Deriving Safety Pre-Condition for Array Bounds Check in Imperative Programs

Supervised by:

A/P Wei-Ngan Chin

Student:

Corneliu Popeea

School of Computing

National University of Singapore

2001

Acknowledgement

I would like to thank my supervisor, professor Chin Wei-Ngan, for the time and knowledge that he shared with me. Working with him, was a new experience for me: a research project is, in many ways, different than course-work. It takes as well pleasure to work and a special programming discipline, to adapt to the research group formed by professor Chin Wei-Ngan, professor Khoo Siau Cheng and Dana Xu.

Throughout my faculty years, I received precious advices and support from professor Nicolae Tapus and professor Irina Athanasiu. In a special moment, I needed courage and motivation to undertake an important project as this one, and they guided me to the right choice.

I would like to thank The National University of Singapore for sponsoring my research here and especially professor Chin Wei-Ngan and professor Nicolae Tapus for making all this possible.

The romanian atmosphere from Singapore, both warm and inspiring, helped me concentrate on my work, and I must thank for this Mr. Razvan Voicu, his family and my coleagues: Radu Negoescu, Mihail Asavoae and Andrei Frangeti.

Table of Contents

Chapter 1

Introduction

Array bound check optimization has been extensively investigated over the last three decades, with renewed interests recently [12,5,7,8,9]. While successful bound check elimination can bring about measurable gains in performance, the importance of bound check optimization goes beyond these direct gains. In safety-oriented languages, such as Ada or Java, all bound violation must be faithfully reported through precise exception handling mechanism. With this, the presence of bound checks could potentially interfere with other program analyses. For example, data-flow based analysis must take into account potential loss in control flow should array bound violation occurs.

Let us first review the key problem of identifying bound checks for elimination. In general, each check can be classified as either:

- unsafe;
- totally redundant;
- partially redundant.

A check is classified as *unsafe* if either a bound violation is expected to occur, or its safety condition is unknown. As a result, we cannot eliminate such a check. A check is classified as *totally redundant* if it can be proven that no bound violation will occur. Lastly, a check is said to be *partially redundant* if we can identify a pre-condition that could ensure that the check becomes redundant.

To illustrate these three types of checks, consider the following excerpts of code:

```
int access (int A[]){
    ...
    if (i>=3)
        x=A[i];        //L1,H1
    j=-5;
    y=A[j];            //L2,H2
    ...
```

}

Arrays are assumed to start at index 0; each array access is associated with two bound checks that need both to be satisfied in order for this access to be *safe*. The low bound check requirement is that index is positive: Lo: i>=0. The high bound check requirement is: Hi: i<length(A).
For the first array access, under the context (i >= 3) we can prove that the low bound check (i > 0) is *totally redundant*. The high bound check cannot be inferred to be totally redundant from this context. However, it can be made redundant under appropriate pre-conditions - for example: i < length(A) – this check is *partially redundant*.
The second array access uses an index outside of range 0 to length(A)-1; under the context (j = -5) we can prove that the low bound check (j > 0) can not be satisfied, so this check is *unsafe*.
Totally and partially redundant checks are traditionally identified by two separate techniques [1,9]. Forward data flow analysis, which determines *available expressions,* has been primarily used to identify totally redundant checks. An expression (or check) e is said to be *available* at program point p if some expression in an equivalence class of e has been computed on every path from entry to p, and the constituent of e has not been redefined in the control flow graph (CFG). Using this information, the computation of an expression (or check) e at point p is redundant if e is available at that point.
Partially redundant checks are more difficult to handle. Traditionally, a backward dataflow analysis [8] is used to determine the *anticipatability* of expressions. An expression (or check) e is anticipatable at program point p if e is computed on every path from p to the exit of the CFG before any of its constituents are redefined. By hoisting an anticipatable expression to its *safe earliest* program point, selected checks can be made totally redundant. Historically hoisting of anticipatable check is deemed as crucial for eliminating checks from loop-based programs. Unfortunately, hoisting of checks causes bound errors to be flagged at an earlier program point, creating problems for precise exception handling.
To overcome this problem, we can use program specialization (rather than check hoisting) to selectively enforce contexts that are strong enough for eliminating partially

redundant check. In conjunction with program specialization, there is a set of advanced techniques to help determine pre-conditions that can allow partially redundant check to become totally redundant. This method is proposed in “Deriving pre-conditions for array bound check elimination” [0].

1.1. Basic Idea of ABCE Algorithm

The algorithm for eliminating array bound checks is based on the following idea: consider a function f, with a check, *chk(L)*, at program point *L* just prior to some expression *e*.

$f(v_1,...,v_n) = ...L@e...$

Suppose that the contextual information that is available at program point *L* is denoted by a predicate, ctx(L). In general, the check chk(L) can be shown to be totally redundant if the following test holds (where => denotes implication, and |- denotes deduction):

ctx(L) => chk(L) |- True

Note that under the scenario that ctx(L) = False, the check chk(L) will be avoided and hence redundant too. This can occur in dead (or partially dead) branches of program.

Example: The low bound check at L1 from the previous defined function *access* is redundant since (i >= 3) => (i >= 0) |- *True*

Alternatively, chk(L) is said to be unsafe to eliminate if

ctx(L) $\wedge$ chk(L) |- False

This contradiction between the context and check indicates the possibility of bound violation.

Example: The low bound check L2 is unsafe since (j=-5) $\wedge$ (j >= 0) |- *False*

Otherwise, the check is classified as *partially redundant*. Under this situation, we are interested in finding a pre-condition, let us call it pre(L), that would allow chk(L) to be made redundant. In other words, we are interested in deriving a suitable pre(L), such that:

(pre(L) $\wedge$ ctx(L)) $\Rightarrow$ chk(L) |- True

A trivial pre(L) to use is False but we are not interested in such a useless pre-condition. Instead, we are primarily interested in finding a pre-condition that is as weak as possible.

Ideally, we should derive the weakest possible pre(L) that is sufficient to make chk(L) redundant. Derivation of the weakest pre-condition depends on a number of factors, the primary one being that the context at program point L should be as informative as possible. In this project, we use a method, based on sized typing, for synthesizing informative contexts. Based on this contextual constraint the weakest pre-condition can be computed using the formula:

$$pre(L) \equiv \neg\, ctx(L)\ V\ chk(L)$$

Example: The upper bound check H1 is partially redundant. We can compute a pre-condition (that is weakest with respect to a given context) to make the check redundant, as follows:

$$pre(H1) \equiv \neg\, ctx(H1)\ V\ chk(H1)$$
$$\equiv \neg\, (i >= 3)\ V\ i < length\ (A)$$
$$\equiv i < 3\ V\ i < length\ (A)$$

Note that this pre-condition is indeed very weak as it captures both the conditions for satisfying the check, as well as for avoiding the check.

This algorithm uses a new approach to handling partially redundant checks that cleanly separates the forward analysis for synthesizing contexts (aiming for most informative post-condition), from the backward derivation of pre-condition for checks (aiming for weakest pre-condition that allows the checks to be safely eliminated).

1.2. Contributions of ABCE Algorithm

The algorithm presented in “Deriving pre-conditions for array bound check elimination” is based on sized-type inference, specifically on automatically inferring input/output size relation and also determine invariants for parameters of recursive functions over the sizes of data structures used. The inference is performed accurately and efficiently with the help of Omega constraint-solver on Presburger form [22]. The presence of sized type greatly enhances inter-procedural analysis of contextual constraints, which are crucial for identifying both unsafe and totally redundant checks. More importantly, accurate contextual constraint also helps in the derivation of safely preconditions for partially-

redundant checks. With the derived pre-condition, we can provide specialized code to selectively eliminate partially-redundant checks based on the available contexts. The specialization process can be further tuned to provide a range of time/space tradeoff.

1. Partially redundant checks are handled through the backward derivation of safety pre-condition after contextual constraint has been gathered in a separate forward phase. This gives very accurate result for eliminating partially-redundant checks.
2. Deriving weakest pre-condition (with respect to a given contextual constraint) works seamlessly across recursive functions.
3. Inter-procedural optimization is handled through backward propagation of a function's pre-condition to its callers to become a check.
4. Inherent imperative problems introduced by assignments and loops are handled by converting the input code into static single assignment form (SSA). Reasoning on SSA form is much easier because the compiler can keep reference of all the use sites and all definitions sites for the variables used (a link between definition site of variables and its use sites can be established).

Chapter 2

Overview

2.1. Sized Types Information

A normal type system is enhanced with the notion of *sized types*, which captures the size information about the underlying expressions/values. For a function, sized type reveals size relationship amongst the parameters and results of that function.
Sized type is represented by a pair containing an annotated type and a Presburger formula. An annotated type expression augments an ordinary type expression with size variables; the relationship among these variables is expressed in the associated formula.
As an example, let us see what are the sized type for integers, booleans and arrays. For integers variables, the sized type is Int^v, where v captures the integer value. For booleans, it is $Bool^v$, where v can be either 0 or 1 (this constraint is expressed via a suitable Presburger formula), representing the values *False* and *True* respectively. For arrays, it is $Arr^v\ \tau$, where v captures the integer value, and τ represents the type of components.

2.1.1. Size Annotation for Non-Recursive Functions

When encountering a function call in our analyses, we need a relation between the arguments and the result of the function. This relation represents the sized type of a given function.
In our work, the sized type for each function is not calculated. Instead, it should be introduced in the input file as a special comment, between "/*ST" and "*/" tags.
We provide here the source code for *cmp* function, preceded by its sized type:

```
/*ST cmp :: (x, y) -> r st
        ((x<y && r==0-1) || (x>y && r==1)) || (x==y && r==0) */
```

```
int cmp (int x, int y){
        if (x<y) return 0-1;
        else
                if (x==y) return 0;
                else return 1;
}
```

In the case where one of the arguments is not important, it can be replaced by an anonymous variable (represented by "_"). The substitutions (between formal and actual arguments) will not be done for such anonymous parameters. This implies that the resulting formula will be simpler.
If one of the parameters of function or the result is of complex type, then the size types of its components will be represented as a pair. For example the sized type for *getmid* function can be represented as follows:

```
/*ST getmid :: (A, lo, hi) -> (f,s)
        st (A>0) && (f==(lo+hi)/2) */
```

The pair (f,s) is the sized type of components for a variable of type *Pair* declared as follows:

```
class Pair {
        int first;
        int second;
};
```

2.1.2. Loop Invariants

In our language, loops are implemented using *"while"* instructions. Here, we need a static relationship between the variables that change their value inside this loop, let us

call them *recursive variables*. Specifically, this formula should be a relationship between the initial value of *recursive* variables and the value of these variables throughout the loop.
These variables are similar to recursive parameters, and the relation that is captured is known as loop *invariant*.
An example of a loop with an annotated invariant relationship between initial variables (before the loop) and recursive variables is shown below:

```
/*INV (i*<10) && (i*>i)*/
while (i<10) {
        i=i+1;
        x=A[i];
}
```

Notice that the formula contains a recursive variable, *i**. This variable is used to capture the value of variable *i* for the first iteration of the loop and beyond.
In the previous example, the only recursive variable is *i*, and the invariant, namely (*i**>*i*), indicates that *i** is increasing its value, relative to its initial value, *i*.

2.1.3. Recursive Functions Invariants

Recursion can be viewed as a generalization of iteration loop. Being related to iterative loops, we can expect that for each recursive function we need a similar relation to be inserted by the programmer (as an annotation). For example:

```
/*ST sumarray :: A -> l -> i -> r st
        (A*==A) && (l*==l) && (i*>i) */
int sumarray (int A[], int length, int index){
        if (length == index)
                return 0;
```

```
        else
                return A[index]+sumarray(A,length,index+1);
}
```

Parameters for recursive functions can be regarded as recursive variables that change in value from one call to next). In this example, the argument *index* – denoted by its size variable "*i*", is increasing monotonically, so the formula inferred is (i*>i).
We call the initial set of arguments (A, l, i) the *initial arguments*, and that of an arbitrarily nested recursive call (A*, l*, i*) the *recursive arguments*.
All these relations should be included in the input file as annotations (via special comment statements), but they can be inferred automatically. In the paper "Calculating Sized Types", Wei-Ngan Chin and Siau-Cheng Khoo [4] describe such an algorithm for functional programs.

2.2. Bound Check Optimization Algorithm: Overview

Once the sized types of related functions have been inferred, we proceed to handle bound check optimization. The process works in a backward manner, starting with the function at the bottom of the call hierarchy. We briefly discuss the steps involved below, and use two functions from the binary search program depicted below as an illustration.

```
/*ST getmid :: gA -> glo -> ghi -> (f,s)
        st ( gA>0 && f=(glo+ghi)/2 )*/
Pair getmid(int []gA, int glo, int ghi){
        int m = (glo+ghi)/2;
        Pair p;
        p.first=m;
        p.second=gA[m];       //L3,H3
        return p;
}
```

```
int look(int A[], int lo, int hi, int key){
    Pair p;
    /*INV ((lo*<=hi*) && (hi*<=hi) && (lo*>=lo)*/
    while (lo<=hi){
        p=getmid(A,lo,hi);     //L4,H4
        ...
}
```

Step 1. Perform contextual-constraint analysis to determine the contexts at each function call or array access (such checks are denoted as *labeled calls*, and we assign them a simple label).

The implementation of forward analysis is described in chapter 5. For function *getmid*, both the labels L3 and H3 are within identical context. The contextual constraint is computed to be:

$ctx\ (L3) = ctx\ (H3) = (a \geq 0 \wedge 2m <= glo + ghi)$

For each labeled call in a recursive function, we obtain two contexts: a *first* context that encloses the labeled call without invoking any nested recursive call, and a *recursive* context that encloses the labeled call lying arbitrarily deep in the nested recursive-call invocations. Each loop is also analyzed twice: first iteration and repeated iterations will provide two separate checks, which will be, eventually, combined in the final condition. For function *look*, we obtain:

$ctxFst(L4) = ctxFst(H4) = (lo <= hi \wedge A >= 0)$

$ctxRec(L4) = ctxRec(H4) = (lo*<=hi* \wedge hi*<=hi \wedge lo*>=lo)$

Step 2. For each labeled-call site:

a. Convert each non-trivial pre-condition (condition that is neither True nor False) from the call into a check.

   Array-access operations are typically associated with a pair of pre-conditions, both of which must be satisfied for the operation to be safe. These pre-conditions are collectively identified by the keyword Req. (*Req L: i>=0; H: i<length(array)*). Just like labeled calls, every pre-condition is labeled

accordingly. To convert a pre-condition to a check, we perform substitution of size operands. This operation is detailed in chapter 3. For the array access from function *getmid*, the conversion (and simplification) yields:

chk(L3) = (0 <= m)

chk(H3) = (m < a)

b. Determine if the check is unsafe, redundant or partially redundant. If it is partially redundant, then perform backward derivation to compute a pre-condition that makes the check redundant.

For instance, the pre-conditions derived from the checks in *getmid* function are:

pre(L3) = (0 <= glo + ghi)

pre(H3) = (glo + ghi < 2a)

By converting these pre-conditions into checks at L4 and H4, and using the contextual constraints ctxFst(L4) and ctxRec(L4) (obtained earlier), we derive the following pre-conditions for function *look*.

pre(L4) = (hi < lo) V (0 <= lo + hi ∧ 0 <= lo)

pre(H4) = (hi <= lo) V (hi < a ∧ lo + hi < 2a)

Interestingly, we will show later that the pre-conditions derived for the function *bsearch* have the value *True*. This implies that all bound checks associated with the call look(arr,0,length(arr)-1,key) are totally redundant, and can safely be eliminated.

2.3. Why Haskell?

Why use Haskell as a programming language? There are many reasons to use functional programming in general and particularly Haskell.

Functional programs tend to be much more *concise* compared to similar imperative programs. Because of this conciseness, functional programs are easier to understand. A typical example presented as a demonstration in Haskell introductory tutorials is the quick-sort algorithm.

```
quicksort [] = []
quicksort (x:xs) = quicksort [y | y <- xs, y < x]
                   ++ [x]
                   ++ quicksort [y | y <- xs, y >= x]
```

Another advantage of Haskell is its *strongly typed* characteristic. Errors related to misinterpreting types can be identified at compile time. On the other hand, the programmer need not worry about *memory management*: since storage is allocated and initialized implicitly, and recovered automatically by the garbage collector.
Re-usability is encouraged in Haskell by its type-system that is *polymorphic*, and by the use of *higher-order* functions. Even the classes that are predefined (Show, Eq and so on) have methods that are inherited by datatypes derived from these base types.

2.3.1. Defining new types in Haskell – Object oriented features

The keywords involved in type definition are different in Haskell from C++ or Java, so a programmer used with latter languages can be a little disoriented at first.
A definition of a type synonym – prefixed by the *type* keyword – can be used to define a n-tuple, a list or a combination of them over some existing composite types.

```
type Pair = (Int, Int)
type Vector = [Int]
```

In the previous defined datatypes, the components of a record (e.g. Pair) do not have names, and they can be extracted from the record variable by user defined functions. However, there is a possibility of defining names for record's components [6].
A data type defined using *data* keyword can be regarded as an enumeration of types of different parities. The first word from every definition is the type-constructor. In the case that for all alternatives the type-constructor does not have any parameters, the defined type is equivalent with an *enum* from C. However, there are no integers associated; the

only possible reference is using the respective type-constructor. The type-constructor can have parameters (types defined in this way are similar to unions from C), and the new structure can be defined recursively.
Class keyword introduces signatures for methods that can be used on types derived from this datatype. The concept equivalent in Java is an interface containing method's signatures (a similar notion is a virtual function from a C++ class). Haskell permits default implementations for some of the methods. An example is the *Eq* class to capture data types that supplies equality (==) and inequality (/=) operations.

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y     = not (x==y)
```

An *instance* of a class can contain method's implementations, so it is equivalent to a class definition from Java. Note that by deriving a type from a base class, Haskell can provide automatically default implementation for the newly defined type.
For example, the equalities for lists can be defined automatically based on its recursive type structure and on equality of its elements. Thus:

```
instance Eq a => Eq [a] where
  []     == []     =  True
  (x:xs) == (y:ys) =  x==y && xs==ys
  _      == _      =  False
```

However, there is no possibility to control the access (public or private) to methods from Haskell class system. Instead, one can use the module system to hide or reveal components of each class.

2.3.2. Lazy Evaluation and Referential Transparency

Non-strict functional languages have another useful feature: they only evaluate the expressions needed in order to find out the result. This is called *lazy evaluation*. For example, an analysis of input code to remove redundant checks will be carried on (through a function) just until the last array check is analyzed. After this check, there is no reason to carry on the analysis, and the Haskell interpreter "knows" this without further indications from the programmer. In contrast, an imperative programmer needs to examine the input code separately to decide where to stop the analysis.
An expression E is *referential transparent* (and the resulting code is *purely functional*) if any two sub-expressions can be interchanged without changing the value of E. The order of evaluations does not matter because no function uses side effects. An example of a side-effect is the assignment of global variables which is forbidden in pure functional languages.
When side effects can appear (for example, programming the IO monad), then the order of evaluations matters, and the actions (as opposed to evaluations) will be executed in a definite order.

2.3.3. Monads

No side effects are allowed in pure functional languages (including Haskell). Everything that happens in a function can be deduced from its body. The flow of data is explicit; every possible bug is under our eyes and is, therefore, easier to be discovered. This is a big help for modularity. Each function does not depend on its caller to "arrange" a good environment. This is a practice discouraged in imperative programs, but still possible there. Because this is not possible in Haskell, a function can be called safely from anywhere considering that the arguments are passed correctly. On the other hand, having to pass so many arguments from the point of definition to the actual place of use will hide the algorithm itself and make the code hard to understand.
The good parts from both worlds (functional and imperative) can co-exist through monads. The code will continue to be pure (not cluttered by side-effects), and it will be much more readable. This is because not all parameters must be passed explicitly.

Unfortunately, datatypes defined as monads will complicate the program, especially for those that are not used with monadic programming. All the functions will have parameters as abstract datatypes, usually based on other functions.
Haskell defines a constructor class for monads, as follows:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a

  -- Minimal complete definition: (>>=), return
  p >> q  = p >>= \ _ -> q
  fail s  = error s
```

To define an instance of monad class, one must define >>= (equivalent to *then*, executes p then pass the result to q) and *return* bodies. All other operators are based on these two functions.
In Haskell, the first contact with monads is, usually, through the input-output system. From here it can be deduced another important property of monads, namely all the changes done to the contained type (monads can be regarded as containers for values of another type) are guaranteed to be “single-threaded”. This particularity guarantees that the referential transparency of Haskell is not endangered by this way of programming. “Do” notation, available for monads, can be regarded as a way to code in imperative style because of the explicit sequencing of instructions.

Chapter 3

Parser for Java language

3.1. Subset of Java language

The syntax of the language that can be parsed by our analyzer is based on Java. Vector indexes are assumed to be integers, so our analyzer handles them with priority. There should not be any difficulty in using other simple types, such as booleans or chars.
Based on simple types, we can define arrays using a similar syntax to Java. For example, the next instruction declares an array of ten integers:

int A[]=new int[10];

Arrays are zero-indexed, and our analyzer do not use size analysis on the components of arrays. However this is a possible improvement of current project.
A class is seen like a record containing functions that are applied onto the fields of that class. All fields and methods of classes are implicitly declared as public and static.
Similar to a class definition in Java, we can create a record type:

```
class Pair{
        int first;
        int second;
};
```

The instructions that are supported form a small-subset of Java, but they are sufficient for implementing many algorithms. A list of instructions that can be used follows:

- declaration of variables;
- assignment;
- function call;
- if then else instruction;
- while-loop (for-loops can be implemented through while-loops);

- return instruction;

The exceptions that are thrown by a Java program are usually divided into two categories: exceptions that must be handled by a function (it's compulsory to mention these exceptions in the signature of the respective function) and exceptions that can be handled by the virtual machine. The last category is derived from the type RuntimeException and contains exceptions like ArrayIndexOutOfBoundsException or NullPointerException. Because runtime exceptions can be raised by virtually any instruction (which contain an array access, an object field or method access), our analyzer has a complex task in preserving the semantics of the program in the presence of exceptions.
For languages without exceptions, the checks can be moved before their actual occurrence regardless of their safeness. One reason to do this could be hoisting a check – depending on a constant – out of its loop.
Languages that need to handle the mechanism of exception should be analyzed by a method involving *precise exceptions*: any exception that will be raised in a normal execution of program, should be raised at the same point by the optimized program (after having some checks removed). Because our analyze does not try to move checks, we can demonstrate that two executions of the same program on the same environment should raise exception at the same point, preserving the execution flow.

3.2. Internal Representation of a Monadic Parser

Advantages of monadic programming can easily be seen when implementing a parser: the state monad will hide many arguments in the parser type. Also exceptions are easy to propagate, delaying the action to be taken until the adequate moment.
In our program the parser type is declared as follows:

```
data StateM m s a = STM (s -> m (a,s))
type Pos = (Int,Int)
type Parser a = StateM [] (String,Pos) a
```

In this case, first element of the state (an “invisible” argument) is the position in the source code, stored as a pair: line-column. The error messages (generated by the parser) will be much more helpful, because they can specify the exact place where the detected error occurs. More importantly, the state contains the string to be parsed.
The first phase is the lexical analyze, which will separate the input in tokens, so the following phases will be much quicker (and the code easier to write). The lexical analyze is implemented through the “gettoken’” function, which will return the next token from the input stream, ignoring spaces and comments.

gettoken' :: Parser Token

Every parser will modify the input string, taking from it the tokens needed to satisfy the parser – if possible. The parser functions have the type *Parser a*, where *a* is the type that will represent the result of parsing (parts of the abstract syntax tree, such as an expression or an instruction).
Based on the type of tokens (and, also, on their contents) returned, the syntactic analyze can recognize “patterns” of input code. The return values have the following datatypes:

- Identifier – a token that has the type *word*. The function *pIdentifier* will recognize a simple identifier, while *pIdentifier’* will recognize a qualified identifier – a simple identifier prefixed with the name of classes, to whom it belongs.
- Class – it represents the information gathered from a Java class. The identifier is the name of the class, the list of instructions represents the declarations of the fields of the class, and the last parameter is the list of member functions.

  data Class = Class Identifier [Instr] [Function]
- Function – the equivalent of a static java function, has as parameters the type returned (a qualified identifier), the name (a simple identifier), a list of arguments – pairs between the type and the name of each argument, an instruction, a header (a list of size variables for the arguments of the function), a sized type (for functions the sized type is a Presburger formula, and it is represented as an expression from AST), and a list of preconditions. The instruction is constructed with the *Block* type-constructor, so it is, in fact, a list containing all instructions from the body of the function. The size variables from the header of a function are represented through a list of strings, because any of them can be a record. A

precondition is a 3-tuple containing a name, a formula and a safe measure for the precondition. Based on its name, preconditions that are proved to be safe, can guide the context specialization).

```
data Function = Function [Identifier] Identifier [([Identifier],Identifier)]
Instr Header SizedType Preconditions
type Header = [[String]]
type SizedType = Expr
type Preconditions = [(String,Formula,Safeness)]
data Safeness = Complete | Unsafe |NotSure
```

- Expression – possible values are: void expression, a numeric literal (having as parameter the value), a variable (possibly qualified), an array access (the name can be qualified, the indexes are stored as a list of expressions), a function call (the name of the function can be qualified, and the arguments are represented by a list of expressions), or a predefined function. The functions that are predefined are the usual arithmetic and boolean functions.

```
data Expr = Void
          | NLit Identifier
          | Var [Identifier]
          | ArrayAccess [Identifier] [Expr]
          | ECall [Identifier] [Expr]
          | Op OpName [Expr]
data OpName = OpPlus | OpMinus | OpMul | OpDiv | OpAnd | OpOr |
OpEq | OpDif | OpGt | OpGte | OpLt | OpLte
```

- Instruction – the instructions that are supported and their parameters are
  - *if* having as parameters a test condition, the instruction corresponding to *then* branch, and the instruction corresponding to *else* branch. Both instructions are defined as *Block*, so they are in fact a list of instructions.

- a block of instructions - a list of instructions, prefixed and ended with accolades.
- a return instruction - the only parameter is the returned expression;
- *while* having as parameters a test condition, a block of instructions, and a loop invariant, implemented as an expression;
- *assign* - the left-hand side is a qualified identifier, the right-hand side is an expression;
- *declaration* - the type of the newly declared variables are qualified identifiers, the names of the variables are paired with expressions, representing the initialization values. When defining a vector, the second parameter is a list of expressions, standing for dimensions of a vector - possibly multi-dimensional. Currently only one variable can be assigned per instruction.

```
data Instr = If Expr Instr Instr
    | Block [Instr]
    | Return Expr
    | While Expr Instr Invariant
    | IAssign [Identifier] Expr
    | Decl [Identifier] [Expr] [(Identifier, Expr)]
```

The Instruction and Expression types are described as concise as possible. There are no further explanations for the type-constructors that allow instructions or expressions handled only by the parser, but not by the further analysis (boolean literal, assignment as expression, try-catch instruction, function call as instruction).
As an example of a parser function, let us consider the parser for a function definition, given in EBNF (Extended Backus Naur Form) syntax and its implementation:

<function> ::= <size type> <type_result> <name_function>
"(" <argument> ")" <block>*

The function that implements this EBNF syntax, *pFunction*, will return the constructed function. First it will try to identify the size type information. In case that this information

is missing (pSized will fail) the default formula for size type is *True*. *'opt'* will execute its first argument (of type Parser a) and in case that it will fail, the result will be the second argument. The result of pSized is a pair, the first element is a list of size variable (arguments and result for the current function) and the second one is the size type. The parser will continue to identify tokens as required by the EBNF syntax in the same manner. Note that the result of a parser is converted to an argument for the next parser with ">>=" operator.

```
pFunction :: Parser Function
pFunction = pSized `opt` ([],BLit ("True",BLiteral,4)) >>= \(h,st) ->
        pIdentifier' >>= \typef -> pIdentifier >>= \id ->
                try "(" >> (pArguments `opt` []) >>= \args ->
                try ")" >> pBlock >>= \block ->
                return (Function typef id args block h st [])
```

Basic operators for parsing are defined as follows:

```
liftop :: Monad m   => (a -> b -> c)      ->  m a -> m b -> m c
liftop f mp mq=mp >>= \p-> mq >>= \q-> return (f p q)
(<:*>) :: Monad m => m a -> m [a] -> m [a]
(<:*>) = liftop (:)
many   :: Parser a -> Parser [a]
many p = (p <:*> (many p)) `opt` []
some   :: Parser a -> Parser [a]
some p = p <:*> (many p)
atmost1:: Parser [a] -> Parser [a]
atmost1 p = p `opt` []
```

The equivalent for * in EBNF notation is the operator *many*, which will try to apply its first argument (of type Parser a), and then concatenate the result with the same parser

applied many times (type Parser [a]). In case the first parser cannot be applied the result will be an empty list.

The equivalent for + in EBNF notation is the operator *some*, which is defined similarly to *many*, but its first argument **cannot** fail. At least once, the parser *p* must succeed.

The parser *atmost1* is equivalent with ? operator from EBNF. It will try to apply the parser p (the first argument) or fail with an empty list as a result.

The base of implementing a parser that is modular and extensible are operators that are equivalent to those from the definition of a EBNF grammar.

Chapter 4

SSA Renaming

As we are collecting information about variables (relations between them), we need a way to name them. When we consider variables that can be re-assigned, we have a problem. After two assignments of the same variable, there are two values for that particular variable (which will render existing formula to be FALSE, instead of assigning the second value and ignoring the first value).
This problem can be resolved, if we choose a SSA (Static Single Assignment) form for our input program. In SSA form each variable has only one definition in the program text and so, at any moment, is clear at which variable we are referring.
While SSA form is easier to analyze, loops has to be handled carefully. The condition of a "while" loop contains a relation between some variables. This condition may be altered through modifications to these variables (otherwise the true value of condition will remain unchanged, rendering the loop infinite). It is obvious that a variable should modify its value in a loop, considering also that it was initialized before the loop.
Andrew Appel's article "SSA is functional programming" [2] discuss this question, but we can adopt here a simpler solution. In our input source code we have loops, but these loops are not analyzed for an unknown number of times, but exactly twice: once for the first iteration and the second time for subsequent iterations.

4.1. Re-assignment of variables

Let us consider the following fragment of code:

```
int i=0;        //i=0
int x,y;
x=A[i];         //first check
i=i+3;          //i1=i+3
```

y=A[i]; //second check

At the first check the contextual constraint is (i=0), so we can decide on safeness of the check (the lower bound) with the following formula: (i=0) => (i>=0). For the second check, if we do not rename *i* variable we will have the high bound check rendered unsafe, regardless of the dimension of the *A* vector (which loses to much information).
Therefore, we need to generate a new name for the variable reassigned (i -> i_1), and substitute all subsequent use of *i* variable with the new name i_1 (from this instruction towards the next re-assignment of *i* variable).
The re-assignment of a variable can be viewed as being equivalent with a definition of a new local variable:

int i=0;
int x,y;
x=A[i];
int i_1;
i_1=i+3;
y=A[i_1];

This resemblance indicates the way in which the newly declared variable will be eliminated from the pre-conditions to be propagated to other functions. This is similar to the way in which a “real” local variable is eliminated via *forall* quantifier.

4.2. Re-assignment of variables in conditional instructions

With *if_then_else* instructions, it is possible for a variable to be re-assigned just on one path (for example in “then” block), but not the other. Post-condition for *if* statement should contain substitutions for the variables that are not re-assigned on both paths.
As an example, consider the following code:

```
int i=0,j=0,k=0;
int x,y,z;
if (...) {
        i=i+1;          //i1=i+1
        j=2;            //j1=2
        k=k+5;          //k1=k+5
        k=k+6;          //k2=k1+6
}
else {
        i=i+3;          //i1=i+3
        k=4;            //k1=4
}
x=A[i];
y=A[j];
z=A[k];
```

After the *if* instruction, we need to know which *i* variable we shall be referring to. Similarly, if we are considering a variable that is re-assigned in just one path (*j* variable) or re-assigned a different number of times (*k* variable), we should know the final value after if-statement.

After transforming the example to SSA form, the code that results contains highlighted substitutions needed and definitions of newly introduced variables:

```
int i=0,j=0,k=0;
int x,y,z;
int i1,j1,k1,k2;
if (...) {
        i1=i+1;
        j1=2;
        k1=k+5;
        k2=k1+6;
```

```
}
else {
        i1=i+3;
        k1=4;
        k2=k1;
        j1=j;
}
x=A[i1];
y=A[j1];
z=A[k2];
```

This effect is achieved by merging the substitution that are done in two branches of each conditional, so that the re-assigned variables synchronize.

4.3. Re-assignment of variables in while instructions

For a while loop, we have to analyze the instructions twice: once for the first iteration, and the second time for the repeated iterations. The repeated iteration is analyzed by having regular variables replaced with recursive variables (for *i*, the recursive variable is denoted *i**). Syntactically, we wrote it as *RECi* for easy understanding of the code. A re-assignment can be made inside of a loop, and the set of re-assigned variables will be the same for both analysis of the loop.
Let us consider the following simple example:

```
i=0;
while (i<10){
        i=i+1;
}
x=A[i];
```

If we unfold the two analysis of loop, and point out the renamed variables, we will obtain a SSA equivalent for this loop example:

```
i=0;
if (i<10) {                    //first it
        i1=i+1;
        if (i1<10) {           //repeated it
                /* assume invariant */
                RECi1=RECi+1;
                /*assume ¬(RECi1<10) */
        } else {               //only 1 it
                RECi1=i1;
        }
} else {                       // 0 it
        RECi1=i;
}
x=A[RECi1];
```

Note that *RECi* has no specific **value**, since it is a recursive variable that is bounded to an **interval** from the loop invariant. In our example: *RECi* ∈ [i,10].
In the rest of the paper, variables that are re-assigned **in** a loop are denoted **after** the loop with a specific notation. That is, *i* is renamed as *i#* to capture the value of *i* after the loop (i# / i). In the last example, i# is the same with $RECi_1$. If *i* would have been re-assigned multiple times then *i#* will be represented by $RECi_n$ (n is the number of re-assignments inside the loop).

4.4. Renaming Implementation

Renamings (substitutions) are kept as a list with pairs of variables as elements. Let us consider the example from the previous chapter section:

[(k,k_2),(k,k_1),(j,j_1),(i,i_1)] *("then" path)*

[(k,k_1),(i,i_1)] *("else" path)*

The reason why we keep more than a substitution for a variable (*k* in "then" path) is the need to replace the most "current" variable *k*, with the most "current" variable *k* at another moment of time. One example is the while loop. After recursive iterations we will have references to recursive variables – i_1*. After one iteration, the relation refers to just i_1. These two variables will be related in the branch following "one iteration". For the branch following "no iteration" the variable will be related to i_1*. This variable will be the most current *i* at that moment, which is exactly *i*.

The function that can substitute a variable will replace every occurrence of a variable, with the most "recent" substitution. That is the reason why the substitutions are kept in reverse order.

subsVar :: Variable -> Substitutions -> Variable

From the two lists of substitutions (for then and else paths) we can deduce which substitutions must be applied in order to have the most "recent" re-definitions of each variable.

Chapter 5

Context Synthesis

The algorithm for bound check elimination begins by determining the context in which a check occurs. Information about context serves two purposes:

- it can be used to determine if a check is redundant within the specific context;
- in case the check is not redundant, it can be used to derive the pre-condition necessary for eliminating/avoiding the test.

Contextual information is described in Presburger form, and is called *contextual constraint*. This is gathered by a forward analysis, specifically parsing the instructions from a function (block) sequentially.

The forward analysis task is done in our program by a function that have as inputs the following parameters:

- the current instruction (at the beginning the current instruction is the block containing all the instructions from the function to be analyzed; these instructions will be analyzed sequentially, the post-condition deduced from one constituting the pre-condition for the next one).
- pre-condition: this is the constraint that captures the context determined from the current function (starting with the first instruction until the current one). At the starting point of the function this parameter is set to *True*. Changes from pre-condition to post-condition depend on the type of the current instruction, and are detailed in the following section.
- the current function: when encountering a check which is proved to be partially redundant, the inferred precondition needs to be added in the list of current function's preconditions.
- the list of sized types of all functions from the program. At the place of each function call, we need the relation between the inputs and outputs of this function. This relation, together with substitutions between formal and actual arguments, will give a relation between the actual parameters (of the function that is called) and the result.

- variables environment: it stores information regarding variables, namely their type and a flag that is true for a variable that already has assigned a value. At a place of an assignment, we can deduce if the left-hand side needs to be renamed. This is done, based on a boolean flag, indicating if the variable is already assigned a value, as described in previous section.
- type environment: a variable can be of simple type, or record type. Record fields are stored in the type environment together with their types.
- renaming: when a renaming is done, it needs to be stored for further reference. Every appearance of this variable will be represented with the new name (between the moment of re-assignment and a possible, later renaming).

5.1. Evaluating Expressions

Evaluation of expressions poses some problems for array components and function calls. Otherwise expressions are evaluated in the order in which they appear in the abstract syntax tree.

The formula that results from an expression that contains an array bound check is useless because we cannot make any assumption about the values of vector components. For example, an assignment that have an array access as its right-hand side will have its left-hand side *unknown* after this instruction.

In the case of a function call, the sized type of that function should be put in the current context, with the formal arguments replaced by actual arguments. Because the whole instruction that contains this function call is not known at the time that the expression is evaluated, the formula deduced is not yet complete.

In the following example the bold part will have to be added to the final formula when analyzing the whole instruction.

*/*ST cmp :: x -> y -> r*

s.t. (x>y && r=1) || (x<y && r=-1) || (x=y && r=0)/*

A = B + cmp (a,b);

A = B + Bottom (uncompleted_fcall r)

*fcall = exists (formal_args: st(x,y,r) And (a=x) (b=y) (**A=B+r**)*

A case that is resolved in a similar manner is an expression that contains a *divide* operation. The final formula which will be simplified by Omega, must not contain the divide operation. Because of this, we will substitute the divide operation with the following formula:

*e=C/D |- e*D <= C <= (e+1)*D*

As in the previous example, the bold part will be added when analyzing the whole instruction.

A = B + C / D;

A = B + Bottom (uncompleted_div e)

*div = exists (e: e*D <= C and e*(D+1) >= C and **A=B+e**)*

The variable *e* approximates the result of the divide operation (all variables are integers). In the propagated formula *e* will be removed in the same manner as local variables (through *forall* quantifier).

5.2. Context synthesis from specific instructions

The context synthesis function takes a pre-condition and, based on the type of the current instruction, calculates a post-condition. All the parameters of this function are presented in the previous section. Here, we describe the important ones for simplicity.

5.2.1. Block instruction

The instructions from a block are parsed sequentially, with the context and substitutions taken over from the previous instruction.

C (Block (i:instr)) (ctx,subs) = C (Block instr) (ctx',subs') where

–(ctx',subs') = C i (ctx,subs)

5.2.2. Assignment instruction

An assignment instruction is added as equality between the left-hand and the right-hand side. If the left-hand side is already assigned, there will be a substitution added for it.

C (IAssign id expr) (ctx,subs) = (And (Eq id expr) ctx,subs)

5.2.3. Conditional instruction

An if_then_else instruction is analyzed by considering separately "then" and "else" blocks, and afterwards mixing the results (both formulas and different substitutions). Final substitutions are both substitutions made for "then" path and for the "else" path. If a substitution is made for both paths, then it will be put just once in the final list. The formula deduced is an "or" between the contexts inferred on two paths, both having added the substitution that were made for the other path.

C (If cond thenI elseI) (ctx,subs) = (Or (And ctx' (subs'' – subs)) (And ctx'' (subs' – subs)),subsIf) where

–(ctx',subs') = C thenI (And cond ctx)

–(ctx'',subs'') = C elseI (And (Neg cond) ctx)

–subsIf = (subs' – subs) ++ (subs'' – subs) ++ subs

As an example, consider the following code:

```
int i=0,j=0,k=0;
int x,y,z;
```

```
if (...) {
        i=i+1;          //i1=i+1
        j=2;            //j1=2
        k=k+5;          //k1=k+5
        k=k+6;          //k2=k1+6
}
else {
        i=i+3;          //i1=i+3
        k=4;            //k1=4
}
x=A[i];
y=A[j];
z=A[k];
```

After the "if" instruction, we need to know which *i* variable are we referring. There is the same problem if we are considering a variable that is re-assigned just on one path (or is re-assigned a different number of times).
To decide on the safeness of the first check (lower bound) consider the following formula:

$$\{(i=0) \;\&\&\; (condition) \;\&\&\; (i_1=i+1) \;\|\; (i=0) \;\&\&\; (Not\ condition) \;\&\&\; (i_1=i+3)\} \Rightarrow (i_1 >= 0)$$

For the second check, we need to add a substitution (j_1 / j), because on the "else" path, we don't have this re-assignment. After *if* instruction, we can refer to all appearances of *j* variable (until a new re-assignment) as j_1.

$$\{(j=0) \;\&\&\; (condition) \;\&\&\; (j_1=2) \;\|\; \{(j=0) \;\&\&\; (Not\ condition) \;\&\&\; \mathbf{(j_1=j)}\} \Rightarrow (j_1 >= 0)$$

For the third check, we obtain the following formula:

*{(k=0) && (condition) && (k_1=k+5) && (k_2=k_1+6) || {(k=0) && (Not condition) && (k_1=k+4) && **(k_2=k_1)**}*
=> (k_2 >=0)

5.2.4. While instruction

The instructions from a while loop are analyzed twice. For the first iteration, the starting formula is the condition (which must be true, if we are going into the loop) and pre-condition (valid before the loop). For the repeated iteration, the starting formula is an "and" between the invariant of the loop, the loop condition (having replaced the recursive variables), and pre-condition. The second analysis (for repeated iterations) is made by substituting all the variables that change their values in the loop with their recursive counterparts (subsRec function does this substitution). The first and the iterative iterations are made to determine the safeness of checks **from** the loop.
To determine the safeness **after** the loop, the final formula is an "or" between three alternatives:

- the loop condition was false from the beginning (no execution of the loop), so the final formula contains both the negated condition and pre-condition (valid before the loop).
- the condition from the while loop was true on the first iteration, but after execution of the first iteration, it was changed to false (only one iteration). The final formula contains the context gathered after the first analyze, and the negated condition (with correct variables substituted).
- the condition was true for two or more iterations and the post-condition for the loop contains the context gathered after the repeated analyze, and the negated condition (eventually the loop will be exited).

C (While cond instr inv) (ctx,subs) = (Or [zeroIt,oneIt,moreIt],subs'') where
–(ctx',subs') = C instr (And cond ctx,subs) --fst

–(ctx'',subs'') = C instr (And cond inv ctx,subsRec)--rec*

–zeroIt = And ctx (Neg cond) (subs''-subs)

–oneIt = And ctx' (Neg cond <-subs') (subs''-subs')

-moreIt = And ctx'' (Neg cond <- subs'')

Let us consider the following simple example:

```
i=0;
while (i<10){
        x=A[i];
        i=i+1;
        y=B[i];
}
z=C[i];
```

For the first iteration we have the following context: (i=0) && (i<10)

- for the first check (lower bound): (i=0) && (i<10) => (i>=0)
- for the second check (lower bound): (i=0) && (i<10) && (i_1=i+1) => (i_1>=0)

For the repeated iteration, we need to make another set of substitutions to reflect the changes from the variables (for the first iteration) to the "recursive" variables.

- for the first check (lower bound): (i=0) && (i*<10) => (i>=0)
- for the second check (lower bound): (i=0) && (i*<10) && (i_1*=i+1) => (i_1*>=0)

To deduce the safeness of the first two checks, we need to "and" both conditions (from the first iteration, and from the repeated iterations).

For the third check, after the while loop, we need to consider three variants:

- the loop was never executed, because the condition was not true at the beginning (this is not the case for this example);
- the loop was executed only once, and after this the condition was no longer true (replacing i_1 in the loop condition that is no longer *True*);
- the loop was executed at least twice; after the last execution of the loop, the condition was no longer true (replacing i_1* in the loop condition that proves to be *False* after the last iteration).

To deduce the safeness of the third check, we can use the following formula:

*{(i=0) && !(i<10) && **(i_1*=i)**} ||*

*{(i=0) && (i<10) && (i_1=i+1) && !(i_1<10) && **(i_1*=i_1)** } ||*

{(i=0) && (i<10) && (i*>i) && (i_1*=i*+1) && !(i_1*<10) }*

=> (i_1>=0)*

Chapter 6

Backward Analysis

The synthesis of context constraints and invariants is essentially a forward analysis that gathers information about how values are computed and propagated and how the conditions of if and while instructions are inherited. In contrast, the derivation of pre-condition for check elimination is inherently a backward problem. Here, the flow of information goes from callee to caller, with the goal of finding weakest possible pre-condition that ensures the safeness of the operation.

Deriving safety pre-conditions is done through a backward method, which considers each function in turn, starting from the lowest one in the calling hierarchy. Working backwards, each pre-condition that we derive from a callee would be converted into a check for its caller. In this way, we are able to derive the pre-condition for each check, including those that are nested arbitrarily deep inside function calls. The main steps of this algorithm are summarized below:

1. Determine each check to see if it is unsafe, totally redundant or partially redundant.
2. Derive a safety pre-condition for each partially redundant check.
3. To support inter-procedural propagation, convert each required pre-condition of a function into a check at its call site based on the parameter instantiation.

To help describe this method, consider the following functions from the binary search example:

```
/*ST getmid :: gA -> glo -> ghi -> (f,s)
        st ( gA>0 && f=(glo+ghi)/2 )*/
Pair getmid(int []gA, int glo, int ghi){
        int m = (glo+ghi)/2;
        Pair p;
        p.first=m;
        p.second=gA[m];       //L3,H3
```

```
        return p;
}
int look(int A[], int lo, int hi, int key){
        Pair p;
        /*INV ((lo*<=hi*) && (hi*<=hi) && (lo*>=lo)*/
        while (lo<=hi){
                p=getmid(A,lo,hi);      //L4,H4
                ...
}
/*ST bsearch :: bA -> bkey -> r
        st (bA>0) */
int bsearch (int bA[], int bkey){
        return look(bA,0,bA.length()-1,bkey);          //L5,H5
}
```

6.1. Deriving Pre-conditions

We classify each check as totally redundant, partially redundant or unsafe. For each check at L, we use ctx(L) => chk(L) to test for totally redundant checks, and ctx(L) $\wedge$ chk(L) |- False to test for unsafe checks. Otherwise, we attempt to derive a weakest pre-condition for the partially redundant checks.

A better approach to classify each check is to directly compute its weakest pre-condition using:

$pre(L) \equiv \neg\, ctx(L)\ V\ chk(L)$

This pre-condition should be simplified using the invariant context at function entry - *ctxSta*. If pre(L) simplifies to *True*, we classify the check as totally redundant. If pre(L) simplifies to *False* (or unknown due to the limitation of Presburger solver), we classify the check as unsafe. Otherwise, the check is said to be partially redundant.

Example: The two checks in *getmid* function are:

$chk(L3) = (m >= 0)$

$chk(H3) = (m < gA)$

$ctx(L3) = ctx(H3)$

$ctx(L3) = ctxSta(getmid) \wedge (2*m=glo+ghi)$

$ctxSta(getmid) = (gA >= 0)$

Considering the context in which these checks appear, we can obtain:

$pre(L3) \equiv \neg (2*m=glo+ghi \wedge gA >= 0) \vee (m >= 0)$

$\equiv (glo+ghi >= 0)$

$pre(H3) \equiv \neg (2*m=glo+ghi \wedge gA >= 0) \vee (m < gA)$

$\equiv (glo+ghi < 2*gA)$

The derivation of pre(L) is to a large extent dependent on ctx(L). A more informative ctx(L) could lead to a better pre(L).

Deriving pre-conditions for the elimination of checks from recursive functions is more challenging. A key problem is that the check may be executed repeatedly, and any derived pre-condition must ensure that the check is completely eliminated. One well-known technique for the elimination of checks from loop-based program is the loop limit substitution method. Depending on the direction of monotonicity (either incrementing or decrementing loop index), the check of either the first or last iteration of the loop is used as a condition for the elimination of all checks. However, this method is restricted to checks on monotonic parameters, whose limits can be precisely calculated.

The present algorithm can handle generic modifications to parameters. For better precision, our approach separates out the context of the initial recursive call from the context of the subsequent recursive calls. The latter context may use the invariant of recursive parameters from sized typing.

Example: In the function *look* the checks L4 and H4 (the pre-conditions of *getmid* propagated to the call site in *look* function) appear in a loop. For the low bound check, we will provide two separate conditions for the first and repeated iterations:

chkFst(L4) = (0 <= lo+hi)

chkRec(L4) = (0 <= lo+hi*)*

with their respective contexts:

ctxFst(L4) = (lo <=hi)

ctxRec(L4) = inv(loop)

inv(loop) = (lo<=hi* ∧ hi*<=hi ∧ lo*>=lo)*

We next derive the pre-conditions for the two checks separately, as follows:

preFst(L4) = ¬ *ctxFst(L4) V chkFst(L4)*

= *(hi < lo) V (0 <= lo+hi)*

preRec(L4) = ¬ *ctxRec(L4) V chkRec(L4)*

= ∀ *lo*,hi*.* ¬ *(lo*<=hi* ∧ hi*<=hi ∧ lo*>=lo) V (0 <= lo*+hi*)*

= *(0 <= lo)*

Note that *preRec* is naturally expressed in terms of the recursive variables. However, we must re-express each pre-condition in terms of the initial variables. Hence, universal quantification was used to remove the recursive variables *lo*, hi**.*

We can now combine the two pre-conditions together in order to obtain a single safety pre-condition for the recursive check, as shown here:

pre(L4) = preFst(L4) ∧ preRec(L4)

= *(hi < lo) V (0 <= lo+hi ∧ 0 <= lo)*

Through a similar derivation, the other check of H4, based on the pre-condition (lo+hi < 2*A) from *getmid*, yields:

pre(H4) = preFst(H4) ∧ preRec(H4)

$= (hi < lo)\ V\ (hi < A \wedge lo+hi < 2*A)$

The derived pre-conditions are very precise. Apart from ensuring that the checks are safe, it also captures a condition oh how the checks may be avoided (hi < lo).

6.2. Inter-procedural Propagation of Checks

To support inter-procedural propagation of checks, each pre-condition for a partially redundant check must first be converted into a new check at the call site. After that, the process of classifying the check and deriving its safety pre-condition is repeated. Consider two functions:

```
f(v1,...vn) {
        ...chk(L)...            //check L
}
g(w1,...wn) {
        ...f(v1',...vn')...//check C
}
```

Suppose that in *f*, we have managed to derive a non-trivial pre(L) that would make chk(L) redundant. Then, at each call site of *f*, such as *f(v1',...vn')* in the body of function *g*, we should convert the pre-condition of *f* into a new check at the call site. The pre-condition of *f* is converted into a check via a size parameter substitution, *subs(C)*.

$chk(C) = \exists X.pre(L) \wedge subs(C)$

$subs(C) = \Lambda_{i=1}^{n}(eq\,\tau_i\,\tau_i^{'})$

$where\ v_i :: \tau_i;\ v_i^{'} :: \tau_i^{'};\ X = \bigcup_{i=1}^{n} fv(\tau_i)$

Example: Consider the functions *getmid*, *look* and *bsearch* from the binary search example, and the pre-conditions determined in the previous section:.

$pre(L4) = preFst(L4) \wedge preRec(L4)$

$= (hi < lo) \vee (0 <= lo+hi \wedge 0 <= lo)$

$pre(H4) = (hi < lo) \vee (hi < A \wedge lo+hi < 2*A)$

With the pre-conditions determined for the function *look*, we can calculate the two checks for *bsearch*, and then transform them to pre-conditions.

$subs(L5) = subs(H5) = subs(C5)$

$= (A = bA \wedge lo = 0 \wedge hi = bA-1)$

$chk(L5) = \exists A,lo,hi \,.\, pre(L4) \wedge subs(L5)$

$= \exists A,lo,hi \,.\, (hi < lo) \vee (0 <= lo+hi \wedge 0 <= lo) \wedge (A = bA \wedge lo = 0 \wedge hi = bA-1)$

$= (bA <= 0 \vee 1 <= bA)$

$ctx(L5) = (bA > 0)$

$pre(L5) = \neg ctx(L5) \vee chk(L5)$

$= \neg (bA>0) \vee (bA <= 0 \vee 1 <= bA)$

$chk(H5) = \exists A,lo,hi \,.\, pre(H4) \wedge subs(H5)$

$= \exists A,lo,hi \,.\, (hi < lo) \vee (hi < A \wedge lo+hi < 2*A) \wedge (A = bA \wedge lo = 0 \wedge hi = bA-1)$

$= (bA <= 0 \vee 0 <= bA)$

$pre(H5) = \neg ctx(H5) \vee chk(H5)$

$= \neg (bA>0) \vee (bA <= 0 \vee 0 <= bA)$

Both pre-conditions, pre(L5) and pre(H5), depend on the dimension of the vector declared in the *main* function. These two formulas will be simplified to *True*, for a positive length of a vector.

Inter-procedural propagation of checks applies to recursive functions without exception.

Through this inter-procedural propagation, we have successfully determined that the recursive checks of *look* inside *bsearch* are totally redundant. Hence, all bound checks for *bsearch* can be completely eliminated. This is done by providing specialized versions of *look* and *getmid* (without array bound checks) that would be called from *bsearch*.

Chapter 7

FFI (Forward Function Interface): Calling C from Haskell

This project was developed using both Hugs and GHC (Glasgow Haskell Compiler). Hugs interpreter accelerates the cycle of development, but cannot be used for linking with a foreign (for example, C) library.
GHC provides a facility for linking with foreign libraries. To use this facility, one should include the next two options in the ghc command line:
-fglasgow-exts: permits user to specify functions that are imported from other external modules (possibly C++ libraries). An imported function is specified like this:

foreign import fname :: args -> IO args

The name "*fname*" can be used directly in the Haskell code, or it can be defined a "nickname" (different from the name of the function in the external module).
"*args*" should be of primitive types, or defined in the "Foreign" module. This implies the second option of ghc command line:
-package lang: includes "*Foreign*" and "*CForeign*" modules. These modules include other types (beside the primitive types) that can be marshaled to C. The type used as a parameter to (and from) the C library is *CString*, which represents a string terminated with the NULL character (the same as definition as a string in C language).
The same modules provide functions for marshalling and un-marshalling strings. The *newCString* function allocates memory for the C representation of a Haskell string and also performs the marshalling. Function *peekCString* marshals C strings to Haskell (it also de-allocates the memory used for C string).

newCString :: String -> IO CString

peekCString :: CString -> IO String

The function provided by the C library – *simplifyIt* – takes a string (representing a formula), and returns the simplified result calculated by Omega. Because the result of calling the C function is obtained using IO monads, the result is of type IO a.
The result of an IO computation can be converted from *IO a* to *a* type using *unsafePerformIO* function. In order to preserve *referential transparency*, the IO

computation should not perform side effects. This restriction cannot be verified by the compiler, and is left as an onus for the programmer. That is the reason why the function is called unsafe, in order to remind programmers that they must assure it can be executed in a purely functional manner.

unsafePerformIO :: IO a -> a

The function which takes a list of strings, and tries to simplify them (using omega calculator), passing back the result as another list of strings (after converting the IO result) can be defined in an elegant way using the following code:

```
map (\str -> unsafePerformIO (newCString >>= simplifyIt >>= peekCString))
ocStrings
```

Chapter 8

Omega Project

The Omega Library manipulates integer tuple relations and sets described using Presburger formulas.

Presburger formulas are logical formulas that can be built from affine constraints over integer variables. These constraints are linked with logical connectors *Not*, *And*, *Or* and the quantifiers *Exists*, *Forall*. Relations that are passed to Omega Library are transformed into disjunctive normal function and redundant constraints are removed. The resulted relations can contain all the points or none; these two particular cases are linked to *True* and *False*. Otherwise the simplified relation is prefixed by the variables that are used to describe it.

The syntax of Presburger formulas are as follows:

Name	Notation
And	*formula* & *formula* *formula* && *formula* *formula* and *formula*
Or	*formula* | *formula* *formula* || *formula* *formula* or *formula*
Not	! *formula* not *formula*
Exists	exists ($v_1, \ldots, v_n$: *formula*)
Forall	forall ($v_1, \ldots, v_n$: *formula*)
Parentheses	(*formula*)
Constraint	*constraint*

The operations that can be used on relations or sets are as follows:

Name	Syntax	Explanation
Union	$r = r_1$ union r_2	$x \to y \in r$ iff $x \to y \in r_1$ or
	$s = s_1$ union s_2	$x \in s$ iff $x \in s_1$ or $x \in s_1$
Intersection	$r = r_1$ intersection r_2	$x \to y \in r$ iff $x \to y \in r_1$ an
	$s = s_1$ intersection s_2	$x \in s$ iff $x \in s_1$ and $x \in s_1$
Difference	$r = r_1 - r_2$	$x \to y \in r$ iff $x \to y \in r_1$ an
	$s = s_1 - s_2$	$x \in s$ iff $x \in s_1$ and $x \notin s_1$
Complement	$r =$ complement r_1	$x \to y \in r$ iff $x \to y \notin r_1$
	$s =$ complement s_1	$x \in s$ iff $x \notin r_1$
Composition	$r = r_1$ compose r_2	$x \to y \in r$ iff $\exists y$ s.t. $x \to y$
Application	$s = r_1(s_2)$	$x \in s$ iff $\exists y$ s.t. $x \to y \in r_1$
	$r = r_1(r_2)$	Equivalent to r_1 compose r
Join	$r = r_1.r_2$	Equivalent to r_2 compose r
		$x \to y \in r$ iff $\exists y$ s.t. $x \to y$
Inverse	$r =$ inverse r_1	$x \to y \in r$ iff $y \to x \in r_1$
Domain	$s =$ domain r_1	$x \in s$ iff $\exists y$ s.t. $x \to y \in r_1$
Range	$s =$ range r_1	$y \in s$ iff $\exists x$ s.t. $x \to y \in r_1$
Restrict Domain	$r = r_1 <$ *tex2html*$_{\text{verb}_m}$*ark* $><$ *tex2html*$_{\text{verb}_m}$*ark* $> s_2$	$x \to y \in r$ iff $x \to y \in r_1$ an
Restrict Range	$r = r_1 / s_2$	$x \to y \in r$ iff $x \to y \in r_1$ an
Gist	$r =$ gist r_1 given r_2	Computes gist of relation r
Hull	$r =$ hull r_1	
Transitive Closure	$r = r_1+$	Least fixed point of $r^+ \equiv r$
	$r = r_1+$ within b	Least fixed point of r^+ fo
		within iteration space b
Cross-product	$r = r_1 * r_2$	$x \to y \in r$ iff $x \in r_1$ and $y \in$
Create superset	$r =$ supersetof r_1	r is inexact with lower bou
Create subset	$r =$ subsetof r_1	r is inexact with upper bou
Get an example	$r =$ example r_1	$r \subseteq r_1$ and all variables in r
Symbolic example	$r =$ sym_example r_1	$r \subseteq r_1$ and all non-symbolic
		valued

From the previous table, the transitive closure operation can be used to infer the invariants for a loop or recursive function. A disadvantage for this method is that it does not support generalization to give an approximate fixed point, if the least fixed point cannot be found. To overcome the limitation of Omega's transitive closure operation, there is the notion of generalized transitive closure with selective generalization, described in [4]. Basically, it introduces generalization of size relation based on selective grouping and hulling of its various disjuncts. While hulling aids in ensuring termination of fixed-point computation at the expense of accuracy, selective grouping and hulling help maintain accuracy of such computation.

As an example, let us consider the loop from the *look* function. First, we need to compute the constraint relating the initial variables (*lo* and *hi*) and the recursive variables (*lo** and *hi**). Conceptually, this constraint spells out the change for these variables after one *unfolding* of instructions from the loop. For this computation, we can use the same function as for context synthesis. The resulting constraint (which is a *relation* between the initial variables and the recursive variables) is:

U := [A,lo,hi] -> [A,lo*,hi*] : hi >= lo ∧ ∃(m.2m <= hi+lo ∧ hi+lo <= 1+2m ∧ ((lo*=lo) ∧ (hi*=m-1) V (lo*=m+1 ∧ hi*=hi)))*

The first part of the formula *(2m <= hi+lo ∧ hi+lo <= 1+2m)* results from the assignment *(m=(lo+hi)/2)* as explained before. The second part of the formula *((lo*=lo) ∧ (hi*=m-1) V (lo*=m+1 ∧ hi*=hi))* is a disjunction between the two alternatives of the *if* instruction.

Next, we perform inductive computation to infer the change for these recursive variables resulting from arbitrary number of execution of the loop. The result is the loop invariant. Polyhedra analysis (proposed by) is an abstract interpretation approach to find the input/output size relation through fixed-point computation over linear constraint. Both convex-hull operation (to eliminate multiple disjuncts) and widening operation (to generalize a constraint by dropping some conjuncts that cannot be subsumed by others in an ascending chain of constraints) are used as generalization techniques to ensure termination of the analysis. For recursive-call invariant computation, we modify this analysis by ignoring the degenerated case of a recursive definition from our computation. As an example, below is the trace of such computation for the function *look* with the aid of the Omega calculator.

 # First Approximation

U_1 := hull U;

U_1 = [a,lo,hi] -> [a,lo*,hi*] : a=a* ∧ lo <= hi, lo* ∧ hi* <= hi ∧ lo + 2hi* <= hi + 2lo* ∧ 2 + 2lo + 2hi* < hi + 3lo* ∧ hi + 2lo* <= 3 + lo + 2hi* ∧ hi + 4lo* <= 4 + 2lo + 3hi**

 # Second Approximation

U_2 := hull (U_1 union (U_1 compose U));

U_2 = [a,lo,hi] -> [a,lo*,hi*] : a=a* ∧ lo <= hi, lo* ∧ hi* <= hi ∧ hi + 4lo* <= 9 + lo + 4hi* ∧ lo + 2hi* < hi + 2lo* ∧ 2 + 2lo + 2hi* <= hi + 3lo**

 # Third Approximation

U_3 := hull (U_2 union (U_2 compose U));

U_3 := hull (U_2 union (U_2 compose U));

U_3 = [a,lo,hi] -> [a,lo*,hi*] : a=a* ∧ lo <= hi, lo* ∧ hi* <= hi ∧ hi + 8lo* <= 2lo + lo + 8hi* ∧ 2 + 2lo + 2hi* < hi + 3lo* ∧ lo + 2hi* <= hi + 2lo**

 # Apply Generalization by Widening

W_2 := widen (U_2,U_3);

W_2 = [a,lo,hi] -> [a,lo*,hi*] : a=a* ∧ lo <= hi, lo* ∧ hi* <= hi ∧ 2+2lo+2hi* <= hi+3lo* ∧ lo+2hi* < hi+2lo**

 # Is the result a fixed point?

(W_2 compose U) subset W_2;

True

Lines beginning with # are comments; lines ending with semicolon are commands for Omega.

hull U – computes the convex hull of U (viewed as a relation);

widen (U_2,U_3) – generalizes Y_2 to yield a constraint W_2, such that both U_2 and U_3 are instances of W_2;

union means disjunction;

compose combines two constraints by identifying (and eliminating) the target of the former with the source of the latter.

The second and third steps of the above trace are iterative computation of fixed-point computation. The last command checks if a fixed-point is reached.

Chapter 9

Conclusion and future work

Through a novel combination of both forward technique to compute contextual constraint, and backward method to derive weakest pre-conditions, ABCE algorithm represents a comprehensive method for handling both totally redundant and partially redundant checks. Both analysis methods are built on top of a Presburger constraint solver that has been shown to be both accurate and practically efficient [10]. Translation from imperative code to SSA form is done in the same phase as the forward analyze, and the code that is added (substitutions, and conversion from while loop) is not significant in size. However, duplication of formulas in the final condition induces some problems: after an assignment instruction, the left-hand side variable will be replaced with a possible big expression (the right-hand side part of assignment) in all its appearance places. Also, the pre-condition of an if instruction is duplicated in both branches, and loops being converted in conditionals will suffer from the same problem.
While the method proposed by ABCE authors for functional code is complete, our implementation needs annotation in order to handle loops and inter-procedural propagation. Annotations should be replaced from input code, as loop invariants and function's sized type can be inferred in a similar manner as described in chapter 8 (using fix-point computation).
Our test analyzer works on a small subset of Java language, which can be extended. An exception mechanism would introduce new problems related to the flow control. Hoisting checks, used initially to remove safe array accesses, modifies control flow of instructions. Complex data structures, particularly sub-typing, will extend the range and the expressiveness of programs that can be implemented with our test analyzer.
Lastly, we should extend the practical use of this analyzer with bound check specialization. Specializing each call site with respect to its context could be guided by derived pre-conditions and will remove effectively checks detected as safe. There is a cost-benefit tradeoff for checks amalgamating and bound check specialization that should be carefully investigated in order to come up with a practically useful strategy.

Chapter 10

References

[0] W. N. Chin, S. C. Khoo and Dana N. Xu. Deriving pre-conditions for array bound check elimination, Proceedings of the Second Symposium on Programs as Data Objects, pages 2-24, 2001.

[1] A. V. Aho, R.Sethi, and J.D. Ullman. Compilers, Principles, Techniques, and Tools, Addison-Wesley, 1986.

[2] A. W. Appel. Modern compiler implementation in Java. Cambridge University Press, 1998.

[3] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bound checks on demand. In ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 321-333, 2000.

[4] W. N. Chin and S. C. Khoo. Calculating sized types. In 2000 ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation, pages 62-72, Boston, Massachusetts, United States, January 2000.

[5] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In Symposium on Principles of Programming Languages, pages 84-96. ACM Press, 1978.

[6] Haskell 98 Report. http://www.haskell.org/onlinereport

[7] R. Gupta. A fresh look at optimizing array bound checking. In ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 272-282, New Your, June 1990.

[8] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters Program Language Systems*, pages 135-150, Mar-Dec 1994.

[9] P. Kolte and M. Wolfe. Elimination of redundant array sbuscript range checks. In ACM Conference on Programming Language Design and Implementation, pages 270-278. ACM Press, June 1995.

[10] W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. Communications of ACM, pages 102-114, 1992.

[11] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 182-195. ACM Press, June 2000.
[12] N. Suzuki and K. Ishihata. Implementation of array bound checker. In ACM Principles of Programming Languages, pages 132-143, 1977.
[13] Z. Xu, B. P. Miller, and T. Reps. Safety checking of machine code. In ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 70-82. ACM Press, June 2000.

Chapter 11

Appendixes

11.1. Examples

1. The first example illustrates forward analysis through a simple function. This function contains a variable that is re-assigned. The substitutions made – part of SSA form - are indicated as a comment.

```
int elem (int []A){
        int i=0;
        int x;
        i=i+2;          //i1=i+2
        x=A[i];         //chk(i1)
        return x;
}
```

The following pre-conditions are calculated and passed to Omega calculator (the answer is written in bold):

("elem#sub#lo",forall(i1,i:!(1=1 && i=0 && i1=i+2) || i1>=0),Safe)
-> {TRUE}

("elem#sub#hi",forall(i1,i:!(1=1 && i=0 && i1=i+2) || i1< A),NotSure)
-> {[A]: 3 <= A}

2. Here is an example showing how renaming is done for a conditional instruction. Both branches – then and else - are merged after it. The substitutions made – SSA form – are indicated as comments.

```
int elem (int []A){
        int i=0;int j=3;int k=1;
        int x=0;
        i=i+2;          //i1=i+2
        if (x==2) {
                i=i+3;  //i2=i1+3
                i=i+4;  //i3=i2+4
                k=k+1;//k1=k+1
        } else {
                j=i+4;  //j1=i1+4
                i=i+6;  //i2=i1+6
        }
        x=A[i];         //chk(i3)
        return x;
}
```

The following pre-conditions are calculated and passed to Omega calculator (the answer is written in bold):

("elem#sub#lo",forall(k1,i3,i1,k,j,i,i2,j1:!(x=2 && 1=1 && i=0 && j=3 && k=1 && x=0 && i1=i+2 && i2=i1+3
&& i3=i2+4 && k1=k+1 && j1=j && 1=1 ||
!(x=2) && 1=1 && i=0 && j=3 && k=1 && i1=i+2 && j1=i1+4 && i2=i1+6 && i3=i2 && k1=k && 1=1) || i3>=0), Safe)
-> {TRUE}

("elem#sub#hi",forall(k1,i3,i1,k,j,i,i2,j1:!(x=2 && 1=1 && i=0 && j=3 && k=1 && x=0 && i1=i+2 && i2=i1+3
&& i3=i2+4 && k1=k+1 && j1=j && 1=1 ||
!(x=2) && 1=1 && i=0 && j=3 && k=1 && i1=i+2 && j1=i1+4 && i2=i1+6 && i3=i2 && k1=k && 1=1) || i3 < A), NotSure)
-> {[A]: 9 <= A}

3. Here is an example containing a while loop. This illustrates how renaming and merging are done for a loop.

```
int elem (int []A){
        int i=0;int j=3;
        int x=0;
        i=i+2;                          //i1=i+2
        /*INV (i*<10 && i*>i) */
        while (i<10){                   //i1 -> RECi
                i=i+1;                  //i2=i1+1               //RECi1=RECi+1
                j=A[i];
        }
        x=A[i];                         //chk(RECi2)
        return x;
}
```

("elem#sub#lo",forall(i2,i1,j,i:!(i1<10 && 1=1 && i=0 && j=3 && i1=i+2 && i2=i1+1) || i2>=0) &&
forall(RECi1,RECi,RECj,i1,j,i:!(RECi1<10 && RECi1>i1 && 1=1 && i=0 && j=3 && i1=i+2 && RECi1=RECi+1)
|| RECi1>=0), Safe)
-> {TRUE}

("elem#sub#hi",forall(i2,i1,j,i:!(i1<10 && 1=1 && i=0 && j=3 && i1=i+2 && i2=i1+1) || i2>=0) &&
forall(RECi1,RECi,RECj,i1,j,i:!(RECi1<10 && RECi1>i1 && 1=1 && i=0 && j=3 && i1=i+2 && RECi1=RECi+1)
|| RECi1 < A), NotSure)
-> {[A]: 11 <= A}

4. The following example shows propagation of pre-conditions from *access* function to *main* function. *access* function contains a conditional instruction.

```
class Test{
	/*ST access :: x -> y -> res st x>y*/
	int access (int []A,int x,int y){
		int i=0;int j=3;
		int []B=new int[10];
		j=A[i];				//j1
		if (j==2) {
			i=i+3;			//i1
		} else {
			i=i+4;			//i1
			j=i+1;			/j2
		}
		i=B[j+i];			//chk(j2,i1)
		return i;
	}

	int main(){
		int r;
		int AA[]=new int[4];
		r=access (AA,3,4);
	}
```

};

("access#sub#lo",forall(B,j,i:!(1=1 && i=0 && j=3 && B=10) || i>=0),Safe)
-> {TRUE}

("access#sub#hi",forall(B,j,i:!(1=1 && i=0 && j=3 && B=10) || i<A),NotSure)
-> {[A]: 1<=A}

("access#sub#lo",forall(j1,B,j,i,j2,i1:!(j1=2 && 1=1 && i=0 && j=3 && B=10 &&
i1=i+3 && j2=j1 && 1=1
|| !(j1=2) && 1=1 && i=0 && j=3 && B=10 && i1=i+
4 && j2=i1+1 && 1=1) || j2+i1>=0),Safe)
-> {[A,x,y]}

("access#sub#hi",forall(j1,B,j,i,j2,i1:!(j1=2 && 1=1 && i=
0 && j=3 && B=10 && i1=i+3 && j2=j1 && 1=1 || !(j
1=2) && 1=1 && i=0 && j=3 && B=10 && i1=i+4 && j2
=i1+1 && 1=1) || j2+i1<B),Safe)
-> {[A,x,y]}

1 pre-condition propagated to *main* function
substitutions: A -> AA, x -> 3, y -> 4
("main#access#sub#hi",forall(AA:!(1=1 && AA=4)|| exists(A,x,y:A=AA && x=3 &&
y=4 && 1=1 && fora
ll(B,j,i:!(1=1 && i=0 && j=3 && B=10) || i<A))),Safe)
-> {TRUE}

5. The following example shows propagation of pre-conditions from *access* function to *main* function. *access* function contains a loop instruction.

```
class Test{
	/*ST access :: x -> y -> res st x>y*/
	int access (int []A,int x,int y){
		int i=0;int j=3;
		int []B=new int[10];
		/*INV (i*<10 && i*>i) */
		while (i<10){
			i=i+1;
			j=A[i];
		}
		i=B[j+i];
		return i;
	}

	int main(){
		int r;
		int AA[]=new int[4];
		r=access (AA,3,4);
	}
};
```

("access#sub#lo",forall(i1,B,j,i:!(i<10 && 1=1 && i=0 && j=3 && B=10 && i1=i+1) || i1>=0) && forall(RECi1,RECi,RECj,B,j,i:!(RECi<10 && RECi>i && 1=1 && i=0 && j=3 && B=10 && RECi1=RECi+1) || RECi1>=0),Safe)
-> {TRUE}

("access#sub#hi",forall(i1,B,j,i:!(i<10 && 1=1 && i=0 && j=3 && B=10 && i1=i+1) || i1<A)
&& forall(RECi1,RECi,RECj,B,j,i:!(RECi<10 && RECi>i && 1=1 && i=0 && j=3 &&

B=10 && RECi1=RECi+1) || RECi1<A),NotSure)

-> {[A,x,y]: 11<=A}

("access#sub#lo",forall(j1,i1,B,j,i,RECj1,RECi1,RECi,RECj:!(!(i<10) && 1=1 && i=
0 && j=3 && B=10 && RECi1=i && RECj1=j && 1=1 || !(i1<10) && i<10 && 1=1
&& i=0 && j=3 && B=10 &&
i1=i+1 && RECi1=i1 && RECj1=j1 && 1=1 || !(RECi1<10) && RECi<10 && RECi>i
&& 1=1 && i=0 && j=3 && B=10 && RECi1=RECi+1) || RECj1+RECi1>=0),Unsafe)

-> {FALSE}

-> if I change (i=0) to (i=11) **-> {TRUE}**

("access#sub#hi",forall(j1,i1,B,j,i,RECj1,RECi1,RECi,RECj:!(!(i<10) && 1=1 &&
i=0 && j=3 && B=10 && RECi1=i && RECj1=j && 1=1 || !(i1<10) && i<10 && 1=1
&& i=0 && j=3 && B=10 && i1=i+1 && RECi1=i1 &
& RECj1=j1 && 1=1 || !(RECi1<10) && RECi<10 && RECi>i && 1=1 && i=0 &&
j=3
&& B=10 && RECi1=RECi+1) || RECj1+RECi1<B),Unsafe)

-> {FALSE}

1 pre-condition propagated to *main* function:

substitutions: A -> AA, x -> 3, y -> 4

("main#access#sub#hi",forall(AA:!(1=1 && AA=4) || exists(A,x,y:A=AA && x=3 &&
y=4 && 1=1 && forall(i1,B,j,i:!(i<10 && 1=1 && i=0 && j=3 && B=10 && i1=i+1)
|| i1<A) && forall(RECi1,RECi,RECj,B,j,i:!(RECi<10 && RECi>i && 1=1 && i=0 &&
j=3 && B=10 && RECi1=RECi+1) || RECi1<A))),Unsafe)

-> {FALSE}

6. This is the complete binary search example:

class BinSearch {

```
/*ST getmid :: AA -> getmid_LO -> getmid_HI -> (f,s)
        st (AA>0)*/
Pair getmid(int []AA, int getmid_LO, int getmid_HI){
        int getmid_M=(getmid_LO+getmid_HI)/2;
        Pair getmid_P;
        getmid_P.first=getmid_M;
        getmid_P.second=AA[getmid_M];
        return getmid_P;
}
/*ST cmp :: x -> y -> r st
        ((x<y && r==0-1) || (x>y && r==1)) || (x==y && r==0) */
int cmp (int x, int y){
        if (x<y) return 0-1;
        else
                if (x==y) return 0;
                else return 1;
}
/*ST look :: a -> lo -> hi -> key -> r
        st a>0 && ((lo>hi && r==0-1) || (lo<=hi))*/
int look(int A[], int lo, int hi, int key){
        Pair p; int t; int m; int x;int v;
        /*INV ((lo<=hi) && (lo*<=hi*)) && ((hi*<=hi) && (lo*>=lo))*/
        while (lo<=hi){
                p=getmid(A,lo,hi);
                m=p.first;
                x=p.second;
                t=cmp(key,x);
                if (t==0) return m;
                else
                        if (t<0) hi=m-1;
                        else lo=m+1;
```

```
                }
                return 0-1;
        }
        /*ST bsearch :: a -> key -> length -> r
                st a>0 */
        int bsearch (int bA[], int bkey, int blength){
                int v=blength;//A.length()
                return look(bA,0,v-1,bkey);
        }
        int main(){
                int []A = new int[3];
                int length = 3;
                int x=3;
                return bsearch(A,x,length);
        }

};
class Pair {
        int first;
        int second;
};
```

("getmid#sub#lo",forall(getmid_PDOTfirst,getmid_M:!(1=1 &&
exists(ee:ee*2<=getmid_LO+getmid_HI && ee*2+1>=getmid_LO+getmid_HI &&
getmid_M=ee) && getmid_PDOTfirst=getmid_M) || getmid_M>=0),NotSure)
-> {[AA,getmid_LO,getmid_HI]: 0<=getmid_LO+getmid_HI}

("getmid#sub#hi",forall(getmid_PDOTfirst,getmid_M:!(1=1 && exists(
ee:ee*2<=getmid_LO+getmid_HI && ee*2+1>=getmid_LO+getmid_HI &&
getmid_M=ee)
&& getmid_PDOTfirst=getmid_M) || getmid_M<AA),NotSure)

-> {[AA,getmid_LO,getmid_HI]: getmid_LO+getmid_HI<2AA}

2 pre-conditions propagated from *getmid* to *look* function.
substitutions: AA -> A, getmid_LO -> lo, getmid_HI -> hi
("look#getmid#sub#lo",!(lo<=hi && 1=1) || exists
(AA,getmid_LO,getmid_HI:AA=A && getmid_LO=lo && getmid_HI=hi && 1=1 &&
forall(getmid_PDOTfirst,getmid_M:!(1=1 && exists(ee:ee*2<=getmid_LO+getmid_HI
&& ee*2+1>=getmid_LO+getmid_HI && getmid_M=ee) &&
getmid_PDOTfirst=getmid_M)
|| getmid_M>=0)) && forall(RECpDOTfirst,RECpDOTsecond,
RECm,RECx,RECt,REChi,REClo:!(lo<=hi && REClo<=REChi
&& REChi<=hi && REClo>=lo && 1=1) || exists(AA,getmid_LO,getmid_HI:
AA=A && getmid_LO=REClo && getmid_HI=REChi && 1=1 &&
forall(getmid_PDOTfirst,getmid_M:!(1=1 && exists
(ee:ee*2<=getmid_LO+getmid_HI && ee*2+1>=getmid_LO+getmid_HI &&
getmid_M=ee
) && getmid_PDOTfirst=getmid_M) || getmid_M>=0))),NotSure)
-> {[A,lo,hi,key]: hi< lo} union {[A,lo,hi,key]: 0 <= lo+hi && 0 <=lo}

("look#getmid#sub#hi",!(lo<=hi && 1=1) || exists(AA,getmid_LO,
getmid_HI:AA=A && getmid_LO=lo && getmid_HI=hi && 1=1 && forall(
getmid_PDOTfirst,getmid_M:!(1=1 && exists(ee:ee*2<=getmid_LO+getmid_HI
&& ee*2+1>=getmid_LO+getmid_HI && getmid_M=ee) &&
getmid_PDOTfirst=getmid_M)
|| getmid_M<AA)) && forall(RECpDOTfirst,RECpDOTsecond,
RECm,RECx,RECt,REChi,REClo:!(lo<=hi &&
REClo<=REChi && REChi<=hi && REClo>=lo && 1=1) || exists(AA,
getmid_LO,getmid_HI:AA=A && getmid_LO=REClo && getmid_HI=REChi && 1=1
&& forall(getmid_PDOTfirst,getmid_M:!(1=1 &&

exists(ee:ee*2<=getmid_LO+getmid_HI && ee*2+1>=getmid_LO+getmid_HI &&
getmid_M=ee)
&& getmid_PDOTfirst=getmid_M) || getmid_M<AA))),NotSure)
-> {[A,lo,hi,key]: hi<lo} union {[A,lo,hi,key]: hi<A && lo+hi<2A}

2 pre-conditions propagated from *look* to *bsearch* function.
substitutions: A -> bA, lo -> 0, hi -> v-1, key -> bkey
("bsearch#look#getmid#sub#lo,!(v=length) || exists(A,lo,hi,key: A=bA && lo=0 &&
hi=v-1 && key=bkey
&& (hi < lo || (0 <= lo+hi && 0 <= lo)),Safe)
-> {TRUE}

("bsearch#look#getmid#sub#hi, !(v=length) || exists(A,lo,hi,key: A=bA && lo=0 &&
hi=v-1 && key=bkey
&& (hi < lo || (hi < A && lo+hi < 2*A)),NotSure)
-> {[bA,bkey,blength]: blength < 0} union {[bA,bkey,blength]: 1 <= blength <= bA}

1 pre-condition propagated from *bsearch* to *main* function.
substitutions: bA -> A, bkey -> x, blength -> length
("main#bsearch#look#getmid#sub#hi, !(A=3 && length=3 && x=3) ||
exists(bA,blength,bkey: bA=A && blength=length && bkey=x &&
(blength<0 || 1<= blength && blength <=bA)),NotSure)
-> {TRUE}