(xs)\} \rightarrow \\ &\quad (\{j:\text{int} \mid i \leq j < i + \text{len}(xs)\} \rightarrow \alpha \rightarrow \text{unit}) \rightarrow \text{unit} \\ g :: \{j:\text{int} \mid 0 \leq j < \text{len}(a)\} &\rightarrow \text{bool} \rightarrow \text{unit} \end{aligned}$$

Automatic Verification via HMC. As even this simple example illustrates, the annotation burden for verification can be quite high. Instead, we show how our algorithm combines type-based reasoning for complex language features and abstract interpretation for first-order control flow to automatically verify the program without requiring refinement annotations. Our HMC algorithm proceed as follows. First, we use the *source* program to generate a set of constraints which is satisfiable if the program is safe. Second, we translate the constraints into a first-order *imperative* target program which is safe iff the set of constraints is satisfiable. After these two steps, we can analyze the target program with any first-order safety analyzer. If the analyzer determines the target is

¹ Our formalization of the algorithm actually uses a core fragment μML of ML. We keep the following discussion simple by simply referring to ML programs.

safe, we can soundly conclude that the constraints are satisfiable, and hence, the source program is safe. Next, we illustrate these steps using the source program from Figure 1.

Notation. Let X be a set of variables. We use ν, x, y, z and subscripted versions thereof to range over X . A *state* σ is a partial map from variables X to values in the universe $\mathcal{U}(\beta)$ of values of type β . We lift states to maps from expressions to values and predicates to boolean values in the standard manner. We write $[\cdot]$ for the state with empty domain, and write $\sigma[z \mapsto v]$ for the state that maps the variable z to v and all other variables y to $\sigma(y)$.

2.1 Step 1: Constraint Generation

First, we generate a system of *refinement constraints* for the source program [13, 22]. 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.

Templates. For the functions `iteri` and `g` from Figure 1, with the respective ML types

```
i:int → xs:α list → (j:int → α → unit) → unit
j:int → bool → unit
```

we generate the templates

```
i:int → xs:{0 ≤ len(xs)} → (j:κ₁(j, i, xs) → α → unit) → unit
j:κ₂(j, a, xs) → bool → unit
```

The templates refine the ML types with *parameterized refinement variables* that represent the unknown refinements. $\kappa_1(j, i, xs)$ represents the unknown refinement that describes the values passed as the first input to the function `f` that is used by the iterator `iteri`. The values are the first elements of tuples belonging to a ternary relation between the values of `j` and the two other program variables *in-scope* at that point, namely `i` and `xs`. $\kappa_2(j, a, xs)$ represents the unknown refinement that describes the values passed as the first input to `g`. In this case, the values belong to a ternary relation over `j`: the formal and the two variables `a` and `xs` in scope at that program point.

For clarity of exposition, we have use the trivial refinement *true* for some variables (e.g., for α and `bool`); HMC would automatically infer these refinements. We model the length of lists (resp. arrays) with an uninterpreted function `len` from the lists (resp. arrays) to integers, and (again, for brevity) add the refinement stating `xs` has a non-negative length in the type of `iteri`.

Constraints. Informally, refinement constraints reduce the flow of values within the program into subtyping relationships that must hold between the source and target of the flow. A *refinement* r is either a *concrete predicate* p drawn from some predicate language or a parameterized refinement variable $\kappa(x_0, \dots, x_n)$, where κ is a refinement variable of arity n . We assume, without loss of generality, that each κ has a fixed arity. A *refinement type binding* ρ is a triple $\{x:\beta \mid r\}$ comprising a *variable* x that is being bound, a base type β describing the base type of x , and a refinement r that describes

Predicates	$\sigma \models p$	iff $\sigma(p) = \text{true}$
Environments	$\Sigma, [\cdot] \models \emptyset$	
	$\Sigma, \sigma \models G; \{x:\beta \mid r\}$	iff $\Sigma, \sigma \setminus x \models G$ and $\Sigma, \sigma \models \{x:\beta \mid r\}$
Refinements	$\Sigma, \sigma \models \{x:\beta \mid p\}$	iff $\sigma \models p$
	$\Sigma, \sigma \models \{x:\beta \mid \kappa(y_0, \dots, y_n)\}$	iff $(\sigma(y_0), \dots, \sigma(y_n)) \in \Sigma(\kappa)$
Constraints	$\Sigma \models G \vdash \{\mathbf{x}_1:\beta \mid r_1\} \prec \{\mathbf{x}_2:\beta \mid r_2\}$	iff For all σ : $\Sigma, \sigma \models G; \{\nu:\beta \mid r_1\}$ implies $\Sigma, \sigma \models \{\mathbf{x}_1:\beta \mid r_2[\mathbf{x}_1/\mathbf{x}_2]\}$

Fig. 2. Constraint Satisfaction

an invariant satisfied by all the values bound to \mathbf{x} . A *refinement environment* G is a sequence of refinement bindings. A *refinement constraint* $G \vdash \{\mathbf{x}:\beta \mid r_1\} \prec \{\mathbf{x}:\beta \mid r_2\}$ states that when the program variables satisfy the invariants described in G , the type r_1 must be a subtype of r_2 , that is, the set of values described by the refinement r_1 *must be included in* the set of values described by the refinement r_2 .

Satisfaction. A *relational interpretation* for κ of arity n is a subset of $\mathcal{U}(\beta_1) \times \dots \times \mathcal{U}(\beta_n)$ for appropriate types β_1, \dots, β_n for the parameters of κ . A *relational model* Σ is a map from refinement variables κ to relational interpretations. Figure 2 formalizes the notion of when a relational interpretation Σ *satisfies* a constraint. A state satisfies a predicate if the predicate evaluates to true in the state. A state satisfies a predicate refinement binding if the tuple of values of relevant variables belongs to the relation corresponding to the refinement. A state satisfies an environment if it satisfies each binding in the environment. A relational interpretation satisfies a constraint if every state that satisfies the LHS of the constraint also satisfies the RHS of the constraint. A relational interpretation satisfies a set of constraints if it satisfies each constraint in the set.

Constraint Generation. Let $\text{Generate}(e)$ denote the procedure that takes as input a program e and uses the type structure of the program to generate constraints. $\text{Generate}(e)$ proceeds syntactically over the structure of the program. We omit the full description of the procedure $\text{Generate}(e)$, which is similar to the constraint generation procedure for refinement type constraints (e.g., [2, 6, 13, 22]). Theorem 1 summarizes the main property of $\text{Generate}(e)$.

Theorem 1. *If $\text{Generate}(e)$ is satisfiable then e is safe.*

For our example, the following subtyping constraints are generated syntactically from the code. First consider the function `iteri`. The call to `f` generates

$$G \vdash \{\nu:\text{int} \mid \nu = i\} \prec \{\nu:\text{int} \mid \kappa_1(\nu, i, \mathbf{xs})\} \quad (\text{c1})$$

where ν is the parameter's value, and the environment bindings are

$$\begin{aligned} G \doteq & i:\text{int}; \{ \mathbf{xs}:\alpha \text{ list} \mid 0 \leq \text{len}(\mathbf{xs}) \}; \\ & \mathbf{x}:\alpha; \{ \mathbf{xs}' :\alpha \text{ list} \mid 0 \leq \text{len}(\mathbf{xs}') = \text{len}(\mathbf{xs}) - 1 \} \end{aligned}$$

The constraint ensures that at the call-site, the refinement of the actual is included in (*i.e.*, a subtype of) the refinement 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 refinement type system yields the information that the length of xs' is one less than xs as the former is the tail of the latter [12, 26]. Similarly, the recursive call to `iteri` generates

$$G \vdash j : \kappa_1(j, i, \text{xs}) \rightarrow \alpha \rightarrow \text{unit} \prec (j : \kappa_1(j, i, \text{xs}) \rightarrow \alpha \rightarrow \text{unit})[i + 1/i][\text{xs}'/\text{xs}]$$

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

$$G \vdash j : \kappa_1(j, i + 1, \text{xs}') \prec j : \kappa_1(j, i, \text{xs}) \quad (\text{c2})$$

Next, consider the function `mask`. The array accesses in `g` generate

$$G' ; j : \kappa_2(j, a, \text{xs}); y : \text{bool} \vdash \{\nu = j\} \prec \{0 \leq \nu < \text{len}(a)\} \quad (\text{c3})$$

a “bounds-check” constraint where G' has bindings for the other variables in scope, namely $a : \text{bool array}$ and $\{\text{xs} : \text{bool list} \mid 0 \leq \text{len}(\text{xs})\}$. Finally, the flow due to the third parameter for the call to `iteri` yields

$$G' ; \text{len}(a) = \text{len}(\text{xs}) \vdash j : \kappa_2(j, a, \text{xs}) \rightarrow \text{bool} \rightarrow \text{unit} \prec j : \kappa_1(j, 0, \text{xs}) \rightarrow \text{bool} \rightarrow \text{unit}$$

where, on the RHS, we have substituted the actuals 0 and xs for the formals i and xs . The last conjunct in the environment represents the guard from the `if` under whose auspices the call occurs. By standard function subtyping, the above reduces to

$$G' ; \text{len}(a) = \text{len}(\text{xs}) \vdash j : \kappa_1(j, 0, \text{xs}) \prec j : \kappa_2(j, a, \text{xs}) \quad (\text{c4})$$

By Theorem 1, if the set of constraints given by (c1), (c2), (c3), and (c4) is satisfiable, then the program is safe.

2.2 Step 2: Translation to Imperative Program

Our main contribution is a translation from the satisfiability problem for a set of refinement constraints to the safety verification problem for an imperative program.

Imperative Programs. We write imperative programs in $\mu\mathcal{C}$, a first-order imperative language with variables ranging over base types β . An *instruction* is either an assignment $x \leftarrow e$, an assume `assume p`, an assert `assert p`, or the sequencing $I; I$ or non-deterministic choice $I \parallel I$ of two instructions. An *assignment* to a target variable is of one of three kinds. Either (1) $x \leftarrow e$: an expression e over the variables is evaluated and assigned to the target variable x , or, (2) $x \leftarrow \text{nondet}()$: an arbitrary non-deterministically chosen value of the appropriate base type is assigned to x , *i.e.*, the target variable is

“havoc-ed”, or (3) $x \leftarrow F(y_1, \dots, y_n)$: a function F is called, and its return value is assigned to the target variable x . A *function definition* has a name F , a sequence of formal parameters z_1, \dots, z_n , a body instruction I , and a return variable z_0 . A *program* is a set of functions including a distinguished *entry* function F_0 that takes no arguments.

Imperative Semantics. We formalize the call-by-value semantics of μC programs using a standard big-step transition relation between program configurations. A *configuration* is either a state, *i.e.*, a partial map from variables X to values, or a special unsafe configuration Err . All the variables in an μC program are *local*. That is, the variables of each (state) configuration describe the values of the variables of a single “stack-frame”. The transition relation is described by the judgment $P, I \vdash \sigma \hookrightarrow \sigma'$ that stipulates that in the program P , the execution of the instruction I causes the machine to move from a configuration σ to the configuration σ' .

The expression and havoc assignments update the target variable with the RHS and a non-deterministically chosen value respectively. The call assignment updates the target value with any of the possible values returned by the callee (*i.e.*, the value of the return variable of the callee in the exit configuration of the callee.) Dually, the return instruction simply assigns the return value into the return variable z_0 . The assume instruction proceeds without updating the state only if the corresponding predicate holds (and otherwise, the program halts). Thus, μC eschews if-then-else instructions in favor of the more general assume and choice instructions. The assert instruction is like the assume, but if the predicate does not hold, the system transitions into the configuration Err (in which it remains forever.)

Let P be an μC program whose entry function F_0 has the body I_0 . We say that P is μC -safe if there is no transition $P, I_0 \vdash [] \hookrightarrow \text{Err}$.

Translation. The constraints generated in Step 1 encode the semantics of program computations. In the second step, we define a translation $\llbracket C \rrbracket$ that takes a set of constraints C and constructs an μC -program such that C is satisfiable iff $\llbracket C \rrbracket$ is safe. Our translation is based on two observations.

Refinements are Relations. The first observation is that refinement variables in the constraints stand for *relations* between program variables: the set of values denoted by a refinement type $\{x_0 : \beta_0 \mid p\}$ where p is a predicate that refers to the program variables x_0, \dots, x_n of base types β_0, \dots, β_n is equivalent to

$$\{t_0 \mid \exists(t_1, \dots, t_n) \text{ s.t. } (t_0, \dots, t_n) \in R_p \wedge_{i=1}^n t_i = x_i\}$$

where R_p is an $(n + 1)$ -ary relation in $\beta_0 \times \dots \times \beta_n$ defined by p . For example, $\{\nu : \text{int} \mid \nu \leq i\}$ is equivalent to the set $\{t_0 \mid \exists t_1 \text{ s.t. } (t_0, t_1) \in R_{\leq} \wedge t_1 = i\}$, where R_{\leq} is the standard \leq -ordering relation over the integers. In other words, each parameterized refinement variable $\kappa(x_0, \dots, x_n)$ can be seen as the projection on the first co-ordinate of a $(n + 1)$ -ary relation over the variables (x_0, \dots, x_n) . Thus, the problem of determining the satisfiability of the constraints is analogous to the problem of determining the existence of appropriate relations.

From Relations to Imperative Programs. The second observation is that the problem of finding appropriate relations can be reduced to the problem of analyzing a simple

imperative program, which encodes each refinement variable with a function whose input-output semantics correspond to the relation described by the refinement variable.

The imperative program derived from a set of constraints C consists of a set of mutually recursive functions, one for each parameterized refinement variable in C , together with a “main” function that checks the safety property. In particular, for the variable κ_i with arity $n + 1$, the imperative program has a function F_i that enjoys the following *function property*: F_i takes n arguments v_1, \dots, v_n and (non-deterministically) returns a value v_0 iff the tuple v_0, \dots, v_n is in the relation corresponding to κ_i in *every* relational model that satisfies C . Following this intuition, an environment binding $x : \kappa_i(y_1, \dots, y_n)$ can be encoded as a function call $x \leftarrow F_i(y_1, \dots, y_n)$ and each lower-bound constraint on $kvar_i$, *i.e.*, where κ_i appears on the RHS can be encoded as a return from F_i after a prefix of instructions that establishes the conditions of the LHS of the lower-bound constraint. Next, we outline the translation $\llbracket C \rrbracket$, summarized in Figure 3.

Functions from Refinement Variables. Figure 4 shows the imperative program translated from the constraints for our running example. There are two functions F_1 and F_2 , corresponding to the refinement variables κ_1 and κ_2 . The function F_1 encodes the function property for κ_1 . The formals z_1, z_2 encode the second and third elements of the relation κ_1 . The return value encodes the first element of the relation κ_1 . The body of the function is the non-deterministic choice between a set of two blocks which encode κ_1 ’s lower-bound constraints (c1) and (c2) respectively. Similarly, the function F_2 encodes the function property for κ_2 , via a single block that encodes κ_2 ’s only lower-bound constraint (c4).

Bound Translation. The translation gathers all the constraints whose RHS have concrete refinements into a set

$$C \downarrow \perp \doteq \{c \in C \mid c \equiv _ \vdash _ \prec p\}$$

and translates these constraints into the entry function f_0 . Intuitively, in such constraints the RHS defines a concrete “upper bound” on the set of tuples that satisfy LHS. In the translated μC program, the entry function enforces the upper bound via `assert` instructions as described below. The main function F_0 , in which execution starts, encodes the concrete-upper-bound (*i.e.*, “bounds-check”) constraint (c3) which stipulates that the value of the variable j is within bounds. The body of F_0 translates the constraint to an assertion over the corresponding variables. As with the other functions, the main function is the non-deterministic choice of all the blocks that encode the individual upper-bound constraints.

Blocks. To ensure that F_i satisfies the function property, we first gather the set $C \downarrow \kappa_i$ of constraints where κ_i appears on the RHS of the constraint. Formally,

$$C \downarrow \kappa_i \doteq \{c \in C \mid c \equiv _ \vdash _ \prec \kappa_i(_)\}$$

Each constraint in the set $C \downarrow \kappa_i$ is individually translated into a block of straight-line assignments and assumes that has the *block property* that the state at the end of the block, maps the formals z_1, \dots, z_n and the return value z_0 to a tuple of values that must belong in every relational model of κ_i that satisfies the constraint. Thus, the body

instruction of F_i , *i.e.*, the choice composition of all the blocks is such that each tuple of inputs and output of F_i belongs in every relational interpretation of κ_i .

To ensure that the translation of each constraint $G \vdash \{x_1 : \beta \mid r_1\} \prec \{x_2 : \beta \mid r_2\}$ in $C \downarrow \kappa_i$ has the block property, we translate the constraint into a straight-line block of instructions with three parts: a sequence of instructions that establishes the environment bindings ($\llbracket G \rrbracket$), a sequence of instructions that “gets” the values corresponding to the LHS ($\llbracket \{x : \beta \mid r_1\}_{get} \rrbracket$) and a sequence of instructions that “sets” the return value of F_i appropriately ($\llbracket \{x : \beta \mid r_2\}_{set} \rrbracket$).

Get Instructions. Each environment binding is encoded as a local variable, and gets translated as a “get” operation that defines the local variable as follows. Bindings with unknown refinements $\kappa_i(x_0, \dots, x_n)$ are translated into calls $x_0 \leftarrow F_i(x_1, \dots, x_n)$ to F_i with arguments x_1, \dots, x_n , with the return value assigned to x_0 . Bindings with concrete refinements p are translated into non-deterministic assignments followed by an `assume` enforcing that the non-deterministically assigned values satisfy p .

Set Instructions. Each RHS refinement is translated into a “set” operation as follows. A concrete refinement p is translated into an `assert` which enforces that the RHS refinement is indeed an upper bound on the values populating the corresponding type in the inclusion constraints. A parameterized refinement $\kappa_i(x_0, \dots, x_n)$ is translated into an `assume` that establishes the equalities between each x_i and the formal z_i representing the i^{th} tuple element, followed by a `return` x_0 . Thus, the translation guarantees that any execution that reaches the end of the block is such that the tuple of values of the return variable and formals of F_i satisfies the constraint to which the RHS refinement (over κ_i) belongs.

Correctness of Translation. The correctness of the key translation step of the HMC algorithm is stated by Theorem 2. Due to lack of space we defer the proof to [11].

Theorem 2. *[Translation] C is satisfiable iff $\llbracket C \rrbracket$ is μC -safe.*

Consider the constraint (c2) which is translated to the second block in F_1 (*i.e.*, the block after the non-deterministic choice $\|$). The (trivial) environment binding $i : \text{int}$, is encoded as a non-deterministic assignment $i \leftarrow \text{nondet}()$ followed by the (elided) `assume` `true`. The (non-trivial) environment binding $\{xs : \alpha \text{ list} \mid 0 \leq \text{len}(xs)\}$ is encoded as

$$xs \leftarrow \text{nondet}(); \text{assume } (0 \leq \text{len}(xs))$$

where in the encoded program xs takes on values of a basic uninterpreted type ui , and len is an uninterpreted function from ui to int . Similarly xs' gets assigned an arbitrary value, that has non-negative length and whose length is 1 less than that of xs . The LHS of (c2) corresponds to the environment binding $j : \kappa_1(j, i + 1, xs')$. Thus, in the encoded block, the local j is defined via a (recursive) call to $F_1(i + 1, xs')$. The block is terminated by returning the value j , after assuming that function parameters z_1 and z_2 equal the tuple elements i and xs of the RHS parameterized refinement, thereby ensuring that the right set of tuples populate corresponding refinement κ_1 .

The HMC Algorithm. The HMC algorithm takes the ML program, generates constraints (Step 1, `Generate(·)`) and translates them into an imperative program (Step 2,

$C \downarrow \kappa$	$\doteq \{c \in C \mid c \equiv _ \vdash _ \prec \kappa_i(_)\}$
$C \downarrow \perp$	$\doteq \{c \in C \mid c \equiv _ \vdash _ \prec p\}$
$\llbracket C \rrbracket$	$\doteq \text{let } \kappa_1, \dots, \kappa_m = \text{Ref. vars. of } C \text{ in}$ $\quad \llbracket 0, 0, C \downarrow \perp \rrbracket,$ $\quad \llbracket 1, \text{arity } \kappa_1, C \downarrow \kappa_1 \rrbracket$ $\quad \dots,$ $\quad \llbracket m, \text{arity } \kappa_m, C \downarrow \kappa_m \rrbracket$
$\llbracket i, a, \{c_1, \dots, c_n\} \rrbracket$	$\doteq f_i(z_1, \dots, z_a) \{ \llbracket c_1 \rrbracket \dots \llbracket c_n \rrbracket \}$
$\llbracket G \vdash \{x_1 : \beta \mid r_1\} \prec \{x_2 : \beta \mid r_2\} \rrbracket$	$\doteq \llbracket G; \{x_1 : \beta \mid r_1\} \rrbracket_{get};$ $\quad \llbracket \{x_1 : \beta \mid r_2[x_1/x_2]\} \rrbracket_{set}$
$\llbracket \{x : \beta \mid r\}; G \rrbracket_{get}$	$\doteq \llbracket \{x : \beta \mid r\} \rrbracket_{get}; \llbracket G \rrbracket_{get}$
$\llbracket \emptyset \rrbracket_{get}$	$\doteq \text{skip}$
$\llbracket \{x : \beta \mid p\} \rrbracket_{get}$	$\doteq x \leftarrow \text{nondet}();$ $\quad \text{assume } p$
$\llbracket \{x_0 : \beta \mid \kappa_i(x_0, \dots, x_n)\} \rrbracket_{get}$	$\doteq x \leftarrow f_i(x_1, \dots, x_n)$
$\llbracket \{x : \beta \mid p\} \rrbracket_{set}$	$\doteq \text{assert } p$
$\llbracket \{x_0 : \beta \mid \kappa(x_0, \dots, x_n)\} \rrbracket_{set}$	$\doteq \text{assume } (\wedge_{j=1}^n x_j = z_j)$ $\quad \text{return } x$

Fig. 3. Translating Constraints To μC Programs

$\llbracket \cdot \rrbracket$). After this, it runs an off-the-shelf abstract interpretation or invariant generation tool on the translated program, and uses the result of this analysis to determine whether the original ML program is safe.

A *safety verifier* V is a procedure that takes an input program and returns Safe or Unsafe. V is *sound* for a language if for each program x in the language, $V(x) = \text{Safe}$ implies that x is safe. HMC converts a verifier for the (first-order, imperative) language μC to a verifier for the (higher-order, functional) language ML in the following way:

$$\text{HMC}(V) \doteq \lambda e. V(\llbracket \text{Generate}(e) \rrbracket)$$

The correctness of HMC follows by combining Theorems 1 and 2.

Theorem 3. [HMC Algorithm] If V is a sound verifier for μC , then $\text{HMC}(V)$ is a sound verifier for ML.

For the translated program shown in Figure 4, the CEGAR-based software model checker ARMC [21] or the abstract interpretation tool INTERPROC [17] finds that the assertion is never violated. From the invariants computed by the tools, we can find solutions to the refinement variables:

$$\kappa_1 \doteq i \leq \nu < i + \text{len}(\text{xs}) \quad \kappa_2 \doteq 0 \leq \nu < \text{len}(\text{a})$$

```

F0 (){
    a ← nondet();
    xs ← nondet(); assume (0 ≤ len(xs));
    j ← F2(a, xs);
    assert (0 ≤ j < len(a));
}

F2 (z1, z2){
    a ← nondet();
    xs ← nondet(); assume (0 ≤ len(xs));
    assume (len(a) = len(xs));
    j ← F1(0, xs);
    assume (z1 = a ∧ z2 = xs);
    return j;
}

F1 (z1, z2){
    i ← nondet();
    xs ← nondet(); assume (0 ≤ len(xs));
    xs' ← nondet(); assume (0 ≤ len(xs') = len(xs) - 1);
    j ← nondet(); assume (j = i);
    assume (z1 = i ∧ z2 = xs);
    return j;
}

||

i ← nondet();
xs ← nondet(); assume (0 ≤ len(xs));
xs' ← nondet(); assume (0 ≤ len(xs') = len(xs) - 1);
j ← F1(i + 1, xs');
assume (z1 = i ∧ z2 = xs);
return j;
}

```

Fig. 4. Translated Program

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

By exploiting the refinement type structure, HMC reduces verification of programs with advanced language features to verification of simple imperative programs that are amenable to analysis by a wide variety of analysis algorithms and tools.

3 Experiments

To demonstrate the feasibility of HMC, we have instantiated it for OCAML with two off-the-shelf imperative verifiers. We use the liquid types types infrastructure implemented in DSOLVE [22] to generate refinement constraints from OCAML programs. The implementation uses OCAML’s implementation of Hindley-Milner type inference to obtain the ML types for each expression, after which the refinement constraints are generated via a syntax-directed pass. Instead of parameterized refinement variables, these constraints have variables with pending substitutions and a separate set of well-formedness (WF) constraints that define the scope of each κ . In a first post-processing step, we use the WF constraints to introduce parametrized refinement variables in place of the pending substitutions. In a second post-processing step, we perform constraint simplifications like constant propagation and resolution.

We use two back-end imperative verifiers to verify the translated programs: ARMC [21], a counterexample-guided software model checker based on predicate abstraction and interpolation-based refinement, and INTERPROC [17], a static analyzer for recursive programs that uses a set of numerical domains such as polyhedra and octagons to compute invariants over numeric variables. In our experiments, we invoked INTERPROC with a polyhedral domain implemented using the Polka library [9]. For each benchmark, the invariants computed by ARMC and INTERPROC could be used to synthesize refinement types for the original source ML program.

Results. Table 1 shows the results of running the two verifiers on suite of small OCAML examples. In addition to the running time, we report the number of predicates discovered by ARMC. The rows with prefix `na_` are a subset of the array manipulating programs from [22], where the safety objective is to prove array accesses are within bounds. The other rows correspond to the benchmark suite used to evaluate the DEPCEGAR verifier [24], where each program contains a set of assertions designed to enforce safety. For each program we created a buggy version that contains a manually inserted safety violation. We observe that despite our blackbox treatment of ARMC and INTERPROC we obtain running times that are competitive with DEPCEGAR on most of the examples (DEPCEGAR uses a customized procedure for unfolding constraints and creating interpolation queries that yield refinement types).

Refinements Discovered. Most of the atomic predicates discovered by ARMC and INTERPROC fall into the two-variables-per-inequality fragment. However, the example MASK from Section 2 required a predicate that refers to three variables, and thus could not be verified using a simpler domain (*e.g.*, octagons). In this case, INTERPROC determined the following relationship between the input and output variables of F_1 and F_2 (after existentially eliminating local variables):

$$\begin{aligned} F_1 :: z_1 \leq \nu \leq z_1 + \text{len}(z_2) - 1 \\ F_2 :: 0 \leq \nu \leq \text{len}(z_1) - 1 \wedge \text{len}(z_1) = \text{len}(z_2) \end{aligned}$$

These invariants are sufficient to show that the assertion in F_0 always holds.

4 Discussions: Completeness

Since the safety verification problem for higher-order programs is undecidable, the sound HMC cannot also 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 refinement constraints.

For example, in our current formulation, we employ a *context insensitive* form of constraint generation where we use the same template for a (monomorphic) function at different call points. 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 [2, 12, 22]. Nevertheless, there can be a loss of information. Consider the program

Table 1. Experimental Results: ARMC (s) denotes the time taken (in seconds) by ARMC to analyze the translated program in its correct and buggy version; DNF indicates that the tool did not finish on the benchmark. ARMC Preds. denotes the number of predicates iteratively found by ARMC in order to verify the safe benchmarks. INTERPROC (s) denotes the time (in seconds) taken by INTERPROC to analyze the translated program in its correct and buggy version; * means INTERPROC was not precise enough to prove all assertions, *i.e.*, raised false alarms.

Program	ARMC (s) correct / buggy	ARMC Preds.	INTERPROC (s) correct / buggy
na_dotprod-m	0.04 / 0.04	2	0.55 / 0.56
na_arraymax-m	0.32 / 0.05	6	0.40 / 0.23
na_bcopy-m	0.09 / 5.94	3	0.33 / 0.38
na_bsearch-m	0.91 / 0.10	11	9.73 / 2.76
na_insertsort	0.03 / 0.03	0	40.11 / 7.38
na_heapsort	DNF / DNF		*27.99 / 28.26
boolflip	0.23 / 0.19	5	0.05 / 0.09
lock	0.03 / 0.03	0	*0.19 / 0.23
mult-cps-m	0.03 / 0.03	0	0.08 / 0.12
mult-all-m	0.03 / 0.03	1	0.13 / 0.07
mult	0.03 / 0.03	2	0.08 / 0.06
sum-all-m	0.03 / 0.03	1	0.10 / 0.08
sum	0.03 / 0.03	2	0.02 / 0.02
sum-acm-m	0.04 / 0.03	2	*0.10 / 0.13

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

For `check`, our constraint generation produces the template

$$(\{x:\text{bool} \mid \kappa_1\} \rightarrow \{\kappa_2\}) \rightarrow \{\kappa_3\} \rightarrow \{\kappa_4\} \rightarrow \text{unit}$$

which is too weak to show safety as the template “merges” the two call sites for `check`. However, we can regain sensitivity via the following *refined intersection type* [5, 6, 15, 18], for `check`:

$$\wedge \begin{array}{l} (x : \text{bool} \rightarrow \{\nu = x\}) \rightarrow \{\neg\nu\} \rightarrow \{\neg\nu\} \rightarrow \text{unit} \\ (x : \text{bool} \rightarrow \{\nu = \neg x\}) \rightarrow \{\neg\nu\} \rightarrow \{\nu\} \rightarrow \text{unit} \end{array}$$

It is important to note that our translation works holds for *any* set of implication constraints (Theorem 2). Thus, one can improve the precision of HMC, by using a more expressive refinement type system to generate the constraints, without having to modify the back-end invariant generation. For example, to recover completeness in the finite-state case, we can use intersection type system of [15] that uses a finite number of “contexts” to generate the implication constraints, after which a finite-state checker *e.g.*, BEBOP [1] would suffice to give a complete verification procedure. (We omit this in our current implementation as there can be a super-exponential number of implication constraints, and the relational refinements were sufficient for our experiments.)

References

1. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL, pp. 1–3. ACM, New York (2002)
2. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffei, S.: Refinement types for secure implementations. In: CSF (2008)
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI, pp. 196–207 (2003)
4. Cui, S., Donnelly, K., Xi, H.: ATS: A language that combines programming with theorem proving. In: FroCos (2005)
5. Dunfield, J.: A Unified System of Type Refinements. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (2007)
6. Freeman, T., Pfenning, F.: Refinement types for ML. In: PLDI (1991)
7. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL 2004. ACM, New York (2004)
8. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL (2002)
9. Jeannet, B., Miné, A.: APRON: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009)
10. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surveys (2009)
11. Jhala, R., Majumdar, R., Rybalchenko, A.: Refinement type inference via abstract interpretation. CoRR, abs/1004.2884 (2010)
12. Kawaguchi, M., Rondon, P., Jhala, R.: Type-based data structure verification. In: PLDI, pp. 304–315 (2009)
13. Knowles, K., Flanagan, C.: Type reconstruction for general refinement types. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 505–519. Springer, Heidelberg (2007)
14. Knowles, K.W., Flanagan, C.: Hybrid type checking. ACM TOPLAS 32(2) (2010)
15. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: POPL (2009)
16. Kobayashi, N., Ong, C.-H.L.: A type system equivalent to modal μ -calculus model checking of recursion schemes. In: LICS (2009)
17. Lalire, G., Argoud, M., Jeannet, B.: Interproc, <http://bit.ly/8Y310m>
18. Naik, M., Palsberg, J.: A type system equivalent to a model checker. ACM Trans. Program. Lang. Syst. 30(5) (2008)
19. Ong, C.-H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS (2006)
20. Ou, X., Tan, G., Mandelbaum, Y., Walker, D.: Dynamic typing with dependent types. In: IFIP TCS, pp. 437–450 (2004)
21. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: PADL (2007)
22. Rondon, P., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI (2008)
23. Rondon, P., Kawaguchi, M., Jhala, R.: Low-level liquid types. In: POPL (2010)
24. Terauchi, T.: Dependent types from counterexamples. In: POPL. ACM, New York (2010)
25. Unno, H., Kobayashi, N.: Dependent type inference with interpolants. In: PPDP (2009)
26. Xi, H., Pfenning, F.: Dependent types in practical programming. In: POPL (1999)