AUTOMATIC VERIFICATION OF HIGHER ORDER PROGRAMS

by

ZHOU LITAO
(Shanghai Jiao Tong University)

A THESIS SUBMITTED FOR

NGNE CP3106 PERSONAL PROJECT

in

COMPUTER SCIENCE

in the

UNDERGRADUATE DIVISION

of the

NATIONAL UNIVERSITY OF SINGAPORE

2021

Supervisor:
Associate Professor CHIN Wei Ngan

Contents

List of Figures

Chapter 1

Introduction

Many tools have been developed to verify the correctness of a program. One typical method towards program verification is to ask users to write specifications for a given program, and then the verifier will check the correctness of the program by using rules of a logical system to prove that the program behavior conforms to the user-given assertion. The assertion entailment will be encoded to a *verification condition* (VC) in a *satisfiability modulo theory* (SMT) and then discharged by SMT solvers such as Z3. The HIP/SLEEK [2] system has managed to apply separation logic to verifying heap-manipulating imperative programs using this approach.

However, automatically verifying programs with higher order functions is a topic yet to be explored. Many automatic verifiers, such as HIP/SLEEK, Dafny [7] and VeriFast [4] have limited their programming language to a first-order language, which only allows users to write program expressions that operate on individual data elements (e.g., strings, integers, records, variants, etc.). By contrast, higher-order programming languages treat functions as first-class values, so that a higher-order function can operate on functions (*function as parameter*) and return a function as result (*function as returned value*). The use of higher-order functions can increase the modularity since they allow programmers to factor out functions and parameterize functions on other functions. In fact, the use of higher order functions is the essence of *functional programming*. It is worth investigating what approaches will be applicable for automatic verification of functional programs.

1.1 Current Work

The goal of this report is to provide a solution to automatic verification of higher order programs and implement a prototype system to verify several concrete examples of higher order programs. Higher order functional programming languages encompoass many programming language features and this project does not cover all of them. Therefore, the presentation of this report will be driven by higher-order programming examples, to see how far automatic verification can go in a higher-order setting.

In this project we take a subset language of OCaml [11], an industrial-strength functional programming language, and explore the automatic verification of higher order functions based on the structure of HIP/SLEEK system.

Currently, this report focuses on one specific target language OCaml and supports reasoning on pure properties of immutable variables. However the techniques such as abstract predicates and nested function specifications in the design of this project can also be extended to a larger category of programming languages and features. Our system also has the potential to integrate with the HIP/SLEEK system, to enable more sophisticated reasoning on mutable data structures.

1.2 Report Structure

The rest of this report is organized as follows. Chapter 2 provides some preliminary knowledge in program logic and automatic verification. We also conduct a literature review on related work regarding higher order program verification. Chapter 3 describes the design of our prototype verification system. Chapter 4 presents some programming examples to further elaborate on the proposed verification rules. We conclude the entire report as well as discuss further directions for future research in Chapter 5.

Chapter 2

Preliminaries

2.1 Higher order programs

Programs with higher order functions either take other functions as input or return functions as output (or both). Higher order programs are useful in that programmers can write general, reusable code with higher order functions. Consider a famous higher order function `fold` that combines elements of a list in Code 2.1.

```
1 let rec fold_left op acc = function
2   | []   -> acc
3   | h :: t -> fold_left op (op acc h) t
```

Code 2.1: Higher order function that combines list elements

To illustrate, `fold_left f a [b;c;d]` computes the expression `f(f(f(a,b),c),d)`. The behavior of `fold_left` is to fold elements of the list from left to the right and combine the next element using the operator `op`. Since `op` is a function as parameter, the `fold_left` function is a higher order function.

We can use the `fold_left` function to implement many useful functions, as shown in Code 2.2. It is expected that modern verification tools should be able to specify and verify such programs.

```
1 let length l = List.fold_left (fun a _ -> a+1) 0 l
2 let sum l = List.fold_left (fun a b -> a+b) 0 l
3 let rev l = List.fold_left (fun a x -> x::a) [] l
```

Code 2.2: `fold_left` application examples

2.2 Refinement type based reasoning

One related work in specifying and verifying higher order functional programs is the LiquidHaskell project [9]. LiquidHaskell is a static verifier for Haskell, based on refinement types [10]. The precondition and postconditions can be encoded using refined function types like

```
type Pos = {v:Int | v > 0 }
type Nat = {v:Int | v >= 0}
div::  n:Nat -> d:Pos -> {v:Nat | v <= n}
```

which states that the function `div` requires inputs that are respectively non-negative and positive, and ensures that the output is no greater than the first input n.

LiquidHaskell has a type checker that applies SMT to check the validity of the refinement type specifications. The system has been shown its effectiveness in verifying program safety properties, and data structures [5]. One important feature of LiquidHaskell is that its restriction on refinements has been delicately designed to be decidable in type checking while expressive enough to specify program properties. Our project differs from LiquidHaskell in that we choose not to specify properties with refinement type, but allow a more flexible assertion syntax in the style of Hoare logic.

2.3 Hoare logic based reasoning

Hoare logic [3] has long been a natural way of writing down specifications of programs. The main judgement of Hoare logic consists of a program and a pair of formulae `{P}c{Q}`, indicating if program state satisfies property `P`, then after executing program `c`, the program state should satisfy `Q`.

Yoshida et al. [12] proposed an extension of Hoare logic for call-by-value higher-order functions with ML-like local reference generation, and proved the logic sound with respect to a standard semantic model. However, the design of their assertion language and the rules of their logical system are difficult to automate reasoning.

Chapter 3

Design of the system

3.1 Programming Language

We provide a simple functional language in Figure 3.1. A program comprises a list of abstract datatype declarations `datat`, a list of user-defined predicates `spred` and a list of method declarations `meth`. The supscript $*$ is used to denote a list of items. The function body can be composed by constant k^{τ} of a ground type τ, a variable v, a lambda expression e with a bounded variable v, function application of two expressions, if branches, pattern matching on datatype expressions, and primitive operations (such as arithmetic and boolean operations).

Following the OCaml programming convention, a program is presented as a list of `let`-expressions, each of which declares a method to be verified. In order to allow users to write specifications, each method declaration should come with a function specification $\mathcal{F}$. We also allow users to provide predicate instances `spred` in the program. The syntax of function specification $\mathcal{F}$ and logical proposition π will be described in the next section.

3.2 Assertion Language

Since higher order functions can take functions as input or output values, it is essential for our system to be able to describe the function as part of the program assertion. We present the specification language for our system in Figure 3.2. An important feature of our system is that the function specification $\mathcal{F}$ and assertions Φ are mutually recursively defined. So for a higher order function that takes a function

$$
\begin{array}{rrl}
\text{program :} & \texttt{P} & := \ \texttt{datat}^* \ \texttt{spred}^* \ \texttt{meth}^* \\
\text{data type :} & \texttt{datat} & := \ \texttt{type c = tconstr}^* \\
\text{predicate :} & \texttt{spred} & := \ \texttt{spname}\langle v^* \rangle = ((\exists \vec{v}.\pi))^* \\
\text{type constructor :} & \texttt{tconstr} & := \ \texttt{cname of } \tau^* \\
\text{ground types :} & \tau & := \ \texttt{int} \mid \texttt{bool} \mid \texttt{c} \\
\text{general types :} & \gamma & := \ \tau \mid \gamma_1 \to \gamma_2 \\
\text{method definition} & \texttt{meth} & := \ \texttt{let f} : \gamma := e \ \texttt{where } \mathcal{F} \\
 & & \\
\text{function body} & e & := \ k^\tau \mid v \mid \lambda v.e \mid e_1 \ e_2 \\
 & & \quad \mid \ \texttt{if } e_0 \ \texttt{then } e_1 \ \texttt{else } e_2 \\
 & & \quad \mid \ \texttt{match } e \ \texttt{with } (\texttt{cname} \Rightarrow e_i)^* \\
 & & \quad \mid \ \texttt{op } \vec{e}
\end{array}
$$

Figure 3.1: The core language

type argument `f`, users can specify what kind of argument they expect by writing a specification for `f` in the assertion Φ.

To be specific, a program state assertion Φ includes two parts, the pure predicate part and the function specification part. To simplify the verification process, the pure predicate is represented as a disjunctive normal form. It is easy to convert any first order pure proposition Δ that our system uses into disjunctive normal form. In each pure clause, we allow equality comparison between ground types such as integers and booleans, basic arithmetic judgements, datatype judgements and logicial connectives such as conjunction and negation.

As for the logical expressions, apart from basic operators and values from the arithmetic and boolean theory, we also allow logical expressions of the form $f(\vec{x})$, where f is the name of a logical predicate bound by the GIVEN clause. We will see the use of such *abstract predicates* through examples in the next chapter.

The function specification is composed of several parts, namely, function name `f`, function argument names $\vec{\texttt{x}}$, GIVEN variables $\vec{y}$, (which serve as universally quantified auxiliary variables that only appear in specifications), precondition Φ_{pre}, the name of the return value (*anchor*) `r` and postcondition Φ_{post}. Note that function specifications can also appear as sub-expressions in the precondition and postcondition. We require each top-level function specification to be well-formed, i.e. all the variables in the assertion should be closed as a program variable introduced by the function arguments, or a logical variable introduced by the GIVEN binder or existential

quantifiers. The return value name can only occur in the postcondition.

$$
\begin{array}{rrl}
\text{program variables :} & \mathtt{v} := & \mathtt{x}, \mathtt{y}, \mathtt{f}, \mathtt{g}, \ldots \\
\text{function specification :} & \mathcal{F} := & \mathtt{f}(\vec{\mathtt{x}}) \vDash \text{Given } \vec{y},\ \Phi_{\text{pre}} \rightarrowtail_{\mathtt{r}} \Phi_{\text{post}} \\
 & & \\
\text{logical variables :} & v := & a, b, c, f, \ldots \\
\text{assertions :} & \Phi := & \bigvee ((\exists \vec{v}.\pi))^* \wedge \vec{\mathcal{F}} \\
\text{pre/post condition} & \Delta := & \Phi \mid \exists v.\Delta \mid \Delta \wedge \pi \mid \Delta_1 \vee \Delta_2 \\
 & & \\
\text{pure predicates :} & \pi := & s_1 = s_2 \mid s_1 \leq s_2 \quad \text{(arithmetic judgements)} \\
 & \mid & \mathtt{x} :: \mathtt{cname}(\vec{y}) \quad \text{(datatype judgements)} \\
 & \mid & \pi_1 \wedge \pi_2 \mid \neg\pi \\
\text{logical expressions :} & s := & v \mid f(\vec{s}) \mid c \\
 & \mid & s_1 + s_2 \mid -s \mid s_1 \times s_2 \\
 & \mid & s_1 =?s_2 \mid s1 \leq ?s_2
\end{array}
$$

Figure 3.2: The specification language

3.3 The programming interface

3.3.1 Overview of the system

We developed a prototype system in programming language OCaml to implement the automated verification. To make our system useful, we also provide our system with a front-end parser so that users can directly input a real-world OCaml program and write the specification they want to verify in the form of comments along with the code. As we shall see in the next chapter, all example codes can be directly input to the system and our system can report whether each function in the program is verified correct. The general structure of our system is shown in Figure 3.3.

After parsing the program and user-provided specification, our system implements a standard Hoare-style forward verifier, which will next invoke the entailment prover. The forward verifier is built upon a set of forward verification rules to systematically check the precondition is satisfied at each call site, and the declared postcondition is successfully verified under the given precondition for each method declaration. At certain points, the verifier requires checking the entailment between two assertions

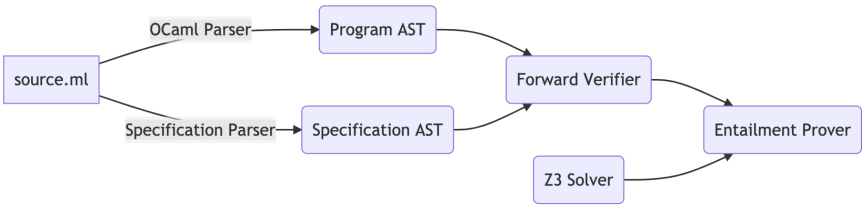

Figure 3.3: Prototype system structure

holds, which will be handled by the backend entailment prover. The benefit of such structure is that our system can be modular and easily extended to support richer features. In the next two sections, we will show in detail how the forward verifier and entailment checker are designed and implemented.

3.3.2 Front-end Implementation

We present the implementation of our front-end parser in this section. This section also serves as a guide to use our system. Our system takes an OCaml program `*.ml` as input. Users should program each method in the format of `let f x = ...` and only use the structures supported by our system. Otherwise the forward verifier will reject the input. The codes enclosed by `(*...*)` will be recognized as comments. However, if users write comments enclosed by `(*@ ...@*)`, they will be recognized by our specification parser and parsed into an abstract syntax tree for the forward verifier to use. Readers can refer to the next chapter to see several example programs that our system can accept and verify. Several implementation details are discussed below.

OCaml frontend We make use of the OCaml 4.12.0 frontend library to parse the program part, so our forward verifier is based on the same program abstract syntax tree as the OCaml compiler accepts. The detail of the AST definition can be found in the OCaml compiler repository[1].

Specification parser The OCaml parser library menhir is used in our implementation. The design of the specification abstract syntax tree follows the definition in Figure 3.2. To improve readability of the specification, users can write the top-level

[1]https://github.com/ocaml/ocaml/blob/4.12/parsing/parsetree.mli

specification in the format of `given` $\vec{y}$ `requires` $\Phi_{\texttt{pre}}$ `ensures[r]` $\Phi_{\texttt{post}}$ The nested specifications in program assertions can be written as `f(x)|={pre}*->:r{post}`.

3.4 Forward Verification

3.4.1 Logical reasoning rules

We present a rule-based declaration for our forward verification system in Figure 3.4. The judgement $\Sigma; \Gamma \vdash \{\Delta_1\}\, e\, {}_{:\texttt{r}}\, \{\Delta_2\}$ indicates *if we evaluate e in the initial state satisfying* Δ_1*, then it terminates with a value, name it* `r`*, and a final state, which together satisfy* Δ_2. All the rules are presented in a forward style, i.e. they expect precondition to be given before computing the postcondition. We write $\Delta[\texttt{b}/\texttt{a}]$ to indicate the assertion obtained from replacing `a` by `b` in Δ. We assume all the introduced anchors and variables are fresh.

We use Σ to denote the dynamic context of logical variables and use Γ to denote the context of specifications available for the expression to call. Our system keeps track of the context during forward verification for each method declaration. [Fv-Spec] and [Fv-Exists] allows the verifier to extract the method declarations or existential variables users write in the precondition to the context.

We use P to denote the program being checked. With pre/post conditions declared for each method in P, we can apply modular verification to a method's body, as has been defined in [Fv-Method]. When a new method is being verified, the variable context will be variables in the function arguments and given clause, and the specification context will be all the specifications in the program.

[Fv-App-Par] deals with partial application, which basically creates a new instance of specification for a partially applied function. Note that the function and the argument should both be evaluated before creating the specification. [Fv-App-Full] deals with full application. When all the arguments of the specification has been applied with program expressions, the verifier will check whether the precondition of the specification holds. Proving entailment will be presented in detail in the next section. If the entailment checker reports success, then the verifier can add the specified post condition to the inferred postcondition. Another thing to note is that the function specification may have variables or predicates to instantiate

$$\textsc{Fv-Method}\quad \frac{\mathcal{F} = \text{Given } \vec{y}, \Phi_{\text{pre}} \rightarrowtail_{\mathtt{r}} \Phi_{\text{post}} \qquad \vec{x}, \vec{y}; P \vdash \{\Phi_{\text{pre}}\}\, e\, {}_{:\mathsf{r}}\, \{\Phi_{\text{post}}\}}{\vdash \texttt{let f=}\lambda\vec{\mathtt{x}}.e \texttt{ where } \mathcal{F}}$$

$$\textsc{Fv-Spec}\quad \frac{\Sigma; \Gamma, \mathcal{F} \vdash \{\Delta_1\}\, e\, {}_{:\mathsf{r}}\, \{\Delta_2\}}{\Sigma; \Gamma \vdash \{\Delta_1 \wedge \mathcal{F}\}\, e\, {}_{:\mathsf{r}}\, \{\Delta_2\}} \qquad \textsc{Fv-Exists}\quad \frac{\Sigma, x; \Gamma \vdash \{\Delta_1\}\, e\, {}_{:\mathsf{r}}\, \{\Delta_2\}}{\Sigma; \Gamma \vdash \{\exists x.\Delta_1\}\, e\, {}_{:\mathsf{r}}\, \{\Delta_2\}}$$

$$\textsc{Fv-App-Par}\quad \frac{\Sigma; \Gamma \vdash \{\Delta_0\}\, f\, {}_{:\mathsf{g}}\, \{\Delta_1\} \qquad \Sigma, \mathsf{g}; \Gamma \vdash \{\Delta_1\}\, x\, {}_{:\mathsf{v}}\, \{\Delta_2\} \qquad \mathsf{g}(\mathsf{a}, \tilde{\mathsf{b}}) \vDash \mathcal{F} \in \Gamma}{\Sigma; \Gamma \vdash \{\Delta_0\}\, f\ x\, {}_{:\mathsf{h}}\, \left\{\Delta_2 \wedge \mathsf{h}(\tilde{\mathsf{b}}) \vDash \mathcal{F}[\mathtt{v}/\mathtt{a}]\right\}}$$

$$\textsc{Fv-App-Full}\quad \frac{\begin{array}{c}\Sigma; \Gamma \vdash \{\Delta_0\}\, f\, {}_{:\mathsf{g}}\, \{\Delta_1\} \qquad \Sigma, \mathsf{g}; \Gamma \vdash \{\Delta_1\}\, x\, {}_{:\mathsf{v}}\, \{\Delta_2\} \\ \mathsf{g}(\mathsf{a}) \vDash \text{Given } \vec{y}, \Phi_{\text{pre}} \rightarrowtail_{\mathtt{r}} \Phi_{\text{post}} \in \Gamma \cup \mathit{Spec}(\Delta_2) \\ \Delta_2 \vdash \Phi_{\text{pre}}(\vec{y})[\mathtt{v}/\mathtt{a}] \qquad \Phi_{\text{post}}(\vec{y})[\mathtt{v}/\mathtt{a}, \mathtt{res}/\mathtt{r}] \vdash \Delta_3\end{array}}{\Sigma; \Gamma \vdash \{\Delta_0\}\, f\ x\, {}_{:\mathsf{res}}\, \{\Delta_3\}}$$

$$\textsc{Fv-Match}\quad \frac{\Sigma; \Gamma \vdash \{\Delta_1\}\, e\, {}_{:\mathsf{v}}\, \{\Delta_2\} \qquad \forall i, \Sigma, \mathtt{v}, \vec{\mathtt{x}_i}; \Gamma \vdash \{\Delta_2 \wedge \mathtt{v} :: \mathsf{cname}_i\, (\vec{\mathtt{x}_i})\}\, e_i\, {}_{:\mathsf{u}}\, \{\Delta_3\}}{\Sigma; \Gamma \vdash \{\Delta_1\}\, \texttt{match } e \texttt{ with } (\mathsf{cname}_i \Rightarrow \lambda\vec{\mathtt{x}_i}.e_i)^*\, {}_{:\mathsf{res}}\, \{\Delta_3[\mathtt{res}/\mathtt{u}]\}}$$

$$\textsc{Fv-If}\quad \frac{\begin{array}{c}\Sigma; \Gamma \vdash \{\Delta_1\}\, e\, {}_{:\mathsf{v}}\, \{\Delta_2\} \qquad \Sigma, v; \Gamma \vdash \{\Delta_2 \wedge \mathtt{v} = \mathtt{true}\}\, e_1\, {}_{:\mathsf{u}}\, \{\Delta_3\} \\ \Sigma, v; \Gamma \vdash \{\Delta_2 \wedge \mathtt{v} = \mathtt{false}\}\, e_2\, {}_{:\mathsf{u}}\, \{\Delta_3\}\end{array}}{\Sigma; \Gamma \vdash \{\Delta_1\}\, \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2\ {}_{:\mathsf{res}}\, \{\Delta_3[\mathtt{res}/\mathtt{u}]\}}$$

$$\textsc{Fv-Cst}\quad \frac{}{\Sigma; \Gamma \vdash \{\Delta\}\, k\, {}_{:\mathsf{res}}\, \{\Delta \wedge \mathtt{res} = k\}} \qquad \textsc{Fv-Var}\quad \frac{\mathtt{v} \in \Sigma \text{ or } \mathtt{v}(\vec{x}) \vDash \mathcal{F} \in \Gamma}{\Sigma; \Gamma \vdash \{\Delta\}\, \mathtt{v}\, {}_{:\mathsf{res}}\, \{\Delta \wedge \mathtt{res} = \mathtt{v}\}}$$

$$\textsc{Fv-Fun-Eval}\quad \frac{\mathtt{v}() \vDash \text{Given } \vec{y},\ \Phi_{\text{pre}} \rightarrowtail_{\mathtt{r}} \Phi_{\text{post}} \in \Gamma \qquad \Delta \vdash \Phi_{\text{pre}}(\vec{y})}{\Sigma; \Gamma \vdash \{\Delta\}\, \mathtt{v}\ {}_{:\mathsf{res}}\, \{\Delta \wedge \mathtt{res} = v \wedge \Phi_{\text{post}}(\vec{y})[\mathtt{res}/\mathtt{r}]\}}$$

Figure 3.4: Proof rules of Hoare logic

in the GIVEN clause, so the verifier should also infer the instances before proceeding the verification.

Apart from the standard structural rules such as [FV-MATCH], [FV-IF], [FV-CST], and [FV-VAR], our system also comprises a rule [FV-FUN-EVAL] for evaluating "zero-ary" function identifiers. Since our specification language supports nested function specifications in the assertion, the actual user-expected specification may be hidden in the assertion and needs unfolding. A verification example that uses this rule can be found in the next chapter.

3.4.2 Implementation

The logical rules above provide some general guidelines to verify a program annotated by user-provided specifications. To implement a forward verifier system, we should fix the order of applying these rules to make it more organized and easier to implement. Furthermore, the instantiating in [FV-APP-FULL] is in general an undecidable problem. Our verifier can only infer some predicate instances for a function application, and the inferred result may lead to a failure in proof. To address these issues, we list some key steps in implementing our forward verifier system.

The general structure of the forward verifier can be defined as a recursive function `infer_exp` that takes as input the proof context $\mathcal{E} = (\Sigma, \Gamma)$, program expression e and pre-condition Δ. Due to the restriction we have discussed about instantiating, the output is designed in a non-deterministic style, i.e. the recursive function will output a set of possible solutions, each solution comprises the its proof context $\mathcal{E}'$, return value name $\mathtt{r}$, and inferred post-condition Δ'.

To implement `infer_exp`, we always normalize the input precondition using [FV-SPEC] and [FV-EXISTS] on entry to extract the specifications into the proof context. Then the verifier will choose a rule based on the structure of the program expression. More than one possible inferred postcondition may be returned by [FV-APP-FULL].

The recursion goes on following the syntax of the program expression e. After e is evaluated, the verifier asks the entailment checker to check whether one of the triples $(\mathcal{E}', \mathtt{r}, \Delta')$ in the result can derive the expected post-condition. If so, the

system can report that the method is verified w.r.t. the user-defined specification.

3.5 Entailment checking

At function call sites and the end of a method declaration, the forward verifier will generate proof obligations of assertion entailments. Our formulae are a combination of pure predicates and function specifications, where abstract predicates are involved. We take the satisfiability modulo theories (SMT) approach to discharge the proof obligations. In our project, we use Z3, an efficient SMT solver with specialized algorithms for solving background theories.

$$\textsc{Ent-Ass}\quad \frac{\Sigma \vdash \Delta_1 \Rightarrow \Delta_2 \qquad \Sigma; \Gamma; \Delta_1 \vdash \vec{\mathcal{F}_1} \rhd \vec{\mathcal{F}_2}}{\Delta_1 \wedge \vec{\mathcal{F}_1} \vdash^{\Sigma;\Gamma} \Delta_2 \wedge \vec{\mathcal{F}_2}}$$

Our job is to encode the proof entailments into a verification condition that Z3 accepts. Based on the solver result `SAT/UNSAT/UNKNOWN`, the entailment checker can return to the forward verifier whether the entailment succeeds. The general idea in [Ent-Ass] is to verify both the derivation of the pure predicates and the *subsumption* relation of the specifications. Note that the verification context is also used in the entailment checker, in that the variables that have been extracted into the context Σ should be universally quantified, and that the specifications available in the context Γ may also be used in the function subsumption check.

Since Z3 library has already provided a high-level API system for the OCaml programming language, most of the pure predicates such as arithmetic operations can be easily encoded as Z3 formulas. We will highlight several challenges and our solutions in the encoding.

Verifying with abstract predicates The syntax of our specification language allows user to write logical expressions with abstract predicate $f(\vec{e})$. What we mean by using an abstract predicate f in a formula $\Delta_1(f) \vdash \Delta_2(f)$ is that by replacing f with any instance that matches the type signature of f we can always prove $\Delta_1 \vdash \Delta_2$. The abstract predicate f is encoded as an uninterpreted function in the SMT formula.

When there are uninterpreted formulas in the constraints, the Z3 solver will try to find instances of formulas that make the formula satisfiable. However, this solving process is different from our goal, since the `SAT` result for a direct encoding VC means $\exists f, \Delta_1(f) \vdash \Delta_2(f)$ holds. The trick here is to encode the negation of the formula, i.e. we add $\Delta_1 \wedge \neg\Delta_2$ to the constraints of the solver. The idea is that the negation of an existential quantifier is universal. Now if the solver returns `UNSAT`, the entailment checker should report success.

Verifying with concrete predicates As the rule [Fv-App-Full] shows, when we verify the application of a higher order specification with abstract predicates, we need to provide a concrete instance to predicates in the given clause. We will also see examples in the next chapter that users are able to write their own concrete predicates to help specify a function. To implement this feature, our solver will leave the original verification goal untouched, and add an extra constraint $\forall \vec{x}, f(\vec{x}) = \ldots$ to the Z3 constraints when certain predicates are instantiated.

Verifying function specifications An important component of our assertions are function specifications. We use the notion of *function subsumption* [1] to model our solution to the specification part. We say if $\mathcal{F}_1$ is a function subsumption of $\mathcal{F}_2$, that is $\mathcal{F}_1 <: \mathcal{F}_2$, then any function satisfying specification $\mathcal{F}_1$ can be used wherever a function satisfying $\mathcal{F}_2$ is required.

We look at the function part of [Ent-Ass] in detail. We write $\Sigma; \Gamma; \Delta \vdash \vec{\mathcal{F}_1} \rhd \vec{\mathcal{F}_2}$ to indicate that under variable context Σ and specification context Γ, if pure proposition Δ holds, then all the function specification $\mathcal{F}$ required by $\vec{\mathcal{F}_2}$ can find a specification $\mathcal{G}$ in either $\vec{\mathcal{F}_1}$ or the specification context, which is of the same function name and is a subsumption of $\mathcal{F}$. This description is formally written in [Ent-Fun-Intro].

$$\textsc{Ent-Fun-Intro}\quad \frac{\forall \mathcal{F} \in \vec{\mathcal{F}}_2, \quad \exists \mathcal{G} \in \vec{\mathcal{F}}_1 \cup \Gamma, \mathcal{G} <:_{\Sigma,\Delta} \mathcal{F}}{\Sigma; \Gamma; \Delta \vdash \vec{\mathcal{F}}_1 \rhd \vec{\mathcal{F}}_2}$$

$$\textsc{Fun-Sub}\quad \frac{\Sigma \vdash \forall \vec{y}_2, \forall \vec{x}_2, \pi \wedge \Phi_2 \Rightarrow \exists \vec{y}_1, \forall \vec{x}_1, (\Phi_1 \wedge (\forall \mathtt{r}, \Psi_1 \Rightarrow \Psi_2))}{\mathtt{f}(\vec{x}) \vDash \text{Given } \vec{y}_1, \Phi_1 \rightarrowtail_{\mathtt{r}} \Psi_1 <:_{\pi,\Sigma} \mathtt{f}(\vec{x}) \vDash \text{Given } \vec{y}_2, \Phi_2 \rightarrowtail_{\mathtt{r}} \Psi_2}$$

To check whether a single function is a subsumption of the other, we encode the verification condition as [Fun-Sub] shows. We assume that the function arguments and the return anchor name of the two functions have been renamed to be consistent. This encoding is in line with Kleymann's adaptation-complete rule of consequence [6]. Note that the precondition is encoded in a *contravariant* way since one can only strengthen the precondition of $\mathcal{F}_1$ to ensure its compatibility to $\mathcal{F}_2$.

Chapter 4

Verification Examples

4.1 Verification with abstract predicates

Higher order programs provide better abstraction to programmers. Consider the `twice` function in Code 4.1. It takes a function input `f` and an integer `x` as input and return the result of applying the function twice to x. Here `f` can be any function that takes an integer input and return an integer output. When we specify or verify the function `twice`, we do not know what concrete function `f` is. On the other hand, we expect our verification process to be modular with respect to independent method declarations.

We propose abstract predicates as a solution. For the `twice` example, we universally quantify the specification with an abstract predicate `fpure`, which is a binary relation of integers. Now we can specify the argument `f` with a specification that the return value of `f` and the input `a` should satisfy relation `fpure(a,res0)`. Then for the `twice` function, we can specify the postcondition with the composition of the binary relation `fpure`.

```
1 let twice f x = f (f x)
2 (*@ declare twice(f,x)
3 given fpure(int,int)
4 requires.    { true with
5                  f(a) |= { true } *->:res0 {fpure(a,res0)} }
6 ensures[res] {EX (n:int), fpure(x,n) & fpure(n,res) } @*)
```

Code 4.1: Higher order twice function

We show that our system is able to verify such method and specification by listing the verification process. First, [Fv-Spec] will extract the specification for `f` to the verification context so that it can be used when verifying the function body. Next, [Fv-App-Full] will be applied twice, producing two anchors `res'`, `res` satisfying `fpure(n,res')` and `fpure(res',res)`. The entailment checker can then instantiate the existential variable `n` to be the intermediate anchor `res'` and check the entailment holds, which finishes the verification.

Next, we demonstrate how the specification we have verified for `twice` is used at the call site. We provide `incr` as an instance to the `twice` function, as Code 4.2 shows.

Note that our system has already integrated some basic specifications for library functions such as `(+)`, which can be considered as an infix syntatic sugar for a function that takes two inputs. Therefore, the specification for `incr` is easy to verify.

The verification of `incr_twice` can be simply done by applying [Fv-App-Full]. Note that the specification for `twice` has an abstract predicate `fpure`, so we should instantiate it as `fpure(x,res) |= res = x + 1` to make the verification succeed. The inference of instantiating is expected to be automatically done based on the existing specification of `incr`. However, to simplify the implementation, our current system requires users to manually provide the instance when an abstract predicate should be instantiated.

```
1 let incr x = x + 1
2 (*@ declare incr(x)
3 requires    { true }
4 ensures[r]  { r = x + 1 } @*)
5
6 let incr_twice x2 = twice incr x2
7 (*@ declare incr_twice (x2)
8  requires    { true }
9  ensures[r]  { r = x2 + 2 } @*)
```

Code 4.2: Higher order twice function application

In the verification of `incr_twice`, there are two invocations to the entailment checker. The first call takes place when applying [Fv-App-Full]. We need to check the derivation from the precondition of `incr_twice` to the precondition of `twice`, which involves the checking of both the pure predicate part and the specification part. The pure part is trivial, and the specification part is also straightforward since specification for argument `f` instantiated with `fpure(x,res) |= res = x + 1` is almost the same as the specification of `incr`, so the subsumption relation holds.

4.2 Function as return values

The next example shows how our system deals with function as return values. Consider use the specification of `div` to verify `div_by_two` in Code 4.3. The forward verifier will use [Fv-App-Par] to replace `y` with `2` in the specification of `div`, so the inferred assertion at the end of the function body will be

```
true with f(x) |= { 2 != 0 } *->:m { m * 2 = x }
```

The subsumption relation between the inferred function specification and the expected specification can then be checked by the entailment checker.

```
1  let div y x =  (x / y)
2  (*@ declare div(y, x)
3  requires    { ~ y = 0 }
4  ensures  [r]{ r * y = x } @*)
5
6  let div_by_two = (div 2)
7  (*@ declare div_by_two()
8  requires    { true }
9  ensures  [f]{ true with
10                 f(x) |= { true } *->:m { m * 2 = x } } @*)
```

Code 4.3: Function as return value

The specification `div_by_two` we have verified above can also be used when applied to higher order functions. Consider a similar use case of `twice` in Code 4.2. Recall that the previous example in Code 4.2 has the specification for `f(a)` readily in the precondition, so after normalization, the forward verifier can directly find `f` defined in the verification context, thus applying [Fv-Var]. By contrast, here the argument `div_by_two` has a specification without any arguments, and the actual specification we want is hidden in the postcondition. This is why we need the rule [Fv-Fun-Eval] to deal with our nested assertion structure. After applying [Fv-Fun-Eval], the remaining verification process is similar to that of Code 4.2

```
1  let div_by_four x = twice div_by_two x
2  (*@ declare div_by_four(x)
3  requires    { true }
4  ensures  [r]{ 4 * r = x } @*)
```

Returned function as parameter

4.3 Recursive predicates

```
1  let rec fib n =
2    if n = 0 then 1 else
3      if n = 1 then 1 else
4        fib (n - 1) + fib (n - 2)
5
6  (*@ declare fib(n)
7  requires { 0<=n }
8  ensures[res] { fibP(n,res) }
9
10 pred fibP(x:int,r:int) |=
11     x = 0 & r = 1 or  x = 1 & r = 1
12 or  EX (r1:int) (r2:int), r = r1 + r2
13       & fibP(x - 1,r1) & fibP(x - 2,r2) @*)
```

Code 4.4: Use recursive predicate to verify Fibonacci

As has been shown in the practice of the HIP/SLEEK system [2], user-definable predicates allow programmers to describe a wide range of functions and data structures with their associated shape, size and bag (multi-set) properties. To improve the expressiveness of our specification language, we also extend the our system to support reasoning on inductive predicates, so that behavior of recursive functions can be captured in the assertions.

A simple recursive function for computing Fibonacci is shown in 4.4. To define what the `x`-th entry of a Fibonacci sequence is, we define an inductive predicate `fibP`, where the predicate name `fibP` itself can appear in the third constructor body. With `fibP` defined, we can specify the function `fib` simply by using the `fibP` in the postcondition to describe the relation between the output `res` and the input `n`. Note that we do not write `fibP` in the given clause, since unlike previous examples with abstract predicates, `fibP` here is a concrete predicate.

After the forward verification, the entailment checker is asked to verify the following condition

```
 n >= 0 & n = 0 & res = 1
| n >= 0 & n != 0 & n = 1 & res = 1
| n >= 0 & n != 0 & n != 1 & fibP(n-1, res1) &
  fibP(n-1, res2) & res = res1 + res2
⇒  fibP(n, res)
```

This can be done by folding the LHS of the formula. In our implementation, we leave the predicate name `fibP` as an uninterpreted function in SMT solver. We also encode the definition of `fibP` as another constraint to the solver, so that the solver can make use of the folding rule to verify the above condition.

4.4 Verify functions on variant types

One common use case of higher order function often comes with certain data structures, so that operations on data structures can be abstracted, allowing programmers to write modular programs. In functional programs, data structures are typically implemented as variant datatypes. Code 4.5 presents an example of the list data structure. Our system supports verification of variant types by allowing users

to write user-definable predicates about data structures. We begin our example with a first order function calculating the length of a list.

```
1  type list = Nil
2            | Cons of (int * list)
3
4  (*@ pred LL(n:int,l:list) |= l::Nil<> & n=0
5      or EX (i:int) (q:list), l::Cons<i,q> & LL(n - 1, q) @*)
6
7  let rec length lst =
8  (*@ declare length(lst:list)
9      given (n:int)
10     requires      { LL(n,lst) }
11     ensures[res]   { res = n }  @*)
12   match lst with
13   | Nil -> 0
14   | Cons (x, xs) -> 1 + length xs
```

Code 4.5: Use inductive predicates to specify functions on variant types

The predicte `LL` is a binary relation on a list `lst` and its length `n`. It is inductively defined as follows: the length should be 0 on an empty list and if `LL(n-1,q)` holds, then appending a node before `q` can derive that the length of the new list is `n`. With `LL` defined, we can specify `length` that if we have a `n` such that the input `lst` satisfies `LL(n,lst)`, i.e. `n` is the length of `lst`, the return value `res` will be exactly the same as `n`.

According to the [Fv-Match] rule, the pattern-matching will create two disjunctive clauses during forward verification. The verifier can use the constraint introduced by pattern matching to determine which clause is used in the `LL` predicate. The `Nil` case is easy to verify. The `Cons` case has a recursive call to `length`. We can instantiate the specification with `n-1`, and apply [Fv-App-Full] to get the inferred postcondition as follows, which is straight forward to derive the expected postcondition `res = n`.

```
 lst::Nil<> & n = 0
| lst::Cons<x,xs> & LL(n-1,xs) & r'=n-1 & res = 1 + r'
```

Our system can also be used to verify higher-order functions that operate on variant datatypes. The key is integrate abstract predicates that describes function parameter behavior into the inductive predicates that characterizes properties on data structures.

Consider verifying the `fold_left` function in Code 4.6, which takes a function argument `f` on two integers, and outputs the result of repeatedly applying `f` to the list `ys`, where each application uses the last computed result and updates the first argument applied to `f`. We can define a predicate `LL_foldl` to reflect the behavior of this function. The predicate has four arguments, namely `x`, `ys`, and `fpure` as the input and `r` as the result. If `ys` is empty, then the result is equal to the input `x`. If the `ys` is a list node of value `y` and successor list `ys'`, there should exist an intermediate result value `z`, such that `z` is the result of `f(x,y)` and we can use `z` as the initial value to fold the rest of the list `ys'`.

```
1  (*@ pred LL_foldl(x,r,ys:list,fpure:int->int->int)
2  |= ys::Nil<> & x=r
3  or EX (ys':list) (y:int) (z:int), ys::Cons<y,ys'> &
4         LL_foldl(z,r,ys',fpure) & fpure(x,y) = z  @*)
5
6  let rec fold_left f x ys =
7  (*@ declare fold_left(f:int->int->int,x:int,ys:list)
8  given         (fpure(int,int):int), (r:int)
9  requires      { LL_foldl(x,r,ys,fpure)
10                     with f(x,y) |= {true} *->:r {r = fpure(x,y)} }
11 ensures[res]  { res = r } @*)
12 match ys with
13 | Nil -> x
14 | Cons (y, ys') -> fold_left f (f x y) ys'
```

Code 4.6: Use inductive predicates to specify functions on variant types

We make use of the user-definable predicate `LL_foldl` to specify `fold_left`. Note that `LL_foldl` takes an abstract predicate `fpure`, so we also need to specify that `fpure` reflects the behavior of the input function `f` in the precondition.

The `Nil` case is straightforward to verify. For the `Cons` branch, the specification of `f` first gives an intermediate result `r'=fpure(x,y)`. Unfolding `LL_foldl` gives an existantial variable `z = fpure(x,y)`. By the congruence rule of uninterpreted function, we can ensure `r' = z`. Next, at the recursive call site of `fold_left`, we can instantiate the `fold_length` specification with `r:=z` and get `res = r`, which completes the verification.

```
1  let fold_length x y = x + 1
2  (*@ declare fold_length(x,y)
3     requires      { true }
4     ensures[res]  { res=x+1 } @*)
5
6  let length xs = fold_left fold_length 0 xs
7  (*@ declare length(xs:list)
8     given (n:int)
9     requires      { LL(n,xs) }
10    ensures[res]  { res=n } @*)
```

Code 4.7: Application of `fold_left`

The specification of higher order function `fold_left` can now be applied to verify more concrete instances. Consider another `length` implementation using the `fold_left` function in Code 4.7. We can instantiate the `fold_left` specification with `fpure(x,y)=x+1` and `r=n` to apply rule [Fv-App-Full]. The enatilment checker is asked to verify that `LL(n,xs)` $\Rightarrow$ `LL_foldl(0,n,xs,fpure(x,y)=x+1)`. The entailment check can be done by induction on the structure of inductive predicates `LL` and `LL_foldl`. We can encode the verification condition to the SLEEK solver to deal with such kind of proof. We leave the integration of SLEEK into our system as future work.

Chapter 5

Conclusion and Future Work

5.1 Research process

This report is a summary of work starting from the late September of 2021. Before that, I took some time to read literature about automatic verification of higher order functions and to get familiar with the HIP/SLEEK system. Then I began proposing a simple solution to verify some higher order programs presented in Chapter 4. The proof rules in Chapter 3 were also driven by these examples. As I gradually enriched the logical system, I also implemented these rules into an OCaml prototype system.

5.2 Evaluation

The solution provided in this report is mainly trying to tackle automatic verification for higher order programs. The impact of this work can be summarised into following main points.

1. We proposed a logical system that can verify programs with higher order functions, which made use of abstract predicates and assertions with specifications as a solution.

2. We implemented a prototype system including a forward verifier and entailment checker.

3. We present some examples to demonstrate some practical use cases for verifying higher order functions.

5.3 Future Research Directions

Due to the limit of time, the current system is preliminary, but based on our framework, there are a few directions that can be explored to make our system more useful.

Prove soundness of the system We have not formally proved the soundness of our logical system. Our specification language makes it convenient for the logical system to perform forward verification. However, our definition of assertions may pose a challenge to model the semantics of the program and prove soundness. Therefore one future direction is to prove the soundness of our system to make our system more reliable.

Extend the logical system Currently our logical system only allows users to write pure predicates to characterize pure programs that do not have any side effects. However, modern functional languages also support mutable variables to improve efficiency. One particular benefit of using Hoare logic to verify programs is that it can be extended to separation logic [8] which provides a modular solution to verify programs with heap states. Our system has the potential to integrate with the HIP/SLEEK system and verify more program strictures.

More powerful instance inference One limit of our current system is that the power of automated verification is constrained by its ability to infer instances when a higher order function is called. Currently our system largely relies on user's manual effort to provide appropriate concrete predicates to proceed the proof. However, we also observe that many instances are naive and should be automatically inferred. Introducing more powerful inference techniques will greatly improve the practical value of our system.

We believe the above (but not limited to) future research directions will advance the higher order program verification techniques presented in this report.

Bibliography

[1] L. Beringer and A. W. Appel, "Abstraction and subsumption in modular verification of C programs", *Formal Methods in System Design*, ISSN: 15728102.

[2] W. N. Chin, C. David, H. H. Nguyen, and S. Qin, "Automated verification of shape, size and bag properties via user-defined predicates in separation logic", *Science of Computer Programming*, vol. 77, no. 9, pp. 1006–1036, Aug. 2012, ISSN: 01676423.

[3] C. A. Hoare, "An axiomatic basis for computer programming", *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969, ISSN: 15577317. [Online]. Available: `https://dl.acm.org/doi/abs/10.1145/363235.363259`.

[4] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "VeriFast: A powerful, sound, predictable, fast verifier for C and Java", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6617 LNCS, 2011, pp. 41–55, ISBN: 9783642203978.

[5] M. Kawaguchi, P. Rondon, and R. Jhala, "Type-based data structure verification", in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009, pp. 304–315, ISBN: 9781605583921.

[6] T. Kleymann, "Hoare logic and auxiliary variables", *Formal Aspects of Computing*, vol. 11, no. 5, pp. 541–566, 1999, ISSN: 09345043.

[7] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6355 LNAI, 2010, pp. 348–370, ISBN: 3642175104.

[8] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures", in *Proceedings - Symposium on Logic in Computer Science*, 2002, pp. 55–74.

[9] P. M. Rondon, M. Kawaguchi, and R. Jhala, "Liquid types", *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 159–169, 2008, ISSN: 15232867.

[10] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement types for Haskell", in *ACM SIGPLAN Notices*, vol. 49, 2014, pp. 269–282, ISBN: 9781450328739. [Online]. Available: `http://rise4fun.com/Dafny/wVGc`.

[11] L. Xavier, D. Damien, F. Alain, G. Jacques, R. Didier, and V. Jérôme, "OCaml - The OCaml Manual", [Online]. Available: `https://ocaml.org/manual/` (visited on 11/05/2021).

[12] N. Yoshida, K. Honda, and M. Berger, "Logical reasoning for higher-order functions with local state", *Logical Methods in Computer Science*, vol. 4, no. 4, 2008, ISSN: 18605974.