B.Comp. Dissertation

Formal verification of programs with arrays

By

Vu An Hoa

Department of Computer Science

School of Computing

National University of Singapore

2010/11

B.Comp. Dissertation

Formal verification of programs with arrays

By

Vu An Hoa

Department of Computer Science

School of Computing

National University of Singapore

2010/11

Project No: H018370
Advisor: Associate Professor Chin Wei Ngan
Deliverables:
Report: 1 Volume
Source Code: online

Abstract

Hip/Sleek is a novel software verification tool developed at National University of Singapore by Chin et al. Its strength lies in the capability of verifying most programs with recursive data structures while arrays, despite being prominent and intensively used in many programming languages, is an aspect that *was* left unsupported for a long time. In this report, we want to summarize our design and implementation of the Hip/Sleek's array reasoning functionalities as well as to illustrate and analyses some illustrating examples. We then discuss its verification power and conclude with some suggested improvement. Subject Descriptors:

D.2.1 Program Specifications
D.2.4 Program Verification
F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords:
Software Engineering, Software / Program Specification, Mathematical Logic, Programming Languages and application, SMT

Implementation Software and Hardware:
Ubuntu Linux, Ocaml 3.11, Hip/Sleek, Z3 2.19 (dependent software)

Acknowledgement

I owe much to Professor Chin Wei Ngan for his close and ardent guidance.

I also appreciate Professor Kan Min Yen and Professor Yang Yue (Department of Mathematics) for their mind-openning courses in Artificial Intelligence and mathematical logic.

Due to Mr Le Binh (Professor Chin's research assistant), I have the understanding of the original implementation of Hip/Sleek which turns out to be great help in my work.

Last but not least, I would like to send my thank and love to my dad and mom and all my friends for their continual support.

List of Figures

List of Tables

Table of Contents

Chapter 1

Introduction

1.1 Program formal verification

Formal verification of computer programs is a long standing topic in the theory of programming language and verification. It concerns with

> " act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics." (Wikipedia)

In short, the problem is to produce a *formal proof* for *formally specified properties of programs*. The precise meaning of the emphasized words will be given in chapter 2. In the mean time, we take a look at an example of a manually formally verified function.

Example. Show that the following function (method) in figure 1.1 always returns $1+2+\cdots+n$ for all input $n \geq 0$.

Proof. We proceed by induction on n.

- $n = 0$: then $sumfirst(n)$ returns 0. Obviously, $1 + \cdots + 0 = 0$.
- Suppose that $n \geq 0$ and $sumfirst(n)$ returns $1 + 2 + \cdots + n$. For input $n + 1$, since $n + 1 \geq 1 > 0$, the function on this input must returns $(n + 1) + sumfirst(n)$. From the

induction hypothesis, we deduce

$$(n+1) + sumfirst(n) = (n+1) + 1 + 2 + \cdots + n = 1 + 2 + \cdots + n + (n+1)$$

so that the statement is valid for $n+1$.

By principle of mathematical induction, $sumfirst(n)$ always returns $1 + 2 + \cdots + n$. □

```
int sumfirst(int n)
{
        if (n <= 0)
                return 0;
        else
                return n + sumfirst(n-1);
}
```

Figure 1.1: The first function to verify formally manually

From the above example and description of the problem, formal verification does not depend on the model of computation. That is to say, we neither consider the limitation of a variable of type **int** in the C programming language nor the fact that commutativity does not hold for floating number addition and multiplication (i.e. it might happen that $a + b \neq b + a$ and $a \times b \neq b \times a$ where a, b are floating point variables.) The proof in the example definitely fails if the real model is taken into consideration: when the input is sufficiently large, we get negative value for the output. So in fact, we are verifying *algorithms*, not actually programs. Nevertheless, this restriction is merely a minor one because programs can be deemed reliable once they have verified algorithms. Moreover, the reality is so complicated that it is so intractable to verify our programs. (There are, however, verifiers which tackle this general problem.)

Formally verified programs are highly desirable, especially for a practical programmer. Once his/her program's algorithm is proved to be correct, it is free from *algorithmic* bugs. So even if (s)he cannot claim it always works (due to the omnipresence of unexpected events such

aselectrical short circuit, solar storms, etc.), (s)he can claim that if the program does fail, it is not his/her fault.

The landmark in the theory of verification was introduced by C. A. R. Hoare in his 1969 paper (Hoare, 1969) entitled "An axiomatic basis for computer programming". In this work, Hoare borrowed the axiomatic method developed in the early 1900 by the mathematicians' community. He introduced the Hoare's triples and gave a detail framework to verify imperative programs. We will revisit this topic when we discuss about our implementation. Now, we shall state our problem in the next section.

1.2 The problem

Our existing Hip/Sleek is a powerful tool to verify recursive data structures such as linked list, stack, queue, binary tree, etc. (Hip is our front end that parses and verifies programs while Sleek is our separation logic checker engine.) However, it provided no support for programs with arrays. The reason is fairly simple: arrays are *not* recursive data structure in the sense that they are neither defined nor operated on recursively. A comparison between array and linked list should illustrate this difference:

- Array is stored as a contiguous chunk in the memory while linked lists are not.
- Array provides random access and modification because the memory location of an array's element can be computed directly from the index while in the linked lists, we need to traverse all the earlier elements via the pointers.

In this project, we try to extend this system to handle programs with arrays. This is an extremely important problem because arrays are intensively used in many applications: linear algebra softwares, computer graphics, dynamic programming, the list goes on. Arrays are even used to implement recursive data structures. The reason for its ubiquity is the instrinsic property: operations on arrays are efficient. Accessing and modifying array elements requires constant time (i.e. $O(1)$ time complexity) while they are usually linear (i.e. $O(n)$) or in the best cases, logarithmic time (i.e. $O(\log n)$) for most recursive data structures we know. (There

are trade offs for efficiency but here we are not concern with them.) Hence, it is an essential data structure for time efficient algorithms and hence, modern softwares.

On the other hand, this is also a challenging problem, again because of array's operational flexibility. For example, while recursive data structures offer obvious recursive method of operations, it is not true for arrays. In fact, we can recurse on any portion of the arrays (which is important for quick sort, merge sort algorithms.)

In the time limit of this project, we focus on classical operations on arrays, for instance, computing the sum of elements of the arrays, sort the array, etc.

1.3 Organization of this report

In this section, I shall give an overview of the coming chapters.

In chapter 2, I make a brief revision of first order logic and a description of the Satisfiability Modulo Theory problem. These background topics are not only relevant to understand our implementation but give us a basic framework to evaluate the program as well. The material on first order logic is presented as it usually is in common textbooks (except for the section on formal deduction) so readers who are familiar with logic and formal systems can skip this chapter and go straight to chapter 3 on our implementation.

The remaining part of the report describes our implementation (chapter 3) together with several showcase examples. We shall end up with some critical evaluation (chapter 4) on the current system and also compare it with an existing solution, namely the combination Frama-C with the Jessie plug in and Why framework.

Chapter 5 concludes this report with some highlights on feasible enhancements.

Chapter 2

Review of classical first order logic

2.1 First order language

The language of first order logic is a set consisting of

- Logical symbols
  - Logical connectives: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
  - Quantifiers: $\forall$ and $\exists$
  - Equality: $=$
  - Variables $V = \{v_1, v_2, ...\}$
  - Parentheses: (and)
- Parameters
  - Functions $F = \{f_1, f_2, ...\}$
  - Relations $R = \{R_1, R_2, ...\}$
  - Constants $C = \{c_1, c_2, ...\}$

Each symbol for function and relation is associated with a natural number called its rank or arity. This number is either the number of inputs that the function requires or the number of objects the relation binds.

Notice that we have infinitely many languages for first-order logic because there are infinitely many choices of the parameters. Since the logical symbols are always included, I shall omit them in language specification. My notational convention is

$$\mathcal{L} = F \cup R \cup C$$

where the superscript number is used to denote the arity of function (or relation) symbols. For illustration, $\mathcal{L} = \{A^2\} \cup \emptyset \cup \emptyset$ defines a language $\mathcal{L}$ of first order logic with all logical symbols, one binary function A, and there is no relation or constant.

Starting with a first order language, we define the notion of terms and formulas.

Definition. A *term over a first order language* $\mathcal{L}$ is defined inductively

- A variable or a constant symbol in $\mathcal{L}$ is a term.
- If $f_i \in \mathcal{L}$ is a function symbol of arity k and $t_1, \cdots, t_k$ are terms (over $\mathcal{L}$) then $f_i t_1 t_2 \cdots t_k$ is a term.

In the earlier example, Av_1v_2, $AAv_1v_2v_3$ are terms over the language $\mathcal{L} = \{A^2\} \cup \emptyset \cup \emptyset$.

Definition. *Wellformed formulas over a first order language* $\mathcal{L}$ are defined inductively:

- $t_1 = t_2$ is a wellformed formula for any t_1, t_2 being terms over $\mathcal{L}$.
- If $R_i \in \mathcal{L}$ is a k-ary relation symbol and $t_1, t_2, ..., t_k$ are terms over $\mathcal{L}$, then $R_i t_1 t_2 ... t_k$ is also a wellformed formula.
- If α, β are wellformed formulas and v_i is a variable symbol, so are $(\neg\alpha)$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \to \beta)$, $(\alpha \leftrightarrow \beta)$, $\exists v_i \alpha$, $\forall v_i \alpha$.

Formulas in the first two cases are called *atomic formulas*.

Definition. A variable v_i *occurs freely* in α (or v_i is a *free variable* of α) if and only if (i) α is atomic and v_i appears in α; (ii) $\alpha = (\neg\beta)$ or $\alpha = (\beta \square \gamma)$ and either v_i occurs freely in β or in γ where $\square \in \{\wedge, \vee, \to, \leftrightarrow\}$; (iii) $\alpha = \exists v_j \beta$ or $\alpha = \forall v_j \beta$ for $j \neq i$ and v_i occurs freely in β.

Example. Over $\mathcal{L} = \{A^2\} \cup \emptyset \cup \emptyset$, $Av_1v_2 = v_1$, $(Av_1v_2 = v_1 \to v_1 = v_2)$, $\forall v_2(Av_1v_2 = v_1)$ are wellformed formulas. In the first two formulas, v_1, v_2 are free variables while in the last, only v_1 is a free variable.

We will just say "formula" to mean "wellformed formula".

2.2 Formal deduction

At this stage, terms and formulas are meaningless sequence of symbols. *Formal deduction* (or *derivation*) is the idea of syntactical (symbollic) manipulations on the formulas to derive new ones. Machines, even though fails to understand the semantics, can perform efficiently syntactical manipulation.

There are a number of deduction systems for first order logic, for examples, Hilbert's proof system and Gentzen's natural deduction. Different they are, they are in fact equivalent (in the sense that formula derived by one system can be derived by the other). Hilbert's system contains a number of *logical axioms* and one single manipulation rule (Modus Ponens) and hence, it is more convenient to use in mathematics. To us, we will use Gentzen's system as it assembles a more familiar reasoning system.

Gentzen's natural deduction provides $2 \times 7 + 2$ rules including two special rules to handle equality and the pair introduction/elimination rule for other logical symbols. Those rules are given in table 2.1.

Definition. For any formula α and collection of formulas Γ, the notation $\Gamma \vdash \alpha$ is to be read "α is *derivable* from Γ". Recursively, we define $\Gamma \vdash \alpha$ if and only if either

- $\alpha \in \Gamma$; or
- $\Gamma \vdash \alpha$ according to Gentzen's rules of natural deduction.

Example. Consider $\mathcal{L} = \emptyset \cup \{R^1\}\emptyset$. We claim that $\Gamma \vdash \alpha$ for $\Gamma = \{\exists v_1(Rv_1)\}$ and $\alpha = \neg\forall v_2\neg(Rv_2)$. We have

$$\{Rv_3\} \cup \{\forall v_2\neg(Rv_2)\} \vdash \neg(Rv_3)$$

Table 2.1: Gentzen's deduction rules

Symbol	Introduction	Elimination
Negation $\neg$	If $\Gamma \cup \{\alpha\} \vdash \beta$ and $\Gamma \cup \{\alpha\} \vdash \neg\beta$ then $\Gamma \vdash \neg\alpha$	$\Gamma \cup \{\neg\neg\alpha\} \vdash \alpha$
Conjuction $\wedge$	$\Gamma \cup \{\alpha, \beta\} \vdash \alpha \wedge \beta$	$\Gamma \cup \{\alpha \wedge \beta\} \vdash \alpha$ and $\Gamma \cup \{\alpha \wedge \beta\} \vdash \alpha$
Disjunction $\vee$	$\Gamma \cup \{\alpha\} \vdash \alpha \vee \beta$ and $\Gamma \cup \{\beta\} \vdash \alpha \vee \beta$	If $\Gamma \cup \{\alpha\} \vdash \gamma$ and $\Gamma \cup \{\beta\} \vdash \gamma$ then $\Gamma \cup \{\alpha \vee \beta\} \vdash \gamma$
Implication $\rightarrow$	If $\Gamma \cup \{\alpha\} \vdash \beta$ then $\Gamma \vdash \alpha \rightarrow \beta$	$\Gamma \cup \{\alpha, \alpha \rightarrow \beta\} \vdash \beta$
Equivalent $\leftrightarrow$	$\Gamma \cup \{\alpha \rightarrow \beta, \beta \rightarrow \alpha\} \vdash \alpha \leftrightarrow \beta$	$\Gamma \cup \{\alpha \leftrightarrow \beta\} \vdash \alpha \rightarrow \beta$ and $\Gamma \cup \{\alpha \leftrightarrow \beta\} \vdash \beta \rightarrow \alpha$
Universal $\forall$	$\Gamma \cup \{\alpha(v_i)\} \vdash \forall v_i \alpha$	$\Gamma \cup \{\forall v_i \alpha\} \vdash \alpha[v_i/t]$
Existential $\exists$	$\Gamma \cup \{\alpha[v_i/t]\} \vdash \exists v_i \alpha$	If $\Gamma \cup \{\alpha[v_i/t]\} \vdash \gamma$ then $\Gamma \cup \{\exists v_i \alpha\} \vdash \gamma$

Notes:

There are restrictions on Introduction and Elimination rules for logical quantifiers. I omit them here for the sake of simplicity.

The notation $\alpha[v_i/t]$ denotes a formula obtained by replacing *unquantified* instances of v_i in α by the term t. It can be defined recursively: if α is atomic, $\alpha[v_i/t]$ is simply obtained by replacing all v_i by t; if $\alpha = (\neg\beta)$ or $\alpha = \exists v_j \beta$ or $\alpha = \forall v_j \beta$ for $j \neq i$ then $\alpha[v_i/t]$ is $\neg\beta[v_i/t]$ or $\exists v_j \beta[v_i/t]$ or $\forall v_j \beta[v_i/t]$ respectively; if $\alpha = (\beta \square \gamma)$ then $\alpha[v_i/t] = \beta[v_i/t] \square \gamma[v_i/t]$ for $\square \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$; otherwise, $\alpha[v_i/t] = \alpha$.

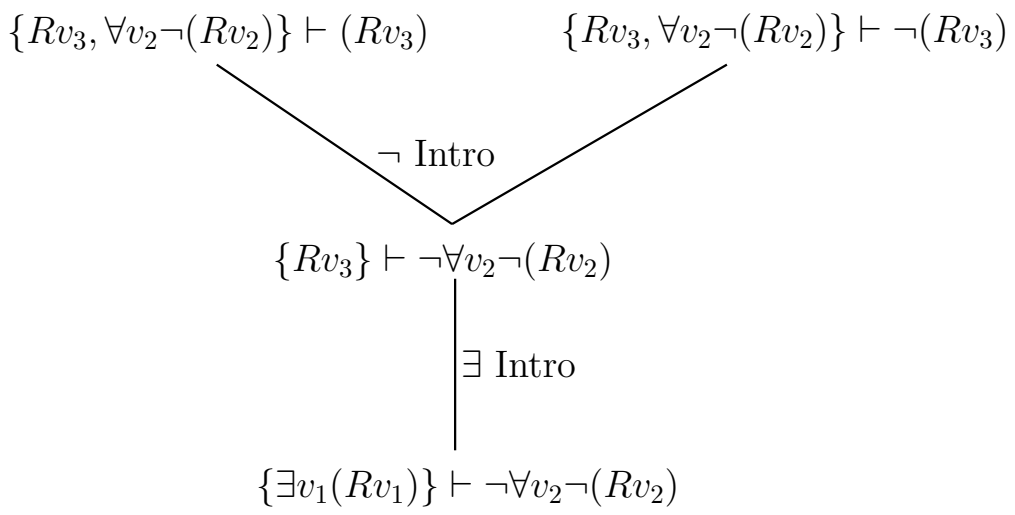

Figure 2.1: Formal proof tree for $\{\exists v_1(Rv_1)\} \vdash \neg\forall v_2 \neg(Rv_2)$

by universal elimination. Also

$$\{Rv_3\} \cup \{\forall v_2 \neg(Rv_2)\} \vdash (Rv_3)$$

because the RHS belongs to LHS. Hence, by negation introduction, we obtain

$$\{Rv_3\} \vdash \neg\forall v_2 \neg(Rv_2).$$

Then utilizing existential introduction on $\alpha := Rv_1$ and $v_i := v_3$, we get

$$\{\exists v_1(Rv_1)\} \vdash \neg\forall v_2 \neg(Rv_2)$$

which is exactly what we want to show.

The previous step-by-step reasoning can be put in form of a tree as in figure 2.1. In general, if $\Gamma \vdash \alpha$ then there must be a tree, which we call a formal proof for $\Gamma \vdash \alpha$.

Definition. A *formal proof* for $\Gamma \vdash \alpha$ is a tree whose nodes are labeled by $\Delta \vdash \beta$ where Δ is set of formulas and β is a formula. The relation between a node and its children is that by applying one Gentzen's rule with the information given by the children, we get the content of the node.

We realize from the examples that intuitively speaking, most mathematical proofs can be formalized into a formal proof.

The purpose of this section is to introduce the method of formal deduction and hence, explain precisely the meaning of the phrase "formal method of mathematics" that was introduced in chapter 1. The objective of formal verification is thus construct the *formal proof* for the correctness of a program. Notice that a formal proof can be verified easily (i.e. there is an algorithm to check correctness of a proof.) Thus, the output of a verifier can be inputed into a proof verifier to double check the property and ensure its reliability.

2.3 Structure, instantiation and validity

Definition. A *structure* $\mathfrak{S}$ for a first-order language $\mathcal{L}$ consists of

- A *universe of discourse* (or just *universe*): a non-empty set denoted by $|\mathfrak{S}|$.
- Parameters' *interpretations*
  - Each constant c_i is associated with a value $c_i^{\mathfrak{S}}$ in $|\mathfrak{S}|$.
  - Each k-ary function f_i is associated with a function $f_i^{\mathfrak{S}} : |\mathfrak{S}|^k \to |\mathfrak{S}|$.
  - Each k-ary relation R_i is associated with a k-ary relation on $|\mathfrak{S}|$ i.e. a subset of $|\mathfrak{S}|^k$.

Definition. A *variable instantiation* ε in a structure $\mathfrak{S}$ is a map that associates each variable symbol v_i to a value $v_i^{\mathfrak{S}}$ in $\mathfrak{S}$.

Given a structure and a variable instantiation, we can associate values (semantics) to terms as well and we can also decides validity of formulas. The notion term instantiation and validity of a formula will be made precise in the following definitions.

Definition. Let $\mathfrak{S}$ be a structure for $\mathcal{L}$ and ε be a variable instantiation. The *term instantiation extending* ε is a map $\overline{\varepsilon}$ that associates each terms over $\mathcal{L}$ with a value in $\mathfrak{S}$ and is defined by

- For each variable v_i, $\overline{\varepsilon}(v_i) = \varepsilon(v_i)$.
- If f_i is a k-ary function and $t_1, t_2, ..., t_k$ are terms then

$$\overline{\varepsilon}(f_i t_1 t_2 ... t_k) = f_i^{\mathfrak{S}} \overline{\varepsilon}(t_1) ... \overline{\varepsilon}(t_k).$$

Definition. Suppose that $\mathfrak{S}$ is a structure over $\mathcal{L}$, ε is a variable instantiation and φ is a formula. We denote $\models_{\mathfrak{S}} \varphi[\varepsilon]$ to stand for φ is *valid* (i.e. φ is *true*) in $\mathfrak{S}$ with respect to ε and $\not\models_{\mathfrak{S}} \varphi[\epsilon]$ otherwise. The validity of a formula is defined recursively as follow

- $\models_{\mathfrak{S}} t_1 = t_2[\varepsilon]$ if and only if $\overline{\varepsilon}(t_1) = \overline{\varepsilon}(t_2)$
- $\models_{\mathfrak{S}} R_i t_1 t_2 ... t_k[\varepsilon]$ if and only if $(\overline{\varepsilon}(t_1), \overline{\varepsilon}(t_2), ..., \overline{\varepsilon}(t_k)) \in R_i^{\mathfrak{S}}$
- $\models_{\mathfrak{S}} \neg\alpha[\varepsilon]$ if and only if $\not\models_{\mathfrak{S}} \alpha[\varepsilon]$
- $\models_{\mathfrak{S}} \alpha \wedge \beta[\varepsilon]$ if and only if $\models_{\mathfrak{S}} \alpha[\varepsilon]$ and $\models_{\mathfrak{S}} \beta[\varepsilon]$
- $\models_{\mathfrak{S}} \alpha \vee \beta[\varepsilon]$ if and only if either $\models_{\mathfrak{S}} \alpha[\varepsilon]$ or $\models_{\mathfrak{S}} \beta[\varepsilon]$
- $\models_{\mathfrak{S}} \alpha \to \beta[\varepsilon]$ if and only if either $\not\models_{\mathfrak{S}} \alpha$ or $\models_{\mathfrak{S}} \beta[\varepsilon]$
- $\models_{\mathfrak{S}} \alpha \to \beta[\varepsilon]$ if and only if both $\models_{\mathfrak{S}} \alpha \to \beta[\varepsilon]$ and $\models_{\mathfrak{S}} \beta \to \alpha[\varepsilon]$
- $\models_{\mathfrak{S}} \forall v_i \alpha$ if and only if for any $d \in |\mathfrak{S}|$, $\models_{\mathfrak{S}} \alpha[\varepsilon^d_{v_i}]$
- $\models_{\mathfrak{S}} \exists v_i \alpha$ if and only if there exists $d \in |\mathfrak{S}|$ such that $\models \alpha[\varepsilon^d_{v_i}]$

where $\varepsilon^d_{v_i}$ denotes the variable instantiation that maps the variable v_i to d and agrees with ε on the other variables i.e.

$$\varepsilon^d_{v_i}(v_j) = \begin{cases} d & \text{if } j = i \\ \varepsilon(v_j) & \text{otherwise} \end{cases}.$$

Example. Consider the first order language $\mathcal{L} = \{S^1, A^2, M^2\} \cup \emptyset \cup \{z\}$. Let $\mathfrak{S}$ be a structure with universe $|\mathfrak{S}| = \mathbb{N}$ and $S^{\mathfrak{S}} : x \mapsto x + 1$, $A^{\mathfrak{S}} : (x, y) \mapsto x + y$, $M^{\mathfrak{S}} : (x, y) \mapsto x \times y$ and $z^{\mathfrak{S}} = 0$ be the interpretation of the parameters in $\mathfrak{S}$ and ε be the map $v_i \mapsto i \in \mathbb{N}$. Then we have $\models_{\mathfrak{S}} \forall v_1 (Azv_1 = v_1)[\varepsilon]$, $\models_{\mathfrak{S}} \forall v_1 (Mzv_1 = z)[\varepsilon]$, for all index i, $\models_{\mathfrak{S}} Sv_i = v_{i+1}[\varepsilon]$.

Definition. Let Γ be a set of formulas and α be a formula. We denote by $\Gamma \models \alpha$ if for any structure $\mathfrak{S}$ and any instantiation ε, whenever $\models_{\mathfrak{S}} \varphi[\varepsilon]$ for every $\varphi \in \Gamma$, we have $\models_{\mathfrak{S}} \alpha[\varepsilon]$.

2.4 Many-sorted first order logic

Many-sorted first order logic is a syntactical extension of first order logic. Nevertheless, the expressiveness is the same. In this version, we extend the language to contain symbols for *sort* or *types*. The universes for the sorts are independent. (The original version has only one sort.)

Example. Consider the first order language with two sorts Z, R and a binary relation P on $Z \times R$, $\forall v_1 \in Z \forall v_2 \in R(Pv_1v_2)$ is a wellformed formula. A formula is wellsorted if the quantification matches the domains. The formula in the example above is wellsorted. The formula $\forall v_1 \in Z \forall v_2 \in R(Pv_2v_1)$ is not wellsorted.

Many-sorted first-order logic is more frequently used in mathematics than the standard version. Most of us should be familiar with formulas like

$$\forall x \in \mathbb{R} \exists n \in \mathbb{Z}(n \geq x).$$

These are exactly instances of many-sorted first order logic. For simplicity, we choose not to elaborate more on this extension.

2.5 Satisfiability Modulo Theory (SMT)

The satisfiability problem for first order logic is posed analogously to the problem of satisfiability for propositional logic (commonly known as *Boolean SAT*). The precise statement is given below

Definition. Given a set of formula Γ over a first-order language $\mathcal{L}$, decide whether there exists a structure $\mathfrak{S}$ and variable instantiation ε such that for every $\varphi \in \Gamma$, we have $\models_{\mathfrak{S}} \varphi[\varepsilon]$. If there is such $\mathfrak{S}$ and ε, we say that Γ is *satisfiable*, otherwise, Γ is said to be *unsatisfiable*.

We shall call this problem *first order satisfiability problem* (First order SAT).

Example. $\{\forall v_1(v_1 \neq v_2)\}$ is unsatisfiable while for any $k \in \mathbb{N}$,

$$\left\{ \exists v_1 \exists v_2 \exists v_3 ... \exists v_k \left(\bigwedge_{i \neq j} \neg(v_i = v_j) \right) \right\}$$

is satisfied by any structure $\mathfrak{S}$ whose universe $|\mathfrak{S}|$ has at least k elements.

An interesting remark is that the original Boolean SAT is also an instance of First order SAT. Consider the language

$$\mathcal{L} = \{N^1, A^2, O^2\} \cup \emptyset \cup \{T, F\}$$

The intention of the function symbols is to be interpreted as Boolean functions corresponding to the logical connectives. For each propositional formula, we can construct a corresponding term recursively

- The term for a propositional symbol P_i is the variable v_i.
- If α, β are formulas in propositional logic with corresponding terms t_α and t_β over $\mathcal{L}$, then the term corresponding to $\neg\alpha$, $\alpha \wedge \beta$ and $\alpha \vee \beta$ are Nt_α, $At_\alpha t_\beta$, and $Ot_\alpha t_\beta$ respectively.

Suppose that we want to check satisfiability of a propositional formula α. Let t be its corresponding term in $\mathcal{L}$. Let

$$\begin{aligned}\Gamma_\alpha := \{&\neg T = F, \forall v_1(v_1 = T \vee v_1 = F), NT = F, NF = T,\\ &\forall v_1 \forall v_2(Av_1v_2 = T \leftrightarrow v_1 = T \wedge v_2 = T),\\ &\forall v_1 \forall v_2(Ov_1v_2 = F \leftrightarrow v_1 = F \wedge v_2 = F),\\ &t = T\}\end{aligned}$$

If Γ is satisfiable and hence is satisfied by some structure $\mathfrak{S}$ and an instantiation ε, it must be the case that $|\mathfrak{S}| = \{T^{\mathfrak{S}}, F^{\mathfrak{S}}\}$ by the first two conditions. And if we choose the assignment $P_i :=$ **true** if $\varepsilon(v_i) = T^{\mathfrak{S}}$ and $P_i :=$ **false** otherwise then we satisfy the formula α. For example, take $\alpha = P_1 \wedge P_2 \wedge (\neg P_2 \vee P_3)$ and so $t = Av_1Av_2ONv_2v_3$. The set Γ_α is satisfiable and so α is satisfiable.

Having to construct the whole structure is so general (and hence is difficult). In practice, we usually are not interested in constructing the structure. Instead, we have a structure and try to find the instantiation of the variables that makes a collection of formulas true. This problem can be stated formally as

Definition. Given a set of formula Γ over a first-order language $\mathcal{L}$ and a structure $\mathfrak{S}$. Decide whether there is a variable instantiation ε such that for every $\varphi \in \Gamma$, we have $\models_{\mathfrak{S}} \varphi[\varepsilon]$. If there is, produce the instantiation ε.

For first order logic, the first order SAT problem in general is not decidable according to Church-Turing's theorem. (Interested readers can confer (Church, 1936) and (Turing, 1936) for exact details.) Neither is its structure-restricted version: Even though the structure is fixed, the problem is not trivial and deciding truth value is still undecidable. To see this, we take an example: the formula

$$\begin{aligned}\varphi = \forall n((\exists m(n = 2m) \wedge n > 2) \rightarrow \\ \exists p \exists q(n = p + q \wedge \forall u \forall v(u \times v \neq p \wedge u \times v \neq q))\end{aligned}$$

is a first-order formula which is the formal statement of Goldbach's conjecture in the language of arithmetic. The truth value of φ is hitherto not decided in the standard structure with universe $\mathbb{N}$ and natural interpretations of $2, +, \times$.

There are special cases in which this problem is decidable. The classical examples are *Boolean SAT*, *Presburger arithmetic* (arithmetic with only addition), *Skolem arithmetic* (arithmetic with only multiplication), *elementary real number algebra and geometry* (Tarski's elimination of quantifiers), etc. To sum up, decidability of SMT problem depends on the parameters of the underlying logic i.e. the expressiveness of logic.

Satisfiability Modulo Theory is a blend between two described versions of first order SAT problem. The phrase *"modulo theory"* can be explained as follow: In the practical context, we are not interested in any arbitrary structure but we concern more about some standard theories (such as theory of integers, real numbers, arrays, etc.) These theories have a standard language and a standard (intended) structure. For instance, the language for theory of integer $\mathcal{L}_{\mathbb{Z}}$ includes arithmetical functions $\{+, -, \times, /\}$, comparison relations $\{<, >, \geq, \leq\}$, and integral constants. The intended structure $\mathfrak{S}$ for $\mathcal{L}_{\mathbb{Z}}$ is a structure with universe $|\mathfrak{S}| = \mathbb{Z}$ and standard interpretations of the functions and relations.

For a language $\mathcal{L}$, a extended language $\mathcal{L}'$ is a language obtained by adding new parameters

to $\mathcal{L}$. In SMT problem, the user is free to add new prameters to the language. The problem is to find (in addition to the instantiation ε) an interpretation for the additional parameters.

Definition (Satisfiability Modulo Theory)**.** Let $\mathfrak{S}$ be a structure over a language $\mathcal{L}$ and $\mathcal{L}'$ is an extended language of $\mathcal{L}$. Given a collection of formulas Γ over $\mathcal{L}'$, decide if $\mathfrak{S}$ can be extended to a structure $\mathfrak{S}'$ for $\mathcal{L}'$ for which there is an instantiation ε such that Γ is satisfied in $\mathfrak{S}'$ with respect to ε.

Example. Let $\mathcal{L}$ be the language obtained by adding a unary relation P to $\mathcal{L}_{\mathbb{Z}}$. Then $\Gamma = \{\forall v_1(Pv_1 \leftrightarrow \forall v_2 \forall v_3(v_1 = v_2 \times v_3 \rightarrow v_2 = v_1 \vee v_2 = 1 \vee v_2 = -v_1 \vee v_2 = -1))\}$ is satisfiable if we interpret P as the collection of prime numbers and their negations.

In our context, we are interested in this problem because our data types (like **int**, **float**, ...) already has their semantics defined and hence, the universe as well as the basic operations should be fixed in their interpretation. We do want to allow uninterpreted symbol because it is useful to simplify our formulas as we have just seen the definition of the primality relation. Note that the presence of these uninterpreted symbol usually make it difficult to solve the SMT problem and in fact, most solver cannot output "sat" even in case the set of formulas is satisfiable (because verifying the formulas might need an infinite amount of checking.) However, we are interested in unsatisfiability, not satisfiability; in which case, SMT solvers still can provide satisfactory results.

2.6 SMTLIB and existing SMT solvers

SMTLIB is "an international initiative aimed at facilitating research and development in SMT". The organization provides the standard for SMT solvers including:

- Rigorous descriptions of background theories such as standard theories of arithmetics, real numbers, bit vectors, arrays, etc.
- Common input-output language i.e. language to write formula, specify logic, theory, etc. as well as to interact with the solver.

- Library of benchmarks for solver evaluation.

A precise description of SMTLIB syntax can be found by the publications (Barrett, Stump, & Tinelli, 2010b) and (Barrett, Stump, & Tinelli, 2010a).

There are many SMT solvers available today. In SMTLIB's record, the following solvers are well maintained today: Alt-Ergo, Barcelogic, Beaver, Boolector , CVC3, DPT, MathSAT, OpenSMT, SatEEn, Spear, STP, SWORD, UCLID, veriT, Yices, Z3.

There is one remark I want to make on the output of the solvers. Suppose that we want to check satisfiability of Γ over the extended $\mathcal{L}'$. In SMTLIB standard, there are three possible outputs

- `sat`: the solver can *find* an extended structure $\mathfrak{S}'$ and an instantiation ε that make every member of Γ valid.
- `unsat`: the solver can *demonstrate* that no such $\mathfrak{S}'$ and ε exists. Since the proof must be *finite*, Γ is in fact unsatisfiable in this case.
- `unknown`: the solver cannot determine satisfiability.

The restriction on the output is to ensure faithfulness (i.e. to eliminate false results). Most solvers have options to access the structure it can find in `sat` case. Unfortunately, many of them do not provide proofs for `unsat` case. Hopefully, there are on-going development in this aspect, such as (Böhme, 2009).

I conclude this section on first-order logic here and leave interested readers with more details in standard treatments in mathematical logic textbooks ((George Boolos, 1985),(Mendelson, 1997),(Enderton, 2001)) that can be found in the reference at the end of this report. (A remark is that the presentation of formal deduction is my own version and hence, is different from most textbooks'. But they are equivalent notions.) The references (Russell, Norvig, Candy, Malik, & Edwards, 1996), (Duffy, 1991), (Gallier, 1985), (Robinson, 2001) might be valuable to those who wish to learn the state-of-the-art in theorem proving and SMT solving. So are many recent papers and documentations that can be found on the enlisted tools' websites.

Chapter 3

Implementation

3.1 Our verification method

Hip/Sleek uses the symbollic execution mechanism to prove correctness of programs. This technique is widely used in formal verification softwares.

Hip/Sleek makes two important assumptions:

- All loops terminates.
- Every recursive function is well-founded.

In our extension of the system to array, we add an additional assumption:

- There is no memory aliasing.

We did not want to consider the memory issue of arrays because our implementation focused on the 'value' model of arrays i.e. array are manipulated just like an integer. The memory issue can be dealt with when we extend our array model using separation logic (confer (Reynolds, 2002) for a detail description). This topic is roughly explored in (Cuong, 2010).

In the scope of this project, we investigate the *value model* of array. That is to say, arrays operate just like integer or floating point number. To be precise, each array updation (i.e. `a[i] = v`) is converted into a function `update_array(a, i, v)` which returns a *new* array. This model fits some programming language in which array underlying memory allocation and operation are performed by the compiler without explicit awareness of users.

Statement	a'	b'	c'
	1	2	3
a := a + 1	2	2	3
b := a + 2	2	4	3
c := a * b	2	4	8

(a) Program tracing

Statement	a'	b'	c'
	a_0	b_0	c_0
a := a + 1	$a_0 + 1$	b_0	c_0
b := a + 2	$a_0 + 1$	$a_0 + 1 + 2 = a_0 + 3$	c_0
c := a * b	$a_0 + 1$	$a_0 + 3$	$(a_0 + 1)(a_0 + 3) = a_0^2 + 4a_0 + 3$

(b) symbolic execution

Table 3.1: Program tracing vs. symbolic execution

3.1.1 Symbollic execution - Hoare's triples

Symbollic execution of a program (or a function) refers to the idea of *tracing* the *state* of program (or a function) in an abstractly algebraic way. (Program tracing is a debugging terminology which means *go through the sequence of program statements one by one.* The state of a program mainly consists of the memory configuration.)

Example. We use the notations a', b', c' stand for the current value of the variable a, b, c respectively at the program point. Table 3.1 traces and executes a simple program symbolicly. With symbolic execution, we can identify the relationship between the output and the input in general.

At this stage, we have a vague idea of symbolic execution. We shall borrow Hoare's idea in (Hoare, 1969) to introduce a simple formal system (or language) that captured this idea. Similar

to what we present in chapter 2, we describe Hoare's language, Hoare's deduction system and finally Hoare's concept of validity.

Hoare's Language

For simplicity, we consider the case when we have only one type, integer. The general case can be dealt with by extending the underlying logic and programming language.

Hoare's language consists of the language for full arithmetic $\mathcal{L}_A := \{+^2, -^2, \times^2, /^2\} \cup \{>, <\} \cup \mathbb{Z}$ (or a subset of it) together with a minimal set of programming operators and constructs, namely $\{:=, \mathsf{while}\ ,\ \mathsf{do}\ , \mathsf{if}\ ,\ \mathsf{then}\ ,\ \mathsf{else}\ , ; \}$. (Note that the branching construct $\mathsf{if}\ \beta\ \mathsf{then}\ T\ \mathsf{else}\ E$ can be expressed in form $v_\infty := 0; \mathsf{while}\ (\beta \wedge v_\infty = 0)\ \mathsf{do}\ (T; v_\infty := 1); \mathsf{while}\ (\neg\beta \wedge v_\infty = 0)\ \mathsf{do}\ (E; v_\infty := 1)$ where v_∞ refers to an unused variable. It can be seen that any computation can be done using this simple language.)

In addition to wellformed formulas, we have notions of *program statement* and *Hoare's triple*.

Definition. *Program statements* can be defined recursively.

- $v_i := t$ is a program statement where v_i is a variable and t is a term over $\mathcal{L}_A$.
- if β is a *quantifier-free* formula over $\mathcal{L}_A$ and P, Q are program statements then $\mathsf{while}\ \beta\ \mathsf{do}\ P$ and $\mathsf{if}\ \beta\ \mathsf{then}\ P\ \mathsf{else}\ Q$ are program statements.
- if P, Q are program statements then $(P; Q)$ is a program statement.

Definition. A Hoare's triple is a string of form

$$\alpha\{P\}\beta$$

where α, β are first order logic formulas, usually called *precondition* and *postcondition* (respectively) and P is a statement.

The intended meaning of a Hoare' triple is: if α is true before P is executed then β must be true after P executes and terminates. We shall define this formally later.

Example. $v_1 := 1; v_2 := v_1 \times 2; \mathsf{while}\ (\neg v_1 = v_2)\ \mathsf{do}\ v_1 := v_1 + 1$ is a program statement. $v_5 = 1\{v_2 := 3\}v_5 = 2$ is a Hoare's triple.

Deduction system for Hoare's triples

In this formal system, we are mainly interested in derivation systems for Hoare's triples. First, let Λ be the collection of axioms for arithmetics i.e. $\Lambda = \{v_1 \times v_2 = v_2 \times v_1, v_1 \times (v_2 + v_3) = v_1 \times v_2 + v_2 \times v_3, ...\}$. (For simplicity, we do not describe Λ in full but it is worth emphasizing that Λ is finite.) Now we describe Hoare's derivation system which consists of four rules:

We use $\Vdash$ instead of $\vdash$ as the latter is already used for formal deduction of first order logic.

(D0) $\emptyset \Vdash \alpha[v_i/t]\{v_i := t\}\alpha$

(D1) If $\Lambda \vdash \beta \to \gamma$ then $\alpha\{P\}\beta \Vdash \alpha\{P\}\gamma$. If $\Lambda \vdash \gamma \to \alpha$ then $\alpha\{P\}\beta \Vdash \gamma\{P\}\beta$. Recall that the derivations $\Lambda \vdash \beta \to \gamma$ and $\Lambda \vdash \gamma \to \alpha$ are derivations *in first order logic* using Gentzen's rules.

(D2) $\alpha\{P_1\}\beta$, $\beta\{P_2\}\gamma \Vdash \alpha\{P_1; P_2\}\gamma$ (The left hand side is a set of triples but we choose to drop the set notation for it confuses with the pair $\{\}$ in the notation of a triple.)

(D3) $\alpha \wedge \beta\{P\}\alpha \Vdash \alpha\{\mathsf{while}\ \beta\ \mathsf{do}\ P\}\neg\beta \wedge \alpha$ (The formula α is commonly known as *loop invariant*.)

(D4) $\alpha \wedge \beta\{P\}\gamma, \alpha \wedge \neg\beta\{Q\}\gamma \Vdash \alpha\{\mathsf{if}\ \beta\ \mathsf{then}\ P\ \mathsf{else}\ Q\}\gamma$

Example. By (D0),

$$\emptyset \Vdash 5 = 5\{v_1 := 5\}v_1 = 5 \qquad (1)$$

and

$$\emptyset \Vdash v_1 = 5 \wedge v_1 + 4 = 9\{v_2 := v_1 + 4\}v_1 = 5 \wedge v_2 = 9.$$

As $\Lambda \vdash v_1 = 5 \to v_1 = 5 \wedge v_1 + 4 = 9$,

$$\emptyset \Vdash v_1 = 5\{v_2 := v_1 + 4\}v_1 = 5 \wedge v_2 = 9 \qquad (2)$$

by (D1). Then, applying (D2) on (1) and (2), we have

$$\emptyset \Vdash 5 = 5\{v_1 := 5; v_2 := v_1 + 4\}v_1 = 5 \wedge v_2 = 9.$$

In the future, we shall omit all occurences of $\emptyset$ on the left hand side.

Program semantics and validity of Hoare's triples

Let us fix the standard structure $\mathfrak{Z}$ for $\mathcal{L}_A$ with universe $|\mathfrak{Z}| = \mathbb{Z}$, the set of integers.

Definition. Let $\varepsilon, \varepsilon'$ be variable instantiations in $\mathfrak{Z}$ (i.e. functions that maps the set of variables $\{v_i : i \in \mathbb{N}\}$ to the set of integers) and P be a program statement. The notion "ε' is the state after executing P starting from ε", denoted by $\{P\} : \varepsilon \rightsquigarrow \varepsilon'$, recursively.

- $\{v_i := t\} : \varepsilon \rightsquigarrow \varepsilon_{v_i}^{\bar{\varepsilon}(t)}$
- $\{(P;Q)\} : \varepsilon \rightsquigarrow \varepsilon'$ if there exists ε_0 such that $\{P\} : \varepsilon \rightsquigarrow \varepsilon_0$ and $\{Q\} : \varepsilon_0 \rightsquigarrow \varepsilon'$
- $\{\mathsf{while}\ \beta\ \mathsf{do}\ P\} : \varepsilon \rightsquigarrow \varepsilon'$ if there exists a finite sequence of instantiations $\varepsilon_0, \varepsilon_1, ..., \varepsilon_n$ such that

$$\begin{aligned} \models_{\mathfrak{Z}} \beta[\varepsilon] \quad & \{P\} : \varepsilon \rightsquigarrow \varepsilon_0 \\ \models_{\mathfrak{Z}} \beta[\varepsilon_0] \quad & \{P\} : \varepsilon_0 \rightsquigarrow \varepsilon_1 \\ & ... \\ \models_{\mathfrak{Z}} \beta[\varepsilon_n] \quad & \{P\} : \varepsilon_n \rightsquigarrow \varepsilon' \\ \not\models_{\mathfrak{Z}} \beta[\varepsilon'] & \end{aligned}$$

  (Note that if the loops does not terminates, for any ε', $\{P\} : \varepsilon_n \not\rightsquigarrow \varepsilon'$ and we can use this characterization to define the concept of non-termination and then, termination of program statements.)

- $\{\mathsf{if}\ \beta\ \mathsf{then}\ P\ \mathsf{else}\ Q\} : \varepsilon \rightsquigarrow \varepsilon'$ if either

$$\models_{\mathfrak{Z}} \beta[\varepsilon] \quad \text{and} \quad \{P\} : \varepsilon \rightsquigarrow \varepsilon'$$

  or

$$\not\models_{\mathfrak{Z}} \beta[\varepsilon] \quad \text{and} \quad \{Q\} : \varepsilon \rightsquigarrow \varepsilon'.$$

Definition. The Hoare's triple $\alpha\{P\}\beta$ is valid if and only if for every ε and ε': if $\models_{\mathfrak{Z}} \alpha[\varepsilon]$ and $\{P\} : \varepsilon \rightsquigarrow \varepsilon'$ then $\models_{\mathfrak{Z}} \beta[\varepsilon']$. We denote by $\models \alpha\{P\}\beta$.

Intuitively, it says that $\alpha\{P\}\beta$ is valid if whenever the precondition α holds before P is executed, β holds after P executes and terminates.

Having presented the Hoare's formal system, we make two remarks.

First, Hoare's system is proved to be partially sound i.e. if $\emptyset \Vdash \alpha\{P\}\beta$ and P terminates then $\models \alpha\{P\}\beta$. That is the reason for our first assumption.

Second, this deductive system does not introduce an algorithm to find the "proof" (i.e. to construct the derivation or how to apply the rules). In practice, we usually use a variant of Dijkstra's *weakest precondition* method (cf. (Dijkstra, 1975)), namely the *strongest postcondition* one.

In the framework of the described formal system, each specification can be written by two first order formulas α and β. Proving correctness becomes a validity checking problem $\models \alpha\{P\}\beta$ and thus, reduces to $\emptyset \Vdash \alpha\{P\}\beta$ if termination is assumed. The basic reasoning system using strongest postcondition method is already implemented in Hip/Sleek and we only need to handle the cases of statements concerning array.

3.1.2 Relations and arrays

Each array in our program is converted to the equivalent SMT array. For each relation, we add a corresponding *uninterpreted function symbol* (into the language) together with the *axiomatization* obtained from the definition of the relation (into the collection of axioms Λ).

Example. The relation for sortedness is defined as

$$sorted(a, i, j) \leftrightarrow \forall k(i \leq k \leq j - 1 \rightarrow a[k] \leq a[k + 1])$$

meaning: $a[i..j]$ is sorted if and only if $a[k] \leq a[k + 1]$ for every indices k between i and $j - 1$. Clearly, if the condition is satisfied, we must have

$$a[i] \leq a[i + 1] \leq ... \leq a[j - 1] \leq a[j]$$

which is what we mean by sortedness. So we set

$$\mathcal{L} := \mathcal{L} \cup \{sorted\}$$

and

$$\Lambda := \Lambda \cup \{\forall a \forall i \forall j(sorted(a, i, j) \leftrightarrow \forall k(i \leq k \leq j - 1 \rightarrow a[k] \leq a[k + 1]))\}.$$

In this example, the symbol *sorted* serves as an abbreviation for the formula $\forall k(i \leq k \leq j-1 \rightarrow a[k] \leq a[k+1])$ which is a formula in our original language. Later, we will see that the definition can be recursive; in which case, the new symbol is not purely an abbreviation.

Adding additional symbol and axiomatization allows our underlying solver to reason with them (symbolicly). We shall see more details in the examples.

3.1.3 First order implication checking

To apply Hoare's rule (D2), we need to perform first order reasoning.

For each entailment (i.e. $\Lambda \cup \Gamma \vdash \alpha$), we convert it to a satisfiability modulo theory problem: we first ask whether $\Lambda \cup \Gamma \cup \{\neg\alpha\}$ is unsatisfiable. If the result is negative (i.e. `unsat`), we know that $\Lambda \cup \Gamma \vdash \alpha$. If the result is affirmative, we know that $\Lambda \cup \Gamma \nvdash \alpha$. If we cannot decide the satisfiability of $\Lambda \cup \Gamma \cup \{\neg\alpha\}$, we will take it that it is satisfiable and the entailment is wrong.

The correctness of our method is guaranteed by soundness and completeness theorem for first order logic and of course, the faithfulness of the SMT solver we use, Z3. We state the theorem (without proof) here.

Theorem (Soundness and completeness of first order logic)**.** $\Gamma \models \alpha$ *if and only if* $\Gamma \vdash \alpha$.

The standard proof is given in (Enderton, 2001) or Gödel original paper's (Gödel, 1930).

As we remark in the last chapter, the output `unsat` from the solver really means that Γ is unsatisfiable. So the theorem guarantee the existence of a proof tree.

3.1.4 Unimplemented features

Because Z3 does not provide option to generate proofs for unsatisfiability, proof generation is not implemented. Also, we worked only with array of integers. (That is to say array of objects are not currently supported.)

Example	Source input file
Sum of all elements of array	`arr_sum.ss`
Insertion sort	`arr_insertsort.ss`
Selection sort	`arr_selectionsort.ss`
Quick sort	`arr_qsort.ss`
Set data structure	`arr_setops.ss`

Table 3.2: Corresponding source file for the examples

3.2 Examples

Previous section introduces briefly our implementation. An implementation journal is given in appendix A. In this section, five examples are given. These examples show the ability of our system as well as our weakness which will be identified in chapter 4. We shall work out the first example in detail to illustrate our algorithm. The others are included for reference.

3.2.1 Sum of all elements of array

In this section, we write a simple example: to verify the function that compute the sum of a portion of an array. We begin by defining what do we mean by the sum in figure 3.1. This figure shows the way we define new relations in Hip/Sleek.

```
// right recursive definition of the sum
relation sumarray(int[] a, int i, int j, int s)
    == (i > j & s = 0 |
    i <= j & sumarray(a,i+1,j,s - a[i])).
```

Figure 3.1: Defining the relation to state what we meant by the sum of an array.

Now we implement the function. Typically, taking the sum of an array is usually performed using a `for` loop but it is not available in Hip/Sleek language. Hence, we use a simple recursive function to do that. With the defining relation for the sum, we can then specify the pre-condition

and the post-condition for the function. In this case, there is no condition, we simply put **true** there. For the post-condition, we requires that the result that the function returns is the sum

$$\sum_{k=i}^{j} a[k]$$

which is written as $sumarray(a, i, j, res)$. In our language, the special symbol (or variable) **res** is allocated for the output of a function.

```
int compute_sum_array(int[] a, int i, int j)
    requires true
    ensures sumarray(a, i, j, res);
{
    if (i > j)
        return 0;
    else
        return a[i] + compute_sum_array(a, i+1, j);
}
```

Figure 3.2: Computing the sum of elements of an array between two indices

Having completed the function with its specification, we are ready to invoke Hip/Sleek to verify this specification. To do that, invoke `hip -tp z3 arr_sum.ss` at the terminal (or command prompt for Windows OS). One can read off from the output that the specification is correct.

Now, we turn to understand how Hip/Sleek solve the problem.

First, let us construct a formal version of the function by eliminating the return statement (and also replace the function name by a simpler name, P):

$$\{P(a, i, j, res)\} \quad : \quad \{\mathsf{if}\ (i > j)\ \mathsf{then}\ res := 0\ \mathsf{else}\ (P(a, i+1, j, s); res := a[i] + s)\}.$$

(Note that there is an extension of the formal system described earlier here to account for functions and arrays. Also, we choose not to have return type but use *referenced* parameters to return value.)

As we mentioned in the last section, proving correctness of the function **compute_sum_array** is equivalent to deriving

$$\Vdash \mathbf{true}\{P(a,i,j,res)\}sumarray(a,i,j,res).$$

(Recall that the pre-condition α is **true** and the post condition is $sumarray(a,i,j,res)$. Here, $\{P(a,i,j,res)\}$ is a program with a single function call.)

Figure 3.3 shows the derivation tree of Hip/Sleek automatic reasoning. (For formatting purpose, in this figure, *sa* is used in place of *sumarray*.)

First of all, using (D4), the problem reduces to two derivations

$$\begin{cases} \Vdash i > j\{res := 0\}sumarray(a,i,j,res) & (1) \\ \Vdash (\neg i > j)\{P(a,i+1,j,s); res := a[i]+s\}sumarray(a,i,j,res) & (2) \end{cases}$$

From (D1), (1) is reduced to $\Vdash i > j\{res := 0\}i > j \wedge res = 0$ which is evident by (D0) and $\vdash i > j \wedge res = 0 \rightarrow sumarray(a,i,j,res)$.

For (2), first expand to $\Vdash (\neg i > j)\{P(a,i+1,j,s)\}(\neg i > j) \wedge sumarray(a,i+1,j,s)$ which is evident from our assumption that any recursive function is well-founded and

$$\Vdash (\neg i > j) \wedge sumarray(a,i+1,j,s)\{res := a[i]+s\}sumarray(a,i,j,res) \qquad (3)$$

Then (3) is again reduced to $\Vdash (\neg i > j) \wedge sumarray(a,i+1,j,s)\{res := a[i]+s\}(\neg i > j) \wedge sumarray(a,i+1,j,s) \wedge res = a[i]+s$ (valid by the general rule that we state earlier) and $\vdash (\neg i > j) \wedge sumarray(a,i+1,j,s) \wedge res = a[i]+s \rightarrow sumarray(a,i,j,res)$ using (D2).

So the problem is solved if we can show the two first order derivations:

$$\vdash i > j \wedge res = 0 \rightarrow sumarray(a,i,j,res)$$

and

$$\vdash (\neg i > j) \wedge sumarray(a,i+1,j,s) \wedge res = a[i]+s \rightarrow sumarray(a,i,j,res).$$

This task is passed to Z3 to solve.

Via this example, we have seen from the example the underlying mechanism of Hip. It transforms (more precisely, reduces) the derivation $\Vdash \alpha\{P\}\beta$ into "simpler" derivations and

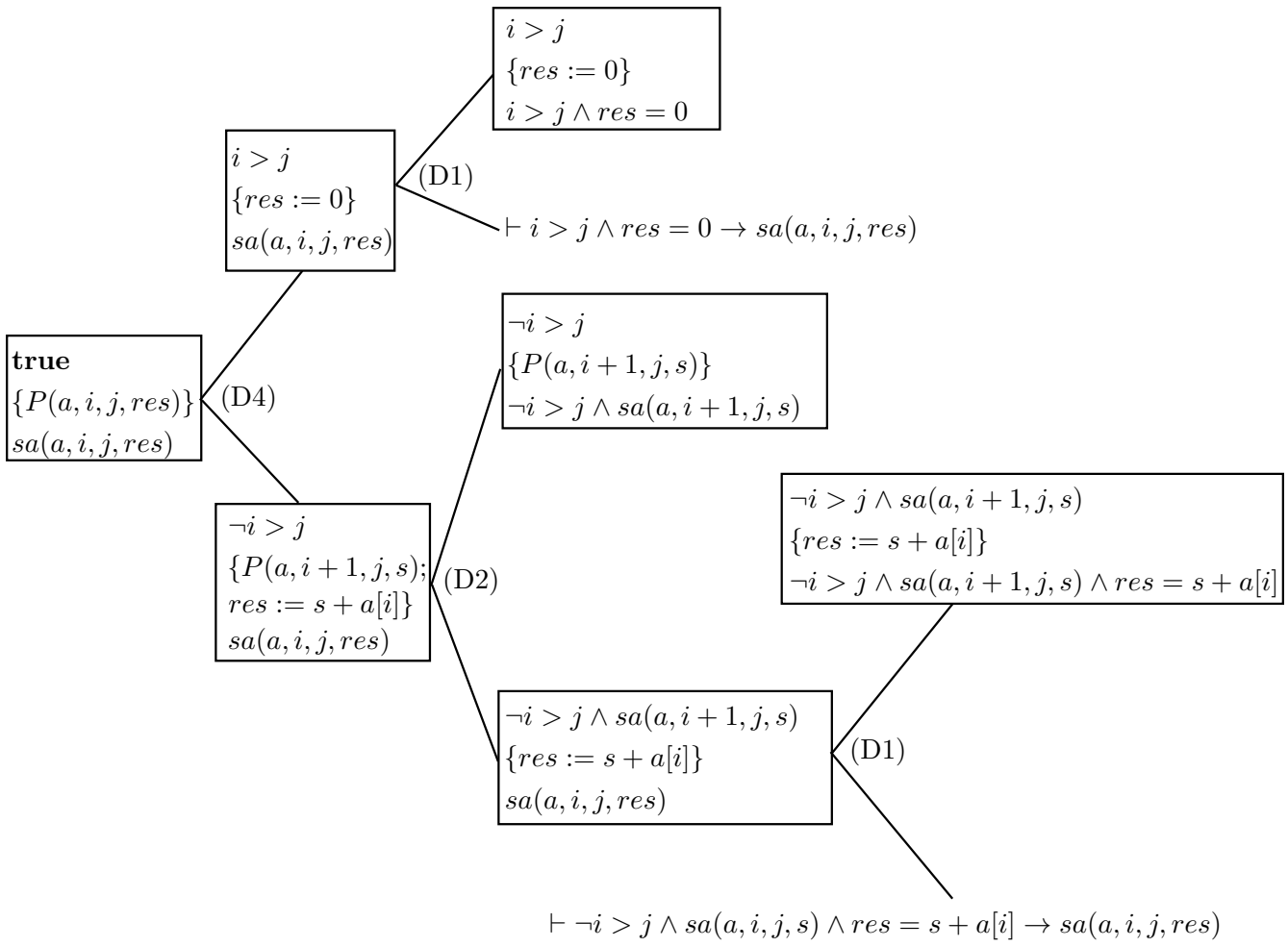

Figure 3.3: Hoare formal reasoning performed by Hip

some first order deduction problems using the rules (D0-4). Then it recurses to solve the simpler problems and requests the SMT solver to solve the logical deduction problems.

(To view the logical implication problems that are generated, supply `hip` with the additional option `--imply-calls`. The options `--smtout` and `--smtinp` is used to dump the original output of the SMT solver and the SMT input file.)

3.2.2 Insertion sort

```
// Two array are identical except between i & j
relation idexc(int[] a, int[] b, int i, int j)
            == forall(k : (i<=k & k<=j | a[k] = b[k])).


// a[i..j] is sorted
relation sorted(int[] a, int i, int j) == (i >= j |
            forall (k : (k < i | k >= j | a[k] <= a[k+1]))).
```

Figure 3.4: Relations idexc to specify range of modification and sorted to specify sortedness property of an array slice

Figure 3.4 defines the main relation *sorted* which is for us to write the specification later. Note that there are many ways to specify this relation, one of which is shown above, the other two alternatives are:

$$(i >= j | i < j \,\&\, a[i] <= a[i+1] \,\&\, sorted(a, i+1, j))$$

and

$$(i >= j | i < j \,\&\, a[j-1] <= a[j] \,\&\, sorted(a, i, j-1)).$$

And we realize that non-recursive version works best due to its *symmetry*.

The additional relation *idexc* (identical except) is to specify that the function only modify the elements of a between i and j and left the other unchanged. Due to the fact that our array model cannot capture this information automatically, we need to specify it explicitly in our specification. Typically, functions that modify the content of the array should specify the range of the array it operates on.

Insertion sort can be viewed as a left-recursive process. To sort the array $a[0..n]$, we first sort $a[0..n-1]$ and then insert the element $a[n]$ to the correct position between $a[0..n-1]$ to make the final array sorted. This idea translates to the implementation in figure 3.5.

```
// Sort a[0..n] using insertion sort algorithm
void insertion_sort(ref int[] a, int n)
    requires true ensures sorted(a',0,n);
{// Sort a[0..n-1]. Then insert a[n] between the
    if (n > 0) {// sorted a[0..n-1] to make a[0..n] sorted.
        insertion_sort(a, n-1);
        insertelm(a,n);
    }
}
```

Figure 3.5: Insertion sort

Now we implement the supplementary function **insertelm** in figure 3.6 which assumes that the array slice $a[0..n-1]$ is sorted and inserts the element $a[n]$ between $a[0..n-1]$ (and shifts the array so that to make $a[0..n]$ sorted.)

```
void insertelm(ref int[] a, int n)
    requires sorted(a,0,n-1)
    case {
        n <= 0 -> ensures a'=a;
        n > 0 -> ensures sorted(a',0,n) & idexc(a,a',0,n) &
                                    (a'[n] = a[n] | a'[n] = a[n-1]);}
{
    if (n > 0 && a[n] < a[n-1]) {
        int t = a[n];        a[n] = a[n-1];        a[n-1] = t;
        insertelm(a,n-1);
    }
}
```

Figure 3.6: Insertion an element to a sorted array

3.2.3 Selection sort

```
// s is an upper bound of a[i..j]
relation upperbnd(int[] a, int i, int j, int s) == (i > j |
            forall ( k : (k < i | k > j | a[k] <= s))).

//Upper bound preserving: if s is an upper bound of a[i..j]
// then s is also an upper bound of b[i..j]
relation upperbndprev(int[] a, int[] b, int i, int j) ==
            forall(s : (!(upperbnd(a,i,j,s)) | upperbnd(b,i,j,s))).

void selection_sort(ref int[] a, int i, int j)
    requires 0<=i & 0<=j
    ensures sorted(a',i,j) & idexc(a',a,i,j) & upperbndprev(a,a',i,j);
{
    if (i < j)
    {
        int k = array_index_of_max(a,i,j);

        // swap a[k] & a[j] so that a[j] now hold the maximum
        int temp = a[k];        a[k] = a[j];        a[j] = temp;

        // and sort the remaining part
        assert upperbnd(a',i,j-1,temp');
        assume upperbnd(a',i,j-1,temp');
        selection_sort(a, i, j-1);
    }
}
```

Figure 3.7: Relations upperbnd and upperbndprev. Recursive implementation of selection sort.

```
// Smallest index that gives maximum value in the array
int array_index_of_max(int[] a, int i, int j)
    requires i>=0 & j>=0
    case {
        i > j -> ensures res = -1;
        i <= j -> ensures i <= res & res <= j &
                                    upperbnd(a, i, j, a[res]);
    }
{
    int k = -1;
    if (i <= j)
    {
        k = array_index_of_max(a,i+1,j);
        if (k == -1 || a[i] >= a[k])
            k = i;
    }
    return k;
}
```

Figure 3.8: Find the index of the maximum element of the array

3.2.4 Quick sort

For quick sort algorithm, we introduce the relation $bnd(a, i, j, low, high)$ to indicate the bounds of the array elements

$$\forall k(i \leq k \leq j \rightarrow low \leq a[k] \leq high$$

as in figure 3.9.

Quick sort is implemented by establishing the function **arraypart** as traditionally done. This function is to partition the array section $a[i..j]$ into three parts:

- $a[i..k]$ contains elements whose values are all less than x;
- $a[n] = x$ for all $k + 1 \leq n \leq t - 1$;
- $a[t..j]$ contains elements greater than x.

Figure 3.10 shows our implementation. The indices k and t should be set appropriately after the function finishes. Two additional parameters l and h is to indicate the bounds of $a[i..j]$. The necessity of these parameters will be explained in the next chapter. Partitioning can be easily implemented by right recursion. Details on the implementation can be retrieved in the input file.

```
// a[k] = s for all k between i & j
relation alleqs(int[] a, int i, int j, int s) ==
        (i > j | forall ( k : (k < i | k > j | a[k] = s))).

// low,high are the lower and upper bound for a[i..j]
relation bnd(int[] a, int i, int j, int low, int high) ==
        (i > j | i<=j &
        forall ( k : (k < i | k > j | low <= a[k] <= high))).
```

Figure 3.9: Relation for upper and lower bound of an array

Due to the special parameters, quick sort has to be modified with the additional parameters.

```
void arraypart(ref int[] a, int i, int j, int x,
                    ref int k, ref int t, int l, int h)
    case {
        i > j -> ensures k' = i - 1 & t' = j + 1 & a' = a;
        i <= j -> requires bnd(a,i,j,l,h) ensures i - 1 <= k'
            & k' <= j & i <= t' & t' <= j + 1  & bnd(a',i,k',l,x-1)
            & alleqs(a',k'+1,t'-1,x) & bnd(a',t',j,x+1,h)
            & idexc(a',a,i,j) & bnd(a',i,j,l,h);
    }
{
    if (i <= j)
    {
        if (a[i] < x) arraypart(a, i + 1, j, x, k, t,l,h);
        else if (a[i] > x) {
            int temp = a[i];     a[i] = a[j];    a[j] = temp;
            arraypart(a, i, j - 1, x, k, t,l,h);
        } else { // a[i] == x
            arraypart(a, i + 1, j, x, k, t,l,h);
            a[i] = a[k];
            a[k] = x;
            k = k - 1;
        }
    }
    else { k = i - 1;     t = j + 1; }
}
```

Figure 3.10: Array partitioning

```
/**
 Sort a[i..j] with the knowledge of the bounds
 of the arrays: l <= a[k] <= h for all i <= k <= j
 **/
void qsort(ref int[] a, int i, int j, int l, int h)
    case {
        i >= j -> ensures a=a';
        i < j -> requires bnd(a,i,j,l,h) ensures
            bnd(a',i,j,l,h) & sorted(a',i,j)
                & idexc(a',a,i,j);
    }
{
    if (i < j)
    {
        int k, t;
        int x = a[i];
        arraypart(a,i,j,x,k,t,l,h);
        qsort(a,i,k,l,x-1);
        qsort(a,t,j,x+1,h);
    }
}
```

Figure 3.11: Quick sort

3.2.5 Utilizing array to implement set data structure

To illustrate the fact that array can be used to implement abstract data structure, this example is set out to implement the abstract set of integers. We use a pair (S, n) of an array S and an natural number n (cardinality) to encode the set whose elements are from $S[0..n-1]$. A valid

pair is with $n \geq 0$ and $S[0..n-1]$ contains no duplicate. The relation *isset* selects the valid pairs. Other relations in 3.12 describes relations between sets or set and its member.

```
// s is a set with n elements
relation isset(int[] s, int n) == 0 <= n &
     forall(i,j: (i < 0 | i >= n | j < 0 | j >= n | i = j |
           i != j & s[i] != s[j])).

relation idbwn(int[] s, int[] t, int i, int j) ==
     forall(k : (k < i | k > j | s[k] = t[k])).

// s is a subset of t
relation issubset(int[] s, int n, int[] t, int nt) ==
     forall(i : (i < 0 | i >= n | member(t,0,nt-1,s[i]))).

// u is the union of a & b
relation isunion(int[] u, int n, int[] a, int na, int[] b, int nb) ==
     issubset(a,na,u,n) & issubset(b,nb,u,n).

// x in s[i..j]
relation member(int[] s, int i, int j, int x) ==
        exists(k: i <= k & k <= j & s[k] = x).
```

Figure 3.12: Standard set relations

The first function is to check whether an integer belongs to a set. Membership checking function *contains* is implemented in figure 3.13.

```
bool contain(int[] S, int i, int j, int x)
    requires true
    ensures (res | !(member(S,i,j,x))) & (!res | member(S,i,j,x));
{
    if (i <= j)
    {
        if (S[i] == x)
            return true;
        else
            return contain(S,i+1,j,x);
    }
    else
        return false;
}
```

Figure 3.13: Membership checking

Next, we provide the function *insert* to insert an element into a set (figure 3.14) which is then used to compute the union (figure 3.15).

```
/** Modify the pair (S,n) to get the set S U {x} **/
void insert(ref int[] S, ref int n, int x)
    requires isset(S,n)
    ensures isset(S',n') & member(S',0,n'-1,x)
            & idbwn(S,S',0,n-1) & issubset(S,n,S',n');
{
    if (!contain(S,0,n-1,x))
    {
        S[n] = x;
        n = n + 1;
        assume (!(idbwn(S,S',0,n-1)) | n' < n | issubset(S,n,S',n'));
    }
}
```

Figure 3.14: Insert an element to a set

```
/** Modify the pair (S,n) to get the set S U T **/
void unionsets(ref int[] S, ref int n, int[] T, int nT)
    requires isset(S,n) & isset(T,nT)
    ensures isset(S',n') & isunion(S',n',S,n,T,nT);
{
    if (nT > 0)
    {
        unionsets(S,n,T,nT-1);
        insert(S,n,T[nT-1]);
    }
}
```

Figure 3.15: Set union

Chapter 4

Evaluation

4.1 Running time of the examples

Example	Run time (Unix)	Run time (Windows)
Sum of all elements of array		1.843
Insertion sort		4.062
Selection sort		4.875
Quick sort		9.609
Set data structure		4.89

Table 4.1: Summary of running time of examples. Running time measures are in seconds.

4.2 Lessons from the examples

The ability to support user-defined relations (even recursively defined ones) greatly improves the expressiveness of Hip/Sleek language. We can see that a lot of specification can be written if we do have relations. In fact, it is very difficult to express *the sum of an array* if we do not have recursive relations. (Even so, it is possible to do it but a typical user does not know how.)

Being able to express complicated ideas, the current system is still primitive in verification. And in this section, we unveil its real verifying power.

4.2.1 Non-trivial applications of induction

In figure 3.2, we present a *right recursive* way of computing the sum of an array. That is to say, the sum is computed with the following paranthesized

$$\sum_{k=i}^{j} a[k] = a[i] + \underbrace{(a[i+1] + (a[i+2] + (... + a[j])))}_{\sum_{k=i+1}^{j} a[k]}$$

Since summation is a symmetric process (i.e. + is commutative), we can compute the sum in a different order such as

$$\sum_{k=i}^{j} a[k] = \underbrace{(((a[i] + a[i+1]) + a[i+2]) + ... + a[j-1])}_{\sum_{k=i}^{j-1} a[k]} + a[j].$$

This method of computation is called *left recursive*. This presents to us a different function to find the sum in figure 4.1.

```
int compure_sum_left(int[] a, int i, int j)
    requires true
    ensures sumarray(a, i, j, res);
{
    int r;
    if (i > j)
        r = 0;
    else
        r = compure_sum_left(a, i, j-1) + a[j];
    return r;
}
```

Figure 4.1: A different way of computing the sum.

Unforturnately, Hip/Sleek cannot prove the correctness in this case. Run the example with directives for more information, we realize that it fails to establish

$$\left\{ \begin{array}{c} i \leq j, sa(a,i,j-1,v_1), res = v_1 + a[j], \\ \forall a \forall i \forall j \forall s (sa(a,i,j,s) \leftrightarrow i > j \wedge s = 0 \vee i \leq j \wedge sa(a,i+1,j,s-a[j])) \end{array} \right\} \vdash sa(a,i,j,res)$$

where the symbol *sa* is used in place of *sumarray*.

With further analysis, we realize that the implication requires a proof by a non-trivial application of induction, namely induction on the difference $j - i$, which our system is currently incapable of. (In fact, one can prove that such derivation is impossible without the induction schema. The proof of this fact is included in appendix C.)

Another occasion in which a proof by induction is necessary is the case of quick sort algorithm. We have seen the appearance of the annoying additional parameters for array bounds. Should we write our functions and specifications differently as in figure 4.2, the verification fails because of the difficulty of the preservation of bounds.

```
// s is strict upper/lower bound of a[i..j]
relation strupperbnd(int[] a, int i, int j, int s) ==
        (i > j | forall ( k : (k < i | k > j | a[k] < s))).
relation strlowerbnd(int[] a, int i, int j, int s) ==
        (i > j | forall ( k : (k < i | k > j | a[k] > s))).

// strict upper and lower bound preserving
relation upperbndprev(int[] a, int[] b) ==
    forall(i,j,s : (!(strupperbnd(a,i,j,s)) | strupperbnd(b,i,j,s))).
relation lowerbndprev(int[] a, int[] b) ==
    forall(i,j,s : (!(strlowerbnd(a,i,j,s)) | strlowerbnd(b,i,j,s))).

// new specification for arraypart
void arraypart(ref int[] a, int i, int j, int x, ref int k, ref int t)
    case {
    i > j -> ensures k' = i - 1 & t' = j + 1 & a' = a;
    i <= j -> ensures i - 1 <= k'  & k' <= j & i <= t' & t' <= j + 1
            & strupperbnd(a', i, k', x) & alleqs(a', k'+1, t'-1, x)
            & strlowerbnd(a', t', j, x) & idexc(a', a, i, j) ;
    }

// new specification for quick sort
void qsort(ref int[] a, int i, int j)
    requires true
    ensures sorted(a',i,j) & idexc(a',a,i,j)
        & upperbndprev(a,a') & lowerbndprev(a,a');
```

Figure 4.2: Expecting array partition and quick sort

	Hip/Sleek	Frama-C
Recursive definition	Y	N
Memory safety	N	Y
Loop termination	N	Y
SMT solvers/ATP	Z3	Coq, CVC, Z3, Simplify, ...

Table 4.2: Functionalities comparison between Hip/Sleek and Frama-C

4.2.2 Intermediate consequence discovery

Let us recall the pair **assert/assume** in the selection sort algorithm. The reason for the presence of this pair of statements is because the formula [....] is an essential intermediate result that we need in order to prove the correctness.

4.2.3 Insufficiency of the specification language and array model

We notice that in *array-modifying* examples, we usually have to include the condition *idexc* to state that the memory outside the given range of the array is not touch. These instances show the insufficiency of our model. This can be dealt with if we make use of separation logic. In such event, we can also verify the memory safety of our programs. An incomplete outline of this approach is already given in (Cuong, 2010).

4.3 A comparison with Frama-C on verification of programs with arrays

Frama-C is an efficient verification platform for the ANSI C programming language. Its specification language is based on the standardized **A**NSI/ISO **C** **S**pecification **L**anguage (ACSL) described in (Baudin, Filliâtre, Marché, Monate, Moy, & Prevosto, 2008). To do formal verification with Frama-C, one needs the plug-in Jessie as well as Why . The former generate the programs and specification in Why 's input language. Then Frama-C calls Why to do the verification. (So in fact, it is the Why that is doing the real work. Why developers actually ship

it with Caduceus, their own version of C verifier. Interested reader can confer (Filliâtre, 2009), (Filliâtre, 2003) and (Filliâtre & Marché,) for a thorough discussion of this tool published by its author.)

The input language of Why does not support array directly. How does Jessie generates Why conditions? It makes use of the pointer model: Jessie developed its own *theory* of pointer (with the standard model being C's semantics). Being close to the real model of execution, it reaps the direct advantage of simple translation (because array in C programs can be translated to its pointer equivalence) and the enhanced capability due to the multiple prover approach of Why . Furthermore, it can make use of this framework to verify the memory safety of the program, a feature which we currently do not support.

An interesting feature which Frama-C supports is proving termination. However, it can do that at the expense of the user to find out *loop variant*, a non-negative integer function (on the loop inputs) that decreases after each loop execution. (Programmers, even experienced ones, might find that detecting loop variant is not a trivial task.)

Frama-C is a successful tool and is used in the industry. We currently beat it on the aspect of recursive relations. (Clearly, simple computation such as taking sum of the array cannot be verified in Frama-C if we cannot write the specification.) Nevertheless, we believe that Frama-C will advance soon and thus we have to start making great leap forward should we want to win this game.

The comparison with Frama-C ends the main part of my report. The next chapter summarizes what we did and offers our vision on further development.

Chapter 5

Conclusion

5.1 Contributions

After this one year project, we achieved an (incomplete) solution to verification of programs with arrays. A collection of comprehensive examples that illustrates the capability of our system were given. Via those examples, we were able to analyse the strenght and weakness of our system in comparison to existing solutions like Frama-C.

5.2 Future works

At this point, Hip/Sleek array capability is still primitive. There are a number of directions which we could look forward to. An array model obtained by separation logic is promising as it enhances our reasoning power on memory safety issue which is an important matter. Furthermore, incorporating existing work on proof generation with Z3 into our system is a necessary step toward a full formal verifier. Beyond that, smart induction and invariant detection serves as challenging research topics.

References

Barrett, C., Stump, A., & Tinelli, C. (2010a). The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

Barrett, C., Stump, A., & Tinelli, C. (2010b). The SMT-LIB Standard: Version 2.0. In A. Gupta, & D. Kroening (Eds.), *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010.

Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., & Prevosto, V. (2008). *Acsl: Ansi/iso c specification language.* `http://frama-c.cea.fr/acsl.html`.

Böhme, S. (2009). Proof reconstruction for Z3 in Isabelle/HOL. *Workshop on Satisfiability Modulo Theories (SMT 2009)*, 2009.

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, *58*(2), April, 1936, 345–363.

Cuong, N. V. (2010). An Expressive Separation Logic Specification.

Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, *18*, August, 1975, 453–457.

Duffy, D. A. (1991). *Principles of automated theorem proving.* New York, NY, USA: John Wiley & Sons, Inc.

Enderton, H. B. (2001). *A mathematical introduction to logic.* San Diego: Academic Press.

Filliâtre, J.-C. (2003). *Why: a multi-language multi-prover verification tool* (Research Report 1366). LRI, Université Paris Sud. `http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz`.

Filliâtre, J.-C. (2009). Invited tutorial: Why — an intermediate language for deductive program verification. In H. Saïdi, & N. Shankar (Eds.), *Automated Formal Methods (AFM09)*, Grenoble, France, 2009.

Filliâtre, J.-C., & Marché, C. The Why/Krakatoa/Caduceus platform for deductive program verification (pp. 173–177).

Gallier, J. H. (1985). *Logic for computer science: foundations of automatic theorem proving.* New York, NY, USA: Harper & Row Publishers, Inc.

George Boolos, R. J. (1985). *Computability and logic.* Cambridge: Cambridge University Press.

Gödel, K. (1930). Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte f. Math.*, *37*, 1930, 349–360.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, *12*, October, 1969, 576–580.

Mendelson, E. (1997). *Introduction to mathematical logic.* New York: Chapman & Hall.

Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02 (pp. 55–74), Washington, DC, USA, 2002: IEEE Computer Society.

Robinson, J. (2001). *Handbook of automated reasoning: Volume 1.* Cambridge, MA, USA: MIT Press.

Russell, S. J., Norvig, P., Candy, J. F., Malik, J. M., & Edwards, D. D. (1996). *Artificial intelligence: a modern approach.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, *2*(42), 1936, 230–265.

Appendix A

Implementation journal

To assist future developers, I include the detail of my implementation in this appendix.

A.1 Sleek

1. Support arrays and relation in parsers

slexer.mll & [ilexer.mll]: add token REL for relation

sparser.mly : add new fields to the changed types & add rule for relation and arrays

```
%type <Iast.rel_decl> rel_decl


non_empty_command
  : rel_decl {
      RelDef $1
    }
;


rel_decl
  : rel_header EQEQ rel_body DOT{
{ $1 with rel_formula = $3;}
  }
```

```
  | rel_header EQ error {
  report_error (get_pos 2) ("use == to define a relation")
}
;

typed_id_list_opt
: { [] }
| typ IDENTIFIER {
[($1,$2)]
}
| typ IDENTIFIER COMMA typed_id_list_opt {
($1,$2) :: $4
}
;

rel_header
  : REL IDENTIFIER OPAREN typed_id_list_opt CPAREN {
  { rel_name = $2;
rel_typed_vars = $4;
rel_formula = P.mkTrue (get_pos 1); }
  }
;

rel_body
  : pure_constr { $1 }
;
```

sleekcommons.ml : add new case of in command type.

```
type command =
```

```
| ...
| RelDef of I.rel_decl (* An Hoa *)
| ...
```

2. Change the input representation and core representation.

iast.ml : Add field

```
{ mutable prog_rel_decls : rel_decl list }
```

to type program.

Create type for relation declaration:

```
rel_decl = { rel_name : ident;
 rel_typed_vars : (typ * ident) list;
 rel_formula : P.formula ; }
```

ipure.ml : Add case [ArrayAt] to type [exp] to handle array access Add case [RelForm] to type [bformula] to handle array access Handle new cases in functions.

cast.ml : Similar changes as in [iast.ml]

cpure.ml : Add array type in the type [typ] and fix related functions (for instance, [get_exp_type\verb])
Add case [ArrayAt] to type [exp] to handle array access Add case [RelForm] to type [bformula] to handle array access Handle new cases in functions.

3. Process relation definitions: Translate input representation to core representation.

astsimp.ml : `trans_pure_exp`

4. Generate SMT formula for relation and array in entailment checking.

smtsolver.ml

5. Eradicate remaining warnings generated from unmatched rules in other files: [mcpure.ml,coq.ml,cvclite.ml

6. Pretty printing for relation and array.

cprinter.ml

iprinter.ml

7. Extend type checking system.

astsimp.ml : make changes to [collect_type_info_pure], [collect_type_info...] to account for array and relations

8. Handle relation declaration in sleek executable.

smtsolver.ml : add an internal type [relation_definition] for relations, a global field [rel_defs] to store the relations and a function [add_rel_def] to add new relation into this module.

```
type relation_definition =
| RelDefn of (ident * CP.spec_var list * CP.formula)

let rel_defs = ref ([] : relation_definition list)

let add_rel_def rdef =
rel_defs := !rel_defs @ [rdef]
```

astsimp.ml : handle translation of input relation [Iast.rel_decl] to core representation in [Cast.rel_decl] by adding function [trans_rel].

sleekengine.ml : add function [process_rel_def] to process relation declarations - 3 things to handle: add the relation declaration to the input storage, translate to core representation and store, and request the instance of smtsolver module to store the relation

```
let process_rel_def rdef =
  if check_data_pred_name rdef.I.rel_name then
let tmp = iprog.I.prog_rel_decls in
  try
iprog.I.prog_rel_decls <- ( rdef :: iprog.I.prog_rel_decls);
let crdef = AS.trans_rel iprog rdef in cprog.C.prog_rel_decls <- (crdef
```

```
Smtsolver.add_rel_def (Smtsolver.RelDefn (crdef.C.rel_name,crdef.C.rel_v
  with
| _ ->  dummy_exception() ; iprog.I.prog_rel_decls <- tmp
  else
print_string (rdef.I.rel_name ^ " is already defined.\n")
```

sleekengine.ml : change [check_data_pred_name] to take care of relation as well

sleek.ml : dispatch the parsed relation declaration to [sleekengine] for processing.

9. Test examples.

A.2 Hip

1. Parser

2. Input & core representation

iast.ml Add new type [exp_arrayat] and new case for type [exp]

```
exp_arrayat = { exp_arrayat_array_name : ident;
        exp_arrayat_index : exp;
exp_arrayat_pos : loc; }


exp =
| ArrayAt of exp_arrayat
| ...
```

I choose not to modify core AST because it will then take huge effort to solve the cascade changes in verification system. Instead, I choose to allocate the [exp::SCall] with

cast.ml `exp_scall_method_name="update_array___"`

and

`exp_scall_method_name="get_element_array___"`

for array updation and access. This minimizes the amount of changes as latter on, there is no need to change the existing context transformation mechanism and verification engine in [typechecker.ml].

ipure.ml,cpure.ml No change: HIP only works on programs, not formulas.

3. AST simplification

astsimp.ml [prim_str] : append the primitive functions with new primitives for array access and update

```
int array_get_elm_at___(int[] a, int i) requires true ensures res = a[i]
int[] update___(int[] a, int i, int v) requires true ensures update_arra
```

[trans_exp] : Handle the case of the array access and array value assignment instructions in function [trans_exp]. For array access, replace it by the predefined [cast] SCall instruction "get_element_array___" and for array assignment (under assignment instruction i.e. case I.Assign, subcase I.AssignOp), replace by a [cast] SCall instruction "update_array___". Then to translate an array assignment, we first make it

```
(IAST)     a[i] = v;
```

in to the equivalent function call

```
(IAST)     a = update___(a, i, v)
```

And call the existing trans_exp to make it into an SCall:

```
(CAST)     exp::Assign{a,exp::SCall{update___,[a,i,v]}}
```

Similarly, array access instruction a[i] will be made into

```
(IAST)     a =  array_get_elm_at___(a, i, v)
```

and then to

```
(CAST)     exp::Assign{a,[exp::Scall]{array_get_elm_at___[a, i, v]}}
```

Thus, no work needs to be done except the transformation from the original IAST [exp] to the equivalent IAST function call.

[trans_prog] : We also translate relations in case of the function

[trans_exp] : translate input representation of instruction [Iast.exp] to core representation [Cast.exp]

[trans_type] : translate array type

4. Handle the predefined relation in SMT solver.

smtsolver.ml : In case [CP.RelForm] of [smt_of_b_formula], we need to handle the relation "update_array" separately: There is no definition from the user for this relation! It is for our internal convenient only.

```
let rec smt_of_b_formula b qvars =
  | ...
  | CP.RelForm (r, args, l) ->
let smt_args = List.map smt_of_exp args in
if (r = "update_array") then
let orig_array = List.nth smt_args 0 in
let index = List.nth smt_args 1 in
let value = List.nth smt_args 2 in
let new_array = List.nth smt_args 3 in
"(= " ^ new_array ^ " (store " ^ orig_array ^ " " ^ index ^ " " ^ value
else
          "(" ^ r ^ " " ^ (String.concat " " smt_args) ^ ")"
```

5. Pretty printing for array access expressions.

6. Add dummy empty array to serve as a default value for array variable. Without this, one cannot declare arrays inside functions.

cast.ml : add type [exp_emparray] and new case [EmptyArray] for type [exp]

```
exp_emparray = { exp_emparray_type : P.typ;
exp_emparray_pos : loc }
exp =
  | ...
  | EmptyArray of exp_emparray
  | ...
```

astsimp.ml : Change [default_value] to return an EmptyArray for the type array.

```
and default_value (t : CP.typ) pos : C.exp = match t with
  | ...
  | CP.Array t -> C.EmptyArray { C.exp_emparray_type = t; C.exp_emparray
```

typechecker.ml : handle the case [EmptyArray] in function [checkexp_a]::[checkexp1] - similar to case [Null] - just ignore! We do not care whether the array is empty or not.

A.3 Bugs

cpure.ml : function [afv] forgets to collect the array variable in array access expression as free variable

smtsolver.ml : New version of Z3 does not print "success" and only print "sat/unsat/unknown" results for each (check-sat) command. After that it will print the warning/errors. Sometimes errors are printed before the sat/unsat. Hence, it is safer to read the entire output and check for instances of "sat" or "unsat" or "unknown" instead of reading the first/last line.

smtsolver.ml : Bug detected in used relations filtering - on encountering a relation, we need to collect all relations that it depends on as well.

cpure.ml : Error in formula substitution and application for case [ArrayAt] - need to substitute the array variable whenever applicable.

A.4 Enhancements

smtsolver.ml : Enhancement - In many cases, putting unused relation definitions and axiomatizations make it difficult for the SMT solver to deduce the satisfiability. So we perform a filtering of used relations

cpure.ml : cache the dependent relations to prevent redundant computations

scriptarguments.ml : add new hip/sleek switches "–smtinp", "–imply-calls" and "–smtout" argument to choose whether to dump the generated SMT input, output or the implication formula.

Appendix B

Hip/Sleek grammar

Appendix C

Selected proofs