HMC: Verifying Functional Programs Using Abstract Interpreters

Ranjit Jhala[1], Rupak Majumdar[2], and Andrey Rybalchenko[3]

[1]UC San Diego [2]MPI-SWS [3]TU München

Abstract. We present Hindley-Milner-Cousots (HMC), an algorithm that allows any interprocedural analysis for first-order imperative programs to be used to verify safety properties of typed higher-order functional 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 that is safe iff the constraints are satisfiable. Thus, in one swoop, HMC makes tools for invariant generation, *e.g.,* based on abstract domains, predicate abstraction, counterexample-guided refinement, and Craig interpolation be directly applicable to verify safety properties of modern functional languages in a fully automatic manner. We have implemented HMC and describe preliminary experimental results using two imperative checkers – ARMC and INTERPROC– to verify OCAML programs. Thus, by composing type-based reasoning grounded in program syntax and state-based reasoning grounded in abstract interpretation, HMC opens the door to 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, which traditionally capture only coarse data-type properties (*e.g.,* functions expecting `int`s are only passed `int`s), but not precise semantic properties (*e.g.,* the second argument of a division is non-zero, or an array index is within bounds).

For first-order imperative programming languages, automatic tools based on abstract interpretation, such as ASTREE [4], SLAM [2], BLAST [11], *etc.*, can infer program invariants and prove many semantic properties of practical interest. Most of these systems faithfully model the sematics of base values like `int`s, but 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 without additional programmer annotations. In particular, our algorithm allows any verifier for first-order imperative programs (*e.g.,* C) to be used for verifying safety properties of typed higher-order functional programs (*e.g.,* ML). Thus, in one swoop, HMC makes first-order program verifiers based on

abstract interpretation (*e.g.,* [4,21]), CEGAR (*e.g.,* [2,12,13,22,28]), invariant generation (*e.g.,* [10, 31]), *etc.* directly applicable to the verification of safety properties of modern higher-order languages in a fully automatic manner.

To get from ML to C, our HMC algorithm uses a path paved by the notion of *refinement type checking* [3, 18, 25, 34], 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 $\{\texttt{x:int} \mid \texttt{x} < 100\}$ `list` describes a list of integers, each of which is smaller than 100 and $\texttt{int} \rightarrow \{\texttt{x:int} \mid \texttt{x} \neq 0\} \rightarrow \texttt{int}$ describes the integer division function whose second (curried) argument must be non-zero. By piggybacking atop type-structure, refinements can express sophisticated data structure invariants as well [6, 8, 16]. While refinement type checking can be used to verify functional programs, the programmer must manually provide the refinements which is analogous to the burden of writing loop-invariants and pre- and post-conditions in the imperative setting. HMC eliminates the need for programmer annotations and thereby enables automatic checking via a two-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* which represent the unknown refinement types for various program expressions. The templates employ *refinement variables* κ as placeholders for the unknown refinement predicates 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 κ implication constraints, whose satisfaction implies program safety [17, 29].

Step 2: Constraint Translation. Next, HMC transforms the implication constraints into a first-order imperative program that is safe iff 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 (*i.e.,* sets of tuples of) 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 $\texttt{F}_\kappa$ whose input-output semantics (*i.e.,* the sets of tuples of $n-1$ inputs and output of the function) correspond to an n-ary relation that satisfies the constraints on κ. Using these insights we design an algorithm that translates type-bindings into function calls, implications into assigments and assumes, yielding a first-order imperative program that is safe iff 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. The most immediate dividend of our approach is that HMC allows one to apply any of the

well-developed semantic imperative program analyses 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, HMC allows us to apply any existing analysis "as is".

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. In particular, 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, booleans, and strings. Thus, HMC provides a simple way to incorporate independent progress in type systems for specifying complex control and 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 a more or less precise abstract domain (*e.g.,* using polyhedra instead of octagons), as needed in a given application domain. Moreover, the constraint translation is entirely independent of the source language, and hence, HMC can be applied to any language for which suitable refinement constraints can be generated, e.g., ML [29] and C [30].

To demonstrate the feasibility of our approach, we have developed two OCAML safety verifiers – HMC(ARMC) and HMC(INTERPROC) – which use the CEGAR-based ARMC [28] software model checker and the Polyhedra-based INTERPROC [21] 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 [34] or their constituent predicates [29]), or an elaborate customization and adaptation of the counterexample-guided abstraction refinement paradigm [32]. Thus, we show, for the first time, how abstract interpretation can be lifted "as-is" to the automatic safety verification of modern, higher-order functional languages.

Related Work. Our starting point was the vast body of work in the verification of imperative programs (see, e.g., [15] for a survey), including tools such as Slam [2], Blast [12], and Astree [4], and to "lift" the techniques to higher-order programming languages. We were influenced by work on refinement types [9, 17] implemented in dependent ML [34] and, more recently, combined with predicate abstraction [16, 29], but wanted to eliminate the need for explicit annotations (or predicates).

Kobayashi [19, 20] 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) [24], which has been augmented to perform CEGAR [32, 33]. For safety verification, HMC shows a promising alternative, enabling us to use the vast literature on invariant generation for first order programs (using abstract interpreters or model checkers).

While we restrict to a simple input language for ease of explanation, our constraint language is generic and can express refinement constraints arising out of quite expressive source languages, such as the source languages used in liquid types [29], F9 [3], or

```
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

C [30], 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 begin with an example that illustrates how HMC reduces safety verification of ML programs with polymorphism, higher-order functions and recursive structures to safety verification of first-order, imperative programs.

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 $\mathtt{j}^{th}$ 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, $0 \leq \mathtt{j} < \mathtt{len(a)}$. 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 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, 17, 29]. The predicates are additional constraints that must be satisfied by every value of the type. 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,

$\{\nu{:}\texttt{int} \mid 0 \leq \nu < \texttt{len(a)}\}$ denotes the set of integers that are between 0 and the value of the expression `len(a)`. Thus, the (unrefined) type `int` abbreviates $\{\nu{:}\texttt{int} \mid \mathit{true}\}$. 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, $\{\texttt{x:int} \mid 0 \leq \texttt{x}\} \rightarrow \{\nu{:}\texttt{int} \mid \nu = \texttt{x} + 1\}$ 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 $\texttt{x:}\beta$ for the type $\{\texttt{x:}\beta \mid \mathit{true}\}$, and $\texttt{x:}r$ for $\{\texttt{x:}\beta \mid r\}$, 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) <- e`) in `g` which is an abbreviation for `get a j` (resp. `set a j e`.) By giving `get` and `set` the types

$$\texttt{a:}\alpha\ \texttt{array} \rightarrow \{\texttt{i:int} \mid 0 \leq \texttt{i} < \texttt{len(a)}\} \rightarrow \alpha\ ,$$
$$\texttt{a:}\alpha\ \texttt{array} \rightarrow \{\texttt{i:int} \mid 0 \leq \texttt{i} < \texttt{len(a)}\} \rightarrow \alpha \rightarrow \texttt{unit}\ ,$$

we can specify that in any program the array accesses must be within bounds. More generally, arbitrary safety properties can be specified [29] by giving `assert` the refinement type $\{\texttt{p:bool} \mid \texttt{p} = \mathit{true}\} \rightarrow \texttt{unit}$.

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 $\texttt{j:int} \rightarrow \texttt{y:bool} \rightarrow \texttt{unit}$, *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 [3, 8, 34] 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:

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

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 an equivalent *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 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.

Step 1: Constraint Generation First, we generate a system of refinement constraints for the source program [17, 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.

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

$$\texttt{i:int} \rightarrow \texttt{xs:}\alpha\ \texttt{list} \rightarrow (\texttt{j:int} \rightarrow \alpha \rightarrow \texttt{unit}) \rightarrow \texttt{unit}$$
$$\texttt{j:int} \rightarrow \texttt{bool} \rightarrow \texttt{unit}$$

we generate the templates

$$\texttt{i:int} \rightarrow \texttt{xs:}\{0 \leq \texttt{len(xs)}\} \rightarrow (\texttt{j:}\kappa_1(\texttt{j},\texttt{i},\texttt{xs}) \rightarrow \alpha \rightarrow \texttt{unit}) \rightarrow \texttt{unit}$$
$$\texttt{j:}\kappa_2(\texttt{j},\texttt{a},\texttt{xs}) \rightarrow \texttt{bool} \rightarrow \texttt{unit}$$

The templates refine the ML types with *parameterized refinement variables* that represent the unknown refinements. $\kappa_1(\texttt{j},\texttt{i},\texttt{xs})$ 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`. $\kappa_2(\texttt{j},\texttt{a},\texttt{xs})$ 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. After creating the templates, we make a syntax-directed pass over the program to generate constraints that 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 $G \vdash T_1 \prec T_2$, 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 `iteri`. The call to `f` generates

$$G \vdash \{\nu\texttt{:int} \mid \nu = \texttt{i}\} \prec \{\nu\texttt{:int} \mid \kappa_1(\nu,\texttt{i},\texttt{xs})\} \tag{c1}$$

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

$$G \doteq \texttt{i:int};\ \{\texttt{xs:}\alpha\ \texttt{list} \mid 0 \leq \texttt{len(xs)}\};\ \texttt{x:}\alpha;$$
$$\{\texttt{xs}'\texttt{:}\alpha\ \texttt{list} \mid 0 \leq \texttt{len(xs}') = \texttt{len(xs)} - 1\}$$

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 $\texttt{xs}'$ is one less than `xs` as the former is the tail of the latter [16, 34]. Similarly, the recursive call to `iteri` generates

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

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

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

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

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

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

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

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

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

We prove that (Theorem 1) if the set of constraints (c1), (c2),(c3), and (c4) is satisfiable, then there is a valid refinement typing of the program, and hence the program is safe.

Step 2: Translation to Imperative Program The constraints generated in Step 1 encode the semantics of program computations. In the second step, we reduce 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 refinement variables in the constraints stand for *relations* between program variables: the set of values denoted by a refinement type $\{\texttt{x}_0{:}\beta_0 \mid p\}$ where p is a predicate that refers to the program variables $\texttt{x}_0, \ldots, \texttt{x}_n$ of base types $\beta_0, \ldots, \beta_n$ is equivalent to

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

where R_p is an $(n+1)$-ary relation in $\beta_0 \times \ldots \times \beta_n$ defined by p. For example, $\{\nu{:}\texttt{int} \mid \nu \leq \texttt{i}\}$ is equivalent to the set $\{t_0 \mid \exists t_1 \text{ s.t. } (t_0, t_1) \in R_{\leq} \wedge t_1 = \texttt{i}\}$, where $R_{\leq}$ is the standard $\leq$-ordering relation over the integers. In other words, each parameterized refinement variable $\kappa(\texttt{x}_0, \ldots, \texttt{x}_n)$ can be seen as the projection on the first coordinate of a $(n+1)$-ary relation over the variables $(\texttt{x}_0, \ldots, \texttt{x}_n)$. 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. In particular, for each parameterized refinement variable κ_i with arity $n+1$, the imperative program has a function $\texttt{F}_i$ that enjoys the *function property*: $\texttt{F}_i$ takes n arguments $v_1, \ldots, v_n$ and (non-deterministically) returns a value v_0 iff the constraints demand that the tuple $v_0, \ldots, v_n$ be in the relation corresponding to κ_i. Following this intuition, an environment binding $\texttt{x}{:}\kappa_i(\texttt{y}_1, \ldots, \texttt{y}_n)$ can be encoded as a function call $\texttt{x} \leftarrow \texttt{F}_i(\texttt{y}_1, \ldots, \texttt{y}_n)$ and each lower-bound constraint on $kvar_i$, *i.e.,* where κ_i appears on the RHS can be encoded as a return from $\texttt{F}_i$ after a prefix of instructions that establishes the conditions of the LHS of the lower-bound constraint.

Functions. Figure 2 shows the imperative program translated from the constraints for our running example. The function $\texttt{F}_1$ encodes the function property for κ_1. The formals $\texttt{z}_1, \texttt{z}_2$ 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 $\texttt{F}_2$ encodes the function property for κ_2, via a single block that encodes κ_2's only lower-bound constraint (c4). The main function $\texttt{F}_0$, 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 $\texttt{F}_0$ 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. Each constraint is encoded as block of instructions Each environment binding is encoded as a local variable. The block has a sequence of assignments that define these local variables. An environment binding that corresponds to a concrete refinement p, is encoded as a non-deterministic assignment followed by an assume operation (a conditional) that establishes that the value assigned satisfied the given refinement p. An environment binding that corresponds to a parameterized refinement $\kappa_j(\texttt{y}_0, \ldots, \texttt{y}_m)$ is encoded as a function call $\texttt{y}_0 \leftarrow \texttt{F}_j(\texttt{y}_1, \ldots, \texttt{y}_m)$. The block is terminated by either a return of the first element of the tuple defined by the lower-bound constraint, or an assert stating that the tuple satisfies the concrete predicate of an upper-bound constraint.

Consider the constraint (c2) which is translated to the second block in $\texttt{F}_1$ (*i.e.,* the block after the non-deterministic choice $[]$). The (trivial) environment binding `i:int`, is encoded as a non-deterministic assignment $\texttt{i} \leftarrow \texttt{nondet}()$ followed by the (elided) assume $\texttt{assume}\ \mathit{true}$. The (non-trivial) environment binding $\{\texttt{xs}{:}\alpha\ \texttt{list} \mid 0 \leq \texttt{len}(\texttt{xs})\}$

is encoded as

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

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 $\texttt{j}{:}\kappa_1(\texttt{j}, \texttt{i}+1, \texttt{xs}')$. Thus, in the encoded block, the local j is defined via a (recursive) call to $\texttt{F}_1(\texttt{i}{+}1, \texttt{xs}')$. The block is terminated by returning the value j, after assuming that function parameters $\texttt{z}_1$ and $\texttt{z}_2$ 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.

HMC Algorithm The HMC algorithm takes the ML program, generates constraints and translates them into an imperative program. After this, we can run any off-the-shelf abstract interpretation or invariant generation tool on the translated program, and use the result of the analysis to determine whether the original ML program is safe.

For the translated program shown in Figure 2, the CEGAR-based software model checker ARMC [28] or the abstract interpretation tool INTERPROC [21] finds that the assertion is never violated. From the invariants computed by the tools, we can find solutions to the refinement variables:

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

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

The attractiveness of the HMC translation is the separation of concerns between the handling of advanced language features (through syntactic subtyping) and of data invariants (through abstract interpretation of imperative programs). This, in particular, implies that the translated programs fall in a particularly pleasant subclass which do not have any advanced language features like higher-order functions, polymorphism, and recursive data structures, or variables over complex datatypes that are the bane of semantic analyses for imperative programs.

In contrast, the HMC algorithm uses type structure to reduce verification of advanced language features to verification of simple imperative programs that are amenable to analysis by a wide variety of analysis algorithms and tools.

3 Formalization

We now formalize the details of the HMC algorithm. We briefly describe constraint generation, which is similar to prior work [29], and focus on the translation to first-order programs which is the main technical contribution of our work. The reader may consult the appendix for complete details.

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.*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

```
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. 2. Translated Program

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 $\sigma[\mathtt{z} \mapsto v]$ for the state that maps the variable $\mathtt{z}$ to v and all other variables y to $\sigma(y)$.

From μML to Constraints In the first step, HMC generates a set of refinement constraints whose satisfaction implies that the program is safe.

Functional Language. The source functional language for HMC is μML, a variant of the λ-calculus with ML-style parametric polymorphism. The language's syntax includes variables, constants, λ-abstractions (functions), applications (calls) and let-bindings (Figure 3). We formalize the eager, call-by-value semantics of μML using the standard small-step operational semantics (Figure 3). To facilitate safety specifications, μML includes a special `assert` function that returns an error value Err when called with the value `false`. Let e be a μML program. We say that e *is* μML*-safe* if there is no derivation of the form $\mathsf{e} \overset{*}{\rightsquigarrow} \mathsf{Err}$, *i.e.,* if e never reduces to Err.

Constraints. A *refinement* r is either a *concrete predicate* p drawn from the refinement logic (Figure 4), or a *parameterized refinement variable* $\kappa(\mathtt{x}_0, \ldots, \mathtt{x}_n)$, 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 $\{\mathtt{x}{:}\beta \mid r\}$ comprising a *variable* x that

is being bound, a base type β describing the base type of $\mathtt{x}$, and a refinement r that describes an invariant satisfied by all the values bound to $\mathtt{x}$. A *refinement environment* G is a sequence of refinement bindings. A *refinement constraint* $G \vdash \{\mathtt{x}{:}\beta \mid r_1\} \prec \{\mathtt{x}{:}\beta \mid r_2\}$ states that when the program variables satisfy the invariants described in G, 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. Figure 5 formalizes the notion of constraint satisfaction. A *relational interpretation* for κ of arity n, is a subset of $\mathcal{U}(\beta)^n$. A *relational model* Σ is a map from refinement variables κ to relational interpretations. Figure 5 formalizes a satisfaction relation between a relational interpretation Σ and an inclusion 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. The first step of HMC is a syntax-directed procedure $\mathsf{Generate}(e)$, summarized in Figure 7 that takes as input a μML program e, and uses the type structure of the program to generate a set of constraints whose satisfiability implies the safety of the program.

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

Our constraint generation process is similar to that of refinement type constraints [3, 9, 17, 29], except for the explicit "refinements-as-relations" view which critically enables the translation in the second step.

From Constraints to μC In the second step, formalized in Figure 8, HMC translates the set of constraints into an imperative program.

Imperative Language. The target imperative language for HMC is μC, a first-order imperative language with a single kind of variables of base type β. An *instruction* is either an assignment, an assume, an assert, or the sequential or non-deterministic (branch) composition of two instructions. A *function* comprises a sequence of formal parameters $\mathtt{z}_1, \ldots, \mathtt{z}_n$, a body instruction $\mathtt{I}$, and a return variable $\mathtt{z}_0$. A *program* is a set of functions including a distinguished *entry* function that takes no arguments. We formalize the semantics of μC using a standard big-step transition relation. Let $\mathtt{P}$ be a μML program. We say that $\mathtt{P}$ *is* μC*-safe* if there is no transition $\mathtt{P}, \mathtt{I}_0 \vdash [\cdot] \hookrightarrow \mathsf{Err}$.

Function Translation. The translation of a set C of constraints maps each refinement variable κ_i to a function $\mathtt{F}_i$. The result of the translation of the refinement variables is a system of mutually recursive functions, as we describe below. Consider a refinement variable $\kappa_i(\mathtt{x}_0, \ldots, \mathtt{x}_n)$. The translated function $\mathtt{F}_i$ has the *function property* that $\mathtt{F}_i(v_1, \ldots, v_n)$ returns v_0 iff *every* relational model that satisfies C maps κ_i to a set that includes the tuple $v_0, \ldots, v_n$.

For the example in Section 2, we create two functions $\mathtt{F}_1$ and $\mathtt{F}_2$ for the refinement variables κ_1 and κ_2.

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

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

and translates these constraints into the entry function $\mathtt{f}_0$. 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.

In Section 2, the function $\mathtt{F}_0$ encodes the constraint (c3).

Block Translation. To ensure that $\mathtt{F}_i$ 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 $\mathtt{z}_1, \ldots, \mathtt{z}_n$ and the return value $\mathtt{z}_0$ to a tuple of values that must belong in every relational model of κ_i that satisfies the constraint. Thus, the body instruction of $\mathtt{F}_i$, *i.e.,* the choice composition of all the blocks is such that each tuple of inputs and output of $\mathtt{F}_i$ belongs in every relational interpretation of κ_i.

To ensure that the translation of each constraint $G \vdash \{\mathtt{x}_1 : \beta \mid r_1\} \prec \{\mathtt{x}_2 : \beta \mid r_2\}$ 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 ($[\![G]\!]$), a sequence of instructions that "gets" the values corresponding to the LHS ($[\![\{\mathtt{x} : \beta \mid r_1\}_{get}]\!]$) and a sequence of instructions that "sets" the return value of $\mathtt{F}_i$ appropriately ($[\![\{\mathtt{x} : \beta \mid r_2\}]\!]_{set}$).

Get Instructions. Each environment binding gets translated as a "get" operation as follows. Bindings with unknown refinements $\kappa_i(\mathtt{x}_0, \ldots, \mathtt{x}_n)$ are translated into calls to $\mathtt{F}_i$ with arguments $\mathtt{x}_1, \ldots, \mathtt{x}_n$, with the return value assigned to $\mathtt{x}_0$. 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 `assume` 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 $\kappa_i(\mathtt{x}_0, \ldots, \mathtt{x}_n)$ is translated into an `assume` that establishes the equalities between each x_i and the formal $\mathtt{z}_i$ representing the i^{th} tuple element, followed by a `return` $\mathtt{x}_0$. 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 $\mathtt{F}_i$ satisfies the constraint to which the RHS refinement (over κ_i) belongs.

In Figure 2, the function $\mathtt{F}_1$ encodes the two constraints (c1), (c2) as a nondeterministic choice between their block translations. For each constraint, its block consists of the "get" and "set" operations as described above. The following theorem formalizes the correctness of $[\![C]\!]$.

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

The HMC Algorithm We combine the the constraint generation and translation procedures to obtain the HMC algorithm. 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, $\mathsf{V}(x) = \mathsf{Safe}$ 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 μML in the following way:

$$\mathrm{HMC}(\mathsf{V}) \doteq \lambda e.\mathsf{V}([\![\mathsf{Generate}(e)]\!])$$

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

Theorem 3. *[*HMC *Algorithm] If* V *is a sound verifier for* μC*, then* HMC(V) *is a sound verifier for* μML*.*

4 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 [29] 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 similar to Figure 7. 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 [28], a counterexample-guided software model checker based on predicate abstraction and interpolation-based refinement, and INTERPROC [21], 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 [14]. 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 [29], 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 [32], 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

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

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; *_ indicates that INTERPROC was not precise enough to prove all assertions, *i.e.,* raised false alarms.

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 $\mathtt{F}_1$ and $\mathtt{F}_2$ (after existentially eliminating local variables):

$$\mathtt{F}_1 :: \mathtt{z}_1 \leq \nu \leq \mathtt{z}_1 + \mathtt{len}(\mathtt{z}_2) - 1$$
$$\mathtt{F}_2 :: 0 \leq \nu \leq \mathtt{len}(\mathtt{z}_1) - 1 \wedge \mathtt{len}(\mathtt{z}_1) = \mathtt{len}(\mathtt{z}_2)$$

These invariants are sufficient to show that the assertion in $\mathtt{F}_0$ always holds.

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. 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.
5. M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*. Springer, 2003.
6. S. Cui, K. Donnelly, and H. Xi. Ats: A language that combines programming with theorem proving. In *FroCos*, 2005.
7. L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, 1982.
8. J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2007.
9. T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, 1991.
10. A. Gupta and A. Rybalchenko. InvGen: An efficient invariant generator. In *CAV*, 2009.
11. T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04*. ACM, 2004.
12. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
13. H. Jain, F. Ivancic, A. Gupta, and M. K. Ganai. Localization and register sharing for predicate abstraction. In *TACAS*, 2005.
14. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, pages 661–667, 2009.
15. R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surveys*, 2009.
16. M. Kawaguchi, P. Rondon, , and R. Jhala. Type-based data structure verification. In *PLDI*, pages 304–315, 2009.
17. K. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *ESOP*, 2007.
18. K.W. Knowles and C. Flanagan. Hybrid type checking. *ACM TOPLAS*, 32(2), 2010.
19. N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL*, 2009.
20. N. Kobayashi and C.-H.L. Ong. A type system equivalent to modal μ-calculus model checking of recursion schemes. In *LICS*, 2009.
21. G. Lalire, M. Argoud, and B. Jeannet. Interproc. http://bit.ly/8Y310m.
22. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*. 2006.
23. M. Naik and J. Palsberg. A type system equivalent to a model checker. *ACM Trans. Program. Lang. Syst.*, 30(5), 2008.
24. C.-H.L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, 2006.
25. X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP TCS*, pages 437–450, 2004.
26. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
27. B. C. Pierce and D. N. Turner. Local type inference. In *POPL*, pages 252–265, 1998.
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. P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, 2010.
31. S. Sankaranarayanan, H.B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, 2005.
32. T. Terauchi. Dependent types from counterexamples. In *POPL*. ACM, 2010.
33. H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP*, 2009.
34. H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.

A Definitions

In this section, we formally define the key players of HMC, namely a *source* higher-order functional language, an *intermediate* logical constraint language, and a *target* first-order imperative language. For ease of exposition, we restrict the description to a minimal set of language features. However, our implementation deals with other language features, such as those described in Section 2.

Base Types. Our language has a fixed set of *base types*, comprising `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, recursive types *etc.* In the sequel, for ease of exposition, we assume that there is a single base type β containing the universe of base values $\mathcal{U}(\beta)$.

Variables and States. 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 $\mathcal{U}(\beta)$. We lift states to maps from expressions to values and predicates to boolean values in the standard manner. We write $[\cdot]$ for the state whose domain is empty, and write $\sigma[\mathtt{z} \mapsto v]$ for the state that maps the variable `z` to v and all other variables y to $\sigma(y)$.

A.1 μML: Higher-Order Functional Language

We start by formalizing the syntax and semantics of μML, a variant of the λ-calculus with ML-style polymorphism.

Terms. An expression `e` of μML is generated by the grammar

$$\mathtt{e} ::= \mathtt{x} \mid \mathtt{c} \mid \lambda x.\mathtt{e} \mid \mathtt{e}\ \mathtt{e} \mid \mathtt{let\ x} = \mathtt{e\ in\ e}$$

where `x` ranges over variables and `c` over constants. Terms $\lambda x.\mathtt{e}$ are called λ-*abstractions*, `e e` called *function applications*, and $\mathtt{let\ x} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2$ `let`−bindings. We discuss μML types in Section B.1.

Constants. The basic units of computation are the constants `c` built into μML, which include *base constants*, corresponding to integers and boolean values, and *primitive functions*, which encode various operations like addition, comparison, and so on. Recursive functions are expressed via the fixpoint combinator constant `fix`. Thus, an OCAML-style let-rec binding $\mathtt{let\ rec\ f\ x} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2$ is expressed in μML as $\mathtt{let\ f} = \mathtt{fix}\ (\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{e}_1)\ \mathtt{in}\ \mathtt{e}_2$. Safety properties are expressed via the constant `assert` which is a primitive operation that takes as input a boolean and returns the input if it is `true` and returns the error value Err otherwise.

Semantics. The call-by-value dynamic semantics of μML are formalized using the standard small-step (contextual) operational semantics, whose rules are shown in Figure 3. We write $\rightsquigarrow$ for the single evaluation step relation for μML expressions, and write $\overset{*}{\rightsquigarrow}$ to describe the reflexive, transitive closure of $\rightsquigarrow$.

$$
\begin{array}{lll}
v ::= & & \textbf{Values:} \\
& \mid \mathtt{c} & \text{constants} \\
& \mid \lambda \mathtt{x.e} & \lambda\text{-terms} \\
C ::= & & \textbf{Contexts:} \\
& \mid \bullet & \text{hole} \\
& \mid C\ e & \text{application left} \\
& \mid v\ C & \text{application right} \\
& \mid \mathtt{let\ x} = C\ \mathtt{in\ e} & \text{let-binding}
\end{array}
$$

μML **Transition Relation** $\boxed{\mathtt{e} \rightsquigarrow \mathtt{e}'}$

$$
\frac{v \in \mathsf{Dom}([\![\mathtt{c}]\!])}{\mathtt{c}\ v \rightsquigarrow [\![\mathtt{c}]\!](v)}\ [\textsc{Con}] \qquad \frac{v \notin \mathsf{Dom}([\![\mathtt{c}]\!])}{\mathtt{c}\ v \rightsquigarrow \mathsf{Err}}\ [\textsc{Con-E}]
$$

$$
\frac{}{(\lambda \mathtt{x.e})\ v \rightsquigarrow \mathtt{e}[v/\mathtt{x}]}\ [\textsc{App}] \qquad \frac{}{\mathtt{let\ x} = v\ \mathtt{in\ e} \rightsquigarrow \mathtt{e}[v/\mathtt{x}]}\ [\textsc{Let}]
$$

$$
\frac{\mathtt{e} \rightsquigarrow \mathtt{e}'}{C[\mathtt{e}] \rightsquigarrow C[\mathtt{e}']}\ [\textsc{Ctx}] \qquad \frac{\mathtt{e} \rightsquigarrow \mathsf{Err}}{C[\mathtt{e}] \rightsquigarrow \mathsf{Err}}\ [\textsc{Ctx-E}]
$$

Fig. 3. μML **Syntax and Semantics**

Safety. The set of values includes the constants, functions, and a special Err value. If a constant is applied to a value that is not in the domain of the constant (*e.g.,* calling division with zero as the second parameter, or calling `assert` with `false`), then the application reduces to Err. Let e be a μML program. We say that e *is* μML*-safe* if there is no derivation of the form $\mathtt{e} \overset{*}{\rightsquigarrow} \mathsf{Err}$. In other words, a μML program is safe if it never reduces to Err.

A.2 Refinement Constraints

Next, we formalize logical refinement constraints, by defining the syntax of refinements and constraints and the semantics of constraint satisfaction.

Syntax

Expressions and Predicates. Figure 4 shows the syntax of refinement predicates. We leave the concrete syntax of expressions *expressions* e unspecified, but assume that function applications are treated as applications of uninterpreted functions. For example, in our implementation, e comprised terms in linear arithmetic together with uninterpreted functions. *Predicates* comprise atomic comparisons of expressions, or boolean combinations of (sub-)predicates.

Refinements. A *refinement* r is either a *concrete predicate* p drawn from the refinement logic, or a *parameterized refinement variable* $\kappa(\mathtt{x}_0, \ldots, \mathtt{x}_n)$, where κ is a refinement variable of arity n. We assume, without loss of generality, that each κ has a fixed arity. That is, in a given set of constraints, every occurrence of κ is parameterized with exactly

e	$::= \ldots$	**Expressions**
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
	$\mid \kappa(\mathtt{x}_0, \ldots, \mathtt{x}_n)$	param. reft. variable
β	$::=$	**Types:**
	$\mid \mathtt{int}$	base type of integers
	$\mid \mathtt{bool}$	base type of booleans
	$\mid \mathtt{ui}$	unint. type
ρ	$::= \{\mathtt{x}{:}\beta \mid r\}$	**Refined Types**
G	$::=$	**Environments:**
	$\mid G;\rho$	binding
	$\mid \emptyset$	empty
c	$::= G \vdash \{\mathtt{x}_1{:}\beta \mid r_1\} \prec \{\mathtt{x}_2{:}\beta \mid r_2\}$	**Constraints**

Fig. 4. Predicates, Refinements and Constraints.

$n + 1$ arguments. Concrete predicates represent *known* invariants, while the parameterized variables represent *unknown* invariants that hold of different program values. The notion of parameterized refinement variables 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. This notion is closely related to the notion of variables with pending substitutions [1, 17] and with invariant templates [5].

Refinement Type Bindings and Environments. A *refinement type binding* ρ is a triple $\{\mathtt{x}{:}\beta \mid r\}$ comprising a *variable* x that is being bound, a base type β describing the base type of $\mathtt{x}$, and a refinement r that describes an invariant satisfied by all the values bound to $\mathtt{x}$. A *refinement environment* G is a sequence of refinement bindings.

A refinement type binding (or just refinement type) describes the set of concrete values of the underlying type which additionally satisfy the refinement predicate. For example, $\{\mathtt{x}{:}\mathtt{int} \mid \mathtt{x} \neq 0\}$ says $\mathtt{x}$ is bound to a value from the set of non-zero integers, and $\{\mathtt{a}{:}\mathtt{int} \mid \mathtt{a} < \mathtt{x} + \mathtt{y}\}$ says $\mathtt{a}$ is bound to a value from the set of integers less than the sum of (the values bound to) $\mathtt{x}$ and $\mathtt{y}$.

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 $\{\mathtt{x}_b{:}\mathtt{bool} \mid \mathtt{x} > 100\}$.

Constraints. Figure 4 shows the syntax of constraints. Informally,

$$G \vdash \{\mathtt{x}{:}\beta \mid r_1\} \prec \{\mathtt{x}{:}\beta \mid r_2\}$$

Predicates	$\sigma \models p$	iff $\sigma(p) = \mathit{true}$
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 $\sigma \models p$
	$\Sigma, \sigma \models \{x{:}\beta \mid \kappa(y_0, \ldots, y_n)\}$	iff $(\sigma(y_0), \ldots, \sigma(y_n)) \in \Sigma(\kappa)$
Constraints	$\Sigma \models G \vdash \{\mathtt{x}_1{:}\beta \mid r_1\} \prec \{\mathtt{x}_2{:}\beta \mid r_2\}$	iff For all σ : $\Sigma, \sigma \models G;\{\nu{:}\beta \mid r_1\}$ implies $\Sigma, \sigma \models \{\mathtt{x}_1{:}\beta \mid r_2[\mathtt{x}_1/\mathtt{x}_2]\}$

Fig. 5. Constraint Satisfaction

states that when the program variables satisfy the invariants described in G, the set of values described by the refinement r_1 *must be included in* the set of values described by the refinement r_2.

Semantics Refinements represent known and unknown relations between program variables at different places in the program. The inclusion constraints describe (set) inclusion relationships that must hold between different relations. Next, we crystallize this intuition, and define the semantics of constraints, by formally defining the notion of constraint satisfaction.

Relational Interpretations. A *relational interpretation* for κ of arity n, is a subset of $\mathcal{U}(\beta)^n$. A *relational model* Σ is a map from refinement variables κ to relational interpretations. In the sequel, we only consider relational models that map each κ to a relation of the arity of κ.

Satisfaction. Figure 5 formalizes the notion of when a relational interpretation Σ *satisfies* an inclusion 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.

A.3 μC: A First-Order Imperative Language

We conclude this section by formalizing the syntax and semantics of μC, a core, first-order imperative language.

Syntax A μC program has a single kind of *base* variables drawn from X, which range over values of type β. An *instruction* (`I`) is either an assignment $\mathtt{x} \leftarrow e$, an assume `assume` p, an assert `assert` p using predicates over the base variables (cf. Figure 4),

or the sequencing $\mathtt{I};\mathtt{I}$ or non-deterministic choice $\mathtt{I}[\!]\mathtt{I}$ of two instructions. We write `skip` as an abbreviation for `assume` *true*. An *assignment* to a target variable is of one of three kinds. Either (1) $\mathtt{x} \leftarrow e$: an expression e over the variables (cf. Figure 4) is evaluated and assigned to the target variable $\mathtt{x}$, or, (2) $\mathtt{x} \leftarrow \mathtt{nondet}()$: an arbitrary non-deterministically chosen value of the appropriate base type is assigned to $\mathtt{x}$, *i.e.,* the target variable is "havoc-ed", or (3) $\mathtt{x} \leftarrow \mathtt{F}(\mathtt{y}_1, \ldots, \mathtt{y}_n)$: a function $\mathtt{F}$ is called, and its return value is assigned to the target variable $\mathtt{x}$. A *function definition* ($\mathtt{F}$) has a name $\mathtt{F}$, sequence of formal parameters $\mathtt{z}_1, \ldots, \mathtt{z}_n$, a function body $\mathtt{I}$, and a return variable $\mathtt{z}_0$. A *program* ($\mathtt{P}$) is a set of functions $\mathtt{F}_0, \ldots, \mathtt{F}_m$, where $\mathtt{F}_0$ is a distinguished *entry function* that takes no arguments.

Next, we formalize the big-step operational semantics of μC. The big-step formulation avoids the need for adding an explicit stack to the semantics, and simplifies the exposition of the proofs of the main equivalence theorems.

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".

Transitions. μC is a standard lexically scoped, imperative language with call-by-value semantics. The transition relation is described by the judgment $\mathtt{P}, \mathtt{I} \vdash \sigma \hookrightarrow \sigma'$ that stipulates that in the program $\mathtt{P}$, the execution of the instruction $\mathtt{I}$ 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 $\mathtt{z}_0$. The assume instruction proceeds without updating the state only if the corresponing predicate holds (and otherwise, the program halts). Thus, μC eschews if-then-else instructions in favor of the 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.) We give the formal details in Figure 10 in the Appendix.

Safety. Let $\mathtt{P}$ be an μC program, whose entry function $\mathtt{F}_0$ has the body $\mathtt{I}_0$. We say that $\mathtt{P}$ *is* μC*-safe* if there is no transition $\mathtt{P}, \mathtt{I}_0 \vdash [\cdot] \hookrightarrow \mathsf{Err}$.

B Algorithm

In this section, we describe how we reduce the safety of a μML program to a set of constraints (Section B.1) and how we reduce the satisfiability of constraints into safety of an μC program (Section B.2). Together the steps combine to yield the HMC Algorithm (Section B.3).

$\mathtt{e}$	$::=$	$\ldots$	**Typed Expressions:**
		$\mid \mathtt{e}{:}\tau$	annotation
		$\mid [\Lambda\alpha]\mathtt{e}$	type-abstraction
		$\mid [\beta]\mathtt{e}$	type-instantiation
$\mathbb{T}(\mathbb{B})$	$::=$		**Type Skeletons:**
		$\mid \mathbb{B}$	base
		$\mid \alpha$	type variable
		$\mid \mathbb{T}(\mathbb{B}) \rightarrow \mathbb{T}(\mathbb{B})$	function
$\mathbb{S}(\mathbb{B})$	$::=$		**Schema Skeletons:**
		$\mid \mathbb{T}(\mathbb{B})$	monotype
		$\mid \forall\alpha.\mathbb{S}(\mathbb{B})$	polytype
τ, σ	$::=$	$\mathbb{T}(\beta), \mathbb{S}(\beta)$	**Types, Schemas**
T, S	$::=$	$\mathbb{T}(\rho), \mathbb{S}(\rho)$	**Templates, Schemas**

Fig. 6. μML: Syntax of Types and Typed Terms

B.1 Step 1: From μML to Constraints

The first step of HMC is a syntax-directed procedure that takes as input a μML program, and uses the type structure of the program to generate a set of constraints whose satisfiability implies the safety of the program. Our constraint generation process is similar to that of refinement type constraints [9, 17, 29], except for the explicit "refinements-as-relations" view which critically enables the translation in the second step, and facilitates the proof of our main technical reduction (Theorem 2).

Refinement Types and Templates

Types. Figure 6 shows the syntax of types (and schemas) of μML, which includes base types β, type variables α, function types (and quantification.) The constraint generation procedure receives as input typed μML programs from the grammar for typed terms shown in Figure 6. We restrict our attention to terms are well-typed according to the standard type-checking rules [26]. Thus, we assume that a classical Hindley-Milner style type inference procedure [7] has inserted appropriate type annotations ($\mathtt{e}{:}\tau$) for each (sub-) term and type generalization ($[\Lambda\alpha]\mathtt{e}$) and instantiation ($[\tau]\mathtt{e}$) operations.

Refinement Types and Subtyping. The constraint generation procedure of HMC is based on the notion of *refinement types* [3, 9, 17, 29] which can be viewed as a type-based analogue of the classical notions of program invariants. Just as invariants represent overapproximations of the states that reach a particular program counter, refinement types represent overapproximations of the set of values to which a particular program expression can evaluate. With invariants, overapproximation is formally established by implication (or subsumption, in the "consequence" rule of Hoare logic). With refinement types, overapproximation is formalized via subtyping. A more thorough discussion of refinement types is beyond the scope of this paper, we refer the reader to [3, 18, 29] for details.

Safety. With classical invariants, safety is established via the validity of a verification condition that checks subsumption between the invariants at different program points. Analogously, with refinement types, safety is established via the the validity of a type derivation that checks subtyping between the types of different sub-expressions. Thus, the main challenge towards automating verification is the inference of appropriate refinements for various program expressions (akin to the challenge of inferring loop invariants).

Templates. Our verification strategy is to create templates that represent the unknown refinement types for various program expressions, and to then traverse the program in a syntax-directed manner, to compute types for other expressions in terms of the templates, and to generate constraints that enforce subtyping between relevant sub-expressions. Figure 6 shows the syntax of *templates* (T) and template *schemas* S. In essence, a template (resp. schema) is a type (resp. schema) with a refinement binding occurring at each of the "leaves". A *template environment* is a map Γ from program variables to templates.

Constraint Generation

Fresh Templates. Procedure $\mathsf{New}(\Gamma, \tau)$, shown in Figure 7, takes as input a template environment Γ and a μML type τ and generates a new template for the unknown refinement type of a value of type τ that contains relations over variables in scope in Γ. For basic types, $\mathsf{New}(\Gamma, \tau)$ first calls $\mathsf{Env}(\Gamma)$ which maps a template environment to a refinement environment by re-binding base variables, and filtering away the complex template bindings. the generation procedure returns a fresh refinement variable κ parameterized with the value variable $\mathtt{x}_0$ and the variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$ bound in the filtered environment. For complex types, the procedure recursively generates fresh templates for the components of the type. For functions whose input has a base type, the binding for the input is added to the environment for the output, which allows the output refinement to refer to, *i.e.,* be related to, the input.

Subtyping Constraints. Procedure $\mathsf{Sub}(\Gamma,\ T_1,\ T_2)$, shown in Figure 7, takes as input a template environment Γ, and two templates T_1 and T_2 and returns as output the set of inclusion constraints that must be satisfied so that T_1 is a subtype of T_2 in the environment Γ. The procedure uses classical type-theoretic subsumption rules (contravariant arguments, covariant return values, *etc.*) to reduce subtyping of complex templates to a set of constraints over base types.

Syntax-directed Traversal. Procedure $\mathsf{Gen}(\Gamma, \mathtt{e})$, shown in Figure 7, takes as input a template environment Γ and a (typed) term $\mathtt{e}$, and traverses the term in the syntax-directed manner of a type checker, to return as output a template T that describes the unknown refinement type of $\mathtt{e}$ and a set of constraints C that capture the subtyping relationships between the types of various sub-terms that must hold so that $\mathtt{e}$ has type T in the environment Γ. The terms of μML are of two classes, those whose (refinement) types are synthesizable from the environment and the types of sub-terms, and those whose types are not [27].

$$
\begin{array}{|ll|}
\hline
\mathsf{Env}(\emptyset) & \doteq \emptyset \\
\mathsf{Env}(\Gamma;\mathtt{x}{:}\{v{:}\beta \mid r\}) & \doteq \mathsf{Env}(\Gamma);\{\mathtt{x}{:}\beta \mid r[\mathtt{x}/\nu]\} \\
\mathsf{Env}(\Gamma;\mathtt{x}{:}_) & \doteq \mathsf{Env}(\Gamma) \\
\hline
\mathsf{New}(\Gamma,\alpha) & \doteq \alpha \\
\mathsf{New}(\Gamma,\beta) & \doteq \textbf{let } \kappa = \text{fresh ref. variable } \textbf{in} \\
 & \quad \textbf{let } G = \mathsf{Env}(\Gamma) \textbf{ in} \\
 & \quad \textbf{let } \mathtt{x}_1,\ldots,\mathtt{x}_n = \mathsf{Dom}(G) \textbf{ in} \\
 & \quad \{\mathtt{x}_0{:}\beta \mid \kappa(\mathtt{x}_0,\mathtt{x}_1,\ldots,\mathtt{x}_n)\} \\
\mathsf{New}(\Gamma,\beta\to\tau) & \doteq \textbf{let } \rho = \mathsf{New}(\Gamma,\beta) \textbf{ in} \\
 & \quad \rho \;\to\; \mathsf{New}(\Gamma;\rho,\tau) \\
\mathsf{New}(\Gamma,\tau_1\to\tau_2) & \doteq \mathsf{New}(\Gamma,\tau_1) \;\to\; \mathsf{New}(\Gamma,\tau_2) \\
\hline
\mathsf{Sub}(\Gamma,\ \alpha,\ \alpha) & \doteq \emptyset \\
\mathsf{Sub}(\Gamma,\ \rho_1,\ \rho_2) & \doteq \{\mathsf{Env}(\Gamma) \vdash \rho_1 \prec \rho_2\} \\
\mathsf{Sub}(\Gamma,\ \rho_1\to T_1,\ \rho_2\to T_2) & \doteq \mathsf{Sub}(\Gamma,\ \rho_2,\ \rho_1)\cup \\
 & \quad \mathsf{Sub}(\Gamma;\rho_2,\ T_1,\ T_2) \\
\mathsf{Sub}(\Gamma,\ T_1\to T_1',\ T_2\to T_2') & \doteq \mathsf{Sub}(\Gamma,\ T_2,\ T_1)\cup \\
 & \quad \mathsf{Sub}(\Gamma,\ T_1',\ T_2') \\
\hline
\mathsf{Gen}(\Gamma,\mathtt{x}:\beta) & \doteq (\{\nu{:}\beta \mid \nu = x\},\emptyset) \\
\mathsf{Gen}(\Gamma,\mathtt{x}:\tau) & \doteq (\Gamma(\mathtt{x}),\emptyset) \\
\mathsf{Gen}(\Gamma,\mathsf{c}:_) & \doteq (ty(\mathsf{c}),\emptyset) \\
\mathsf{Gen}(\Gamma,\mathsf{e}_1\ (\mathsf{e}_2{:}\beta):_) & \doteq \textbf{let } \{\mathtt{x}{:}\beta \mid r\}\to T_1', C_1 = \mathsf{Gen}(\Gamma,\mathsf{e}_1) \textbf{ in} \\
 & \quad \textbf{let } T_2, C_2 = \mathsf{Gen}(\Gamma,\mathsf{e}_2{:}\beta) \textbf{ in} \\
 & \quad (T_1'[e_2/\mathtt{x}], C_1\cup C_2\cup \mathsf{Sub}(\Gamma,\ T_2,\ \{\mathtt{x}{:}\beta \mid r\})) \\
\mathsf{Gen}(\Gamma,\mathsf{e}_1\ (\mathsf{e}_2{:}\tau):_) & \doteq \textbf{let } T_1\to T_1', C_1 = \mathsf{Gen}(\Gamma,\mathsf{e}_1) \textbf{ in} \\
 & \quad \textbf{let } T_2, C_2 = \mathsf{Gen}(\Gamma,\mathsf{e}_2{:}\tau) \textbf{ in} \\
 & \quad (T_1', C_1\cup C_2\cup \mathsf{Sub}(\Gamma,\ T_2,\ T_1)) \\
\mathsf{Gen}(\Gamma,\lambda\mathtt{x}.\mathsf{e}:\tau_\mathtt{x}\to_) & \doteq \textbf{let } T_\mathtt{x} = \mathsf{New}(\Gamma,\tau_\mathtt{x}) \textbf{ in} \\
 & \quad \textbf{let } T, C = \mathsf{Gen}(\Gamma;\mathtt{x}{:}T_\mathtt{x},\mathsf{e}) \textbf{ in} \\
 & \quad (T_\mathtt{x}\to T, C) \\
\mathsf{Gen}(\Gamma,\mathtt{let}\ \mathtt{x} = \mathsf{e}_1\ \mathtt{in}\ \mathsf{e}_2:\tau) & \doteq \textbf{let } T = \mathsf{New}(\Gamma,\tau) \textbf{ in} \\
 & \quad \textbf{let } T_1, C_1 = \mathsf{Gen}(\Gamma,\mathsf{e}_1) \textbf{ in} \\
 & \quad \textbf{let } T_2, C_2 = \mathsf{Gen}(\Gamma;\mathtt{x}{:}T_1,\mathsf{e}_2) \textbf{ in} \\
 & \quad (T, C_1\cup C_2\cup \mathsf{Sub}(\Gamma;\mathtt{x}{:}T_1,\ T_2,\ T)) \\
\mathsf{Gen}(\Gamma,[\Lambda\alpha]\mathsf{e}:_) & \doteq \textbf{let } (T,C) = \mathsf{Gen}(\Gamma,\mathsf{e}) \textbf{ in} \\
 & \quad (\forall\alpha.T, C) \\
\mathsf{Gen}(\Gamma,[\tau]\mathsf{e}:_) & \doteq \textbf{let } T = \mathsf{New}(\Gamma,\tau) \textbf{ in} \\
 & \quad \textbf{let } (\forall\alpha.T', C) = \mathsf{Gen}(\Gamma,\mathsf{e}) \textbf{ in} \\
 & \quad (T'[T/\alpha], C) \\
\hline
\end{array}
$$

Fig. 7. Constraint Generation

Terms with Synthesizable Types. The refinement types for variables, constants, applications and generalizations can be synthesized from those of the subexpressions, and the environment. For example, consider the case for an integer constant $\mathsf{Gen}(\Gamma, \mathtt{i})$. No constraints are generated, and the template returned is the "singleton" concrete refinement type $\{\nu\mathtt{:int} \mid \nu = \mathtt{i}\}$. As another example, consider the case for function application, $\mathsf{Gen}(\Gamma, \mathtt{e}_1\ (\mathtt{e}_2 : \beta))$, where the argument has a basic type. First, Gen is recursively called to obtain the templates and constraints for the sub-terms $\mathtt{e}_1$ and $\mathtt{e}_2$. As the input type of $\mathtt{e}_1$ is a basic type, the output's refinement can refer to the input. Thus the template for the result of the application is the output template of the function, with the all occurrences of the formal x substituted with the actual $\mathtt{e}_2$. When the argument is not a basic type, the argument cannot appear in the refinement for the output [1], and hence, the result has the output template for $\mathtt{e}_1$. In either case, the generated constraints are the union of the constraints generated for $\mathtt{e}_1$ and $\mathtt{e}_2$, and a set of constraints that ensure that the argument $\mathtt{e}_2$ is a subtype of the input type of $\mathtt{e}_1$.

Terms without Synthesizable Types. The refinement types for λ-abstractions, let-in expressions, and polymorphic instantiations (which includes recursive functions as explained below) cannot be synthesized from sub-terms, as we need to carry out an over-approximation of their concrete semantics. For example, consider the case for a let-in binding $\mathsf{Gen}(\Gamma, \mathtt{let\ x} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 \mathtt{:} \tau)$. We may be tempted to ascribe to the term the template found for $\mathtt{e}_2$ but this is unsound as that template can refer to the local x. Thus, for such expressions, we call $\mathsf{New}(\Gamma, \tau)$ to obtain a fresh template of the appropriate type, over variables that are in scope in the environment Γ, and generate constraints which ensure that subtyping holds between $\mathtt{e}_2$ and the entire expression. A similar situation arises for polymorphic instantiation $\mathsf{Gen}(\Gamma, [\tau]\mathtt{e})$ where a fresh template is generated for the unknown refinement for the instantiated monotype. We handle recursive functions via the fixpoint combinator fix of type $\forall\alpha.(\alpha \to \alpha) \to \alpha$. In the generated constraints, the unknown refinement type of a recursive function is represented by the template used to instantiate α at each invocation of fix.

Safety. Recall that μML programs are safe as long as at run-time, each primitive operation (*e.g.,* division, assert) is called with values on which the operation is defined (*e.g.,* non-zero integers, true). To enforce safety, we ensure that each primitive operation c, has a refinement type $ty(\mathtt{c})$ whose input refinement describes the domain of inputs for which the primitive operation is safe. For example,

$$\begin{aligned} ty(\mathtt{assert}) &\doteq \{\mathtt{p:bool} \mid \mathtt{p}\} \to \mathtt{bool} \\ ty(/) &\doteq \mathtt{int} \to \{\mathtt{x:int} \mid \mathtt{x} \neq 0\} \to \mathtt{int} \end{aligned}$$

where β is an abbreviation for $\{\nu\mathtt{:}\beta \mid \mathit{true}\}$. Thus, the safety requirements are captured by the subtyping constraints generated for function applications. The correctness of the constraint generation step is stated by Theorem 1, see Page 11.

The proof of Theorem 1 is shown in two steps. First, we use the correspondence between refinements and relations to demonstrate that the generated constraints are satisfiable iff only valid refinement type derivation exists [17, 29]. Next, we appeal to the

[1] arbitrary functions are barred for the usual soundness reasons

soundness of refinement type checking (which shows that the existence of a type derivation implies safety) to complete the proof. For brevity, we omit the details and refer the reader to [3, 18, 29].

B.2 Step 2: From Constraints to μC

Figure 8 formalizes the translation of a set of constraints C to a μC program.

Function Translation. The translation of a set C of constraints maps each refinement variable κ_i to a function $\mathtt{F}_i$. The result of the translation of the refinement variables is a system of mutually recursive functions, as we describe below. Consider a refinement variable $\kappa_i(\mathtt{x}_0, \ldots, \mathtt{x}_n)$. The translated function $\mathtt{F}_i$ has the *function property* that $\mathtt{F}_i(v_1, \ldots, v_n)$ returns v_0 iff *every* relational model that satisfies C maps κ_i to a set that includes the tuple $v_0, \ldots, v_n$.

For the example in Section 2, we create two functions $\mathtt{F}_1$ and $\mathtt{F}_2$ for the refinement variables κ_1 and κ_2.

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

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

and translates these constraints into the entry function $\mathtt{f}_0$. 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.

In Section 2, the function $\mathtt{F}_0$ encodes the constraint (c3).

Block Translation. To ensure that $\mathtt{F}_i$ 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 $\mathtt{z}_1, \ldots, \mathtt{z}_n$ and the return value $\mathtt{z}_0$ to a tuple of values that must belong in every relational model of κ_i that satisfies the constraint. Thus, the body instruction of $\mathtt{F}_i$, *i.e.,* the choice composition of all the blocks is such that each tuple of inputs and output of $\mathtt{F}_i$ belongs in every relational interpretation of κ_i.

To ensure that the translation of each constraint $G \vdash \{\mathtt{x}_1 : \beta \mid r_1\} \prec \{\mathtt{x}_2 : \beta \mid r_2\}$ 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 ($[\![G]\!]$), a sequence of instructions that "gets" the values corresponding to the LHS ($[\![\{\mathtt{x}:\beta \mid r_1\}_{get}]\!]$) and a sequence of instructions that "sets" the return value of $\mathtt{F}_i$ appropriately ($[\![\{\mathtt{x}:\beta \mid r_2\}]\!]_{set}$).

$$
\begin{array}{ll}
C\downarrow\kappa & \doteq \{c \in C \mid c \equiv _ \vdash _ \prec \kappa_i(_)\} \\
C\downarrow\bot & \doteq \{c \in C \mid c \equiv _ \vdash _ \prec p\} \\
[\![C]\!] & \doteq \textbf{let } \kappa_1, \ldots, \kappa_m = \text{Ref. vars. of } C \textbf{ in} \\
 & \quad [\![0, 0, C\downarrow\bot]\!], \\
 & \quad [\![1, \text{arity } \kappa_1, C\downarrow\kappa_1]\!] \\
 & \quad , \ldots, \\
 & \quad [\![m, \text{arity } \kappa_m, C\downarrow\kappa_m]\!] \\
[\![i, a, \{c_1, \ldots, c_n\}]\!] & \doteq \mathtt{f}_i(\mathtt{z}_1, \ldots, \mathtt{z}_a)\{[\![c_1]\!] \ldots [\![c_n]\!]\} \\
[\![G \vdash \{\mathtt{x}_1 : \beta \mid r_1\} \prec \{\mathtt{x}_2 : \beta \mid r_2\}]\!] & \doteq [\![G; \{\mathtt{x}_1 : \beta \mid r_1\}]\!]_{get}; \\
 & \quad [\![\{\mathtt{x}_1 : \beta \mid r_2[\mathtt{x}_1/\mathtt{x}_2]\}]\!]_{set} \\
[\![\{\mathtt{x} : \beta \mid r\}; G]\!]_{get} & \doteq [\![\{\mathtt{x} : \beta \mid r\}]\!]_{get}; [\![G]\!]_{get} \\
[\![\emptyset]\!]_{get} & \doteq \mathtt{skip} \\
[\![\{\mathtt{x} : \beta \mid p\}]\!]_{get} & \doteq \mathtt{x} \leftarrow \mathtt{nondet}(); \\
 & \quad \mathtt{assume}\ p \\
[\![\{\mathtt{x}_0 : \beta \mid \kappa_i(\mathtt{x}_0, \ldots, \mathtt{x}_n)\}]\!]_{get} & \doteq \mathtt{x} \leftarrow \mathtt{f}_i(\mathtt{x}_1, \ldots, \mathtt{x}_n) \\
[\![\{\mathtt{x} : \beta \mid p\}]\!]_{set} & \doteq \mathtt{assert}\ p \\
[\![\{\mathtt{x}_0 : \beta \mid \kappa(\mathtt{x}_0, \ldots, \mathtt{x}_n)\}]\!]_{set} & \doteq \mathtt{assume}\ (\wedge_{j=1}^{n} \mathtt{x}_j = \mathtt{z}_j) \\
 & \quad \mathtt{return}\ x
\end{array}
$$

Fig. 8. Translating Constraints To μC Programs

Get Instructions. Each environment binding gets translated as a "get" operation as follows. Bindings with unknown refinements $\kappa_i(\mathtt{x}_0, \ldots, \mathtt{x}_n)$ are translated into calls to $\mathtt{F}_i$ with arguments $\mathtt{x}_1, \ldots, \mathtt{x}_n$, with the return value assigned to $\mathtt{x}_0$. Bindings with concrete refinements p are translated into non-deterministic assignments followed by an `assume` that enforces that the refinement holds on the non-deterministically assigned value.

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 $\kappa_i(\mathtt{x}_0, \ldots, \mathtt{x}_n)$ is translated into an `assume` that establishes the equalities between each x_i and the formal $\mathtt{z}_i$ representing the i^{th} tuple element, followed by a `return` $\mathtt{x}_0$. 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 $\mathtt{F}_i$ satisfies the constraint to which the RHS refinement (over κ_i) belongs.

In Figure 2, the function $\mathtt{F}_1$ encodes the two constraints (c1), (c2) as a nondeterministic choice between their block translations. For each constraint, its block consists of the "get" and "set" operations as described above. Theorem 2 formalizes the correctness of $[\![C]\!]$.

B.3 The HMC Algorithm

We combine the the constraint generation and translation procedures to obtain the HMC algorithm. 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, $\mathsf{V}(x) = \mathsf{Safe}$ implies that x is safe. HMC converts a verifier for the (first-order, imperative) language $\mu\mathsf{C}$ to a verifier for the (higher-order, functional) language $\mu\mathsf{ML}$ in the following way:

$$\mathrm{HMC}(\mathsf{V}) \doteq \lambda e.\mathsf{V}([\![\mathsf{Generate}(e)]\!])$$

The correctness of HMC follows by combining Theorems 1 and 2 and is stated in Theorem 3, see Page 13.

B.4 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 [3, 16, 29]. Nevertheless, there can be a loss of information, as demonstrated below. Consider

```
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

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

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* [8, 9, 19, 23], for `check`:

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

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 [19] that uses a finite number of "contexts" to generate the implication constraints, after which a finite-state checker *e.g.,* BEBOP [2] 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.)

$$
\begin{array}{lll}
\mathtt{I} ::= & & \textbf{Instructions:} \\
\quad | & \mathtt{x} \leftarrow e & \text{expr assign} \\
\quad | & \mathtt{x} \leftarrow \mathtt{nondet}() & \text{havoc assign} \\
\quad | & \mathtt{x} \leftarrow \mathtt{F}(\mathtt{y}_1, \ldots, \mathtt{y}_n) & \text{call assign} \\
\quad | & \mathtt{return}\, e & \text{return} \\
\quad | & \mathtt{assume}\, p & \text{assume} \\
\quad | & \mathtt{assert}\, p & \text{assert} \\
\quad | & \mathtt{I}; \mathtt{I} & \text{sequence} \\
\quad | & \mathtt{I} [\!] \mathtt{I} & \text{choice} \\
\mathtt{F} ::= & \mathtt{F}(\mathtt{z}_1, \ldots, \mathtt{z}_n)\{\mathtt{I}\} & \textbf{Functions} \\
\mathtt{P} ::= & \mathtt{F}_0, \ldots, \mathtt{F}_m & \textbf{Programs}
\end{array}
$$

Fig. 9. μC **Syntax**

μC **Transition Relation** $\boxed{\mathtt{P}, \mathtt{I} \vdash \sigma \hookrightarrow \sigma'}$

$$\frac{n \in \mathcal{U}(\beta)}{\mathtt{P}, \mathtt{x} \leftarrow \mathtt{nondet}() \vdash \sigma \hookrightarrow \sigma[\mathtt{x} \mapsto n]}\ [\textsc{Hav}]$$

$$\frac{}{\mathtt{P}, \mathtt{x} \leftarrow e \vdash \sigma \hookrightarrow \sigma[\mathtt{x} \mapsto \sigma(e)]}\ [\textsc{Asgn}]$$

$$\frac{\mathtt{F}(\mathtt{z}_1, \ldots, \mathtt{z}_n)\{\mathtt{I}\} \in \mathtt{P} \qquad \mathtt{P}, \mathtt{I} \vdash [\mathtt{z}_1 \mapsto \sigma(\mathtt{y}_1), \ldots, \mathtt{z}_n \mapsto \sigma(\mathtt{y}_n)] \hookrightarrow \sigma'}{\mathtt{P}, \mathtt{x} \leftarrow \mathtt{F}(\mathtt{y}_1, \ldots, \mathtt{y}_n) \vdash \sigma \hookrightarrow \sigma[\mathtt{x} \mapsto \sigma'(\mathtt{z}_0)]}\ [\textsc{Call}]$$

$$\frac{}{\mathtt{P}, \mathtt{return}\, e \vdash \sigma \hookrightarrow \sigma[\mathtt{z}_0 \mapsto \sigma(e)]}\ [\textsc{Ret}]$$

$$\frac{\sigma(p) = \mathit{true}}{\mathtt{P}, \mathtt{assume}\, p \vdash \sigma \hookrightarrow \sigma}\ [\textsc{Asm}]$$

$$\frac{\sigma(p) = \mathit{true}}{\mathtt{P}, \mathtt{assert}\, p \vdash \sigma \hookrightarrow \sigma}\ [\textsc{Ast-T}] \qquad \frac{\sigma(p) = \mathit{false}}{\mathtt{P}, \mathtt{assert}\, p \vdash \sigma \hookrightarrow \mathsf{Err}}\ [\textsc{Ast-F}]$$

$$\frac{}{\mathtt{P}, \mathtt{I} \vdash \mathsf{Err} \hookrightarrow \mathsf{Err}}\ [\textsc{Err}]$$

$$\frac{\mathtt{P}, \mathtt{I} \vdash \sigma \hookrightarrow \sigma' \qquad \mathtt{P}, \mathtt{I}' \vdash \sigma' \hookrightarrow \sigma''}{\mathtt{P}, \mathtt{I}; \mathtt{I}' \vdash \sigma \hookrightarrow \sigma''}\ [\textsc{Seq}]$$

$$\frac{\mathtt{P}, \mathtt{I} \vdash \sigma \hookrightarrow \sigma'}{\mathtt{P}, \mathtt{I} [\!] \mathtt{I}' \vdash \sigma \hookrightarrow \sigma'}\ [\textsc{Ch-L}] \qquad \frac{\mathtt{P}, \mathtt{I}' \vdash \sigma \hookrightarrow \sigma'}{\mathtt{P}, \mathtt{I} [\!] \mathtt{I}' \vdash \sigma \hookrightarrow \sigma'}\ [\textsc{Ch-R}]$$

Fig. 10. μC **Semantics**

C Proof Sketch for Theorem 2

We give an outline of the proof of Theorem 2. First, we introduce some machinery connecting the semantics of the constraints (*i.e.,* relational interpretations) with the semantics of the translated μC programs (*i.e.,* configurations).

Canonical Interpretations. Let C be a set of inclusion constraints. The *Canonical interpretation* of C, written $\Sigma_{[\![C]\!]}$ maps each κ_i of arity $n+1$ to the set

$$\{(\sigma'(\mathtt{z}_0), \ldots, \sigma'(\mathtt{z}_n)) \mid [\![C]\!], \mathtt{I} \vdash \sigma \hookrightarrow \sigma'\}$$

where $\mathtt{I}_i$ denotes the body instruction of $\mathtt{F}_i$.

The correctness of the function translation is formalized by the following Lemma, which states that for each κ_i, the Canonical interpretation satisfies the constraints with κ_i on the RHS.

Lemma 1. ***[Function Translation]*** *Let C be a set of refinement constraints. For each $\kappa \in C$, we have $\Sigma_{[\![C]\!]} \models C \downarrow \kappa$.*

The correctness of the bound translation is formalized by the following lemma, which says the translated μC program is safe iff the Canonical interpretation satisfies the bound constraints.

Lemma 2. ***[Bound Translation]*** *Let C be a set of refinement constraints. $\Sigma_{[\![C]\!]} \models C \downarrow \bot$ iff $[\![C]\!]$ is safe.*

The proofs of the above lemmas rely on the following lemma that states the correctness of the block translation.

Lemma 3. ***[Block Translation]*** *Let C be a set of refinement constraints, and σ and G be a state and environment with the same domain. $[\![C]\!], [\![G]\!]_{get} \vdash [\cdot] \hookrightarrow \sigma$ iff $\Sigma_{[\![C]\!]}, \sigma \models G$.*

Strengthening. Furthermore, we can show that the Canonical interpretation is *stronger* than *any* interpretation that satisfies the constraints in the following sense. Given two relational interpretations Σ and Σ' with the same domain, we say that Σ is *stronger than* Σ' (written $\Sigma \subseteq \Sigma'$) if for each κ we have $\Sigma(\kappa) \subseteq \Sigma'(\kappa)$.

We prove, by induction on the structure of the big-step derivations that populate the Canonical interpretation, that $\Sigma_{[\![C]\!]}$ is the strongest interpretation that satisfies a set of constraints.

Lemma 4. ***[Strongest Interpretation]*** *Let C be a set of refinement constraints, and Σ be a relational interpretation. If $\Sigma \models C$ then $\Sigma_{[\![C]\!]} \subseteq \Sigma$.*

Furthermore, using the Lemma 3 we prove that strengthening preserves satisfiability of the upper bound constraints.

Lemma 5. ***[Strengthening]*** *Let C be a set of refinement constraints and Σ and Σ' be relational interpretations. If $\Sigma \subseteq \Sigma'$ and $\Sigma' \models C \downarrow \bot$ then $\Sigma \models C \downarrow \bot$.*

Proof. (of Theorem 2) Lemmas 1 and 2 combine to prove that if $[\![C]\!]$ is safe then $\Sigma_{[\![C]\!]} \models C$ and hence, C is satisfiable. To show the other direction, supppose C is satisfiable. Then there exists an interpretation $\Sigma \models C$. By Lemma 4, we deduce that $\Sigma_{[\![C]\!]}$ is stronger than S. As S satisfies C, it satisfies the upper bound constraints of C and so, by Lemma 5 we conclude $\Sigma_{[\![C]\!]} \models C \downarrow \bot$, which, via Lemma 2 implies that $[\![C]\!]$ is safe.