Modular Inference for Array Checks Optimization

Dana N. Xu, Corneliu Popeea, Siau-Cheng Khoo, and Wei-Ngan Chin
Department of Computer Science
School of Computing, National University of Singapore
{xun,popeeaco,khoosc,chinwn}@comp.nus.edu.sg

ABSTRACT

An array check can be safely eliminated if it is valid under the context in which it is invoked. In this paper, we propose a modular inference mechanism that is based on an advanced form of typing, called sized type. We apply this to a core imperative language to discover program contexts for which array checks may be safely eliminated. Our inference is *modular* as it is able to process each method independently, and yet exploit the different contexts of its multiple callers. We achieve this by turning each partially-safe check into a precondition that is propagated up its callers' contexts for further analysis. This approach provides for a highly accurate interprocedural optimization. The results of our modular inference can be used to guide a range of program specializations that trade-off code space for more aggressive optimization. We have implemented a prototype inference system and have also proven its soundness. Initial experiments suggest that such an inference system is viable.

1. INTRODUCTION

Array bound check optimization has been extensively investigated over the last three decades [21, 7, 22, 9], with renewed interests as recently as [2, 24, 15]. Most array optimization techniques (e.g [7, 23, 21]) are focused on the elimination of totally redundant checks. To achieve this, whole program analysis is carried out to propagate analysis information (e.g. availability) to each program point. Even for techniques that handle partially redundant checks, such as partial redundancy elimination (PRE)[3], the focus has been on either moving these checks or restructuring the control flows, but without exploiting path-sensitivity, recursion or interprocedural relational analysis. This weakness goes against the needs of modularity.

In this paper, we propose a new inference mechanism, based on a restricted form of dependent type, called *sized type*[12, 5], that is crafted to maximise the elimination of array-related checks. Our inference is *modular* as it is able to process each method independently, and yet exploits the different contexts of its multiple callers. As we shall see, this provides for a very precise interprocedural optimization.

A key feature of our approach is a three-way classification of checks before optimization. Given a method declaration with a set of parameters V, and a set of checks C. Our approach will classify each check ($c \in C$) that occurs at a given program context *ctx*, as follows:

- *safe* if c is redundant under its context. This holds if the following is *valid*:
$$(ctx \Rightarrow c)$$
- *partially-safe* if c may become redundant under an extra condition. This holds if there exists a *satisfiable pre* (based on only V), such that the following is *valid*.
$$(pre \wedge ctx \Rightarrow c)$$
- *unsafe*, otherwise.

While the above description is general and may be applicable to any kind of checks, we shall be focusing exclusively on array-related checks. Consider an example below.

```
Float foo(Float[] a, Int j, Int n)
   { Float v=0.0; Int i=j+1;
      if (0<i<=n) then v=(ℓ1:a[i])
      else ();
      Int m=len(a); v+(ℓ2:a[m])
    }
```

The set of parameter variables at method declaration is {a, j, n} where a is an array with dimension 0 to len(a)-1. This method contains two array accesses at locations ℓ_1 and ℓ_2. The contexts at these sites may be affected by conditionals, imperative updates and by prior calls. In particular, the three contexts for this example, including one at method entry ℓ_0, are:

$$\begin{array}{lcl} \texttt{ctx}(\ell_0) & = & \texttt{len(a)>0} \\ \texttt{ctx}(\ell_1) & = & \texttt{ctx}(\ell_0) \wedge \texttt{i=j+1} \wedge \texttt{(0<i<=n)} \\ \texttt{ctx}(\ell_2) & = & \texttt{ctx}(\ell_0) \wedge \texttt{i=j+1} \wedge \texttt{m=len(a)} \end{array}$$

Based on the definition of safe checks, we can establish that the low-bound checks (at ℓ_1 and ℓ_2) are safe, since:

$$\texttt{ctx}(\ell_1)\Rightarrow\texttt{(i>=0)} \qquad \texttt{ctx}(\ell_2)\Rightarrow\texttt{(m>=0)}$$

For the high-bound checks (denoted by ℓ_1.H and ℓ_2.H) we can attempt to derive preconditions through universal quantification of the local variables, as follows:

$$
\begin{aligned}
\texttt{pre}(\ell_1.\texttt{H}) &= \forall \texttt{i}\cdot (\texttt{ctx}(\ell_1) \Rightarrow \texttt{i<len(a)}) \\
&= \forall \texttt{i}\cdot \texttt{len(a)>0} \wedge \texttt{i=j+1} \wedge \texttt{(0<i<=n)} \Rightarrow \texttt{i<len(a)}) \\
&= \texttt{len(a)<=0} \vee (\texttt{j<=len(a)-2} \wedge \texttt{1<=a}) \\
&\quad\quad \vee (\texttt{1<=len(a)<=j+1} \wedge \texttt{n<=j}) \\
&\equiv_s \texttt{j<=len(a)-2} \\
\texttt{pre}(\ell_2.\texttt{H}) &= \forall \texttt{i,m}\cdot (\texttt{ctx}(\ell_4) \Rightarrow \texttt{m<len(a)}) \\
&= \forall \texttt{i,m}\cdot (\texttt{len(a)>0} \wedge \texttt{i=j+1} \wedge \texttt{m=len(a)} \Rightarrow \texttt{m<len(a)}) \\
&= \texttt{len(a)<=0} \\
&\equiv_s \texttt{false}
\end{aligned}
$$

The last step (denoted by $\equiv_s$) involves a simplification using an invariant of the check's location at method entry. For the locations, ℓ_1 and ℓ_2, the invariants are $(\exists \texttt{i} \cdot \texttt{ctx}(\ell_1))$ and $(\exists \texttt{i}, \texttt{m} \cdot \texttt{ctx}(\ell_2))$, respectively. This simplification is used to eliminate avoidance[1] and invariant conditions from our preconditions. It is essential for deriving smaller preconditions for propagation. (In the Omega[19] constraint-solver which we are using, this is achieved with an operator called `gist`.) From here, we can see that ℓ_2.H is *unsafe*, while ℓ_1.H is *partially-safe*. Unsafe checks can be marked for runtime testing in support of soft typing, while the precondition of partially-safe checks can be propagated to the method's callers for further analyses.

In this paper, we shall formalise an inference mechanism based on such a three-way classification of array checks. We shall show how preconditions (of partially-safe checks) could be propagated up the function calling hierarchy to achieve inter-procedural optimization. We shall achieve this under the framework of an advanced typing system with support for imperative tracking of size properties across procedures. Our main contributions are:

- **Modular Inference :** We propose a new framework which combines *type inference* and *program specialization* to achieve a modular inference mechanism. Our framework allows partially-safe checks to be propagated as preconditions and unsafe checks to be identified for each method. This enables the checks to be selectively satisfied at different callers' sites, with the support of program specialization.
- **Precision :** We provide a highly precise interprocedural analysis to support the elimination of more checks. We achieve this through the propagation of postconditions in the presence of imperative updates. For recursive methods, we apply fix-point analysis to obtain both the *postconditions* and *invariants* of their recursive calls.
- **Soft Typing :** We propose an integrated approach for array check elimination with support for both *strong typing* and *soft typing*[4]. Our approach allows language designers to specify which checks must be eliminated at compile-time, and which checks may be deferred to run-time.
- **More Optimization :** We generalise the technique of check optimization to handle other kinds of array checks, including *dimensionality* of arrays and *positivity* of its sizes. Our approach can also handle *array indirections* whose elements are being used as indexes. The optimization for this class of bound checks has been mostly ignored in past work.

[1]These are conditions from conditionals that avoid, rather than eliminate the checks. An example is `n<=j`.

P	$\in$	**Program**
	::=	md^*
md	$\in$	**MDef** (Method Definitions)
	::=	$\tau\ m\ (\tau_1\ v_1, \ldots, \tau_k\ v_k)\ \{e\}$
e	$\in$	**Exp** (Expressions)
	::=	$c \mid v \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \tau\ v{=}e_1\ ;\ e_2$ $\mid v = e \mid m(v_1, \ldots, v_k)$
τ	$\in$	**Type**
	::=	$\texttt{t} \mid \texttt{t}[\texttt{Int}, \ldots, \texttt{Int}]$
t	$\in$	**PrimType**
	::=	`Void` | `Int` | `Bool` | `Float`
v	$\in$	**Var** (Program Variables)
c	$\in$	**Cst** (Constant of Primitive Types)
m	$\in$	**MName** (User-defined & Primitive Methods)

Figure 1: The Syntax of Imp

- **Specialization :** The results of our modular inference can be utilized by a range of program specializers that trade-off code space for efficiency. We formalise two versions, *monovariant* and *polyvariant* specializations.
- **Soundness :** A key benefit of type-based approach is its formality. We highlight the correctness of modular inference with a theorem stating that it can only lead to well-typed programs without illegal array accesses.
- **Implementation :** To confirm the viability of our approach, we have built a prototype type inference system together with a monovariant specializer. Preliminary experiments indicate that modular inference is both fast (in comparison to checking) and highly precise (in its elimination of checks).

Sec 2 introduces three core imperative languages (source, intermediate and target) and describes key principles used by modular inference. Sec 3 presents a type inference system to help classify checks at method boundary, together with a soundness theorem. Sec 4 describes how postconditions and invariants of recursive methods are inferred, and how preconditions are derived for recursive checks. Sec 5 shows how the results of modular inference may be utilized by either monovariant or polyvariant specialization. Sec 6 describes our prototype implementation together with initial experimental results. Sec 7 compares with related works, followed by concluding remarks. The Appendix contains a discussion on how modular inference could be further extended to handle array indirections and also `while`-loops, together with details of specializations, dynamic semantics and soundness.

2. KEY PRINCIPLES

In this section, we propose a simple imperative language which we will use to formalise our modular inference mechanism. Though an imperative language has been chosen, its design is largely inspired by the expression-oriented world of functional programming. Moreover, we have adopted a minimalist and disciplined approach to language design, which allows us to focus on key principles and techniques behind modular inference. We expect our proposal to be adaptable for use in other programming paradigms.

2.1 Core Languages

The main goal of our inference process is to transform a program from an imperative language, called Imp, into an equivalent well-typed program in a target language, called Imp_T. The source and target languages are defined in Figs 1 and 3 respectively. Well-typed programs in our target language are expected to be free of array-related errors, other than those that are explicitly flagged through a special primitive, called `error`. To achieve this goal we annotate each Imp_T function with sized precondition and postcondition, that can be verified by a type checker, to guarantee the absence of array-related errors. An example of such a target program (taken from Sec 1) is shown in Fig 2(b), together with its prior source program in Fig 2(a). Note that its postcondition is `false` (corresponding to dead code) due to failure of the last `sub(a,m)` operation. The stated precondition $j{\leq}s{-}2$ is due to a partially-safe check from `sub(a,i)`.

```
Float foo(Float[Int] a, Int j, Int n)
   { Float v=0.0; Int i=j+1;
      if (0<i<=n) then v=sub(a,i)
      else ();
      Int m=len(a,1); v+sub(a,m)
   }
```

(a) Imp Source Program

```
Float foo(Float[Int^s] a, Int^j j, Int^n n)
 where false, (j<=s-2)
   { Float v=0.0; Int i=j+1;
      if (0<i<=n) then v=sub(a,i)
      else ();
      Int m=len(a,1); v+(if m<len(a,1) then
          sub(a,m) else error)
   }
```

(b) Imp_T Target Program

Figure 2: Source and Target Programs

Imp is a non-trivial imperative language with support for assignments, conditionals, local declarations, method calls, and multidimensional arrays. The Imp language is deliberately kept simple to make easier the formulation of static and dynamic semantics. This core language can be extended with abbreviations to make programming more convenient. Some syntactic conveniences which we may use include:

- Multi-declarations block is an abbreviation for nested expression blocks.
  $(d_1\ ;\cdots;\ d_n\ ;\ e) \equiv (d_1\ ;(\cdots(d_n\ ;\ e)\cdots))$
- Sequence is a special case of local block, where e_1 is of `void` type; as shown below.
  $(e_1\ ;\ e_2) \equiv (\texttt{void}\ v = e_1\ ;\ e_2)$
- Expressions may be used where variables are expected, aided by the following equivalences.
  $m(e_1,\ldots,e_n) \equiv$
  $(t_1\ v_1 = e_1\ ;\cdots;\ t_n\ v_n = e_n\ ;\ m(v_1,\ldots,v_n))$
  $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \equiv$
  $(\texttt{bool}\ v = e_1\ ;\ \texttt{if}\ v\ \texttt{then}\ e_2\ \texttt{else}\ e_3)$
- Loops are syntactic abbreviations for tail-recursive functions. With inter-convertibility (see Sec A.2), we use the former for execution and the latter for modular inference.

Arrays are created and manipulated through a set of primitive methods. A new `k`-dimensional array of sizes $[\mathtt{s}_1,\ldots,\mathtt{s}_\mathtt{k}]$ is created using $\mathtt{newArr}(\mathtt{s}_1,\ldots,\mathtt{s}_\mathtt{k},\mathtt{v})$, with all its elements initialised with the value `v`. Each dimension of size `s` covers indexes from 0 to `s`-1. The operation `len(a,i)` returns the size of array `a` at the `i`-th dimension. The operation $\mathtt{sub}(\mathtt{a},\mathtt{i}_1,\ldots,\mathtt{i}_\mathtt{k})$ returns an element of array `a` at index position $(\mathtt{i}_1,\ldots,\mathtt{i}_\mathtt{k})$. Lastly, $\mathtt{assign}(\mathtt{a},\mathtt{i}_1,\ldots,\mathtt{i}_\mathtt{k},\mathtt{v})$ updates the array at index $(\mathtt{i}_1,\ldots,\mathtt{i}_\mathtt{k})$ with the value `v`.

P	$\in$	**Program**	
	$::=$	md^*	
md	$\in$	**MDef**	(Method Definitions)
	$::=$	$\tau\ m\ (\tau_1\ v_1,\ldots,\tau_k\ v_k)\ \texttt{where}\ \Delta,\phi_{pre}\{e\}$	
P_m	$\in$	**PrimLib**	(Primitive Library)
	$::=$	p^*	
p	$\in$	**PDef**	(Primitive Declarations)
	$::=$	$\tau\ m\ (\tau_1\ v_1,\ldots,\tau_k\ v_k)\ \texttt{where}\ \Delta,\phi_{pre}$	
e	$\in$	**Exp**	(Expressions)
	$::=$	$c \mid v \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \tau\ v = e_1\ ;\ e_2$ $\mid v = e \mid m(v_1,\ldots,v_k) \mid \texttt{error}$	
τ	$\in$	**AnnType**	
	$::=$	$\mathtt{t} \mid \mathtt{t}[\mathtt{Int}^{s_1},\ldots,\mathtt{Int}^{s_k}]$	
$\mathtt{t}$	$\in$	**PrimAnnType**	
	$::=$	$\mathtt{Void} \mid \mathtt{Int}^n \mid \mathtt{Bool}^n \mid \mathtt{Float}$	
s,n	$\in$	**SizeVar**	(Size Variables)
ϕ,Δ	$\in$	**F**	(Presburger Formulae)
	$::=$	$b \mid \phi_1\wedge\phi_2 \mid \phi_1\vee\phi_2 \mid \exists n\cdot\phi \mid \forall n\cdot\phi \mid \neg\phi$	
b	$\in$	**BExp**	(Boolean Expression)
	$::=$	$\texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \neq a_2$ $\mid a_1 < a_2 \mid a_1 \leq a_2$	
a	$\in$	**AExp**	(Arithmetic Expression)
	$::=$	$c \mid n \mid c * a \mid a_1 + a_2 \mid -a$	
		where n denotes a size variable	

Figure 3: Imp_T Language with Size Constraints

The target language Imp_T is extended with a form of dependent type system[23, 5] with support for relational size analysis. We achieve this through an annotated type (or indexed type) system, called *sized type*, where annotated variables are used to denote the size properties of the underlying type. For example, the annotated type $\mathtt{Int}^a$ denotes an integer type whose value is captured by the size variable a, and similarly for $\mathtt{Bool}^b$ whose size variable b

md	$\in$	**MDef**	(Method Definitions)
	$::=$	$\tau\ m\ (\tau_1\ v_1,\ldots,\tau_k\ v_k)\ \texttt{where}\ \Delta,\Phi,\Upsilon\ \{e\}$	
p	$\in$	**PDef**	(Primitive Declarations)
	$::=$	$\tau\ m\ (\tau_1\ v_1,\ldots,\tau_k\ v_k)\ \texttt{where}\ \Delta,\Phi,\mathtt{C}$	
e	$\in$	**Exp**	(Expressions)
	$::=$	$c \mid v \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \tau\ v = e_1\ ;\ e_2$ $\mid v = e \mid \ell : m\ (v_1,\ldots,v_k)$	
Φ	$\in$	$\mathcal{P}(\mathbf{Label}^+,\mathbf{F})$	(Labelled Preconditions)
Υ	$\in$	$\mathcal{P}(\mathbf{Label}^+)$	(Unsafe Checks)
$\mathtt{C}$	$\in$	$\mathcal{P}(\mathbf{Label}^+,\mathbf{Exp})$	(Labelled Runtime Checks)
ℓ	$\in$	**Label**	
ϕ,Δ	$\in$	**F**	(Presburger Formulae)
	$::=$	$\cdots \mid q\langle n^*\rangle$	
$\mathcal{Q}$	$\in$	**CLib**	(Constraint Abstraction Set)
	$::=$	$\{(q\langle n^*\rangle = \phi)^*\}$	

Figure 4: Imp_I Intermediate Language

can be 0 or 1, depending on whether its boolean value is `false` or `true`. Also, each `k`-dimensional array is denoted by $t[\texttt{Int}^{s_1}, \ldots, \texttt{Int}^{s_k}]$, with $\{s_1, \ldots, s_k\}$ to capture the sizes of the its dimensions, while t denotes the type of its elements.

The main use of this type system is to express size relations amongst its variables. Applied at method boundaries, it can give accurate interprocedural analysis. To illustrate, consider the types of two primitive operations below.

```
Int^r add (Int^a v1, Int^b v2)   where r=a+b
Bool^r lessThan (Int^a v1, Int^b v2)
   where (a<b∧r=1)∨(a≥b∧r=0)
```

Take note that the constraint after the `where` clause expresses a relationship between the sizes of inputs and output. For size constraint, we restrict to Presburger form, as decidable (and practical) constraint solvers exist, e.g. [19].

While the final goal of inference mechanism is a set of well-typed $\textsc{Imp}_T$ programs, our modular inference uses an intermediate stage to reach this goal, as illustrated in Fig 5(a). The first step involves an inference process that performs three key tasks (i) provide unique label for each method call, (ii) discover the postcondition of each method and (iii) identify all unsafe and partially-safe checks for each method. The second stage uses program specialization to obtain final target programs. It is expected to perform two tasks (i) insert run-time tests for unsafe checks and (ii) derive specialized programs that are well-typed.

$$\boxed{\mathrm{IMP}} \xrightarrow{\text{infer}} \boxed{\mathrm{IMP_I}} \xrightarrow{\text{specialize}} \boxed{\mathrm{IMP_T}}$$

(a) Two-Stage Processing

```
Float foo(Float[Int^s] a, Int^j j, Int^n n)
 where foo⟨s,j,n⟩, {ℓ1.H : j≤s−2)},{ℓ2.H}
   { Float v=0.0; Int i=j+1;
      if (0<i<=n) then v=(ℓ1:sub(a,i))
      else ();
      Int m=ℓ3:len(a,1); v+(ℓ2:sub(a,m))
    }
Q ≡ {foo⟨s,j,n⟩ = false}
```

(b) $\textsc{Imp}_I$ Intermediate Program

Figure 5: Intermediate Stage

Programs for the intermediate stage can be described by a language, called $\textsc{Imp}_I$, which is defined in Fig 4. An example of this is highlighted in Fig 5(b), based on the source program in Fig 2(a). Take note that every call in $\textsc{Imp}_I$ is uniquely labelled. Furthermore, each method declaration supports the identification of a set of *unsafe checks* in Υ, and a set of *partially-safe checks* in Φ. Those not explicitly identified are *safe checks*. In our example, $\{\ell_1.\texttt{H} : \cdots\}$ captures the precondition of a partially-safe check, while $\{\ell_2.\texttt{H}\}$ captures an unsafe check. Each unsafe check originates from a primitive call and will be converted to a run-time test guarding the primitive call. Using a monovariant specializer, we can obtain the target program shown earlier in Fig 2(b).

Another use of this intermediate language is to support the analysis of recursive functions. To formalise this process, we shall use constraint abstractions (adopted from [10]) of the form $q\langle n^*\rangle = \phi$ to denote each parameterized constraint. For uniformity, the size constraint (postcondition) of every method will be captured by a constraint abstraction of the same name. The set of constraint abstractions that is generated for each set of mutual-recursive methods is denoted by $\mathcal{Q}$. Constraint abstractions can be inlined, after fix-point analysis has been applied to solve recursive constraints. This technique will be elaborated in more detail in Sec 4.

2.2 Immutability Principle

Value immutability is one of the hallmarks of purely functional programming where the absence of updates (or side-effects) allow cleaner programs to be modelled and analysed. Our inference mechanism relies on another form of immutability, known as *size-immutability*, which allows size analysis in the presence of arbitrary sharing. Specifically, we track size properties (of arrays and primitives) that are never changed after creation. For example, we track the dimensions of arrays whose size properties are immutable after initialization. Note that assignments allow local variables to point to different values but never change the size-immutable properties of the values themselves.

The main benefit for identifying immutable size variables is that we allow their values to be freely shared without losing track of their size properties (in the presence of assignments). In practice, we may also need to track mutable size properties. In Sec A.1, we shall see that a linear type system with mutable size variables is required to track (updateable) array elements that are being used as indexes. Though this extension has been implemented in our system, we omit its description in this paper to focus on key mechanisms of inference.

Another place where the immutability principle is being adopted is in our handling of the call-by-value[2] parameter passing mechanism. More specifically, we disallow parameter variables from being updated/changed by assignments. There is no loss of generality, as we can always copy parameters to local variables where they may be freely updated. We shall assume the use of a preprocessor to enforce this requirement. Adoption of this principle simplifies our inference rule for method declarations.

2.3 Prime Notation for Imperative

Imperative language features have always been considered a challenge for both compiler-writers and application programmers. To infer postconditions, we have to take imperative updates at assignments into account, whose effects may also be aggregated at method declarations.

In this section, we propose to handle imperative updates in a systematic way with the use the *prime* notation, advocated in [11, 14], to capture the latest value of each size variable. Thus, a variable s' represents new information about the variable s after some expression evaluation. Note that the prime notation is idempotent, namely $s'' = s'$

We define FSV to return all distinct size variables in a formula. For example, $FSV(x' = z + 1 \wedge y = 2) = \{x', y, z\}$. We extend this definition to annotated type, as well as type environment. The function *prime* takes a set of size variables and returns their primed version. For example, $prime(\{s_1, \ldots, s_n\})$

[2]This includes array whose pointer is passed as a value.

$= \{s'_1, \ldots, s'_n\}$. We also extend this to annotated type and type environment, for e.g. $prime(\hat{\tau}) = prime(FSV(\hat{\tau}))$.

Often, we need to express a no-change condition on a set of size variables. This is useful for specifying the initial context. We define a $no\mathcal{X}$ operation which returns a formula for which the original and prime variables are made equal. For example, $no\mathcal{X}(\{x, y, z\}) = (x' = x \wedge y' = y \wedge z' = z)$. We extend this function to annotated types (and type environments), e.g. $no\mathcal{X}(\tau) = no\mathcal{X}(FSV(\tau))$.

Consider the following type environment and code fragment.

$$\Gamma = \{\mathtt{x} : \mathtt{Int}^x, \mathtt{y} : \mathtt{Int}^y, \mathtt{z} : \mathtt{Int}^z, \mathtt{a} : \mathtt{Float}\ [\mathtt{Int}^s]\}$$

```
x=x+1; // x'=x+1
y=x+z; // y'=x+z ∧ x'=x ∧ z'=z
sub(a,y); // y'=y∧s'=s
```

The state change (or otherwise) for each expression is captured by the primed variables in the size contexts (formulae) that appear on the RHS of the above code.

To combine a size context ϕ that has been generated sequentially from its prior context, Δ, we define a *sequential composition* operator $\circ_U$ as follows, where U is the set of size variables that are to be updated. Given $U = \{s_1, \ldots, s_n\}$, the sequential composition of Δ and ϕ is defined as:

$$\begin{array}{ll} \Delta \circ_U \phi = \exists R \cdot \rho'(\Delta) \wedge \rho(\phi) & \\ \text{where} \quad R = \{r_1, \ldots, r_n\} \text{ are new size variables} & \\ \qquad\quad \rho = \{s_1 \mapsto r_1, \ldots, s_n \mapsto r_n\} & \\ \qquad\quad \rho' = \{s'_1 \mapsto r_1, \ldots, s'_n \mapsto r_n\} & \end{array}$$

With $\Delta_{init} = no\mathcal{X}(\Gamma)$, we can compose the effect of the three sub-expressions from the example above, as follows:

$$\begin{array}{ll} \Delta_1 & = \Delta_{init} \circ_{\{x\}} (x'{=}x{+}1) \\ & = (s'{=}s \wedge x'{=}x \wedge y'{=}y \wedge z'{=}z) \circ_{\{x\}} (x'{=}x{+}1) \\ & = \exists x_0 \cdot (s'{=}s \wedge x_0{=}x \wedge y'{=}y \wedge z'{=}z \wedge x'{=}x_0{+}1) \\ & = (s'{=}s \wedge y'{=}y \wedge z'{=}z \wedge x'{=}x{+}1) \end{array}$$

$$\begin{array}{ll} \Delta_2 & = \Delta_1 \circ_{\{x,y,z\}} (y'{=}x{+}z \wedge x'{=}x \wedge z'{=}z) \\ & \quad \vdots \\ & = (s'{=}s \wedge y'{=}x{+}1{+}z \wedge x'{=}x{+}1 \wedge z'{=}z) \end{array}$$

$$\begin{array}{ll} \Delta_3 & = \Delta_2 \circ_{\{y,s\}} (y'{=}y \wedge s'{=}s) \\ & = (s'{=}s \wedge y'{=}x{+}1{+}z \wedge x'{=}x{+}1 \wedge z'{=}z) \end{array}$$

2.4 Soft and Hard Typing

While the type declarations of user-defined methods shall be inferred, those for primitive methods must be given apriori. During analysis, there are two ways to treat array checks from primitive method calls. One way is to demand that failing to satisfy an array check (as either totally-safe or partially-safe) is unacceptable, and thus causing the type inference to abort with a compile-time error. Another way is to allow failing of an array check to be converted to a run-time test, to be inserted in the program after analysis. We call the former kind *hard* checks, and the latter kind *soft* checks - in a spirit similar to strong versus soft typing.

For soft checks, we require corresponding *codes for run-time tests* to be specified in the header of primitive methods. Run-time test is not needed for hard checks. During analysis, determining if a check is hard or soft can be identified by the functions *isHard* and *isSoft*, respectively.

The specification for hard and soft checks can be made at the header declarations of primitive methods. The following are declarations for k-dimensional array primitives.

$$\begin{array}{l} \mathtt{Int}^r\ \mathtt{len}(t[\mathtt{Int}^{n_1}, \ldots, \mathtt{Int}^{n_k}]\ a, \mathtt{Int}^i\ \mathtt{v}) \\ \quad \text{where} \quad \bigvee_{j=1}^{k} (i{=}j \wedge r{=}n_j) \wedge no\mathcal{X}\{i, n_1, .., n_k\}, \{\mathtt{D} : 1{\le}i{\le}k\}, \{\} \end{array}$$

$$\begin{array}{l} t\ \mathtt{sub}(\mathtt{t}[\mathtt{Int}^{n_1}, \ldots, \mathtt{Int}^{n_k}]\ \mathtt{a}, \mathtt{Int}^{s_1}\ \mathtt{v}_1, \ldots, \mathtt{Int}^{s_k}\ \mathtt{v}_k) \\ \quad \text{where} \quad \bigwedge_{i=1}^{k} (0 \le s_i < n_i) \wedge no\mathcal{X}\{n_i, s_i\}_{i=1}^{k}, \{\mathtt{L}_i : 0 \le s_i, \\ \qquad\qquad \mathtt{H}_i : s_i < n_i\}_{i=1}^{k}, \{\mathtt{L}_i : \mathtt{0} <= \mathtt{v}_i, \mathtt{H}_i : \mathtt{v}_i < \mathtt{len}(\mathtt{a}, i)\}_{i=1}^{k} \end{array}$$

$$\begin{array}{l} \mathtt{Void}\ \mathtt{assign}(t[\mathtt{Int}^{n_1}, \ldots, \mathtt{Int}^{n_k}]\ \mathtt{a}, \mathtt{Int}^{s_1}\ \mathtt{v}_1, \ldots, \mathtt{Int}^{s_k}\ \mathtt{v}_k, t\ \mathtt{v}) \\ \text{where} \quad \bigwedge_{i=1}^{k} (0 \le s_i < n_i) \wedge no\mathcal{X}\{n_i, s_i\}_{i=1}^{k}, \{\mathtt{L}_i : 0 \le s_i, \\ \qquad\quad \mathtt{H}_i : s_i < n_i\}_{i=1}^{k}, \{\mathtt{L}_i : \mathtt{0} <= \mathtt{v}_i, \mathtt{H}_i : \mathtt{v}_i < \mathtt{len}(\mathtt{a}, i)\}_{i=1}^{k} \end{array}$$

$$\begin{array}{l} t[\mathtt{Int}^{n_1}, \ldots, \mathtt{Int}^{n_k}]\ \mathtt{newArr}(\mathtt{Int}^{s_1}\ \mathtt{v}_1, \ldots, \mathtt{Int}^{s_k}\ \mathtt{v}_k, t\ \mathtt{v}) \\ \text{where} \quad \bigwedge_{i=1}^{k} (0{<}s_i{\wedge}n_i{=}s_i) \wedge no\mathcal{X}\{s_i\}_{i=1}^{k}, \{\mathtt{S}_i : s_i > 0\}_{i=1}^{k}, \\ \qquad\quad \{\mathtt{S}_i : \mathtt{v_i} > 0\}_{i=1}^{k} \end{array}$$

Each primitive header is augmented with $\Delta, \Phi, \mathtt{C}$ annotations after the `where` keyword. Recall that Δ denotes its postcondition, Φ denotes its set of labelled checks, while $\mathtt{C}$ is a set of runtime testing codes that must be given for each soft check. As shown above, `sub` and `assign` each contains a set of low and high bound checks (labelled $\mathtt{L}_i, \mathtt{H}_i$). The operation `newArr` contains a set of positivity checks (labelled $\mathtt{S}_i$), while `len` has a dimensionality check (labelled $\mathtt{D}$). Of these checks, only the dimensionality check is a hard check without run-time testing, while the rest are given code fragments corresponding to their respective soft checks.

By allowing a mix of hard and soft checks, our approach permits trade-off to be made between the *compile-time guarantee* of hard checks and the *wider-applicability* of soft checks. For example, if all checks are made into hard checks, we may achieve total elimination of array-related checks, but we also rule out certain programs from being accepted by our modular inference.

3. INFERENCE RULES FOR IMP$_I$

The purpose of this inference phase is to analyse and propagate *contexts* at each program point, from which to determine if an array check is *safe* or *unsafe* or if a *precondition* is to be propagated to the method boundary. As we shall see, each propagated precondition will in turn become a check of the current method.

The type judgment to process the entire program is:

$$P_m \vdash_I \ P \leadsto P' \mid \mathcal{Q}$$

It takes a program P and a set of primitive declarations P_m, in order to derive an inferred program. In the process, a set of constraint abstractions is also generated, one for each method. The set of primitive declarations is also type-checked for consistency. Type judgement for expressions is specified as follows:

$$V; \Gamma; \Delta \vdash e \leadsto e_1 :: \tau, \Delta_1, \Phi, \Upsilon$$

Note that V is a set of size variables available at the boundary of the method in which the expression e resides; Γ is a type environment mapping program variables to their annotated types; $\Delta(\Delta_1)$ denotes the size constraint, which holds for size variables associated with Γ (Γ and τ) for expression e before (after) its evaluation; τ is an annotated type. The above judgement states that e will be transformed to e_1 during inference; e_1 has an annotated type τ while e does not contain size variable annotation. Both e and e_1 has the same underlying type under the context Γ and Δ. Furthermore, successful evaluation of e (and e_1) requires the validity of preconditions Φ, and inclusion of the run-time tests Υ. Such evaluation also changes the context Δ to Δ_1.

For convenience, our inference rules ensure that the size variables occurring in the derived annotated type τ are unique; *ie.,* $FSV(\tau) \cap FSV(\Gamma) = \emptyset$. All the inference rules are specified in Fig 6. In the rules, we use $s = fresh()$ and $\ell = fresh()$ to generate a new size variable and a new label, respectively. We also extend it to annotated type, so that $\hat{\tau} = fresh(\tau)$ returns a new type $\hat{\tau}$ with the same underlying type as τ but with fresh size variables instead. The function *equate* (τ_1, τ_2) generates equality constraints for the corresponding size variables of its two arguments, assuming both arguments share the same underlying type. For example, we have $equate(\texttt{Int}^n, \texttt{Int}^{m'}) = (n = m')$. Also, conditional is expressed as $\zeta_1 \lhd b \rhd \zeta_2 =_{df} \begin{cases} \zeta_1 & \text{if } b \\ \zeta_2 & \text{otherwise.} \end{cases}$ For the rest of this section, we highlight important aspects of our inference system via examples. We state the soundness theorem for our inference system, and provide more detail of its formulation in the appendix.

Assignment The assignment [**Var−Assign**] ensures that the size constraint is consistent in accordance with imperative update of LHS. To this end, we link the current size constraint with a new updated state for the affected size variables from the LHS. For example:

$$\frac{\begin{array}{c}\Gamma(v) = \texttt{Int}^n \quad \Gamma(u) = \texttt{Int}^m \\ V;\Gamma;\Delta \vdash \texttt{v} + \texttt{u} \rightsquigarrow \texttt{v} + \texttt{u} :: \texttt{Int}^r, \Delta \wedge r = n' + m', \emptyset, \emptyset \\ \Delta_2 = \exists r \cdot ((\Delta \wedge r = n' + m') \circ_{\{n\}} (n' = r))\end{array}}{V;\Gamma;\Delta \vdash \texttt{v} = \texttt{v} + \texttt{u} \rightsquigarrow \texttt{v} = \texttt{v} + \texttt{u} :: \texttt{void}, \Delta_2, \emptyset, \emptyset}$$

Assuming that $\Delta = (m' = 2 + n' \wedge n' = 5)$, where m' is the (primed) size variable associated with another program variable, say u. Then Δ_2 can be simplified to $(m' = 7 \wedge n' = 12)$. Note that the constraint of m' has been adjusted to preserve the original context; *ie.*, update of variable v does not affect the existing value of variable u. This is automatically handled by the sequential composition operator, namely $\circ_{\{n\}}$, described earlier in Sec 2.3.

Conditional Our [**If**] rule is able to track the size constraint path-sensitivity. Given $\Gamma = \Gamma_0 + \{\texttt{v} :: \texttt{Int}^n, \texttt{u} :: \texttt{Bool}^b\}$, we have:

$$\frac{\begin{array}{c}\Delta_1 = \Delta \wedge (b' = 1) \quad \Delta_2 = \Delta \wedge (b' = 0) \\ V;\Gamma;\Delta \wedge (b' = 1) \vdash \texttt{v} \rightsquigarrow \texttt{v} :: \texttt{Int}^{r_1}, \Delta_1 \wedge (r_1 = n'), \emptyset, \emptyset \\ V;\Gamma;\Delta \wedge (b' = 0) \vdash \texttt{5} \rightsquigarrow \texttt{5} :: \texttt{Int}^{r_2}, \Delta_2 \wedge (r_2 = 5), \emptyset, \emptyset \\ \texttt{e} = \texttt{if u then v else 5} \quad \Delta_4 = \Delta \wedge (r = n' \vee r = 5)\end{array}}{V;\Gamma;\Delta \vdash \texttt{e} \rightsquigarrow \texttt{e} :: \texttt{Int}^r, \Delta_4, \emptyset, \emptyset}$$

where $\Delta_4 = \rho_1(\Delta_1 \wedge (r_1 = n')) \vee \rho_2(\Delta_2 \wedge (r_2 = 5))$, with $\rho_1 = \{r_1 \mapsto r, r_1' \mapsto r'\}$ and $\rho_2 = \{r_2 \mapsto r, r_2' \mapsto r'\}$. Δ_4 can be simplified to $\Delta \wedge ((b' = 1 \wedge r = n') \vee (b' = 0 \wedge r = 5))$.

The substitutions ρ_1 and ρ_2 are the result of the *rename* function. The application $rename(\tau, \tau_1)$ maps all size variables (including their primed counterparts) occurring in τ to the corresponding size variables occurring in τ_1, assuming that τ and τ_1 share the same underlying type. This is used to change the inferred type, τ, to a freshen type, τ_1. We use an operator '$\uplus$' to combine two such disjoint substitutions into one.

Method Call Inference of method invocation is described by the rule [**Call**]. Data are passed from actuals to formals through the *rename* operation. Each call will also be given a unique label to facilitate references to preconditions and run-time tests gathered from it.

As an example, consider the inference of a primitive call $\texttt{sub(z,i)}$ under a context (Γ, Δ) such that $\Gamma = \Gamma_0 + \{\texttt{v} :: \texttt{Int}^v$, $\texttt{z} :: \texttt{Float}[\texttt{Int}^m], \texttt{i} :: \texttt{Int}^i\}$ and $\Delta = (m' = m) \wedge (m' = 10) \wedge (i' = v' + 2) \wedge (v' = v + 1) \wedge (v' > 5)$. The set of boundary variables V is $\{v, m\}$. The two array-bound checks of sub are transformed into the following preconditions:

$$\begin{array}{rcl} \rho = \{n \mapsto m, n' \mapsto m', s \mapsto i, s' \mapsto i'\}, & & \text{and} \\ pre_1 & = & (\Delta \approx\!> \rho(0 \le s))\!\downarrow_V \equiv_s \texttt{true} \\ pre_2 & = & (\Delta \approx\!> \rho(s < n))\!\downarrow_V \equiv_s (v < 7) \end{array}$$

These preconditions are obtained by simplifying ($\equiv_s$) the result of the operations ($\approx\!>$) and $\downarrow_V$. $\approx\!>$ formulates the implication of an array-bound check (expressed in terms of the size variables of the method parameters) by the corresponding calling context (expressed in terms of the size variables of the call arguments). It ensures that all size variables are expressed in terms of those of the call arguments:

$$\Delta \approx\!> \phi = (\Delta \Rightarrow \rho(\phi)) \text{ where } \begin{array}{l} \rho = \{s_1 \mapsto s_1', \ldots, s_n \mapsto s_n'\} \\ \{s_1, \ldots, s_n\} = FSV(\phi) \end{array}$$

The quantification operator $\downarrow_V$ derives the above implication by projecting the constraint to the boundary variable set V through universal quantification of the local variables:

$$\phi\!\downarrow_V = \forall W \cdot \phi \;\text{ where } W = FSV(\phi) - V.$$

Computation of precondition is thus expressed as $(\Delta \approx\!> \phi)\!\downarrow_V$.

Subsequently, pre_1 and pre_2 are used in the relation *mkChk* to determine if the corresponding bound check can be eliminated safely, be left as run-time check, or be decided at a later stage (a partially-safe check). Finally, the postcondition of the call, Δ_2, is inferred and returned. The main inference is described below:

$$\frac{\begin{array}{c} mkChk(pre_1, \ell.\texttt{L}, \emptyset, \emptyset) \quad mkChk(pre_2, \ell.\texttt{H}, \{\ell.\texttt{H} : pre_2\}, \emptyset) \\ \Phi = \{\ell\,.\texttt{H} : pre_2\} \quad \Upsilon = \emptyset \quad \Delta_2 = \Delta \circ_{\{m,i\}} \rho(\Delta_{\texttt{sub}}) \end{array}}{V;\Gamma;\Delta \vdash \texttt{sub(z,i)} \rightsquigarrow \ell : \texttt{sub(z,i)} :: \texttt{Float}, \Delta_2, \Phi, \Upsilon}$$

where $\Delta_2 = (m' = m) \wedge (m' = 10) \wedge (i' = v' + 2) \wedge (v' = v + 1) \wedge (v' > 5) \wedge (0 \le i' < m')$.

Inference of user-defined method call is done similarly.

Method Declaration The type judgment to process each method declaration takes a method declaration *md* of the form:

$$\tau\ m\ (\tau_1\ v_1, \ldots, \tau_k\ v_k)\ \{e\}$$

and infers an annotated counterpart md' of the form:

$$\hat{\texttt{t}}\ m\ (\hat{\tau}_1\ v_1, \ldots, \hat{\tau}_k\ v_k)\ \texttt{where}\ \Delta, \Phi, \Upsilon\ \{e_1\} \mid \mathcal{Q}$$

$$[\textbf{Const-Int}]\quad \frac{s = \mathit{fresh}() \quad \Delta_1 = \Delta \wedge (s = c_{\mathit{Int}})}{V;\Gamma;\Delta \vdash c_{\mathit{Int}} \leadsto c_{\mathit{Int}} :: \mathtt{Int}^s, \Delta_1, \emptyset, \emptyset}$$

$$[\textbf{Const-Bool}]\quad \frac{s = \mathit{fresh}() \quad \psi{=}(s{=}1 \lhd c_{\mathit{Bool}}{=}\mathtt{true} \rhd s{=}0)}{V;\Gamma;\Delta \vdash c_{\mathit{Bool}} \leadsto c_{\mathit{Bool}} :: \mathtt{Bool}^s, \Delta{\wedge}\psi, \emptyset, \emptyset}$$

$$[\textbf{Const-Void-Float}]\quad \frac{c{=}()|c_{\mathit{Float}} \quad t{=}(\mathtt{Void} \lhd c{=}() \rhd \mathtt{Float})}{V;\Gamma;\Delta \vdash c \leadsto c :: t, \Delta, \emptyset, \emptyset}$$

$$[\textbf{Var}]\quad \frac{\begin{array}{c}\Gamma(v) = \hat\tau \quad \hat\tau_1 = \mathit{fresh}(\hat\tau)\\ \phi = \mathit{equate}(\hat\tau_1, \mathit{prime}(\hat\tau))\end{array}}{V;\Gamma;\Delta \vdash v \leadsto v :: \hat\tau_1, \Delta \wedge \phi, \emptyset, \emptyset}$$

$$[\textbf{Var-Assign}]\quad \frac{\begin{array}{c}V;\Gamma;\Delta \vdash e \leadsto e_1 :: \hat\tau_1, \Delta_1, \Phi, \Upsilon \quad \Gamma(v) = \hat\tau \quad X = \mathit{FSV}(\hat\tau_1)\\ \phi = \mathit{equate}(\mathit{prime}(\hat\tau), \hat\tau_1) \quad \Delta_2 = \exists X.\Delta_1 \circ_{\mathit{FSV}(\hat\tau)} \phi\end{array}}{V;\Gamma;\Delta \vdash v{=}e \leadsto v{=}e_1 :: \mathtt{Void}, \Delta_2, \Phi, \Upsilon}$$

$$[\textbf{If}]\quad \frac{\begin{array}{c}\Gamma(v) = \mathtt{Bool}^b \quad V;\Gamma;\Delta \wedge (b'=1) \vdash e_1 \leadsto e_3 :: \hat\tau_1, \Delta_1, \Phi_1, \Upsilon_1\\ \hat\tau = \mathit{fresh}(\hat\tau_1) \quad V;\Gamma;\Delta \wedge (b'=0) \vdash e_2 \leadsto e_4 :: \hat\tau_2, \Delta_2, \Phi_2, \Upsilon_2\\ \rho_i = \mathit{rename}(\hat\tau_i, \hat\tau) \;\; \forall i \in \{1,2\} \quad \Delta_3 = \vee_{i=1}^2(\rho_i\Delta_i)\\ e_5 = \mathtt{if}\ v\ \mathtt{then}\ e_3\ \mathtt{else}\ e_4\end{array}}{V;\Gamma;\Delta \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \leadsto e_5 :: \hat\tau, \Delta_1, \Phi_1{\cup}\Phi_2, \Upsilon_1{\cup}\Upsilon_2}$$

$$[\textbf{Block}]\quad \frac{\begin{array}{c}V;\Gamma;\Delta \vdash e_1 \leadsto e_3 :: \hat\tau_1, \Delta_1, \Phi_1, \Upsilon_1 \quad \hat\tau = \mathit{fresh}(\tau)\\ \Delta_2 = \exists \mathit{FSV}(\hat\tau_1) \cdot (\Delta_1 \wedge \mathit{init}(\hat\tau) \wedge \mathit{equate}(\hat\tau_1, \mathit{prime}(\hat\tau)))\\ V;(\Gamma + \{v :: \hat\tau\});\Delta_2 \vdash e_2 \leadsto e_4 :: \hat\tau_2, \Delta_3, \Phi_2, \Upsilon_2\\ X = \mathit{FSV}(\hat\tau) \cup \mathit{prime}(\hat\tau) \quad \Delta_4 = \exists X.\Delta_3 \quad e_5 = (\hat\tau\ v\ =\ e_3;\ e_4)\end{array}}{V;\Gamma;\Delta \vdash (\tau\ v\ =\ e_1;\ e_2) \leadsto e_5 :: \hat\tau_2, \Delta_4, \Phi_1{\cup}\Phi_2, \Upsilon_1{\cup}\Upsilon_2}$$

$$[\textbf{Call}]\quad \frac{\begin{array}{c}\Gamma(v_i) = \tau_i \;\forall i \in 1..n \qquad \tau = \mathit{fresh}(\hat\tau) \quad U = \bigcup_{i=1}^k \mathit{FSV}(\tau_i)\\ (\hat\tau\ m(\hat\tau_1\ x_1, \ldots, \hat\tau_n\ x_n)\ \mathtt{where}\ \Delta_m,\ \Phi_m, \cdots) \in P \cup P_m\\ \ell = \mathit{fresh}() \quad \rho = \mathit{rename}(\hat\tau, \tau) \uplus \Sigma_{i=1}^n\{\mathit{rename}(\hat\tau_i, \tau_i)\}\\ \mathit{pre}_i \equiv_s (\Delta \ggg \rho\phi_i){\downarrow_V} \quad \mathit{mkChk}(\mathit{pre}_i, \ell.\ell_i^+, \hat\Phi_i, \Upsilon_i) \;\forall i \in 1..k\\ \Phi_m = \{(\ell_1^+ : \phi_1), .., (\ell_k^+ : \phi_k)\} \quad \hat\Phi = \bigcup_{i=1}^k \hat\Phi_i \quad \Upsilon = \bigcup_{i=1}^k \Upsilon_i\end{array}}{V;\Gamma;\Delta \vdash m(v_{1..n}) \leadsto \ell : m(v_{1..n}) :: \tau, \Delta \circ_U \rho(\Delta_m), \hat\Phi, \Upsilon}$$

$$[\textbf{Mtd-Declare}]\quad \frac{\begin{array}{c}\mathit{md} = \tau\ m(\tau_1\ v_1, \ldots, \tau_k\ v_k)\ \{e\}\\ \hat\tau_i = \mathit{fresh}(\tau_i)\ \forall\ i = 1..n \quad \hat\tau = \mathit{fresh}(\tau)\\ V = \bigcup_{i=1}^k \mathit{FSV}(\hat\tau_i) \quad W = V \cup \mathit{FSV}(\hat\tau)\\ \Gamma = \{v_1 : \hat\tau_1, \ldots, v_k : \hat\tau_k\} \quad \Delta_{\mathit{init}} = \mathit{init}(\Gamma)\\ V;\Gamma;\Delta_{\mathit{init}} \vdash e \leadsto e_1 :: \tau, \Delta, \Phi, \Upsilon \quad \mathcal{Q} = \{m\langle W\rangle = \Delta\}\end{array}}{\vdash_I \mathit{md} \leadsto \hat\tau\ m(\hat\tau_1\ v_1, \ldots \hat\tau_k v_k)\ \mathtt{where}\ m\langle W\rangle, \Phi, \Upsilon\{e_1\} \mid \mathcal{Q}}$$

$$[\textbf{Primitive-Declare}]\quad \frac{\begin{array}{c}\Gamma = \{v_1 : \hat\tau_1, \ldots, v_k : \hat\tau_k\} \quad V = \bigcup_{i=1}^k \mathit{FSV}(\hat\tau_i) \quad \Delta = \mathit{init}(\Gamma)\\ \forall(l : e) \in \mathtt{C}.\exists(l : \psi_l) \in \Phi.\;\; V;\Gamma;\Delta \vdash e :: \mathtt{Bool}^s, \Delta_l, \emptyset, \emptyset \quad ((s = 1 \wedge \Delta_l) \;\equiv\; \psi_l)\end{array}}{\vdash_{\mathit{prim}} \hat{\mathtt{t}}\ m\ (\hat\tau_1\ v_1, \ldots, \hat\tau_k\ v_k)\ \mathtt{where}\ \Delta, \Phi, \mathtt{C}}$$

$$[\textbf{Program}]\quad \frac{\begin{array}{c}\vdash_{\mathit{prim}} p \quad \forall p \in P_m\\ \vdash_I \mathit{md}_i \leadsto \mathit{md}'_i \mid \mathcal{Q}_i \;\forall i \in \{1..n\}\end{array}}{P_m \vdash_I \mathit{md}_1 \cdots \mathit{md}_n \leadsto \mathit{md}'_1 \cdots \mathit{md}'_n \mid \bigcup_{i=1}^n \mathcal{Q}_i}$$

$$[\textbf{MkChk-1}]\quad \frac{\mathit{pre} \equiv\ \mathtt{true}}{\mathit{mkChk}(\mathit{pre}, \ell^+, \emptyset, \emptyset)} \qquad [\textbf{MkChk-2}]\quad \frac{\mathit{isSoft}(\ell^+) \quad \mathit{pre} \equiv\ \mathtt{false}}{\mathit{mkChk}(\mathit{pre}, \ell^+, \emptyset, \{\ell^+\})} \qquad [\textbf{MkChk-3}]\quad \frac{\neg(\mathit{pre} \equiv \mathtt{true} \vee \mathit{pre} \equiv \mathtt{false})}{\mathit{mkChk}(\mathit{pre}, \ell^+, \{\ell^+ : \mathit{pre}\}, \emptyset)}$$

Figure 6: Type Inference Rules

where Δ is the postcondition in size constraint, Φ is the set of labelled preconditions from partially-safe checks and Υ is the set of labels for mandatory run-time checks (from unsafe checks). The postcondition is captured using constraint abstractions kept in $\mathcal{Q}$.

Inference is performed on the method body with an initial context that captures both an invariant on the size variables and a no-change initial state on the size variables. This is described by $\mathit{init}(\Gamma)$, where:

$$\begin{array}{lcl}\mathit{init}(\hat\tau) & = & \mathit{no}\mathcal{X}(\hat\tau) \wedge \mathit{init}'(\hat\tau)\\ \mathit{init}'(\mathtt{Bool}^b) & = & 0 \le b \le 1\\ \mathit{init}'(\hat{\mathtt{t}}[\mathtt{Int}^{s_1}, \ldots, \mathtt{Int}^{s_k}]) & = & \bigwedge_{i=1}^k (s_i > 0)\\ \mathit{init}'(\hat\tau) & = & \mathit{True}\end{array}$$

Consider the following declaration md:

```
Float m1(Float[Int] a, Int i) {
   Int j=len(a) ; sub(a, i) + sub(a, j)}
```

Freshly annotated size types are first provided to all the parameters of the method header: $\mathtt{Float\ m1(Float[Int}^s\mathtt{], Int}^i\ \mathtt{i)}$. The initial context is thus $\Delta_{\mathit{init}} = \mathit{no}\mathcal{X}(\{i, s\}) \wedge (s > 0)$. Furthermore, $W = V = \{i, s\}$ and $\Gamma = \{i : \mathtt{Int}^i,\ a : \mathtt{Float[Int}^s\mathtt{]}\}$.

Let:

$$\begin{array}{c}e_1 = \mathtt{int\ j\ =\ len(a)\ ;\ sub(a,i)\ +\ sub(a,j)}\\ e_2 = \mathtt{Int}^j\ \mathtt{j}\ =\ \ell_1 : \mathtt{len(a)}\ ;\ \ell_2 : \mathtt{sub(a,i)}\ +\ \ell_3 : \mathtt{sub(a,j)}\end{array}$$

We first have:

$$V;\Gamma;\Delta_{\mathit{init}} \vdash \mathtt{len(a)} \leadsto \ell_1 : \mathtt{len(a)} :: \mathtt{Int}^r, \Delta_1, \emptyset, \emptyset$$

where $\Delta_1 = \Delta_{\mathit{init}} \wedge (r = s)$. For the call $\mathtt{sub(a,i)}$, we have:

$$\frac{\begin{array}{c}\Gamma_2 = \Gamma + \{\mathtt{j} : \mathtt{Int}^j\} \quad \Delta_2 = \exists r \cdot \Delta_1 \wedge (j = j') \wedge (j' = r)\\ \Delta_3 = \Delta_2 \circ_{\{s,i\}} ((0 \le i < s) \wedge \mathit{no}\mathcal{X}(\{i, s\})\\ \Phi_1 = \{\ell_1.\mathtt{L} : 0 \le i, \ell_1.\mathtt{H} : i < s\}\end{array}}{V;\Gamma_2;\Delta_2 \vdash \mathtt{sub(a,i)} \leadsto \ell_2 : \mathtt{sub(a,i)} :: \mathtt{Float}, \Delta_3, \Phi_1, \emptyset}$$

where $\Delta_2 = \Delta_{\mathit{init}} \wedge (j = j') \wedge (j' = s)$ and $\Delta_3 = \mathit{no}\mathcal{X}(\{i, j, s\}) \wedge 0 \le i' < s' \wedge j' = s'$. For $\mathtt{sub(a,j)}$, we have:

$$V;\Gamma_2;\Delta_3 \vdash \mathtt{sub(a,j)} \leadsto \ell_3 : \mathtt{sub(a,j)} :: \mathtt{Float}, \Delta_4, \emptyset, \{\ell_3.\mathtt{H}\}$$

where $\Delta_4 = \Delta_3 \circ_{\{s,j\}} (0 \le j < s) \wedge \mathit{no}\mathcal{X}(\{j, s\}) = \mathit{no}\mathcal{X}(\{i, j, s\}) \wedge (0 \le i' < s') \wedge (j' = s') \wedge (0 \le j' < s') = \mathtt{false}$. Consequently,

$$\frac{\dfrac{\Delta_5 = \exists j \cdot \Delta_4 = \texttt{false}}{V;\Gamma;\Delta_{init} \vdash e_1 \;\rightsquigarrow\; e_2 :: \texttt{Float}, \Delta_5, \Phi_1, \{\ell_3.\texttt{H}\}} \quad \mathcal{Q} = \{\texttt{m1}\langle W\rangle = \Delta_5\}}{\vdash_I md \rightsquigarrow \texttt{Float m1(Float[Int}^s\texttt{] a, Int}^i\texttt{ i)} \;\; \texttt{where } m1\langle W\rangle, \Phi_1, \{\ell_3.\texttt{H}\}\; \{e_2\} \mid \mathcal{Q}}$$

3.1 Soundness of Inference System

The soundness of our type inference is defined with respect to a type checking system and a specialization process. Briefly, a method which is annotated with a set of preconditions is *well-typed* if it can be determined at compile-time that the execution of the method body, under a context that satisfies the preconditions, will not incur invalid array-accesses. Specifically, after type inference, the method body must be *specialized* to include the run-time tests discovered during inference, before it becomes well-typed. We state the soundness of our inference system below, and refer the reader to the appendices for detail.

THEOREM 1 (SOUNDNESS). *Let P be a program such that $(P_m \vdash_I P \rightsquigarrow P_I \mid \mathcal{Q})$ under type inference. Let $(\triangleright P_I \looparrowright P_T)$ be the specialization of P_I to P_T guided by the inferred run-time tests. Then P_T is well-typed.*

4. RECURSION ANALYSIS

Our type inference rules attempt to determine both a postcondition for each given expression, as well as a set of preconditions for its partially-safe checks. This bidirectional inference is easy for non-recursive expressions. For expressions from recursive methods, we need to determine (i) a fix-point for the postcondition, and (ii) an invariant on its recursive calls. This is on top of the derivation of preconditions for the partially-safe checks. Fortunately, we can handle these tasks with the help of constraint abstractions.

4.1 Deriving Postcondition

The postcondition of recursive constraint can be derived via a fix-point approximation procedure pioneered in [7]. Let us consider a constraint abstraction of the form $q\langle n^*, r\rangle$ where n^* denote inputs, while r denotes its output. For simplicity and without loss of generality, let us assume we have a recursive constraint abstraction with the following form.

$$q\langle n^*, r\rangle = \phi_0 \vee \phi_1[q\langle s^*, r_1\rangle, q\langle t^*, r_2\rangle]$$

Note that $\phi_1[\,,\,]$ is a formula with two holes - containing the recursive invocations, while ϕ_0 is the base case.

According to [7], the fix-point of such an abstraction can be formalised by a series of formulae $q_0, q_1, \cdots$ of the form.

$$q_0\langle n^*, r\rangle = \texttt{false}$$

$$q_{i+1}\langle n^*, r\rangle = \phi_0 \vee \phi_1[q_i\langle s^*, r_1\rangle, q_i\langle t^*, r_2\rangle]$$

For the above fix-point series to converge, we often need to apply approximations through two techniques, known as *hulling* and *widening*. Hulling combines a set of disjuncts $\bigvee \phi_i$ into a conjunct ϕ such that $(\bigvee \phi_i) \Rightarrow \phi$. We denote this process by $\bigvee \phi_i \equiv_h \phi$, and also allow hulling to be applied selectively to a subset of disjuncts. Widening takes a formula $\bigvee \bigwedge \phi_i$ and then drops (by replacing with `true`) one or more ϕ_i from it. Let us denote widening by $\equiv_w$. We shall apply each fix-point iteration and approximation until we obtain a formula $q_p\langle n^*, r\rangle$ such that $q_{p+1}\langle n^*, r\rangle \Rightarrow q_p\langle n^*, r\rangle$. This test indicates that a fix-point $q_p\langle n^*, r\rangle$ has been reached.

Methods with Postconditions:

```
Float sumvec(Float[Int^s] a,Int^i i,Int^j j)
   where sumvec⟨s,i,j⟩,...
{ if i>j then 0.0; else {Int v= ℓ1:sub(a,i);
         v+ℓ2:sumvec(a,i+1,j)
         }   }
Float sum(Float[Int^s] a) where sum⟨s⟩,...
{ Int l=ℓ3:len(a); ℓ4:sumvec(a,0,l-1)   }
```

Constraint Abstractions :

$\mathit{sumvec}\langle s,i,j\rangle \equiv (i{>}j) \vee (i{\leq}j \wedge (0{\leq}i{<}s) \wedge \mathit{sumvec}\langle s, i+1, j\rangle)$
$\qquad\qquad \equiv_{fix} (i{>}j) \vee (0{\leq}i{\leq}s-1, j)$
$\mathit{sum}\langle s\rangle \equiv \texttt{true}$

Figure 7: Sum Vector Program

Consider a simple array summation program in Figure 7, where the constraint abstractions obtained from our inference rules are also given. Take note that the constraint abstraction of the `sumvec` method is initially recursive in form. To obtain a closed-form postcondition for `sumvec`, we can apply fix-point analysis to its constraint abstraction, starting with sumvec_0. We reach a fix-point in the fourth iteration, as shown in Figure 8.

$\mathit{sumvec}_0\langle s,i,j\rangle \equiv \texttt{false}$
$\mathit{sumvec}_1\langle s,i,j\rangle \equiv (i{>}j) \vee ((0{\leq}i < s) \wedge (\exists i_1.\ i_1 = i{+}1 \wedge \texttt{false}))$
$\quad \equiv \quad (i{>}j)$
$\mathit{sumvec}_2\langle s,i,j\rangle \equiv (i{>}j) \vee ((0{\leq}i < s) \wedge (\exists i_1.\ i_1 = i{+}1 \wedge i_1{>}j))$
$\quad \equiv \quad (i{>}j) \vee (0{\leq}i{\leq}s)$
$\mathit{sumvec}_3\langle s,i,j\rangle \equiv (i{>}j) \vee ((0{\leq}i < s) \wedge (\exists i_1.\ i_1 = i{+}1$
$\qquad \wedge ((i_1{>}j) \vee (0{\leq}i_1{\leq}s)))$
$\quad \equiv \quad (i{>}j) \vee (0{\leq}i{\leq}s-2, j) \vee (0{\leq}i{\leq}s \wedge i{=}j)$
$\quad \equiv_h \quad (i{>}j) \vee (0{\leq}i{\leq}j, s-1)$
$\mathit{sumvec}_4\langle s,i,j\rangle \equiv (i{>}j) \vee ((0{\leq}i < s) \wedge (\exists i_1.\ i_1 = i{+}1$
$\qquad \wedge ((i_1{>}j) \vee (0{\leq}i_1{\leq}j, s-1)))$
$\quad \equiv_h \quad (i{>}j) \vee (0{\leq}i{\leq}j, s-1)$
Fix-Point Detected: $\mathit{sumvec}_4\langle s,i,j,r\rangle \Rightarrow \mathit{sumvec}_3\langle s,i,j,r\rangle$

Figure 8: Postcondition Derivation

4.2 Deriving Invariant

Within each recursive method, we may have checks that must be optimized. To deal with this, we compute another constraint, but this time, for just the input size variables (of the parameters). More specifically, we build a one-step size relation to relate the parameters of the next recursive calls with those of the first call. This relation is also subjected to a fix-point analysis, in order to obtain a multi-steps relation. The latter will relate the parameters of an arbitrary recursive call with the parameters of the first call.

The one-step relation can be directly extracted from each recursive constraint abstraction. Given the earlier abstraction, $q\langle n^*, r\rangle = \phi_0 \vee \phi_1[q\langle s^*, r_1\rangle, q\langle t^*, r_2\rangle]$. We can obtain a one-step relation, named I, that attempts to relate the input n^* with that of its recursive call, $\hat{n}^*$, as shown below.

$$I\langle n^*, \hat{n}^*\rangle = \phi_1[\bigwedge(s = \hat{n})^*, q\langle t^*, r_2\rangle]) \vee \phi_1[q\langle s^*, r_1\rangle, \bigwedge(t = \hat{n})^*]$$

With this one-step relation, we can now apply fix-point analysis to obtain its closure operation, as follows:

$$I_1\langle n^*, \hat{n}^*\rangle = I\langle n^*, \hat{n}^*\rangle$$

$$I_{i+1}\langle n^*,\hat{n}^*\rangle = I_i\langle n^*,\hat{n}^*\rangle \vee (\exists z^* \cdot I_i\langle n^*,z^*\rangle \wedge I\langle z^*,\hat{n}^*\rangle)$$

For `sumvec`, the one-step relation is specified as $sumvecI_1$ in Fig. 9. After applying a series of fix-point approximations, we can obtain a recursive invariant, named $sumvecI_3$. This invariant states a property that is true for all recursive calls, except the first call.

$$\begin{array}{l}
sumvecI_1\langle s,i,j,\hat{s},\hat{i},\hat{j}\rangle \equiv_h \\
\quad (\hat{s}{=}s)\wedge(\hat{j}{=}j)\wedge(\hat{i}{=}i{+}1)\wedge(0{\le}i{\le}s{-}1,j) \\
sumvecI_2\langle s,i,j,\hat{s},\hat{i},\hat{j}\rangle \equiv sumvecI_1\langle s,i,j,\hat{s},\hat{i},\hat{j}\rangle \vee \\
\quad \exists a,b,c \ . \ sumvecI_1\langle s,i,j,a,b,c\rangle \wedge sumvecI_1\langle a,b,c,\hat{s},\hat{i},\hat{j}\rangle \\
\quad \equiv_h (\hat{s}{=}s)\wedge(\hat{j}{=}j)\wedge(\hat{i}{-}2, 0{\le}i{<}\hat{i}{\le}s,j{+}1) \\
sumvecI_3\langle s,i,j,\hat{s},\hat{i},\hat{j}\rangle \equiv_h (\hat{s}{=}s)\wedge(\hat{j}{=}j)\wedge(\hat{i}{-}3, 0{\le}i{<}\hat{i}{\le}s,j{+}1) \\
\quad \equiv_w (\hat{s}{=}s)\wedge(\hat{j}{=}j)\wedge(0{\le}i{<}\hat{i}{\le}s,j{+}1) \\
sumvecI_4\langle s,i,j,\hat{s},\hat{i},\hat{j}\rangle \equiv_h (\hat{s}{=}s)\wedge(\hat{j}{=}j)\wedge(0{\le}i{<}\hat{i}{\le}s,j{+}1) \\
\textit{Fix-Point Detected:} \ sumvecI_4\langle s,i,j,\hat{s},\hat{i},\hat{j}\rangle \\
\qquad\qquad \Rightarrow sumvecI_3\langle s,i,j,\hat{s},\hat{i},\hat{j}\rangle
\end{array}$$

Figure 9: Recursive Invariant Derivation

4.3 Deriving Precondition

Let us now see how we can determine the safety of checks inside recursive methods. Firstly, take note that checks of recursive methods may be encountered multiple times due to recursion. We propose to separate out the check of the *first* recursive call, from the checks of the *rest* of the recursive calls. The reason for this is that recursive invariant that we derive is applicable to all recursive calls, except the first. Consequently, the context for the first check and the context for the recursive checks are different. In the case of the array access at label ℓ_1, the two contexts are:

$$\begin{array}{l}
\texttt{ctxFst}(\ell_1) \equiv no\mathcal{X}\{s,i,j\}\wedge s{>}0\wedge i{\le}j \\
\texttt{ctxRec}(\ell_1) \equiv no\mathcal{X}\{s,i,j\}\wedge s{>}0\wedge \hat{i}{\le}\hat{j} \wedge (0{\le}i{<}\hat{i}{\le}s,j{+}1)
\end{array}$$

Using these contexts, we can proceed to derive two preconditions for a pair of low-bound checks, as follows:

$$\begin{array}{l}
\texttt{preFst}(\ell_1.\texttt{L}) \equiv (\texttt{ctxFst}(\ell_1) \Rightarrow i{\ge}0)\downarrow_{\{s,i,j\}} \\
\qquad \equiv (s{\le}0)\vee(j{<}i\wedge 1{\le}s)\vee(0{\le}i) \\
\qquad \equiv_s (0{\le}i) \\
\texttt{preRec}(\ell_1.\texttt{L}) \equiv (\texttt{ctxRec}(\ell_1) \Rightarrow \hat{i}{\ge}0)\downarrow_{\{s,i,j\}} \\
\qquad \equiv \texttt{true}
\end{array}$$

Recall that $\equiv_s$ denotes simplification under the location's invariant at method boundary. A similar calculation for high-bound checks yield:

$$\begin{array}{l}
\texttt{preFst}(\ell_1.\texttt{H}) \equiv (\texttt{ctxFst}(\ell_1) \Rightarrow i < s)\downarrow_{\{s,i,j\}} \\
\quad \equiv (s{\le}0)\vee(j{<}i\wedge 1{\le}s)\vee(i{<}s) \\
\quad \equiv_s (i{<}s) \\
\texttt{preRec}(\ell_1.\texttt{H}) \equiv (\texttt{ctxRec}(\ell_1) \Rightarrow \hat{i} < s)\downarrow_{\{s,i,j\}} \\
\quad \equiv (j{<}s)\vee(s{\le}j\wedge i{\le}-1)\vee(s{\le}j,i\wedge 0{\le}i) \\
\quad \equiv_s (j{<}s)
\end{array}$$

Note that $\texttt{preFst}(\ell_1.\texttt{L})$ and $\texttt{preFst}(\ell_1.\texttt{H})$ are applicable to the first calls of `sumvec`, while $\texttt{preRec}(\ell_1.\texttt{L})$ and $\texttt{preRec}(\ell_1.\texttt{H})$ are applicable to the rest of its recursive calls. This separation allows for a more aggressive optimization since the latter can use the derived invariant, but not the former. For example, the check at $\texttt{preRec}(\ell_1.\texttt{L})$ is redundant irrespective of $\texttt{preFst}(\ell_1.\texttt{L})$.

To fully utilize the two sets of derived preconditions, we synthesize an extra copy of each recursive method. Given:

$$t\ f(\ldots)\ \{\ e[f(\ldots),f(\ldots)]\ \}$$

our system generates a pair of functions:

$$t\ f(\ldots)\ \{\ e[f_1(\ldots),f_1(\ldots)]\ \}$$

$$t\ f_1(\ldots)\ \{\ e[f_1(\ldots),f_1(\ldots)]\ \}$$

with $f(\ldots)$ to denote the first call, while $f_1(\ldots)$ to denote the rest of the recursive calls. In the case of `sumvec`, our inference system generates a pair of functions shown in Fig 10. Note that $\texttt{sumvec}_1$ uses a derived recursive invariant (labelled ℓ_6) to assist in the elimination of checks. This derived invariant is obtained from the multi-steps relation of Fig 9 by using:

$$\texttt{inv}\langle\hat{s},\hat{i},\hat{j}\rangle \equiv \exists s,i,j \cdot (\hat{s}{=}s)\wedge(\hat{j}{=}j)\wedge(0{\le}i{<}\hat{i}{\le}\hat{s},\hat{j}{+}1)$$

$$\equiv (1{\le}\hat{i}{\le}\hat{s},\hat{j}{+}1)$$

```
Float sumvec(Float[Int^s] a,Int^i i,Int^j j)
   where (i > j)∨(0≤i≤s−1, j),
         {ℓ1.L : 0≤i,  ℓ1.H : i<s,  ℓ2.ℓ5.H : j<a}, {}
{ if i>j then 0.0;
   else {Int v= ℓ1:sub(a,i); v+ℓ2:sumvec1(a,i+1,j)
         }   }
Float sumvec1(Float[Int^s] a,Int^i i,Int^j j)
   where (i > j)∨(0≤i≤s−1, j),
         {ℓ6 : 1≤i≤s, j+1,  ℓ5.H : j<a}, {}
{ if i>j then 0.0;
   else {Int v= ℓ5:sub(a,i); v+ℓ6:sumvec1(a,i+1,j)
         }   }
```

Figure 10: Program Inferred for `sumvec`

Going up the call hierarchy, we can further determine that all checks of `sumvec` are in fact safe for its caller at `sum`. We have $\texttt{ctx}(\ell_4) \equiv s{>}0\wedge i{=}0\wedge j{=}s{-}1$, and can confirm the total safety of all its checks via:

$$\texttt{ctx}(\ell_2) \Rightarrow \texttt{pre}(\ell_4.\ell_1.\texttt{L}) \wedge \texttt{pre}(\ell_4.\ell_1.\texttt{H}) \wedge \texttt{pre}(\ell_4.\ell_2.\ell_5.\texttt{H})$$

To summarise, there are a number of benefits in our approach for handling recursive methods. Firstly, both the postcondition and the recursive invariant are derived from the same constraint abstraction. Secondly, the computed recursive invariant did not require any prior precondition to be imposed, a feature that is critical for inference. Lastly, the recursive invariant for each method is computed once but may be used multiple times for different checks within the same method, in line with the modularity principle.

5. SPECIALIZATION

Type inference produces an annotated program such that all run-time tests are collected at method declarations. The next task is to place these run-time tests at their respective primitive operations of the program with the objective that array operations are safe, *and* the array checks are done minimally. To this end, we specialize the existing method definitions with information about run-time tests.

To understand the effectiveness of various approaches to specializing method definitions, we examine the following program (assuming that the labels are predetermined):

```
void main()          t2 m2(···)           t1 m1(···)
{ ···                { ···                { ···
  ℓ5 : m2(···);        ℓ3 : m1(···);        v1 = (ℓ1 : sub(a1, i1));
  ···}                 ···                  ℓ2 : assign(a2, i2, v1) }
                       ℓ4 : m1(···)}
```

Let us assume that the results of inference are as follows:

Pre-Conditions of m1 computed			
from ℓ_1		from ℓ_2	
ℓ_1.L	ℓ_1.H	ℓ_2.L	ℓ_2.H
true	ϕ_1	true	ϕ_2

Pre-Conditions of m2 computed			
from ℓ_3		from ℓ_4	
$\ell_3.\ell_1$.H	$\ell_3.\ell_2$.H	$\ell_4.\ell_1$.H	$\ell_4.\ell_2$.H
true	ϕ_3	ϕ_4	false

Pre-Conditions of main computed from ℓ_5	
$\ell_5.\ell_3.\ell_2$.H	$\ell_5.\ell_4.\ell_1$.H
true	true

This corresponds to the following inferred method headers with partially-safe and unsafe checks.

```
t1 m1(···)···   {ℓ1.H : φ1, ℓ2.H : φ2}, {}
t2 m2(···)···   {ℓ3.ℓ2.H : φ3, ℓ4.ℓ1.H : φ4}, {ℓ4.ℓ2.H}
void main()···  {}, {}
```

Thus, there is only one unsafe check to be residualized at run-time, namely $\ell_4.\ell_2$.H. The other checks are either safe or partially-safe with the possibility of becoming safe at its parent call's context.

A conservative approach to eliminate checks is *monovariant specialization*. This technique aims at modifying the original method definitions such that an array check will be residualized in a specialized method if the check has been marked by *one or more run-time tests* in the program.

Applying this specialization technique to the example given above yields the following specialized code for m1 where ℓ_2.H is explicitly checked at run-time.

```
void main() ..    t2 m2(···)           t1 m1(···) where .., φ1
  { ···           where .., φ3∧φ4      { ···
    m2(···) ;     { ···                  v1 = ( sub(a1, i1));
    ···}            m1(···);             if (i2 < len(a2, 1)) then
                      ···                  assign(a2, i2, v1)
                    m1(···);}            else error }
```

A more aggressive approach to eliminating checks is *polyvariant specialization*. This aims at creating *multiple* specialized methods for each method definition, such that different specialized versions of a method have different sets of array checks being eliminated. Its application on our example program yields the following result:

```
void main() ..      t1 m1_1(···) where .., φ1∧φ2
  { ···             { ···
    m2(···) ;         v1 = sub(a1, i1);
    ···}               v2 = assign(a2, i2, v1) }

t2 m2(···)          t1 m1_2(···)  where .., φ1
where .., φ3∧φ4     { ···
{ ···                 v1 = sub(a1, i1);
  m1_1(···);          if (i2 < len(a2, 1)) then
    ···                 assign(a2, i2, v1)
  m1_2(···);}         else error }
```

Note that two versions of m1 have been created to handle two calls to m1 with different contexts.

Both monovariant and polyvariant specializations are guided by the inferred run-time tests. In addition, polyvariant specialization is aided by a *signature cache*, which records each specialized method created. The detailed formalizations of both specializers are given in the Appendix.

6. IMPLEMENTATION

We have constructed the proposed modular inference system together with a monovariant specializer. The output from our specializer can be verified by a separate type checking system that we have also built. Our implementation includes a pre-processing phase that allows us to use some syntactic sugars for Imp. The entire prototype system was built using Glasgow Haskell compiler[18] extended with the Omega[19] constraint solving library.

Our main objective for building this prototype is to show that such an advanced form of inference mechanism is indeed viable. We tested our system on a set of benchmark programs. Most of them are small but challenging, as they made use of arbitrary recursion. We have also manually converted (to Imp) a couple of medium-sized programs taken from the SciMark benchmark suite [17]. Figure 11 summarises the statistics obtained for each program that we inferred under Linux 9.0 platform on Pentium 2.0 GHz system with 1GBytes main memory. Apart from program size (column 2), we also counted the number of static checks present in the original (column 3) and inferred (column 4) programs. The latter is always higher as partially-safe checks (count in parenthesis of column 4) are re-counted after each propagation up its calling hierarchy.

Due to the precision of our inference system, we were able to eliminate 100% of array checks for all the programs we tested but one. The *sentinel* example had only 50% of its checks eliminated since it make use of a sentinel to guard against falling off one end of the array. We are unable to capture the existential property that is required for such a check elimination. Without the analysis to support indirection arrays, we would also be losing 22% of checks for the sparse matrix multiplication example. The size of constraint annotations (which includes preconditions, postconditions and recursive invariant) are on average around 28.5% of the size of the source program. Thus, our inference eliminates sizeable mental effort required for programmers with only a type checker. Except for hanoi, all fix-points (for postcondition and recursive invariants) were automatically inferred by our prototype system. To handle functions with complicated recursion, like hanoi, we have also provided a mechanism for programmers to supply their own recursive invariants which are verified by our system before they are used.

Note that the time taken for inference includes parsing, preprocessing (to eliminate syntactic sugars), modular inference and specialization, while the time taken for checking involves parsing and type checking. Inference and type checking times appear to correlate well with the number of static checks handled.

Benchmark Programs	Source (lines)	Static Checks Original	Static Checks Inferred		Compile-Time (secs) Inference	Compile-Time (secs) Checking	Dynamic Checks Indirect	Dynamic Checks Eliminated
binary search	50	2	5	(2)	3.41	0.31	-	100%
bubble sort	59	12	21	(9)	2.13	0.73	-	100%
copy sequence	32	4	8	(4)	0.91	0.38	-	100%
dot product	30	4	8	(5)	0.45	0.09	-	100%
hanoi tower	38	16	31	(15)	17.77	9.05	-	100%
initialize array	20	2	4	(2)	0.34	0.07	-	100%
merge sort	80	16	64	(39)	19.8	9.7	-	100%
queen	45	8	13	(5)	3.94	0.95	-	100%
quick sort	67	12	30	(12)	2.32	0.59	-	100%
sentinel	26	4	9	(5)	0.21	0.07	-	50%
sparse multiply	46	12	20	(7)	6.18	2.86	22%	100%
sumvec	33	2	4	(2)	0.60	0.14	-	100%
LU Decomposition (LU)	160	64	116	(48)	26.01	15.10	-	100%
Successive Overrelaxation (SOR)	84	24	44	(20)	7.26	3.73	-	100%

Figure 11: Comparative Statistics

One important feature of our inference is the modular approach taken for interprocedural analysis. Due to a limited number of variables at each method boundary, we believe that our mechanism is reasonably scalable, even though Presburger constraint-solving has exponential time-complexity in the worst case. Furthermore, our approach permits precision/efficiency tradeoff where a simpler constraint domain, such as difference constraints[2] with polynomial-time complexity, may be chosen instead. Finally, we have also succeeded in devising techniques to strip off avoidance conditions from the preconditions of partially-safe checks. This is important for scalability of inference, as smaller formulae are faster to verify. In practice, we are also able to maintain the same level same level of eliminations (for our benchmark programs) with these stronger preconditions.

7. RELATED WORKS

Traditionally, data-flow analysis techniques have been employed to gather information for the purpose of identifying redundant array checks. Within the scope of intra-procedural analysis, these techniques are also used to gather anticipatable information for the purpose of hoisting partially-redundant checks to more profitable locations. The techniques have gradually evolved in sophistication, from the use of family of checks in [13], to the use of difference constraints in [2].

To identify redundant checks more accurately, verification-based methods have also been advocated by Suzuki and Ishihata [21], Necula and Lee [16] and Xu *et al* [24]; whilst Cousot and Halbwachs [7] have advocated the use of abstract interpretation techniques, and Flanagan and Saxe [8] have devised an effective method for discovering loop invariants with minimal user-assistance. Furthermore, Rugina and Rinard [20] proposed an analysis method to synthesize polynomial symbolic bounds without fix-point iteration. While efficient, fixing a target form for the symbolic bound may result in loss in precision, or even failure. Under this category of precise forward analysis, Xi and Pfenning have advocated the use of dependent types for array bound check elimination [23]. Their approach is also limited to totally redundant checks. Moreover, the onus for supplying suitable dependent types rest squarely on the programmers, as only a type checker is available.

The basic idea behind the backward derivation of weakest precondition dates back to the inductive iteration method pioneered by Suzuki and Ishihata [21] in the seventies. More recently, the method has been improved by Xu *et al* [24]. However, the primary focus has been on finding totally redundant checks. Due to this focus, the backward analysis technique proposed in [21] actually gathers both precondition and contextual constraints. Apart from missing out on partially redundant checks, their approach is less accurate than forward methods (such as [7]) since information on local variables are often lost in backward analysis. In contrast, we have used a bidirectional analysis where fix-point iterations is conducted in a forward manner. Apart from improved accuracy, we also better reusability as fix-point analysis is done once per recursive method, rather than once per check!

The initial idea for deriving preconditions for partially redundant checks was first described by the authors[6] for a first-order functional language. However, this early work did not provide any formalization of both the inference and specialization processes. The current work makes considerable advances in several other directions. We have systematically developed a modular inference mechanism and proven it safe for an imperative setting. Automatic analysis of array indirection is now supported and a new technique engineered to obtain smaller preconditions. Most importantly, we have also built an inference system and have conducted experiments to demonstrate both the precision and practicality of our approach.

8. CONCLUDING REMARKS

We have proposed a new modular inference mechanism (and its associated code specializers) for a dependent type system with size relations. Our approach captures postcondition in the presence of imperative updates, and derives safety preconditions for each check encountered. Both the postcondition and safety precondition are propagated interprocedurally, though in opposite directions. Recursive methods are also handled through a fix-point analysis on constraint abstraction derived via inference. The resulting analysis is not only flow and context-sensitive, but is also path-sensitive. It can capture precise size relations between local variables, inputs and outputs.

We have also extended the framework to handle wider classes of programs. Specifically, in the appendix, we discuss the handling of array indirection and while-loop constructs. Initial experiences with a prototype implementation suggest that such an advanced form of type inference is viable.

9. REFERENCES

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotation for Program Understanding. In *ACM OOPSLA*, Seattle, Washington, November 2002.

[2] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *ACM PLDI*, pages 321–333, 2000.

[3] R. Bodik, R. Gupta, and M.L. Soffa. Complete removal of redundant expressions. In *ACM PLDI*, pages 1–14, 1998.

[4] R. Cartwright and M. Fagan. Soft typing. In *ACM PLDI*, pages 278–297, 1991.

[5] W.N. Chin and S.C. Khoo. Calculating sized types. In *ACM SIGPLAN PEPM*, pages 62–72, Boston, Massachusetts, United States, January 2000.

[6] W.N. Chin, S.C. Khoo, and Dana N. Xu. Deriving pre-conditions for array bound check elimination. In *Programs as Data Objects II*, pages 2–24, Aarhus, Denmark, May 2001. Springer Verlag.

[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM POPL*, pages 84–96. ACM Press, 1978.

[8] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *ACM PLDI*, pages 193–205, 2001.

[9] R. Gupta. A fresh look at optimizing array bound checking. In *ACM PLDI*, pages 272–282, New York, June 1990.

[10] J Gustavsson and J Svenningsson. Constraint abstractions. In *Programs as Data Objects II*, pages 63–83, Aarhus, Denmark, May 2001.

[11] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[12] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *23rd ACM Principles of Programming Languages Conference*, pages 410–423, St. Petersburg Beach, Florida, January 1996. ACM Press.

[13] P Kolte and M Wolfe. Elimination of redundant array subscript range checks. In *ACM PLDI*, pages 270–278. ACM Press, June 1995.

[14] L. Lamport. The temporal logic of actions. *ACM Trans. on Programming Languages and Systems*, 16(3):872–923, May 1994.

[15] Mikel Luján, John R. Gurd, T. L. Freeman, and José Miguel. Elimination of Java array bounds checks in the presence of indirection. In *ACM Joint Java Grande-IScope Conf.*, pages 76–85, 2002.

[16] G Necula and P. Lee. The design and implementation of a certifying compiler. In *ACM PLDI*, pages 333–344, 1998.

[17] National Institue of Standards and Technology. Java scimark benchmark for scientific computing. `http://math.nist.gov/scimark2/`.

[18] S Peyton-Jones and et al. Glasgow Haskell Compiler. http://www.haskell.org/ghc.

[19] W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of ACM*, 8:102–114, 1992.

[20] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM PLDI*, pages 182–195. ACM Press, June 2000.

[21] N. Suzuki and K. Ishihata. Implementation of array bound checker. In *ACM POPL*, pages 132–143, 1977.

[22] J. Cocke V. Markstein and P. Markstein. Optimization of range checking. In *Proceedings of a Symposium on Compiler Optimization*, pages 114–119, June 1982.

[23] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *ACM PLDI*, pages 249–257. ACM Press, June 1998.

[24] Z. Xu, B.P. Miller, and T. Reps. Safety checking of machine code. In *ACM PLDI*, pages 70–82. ACM Press, June 2000.

APPENDIX

A. DISCUSSIONS

We discuss two extensions to our modular inference system, to allow it to handle wider classes of programs. Both extensions have been implemented in our system.

A.1 Array Indirections

There is a class of programs which has been largely ignored in past work on array bound checks elimination. This class of programs uses indexes, known as array indirections, that are kept as elements of integer arrays. A well-known example is the sparse matrix computation.

To support programs which use indirection arrays, we must track the max/min bounds of their elements. Precise tracking will allow us to determine the bounds of indexes retrieved from such integer arrays. As the array elements may be changed by the `assign` primitive, these max/min bounds may also change during program execution. Such size properties are therefore *mutable*. To handle them safely, we require a linear type system, such as the one proposed in [1], that could be used to identify references to *unaliased* integer arrays. Only the max/min bounds of such unaliased arrays could be safely tracked in a precise manner.

In addition to alias annotation, the main extra machinery is a set of enhanced primitive declarations for unaliased integer arrays. Note that max/min bounds are changed by the `newArr` and `assign` primitives, and utilized by the `sub` primitive.

$$\begin{array}{l}\texttt{Int}^r\ \texttt{sub}(\texttt{t}^a[\texttt{Int}^{n_1},..,\texttt{Int}^{n_k}]\ \texttt{a},\texttt{Int}^{s_1}\ \texttt{v}_1,..,\texttt{Int}^{s_k}\ \texttt{v}_k)\\ \quad\text{where}\ \bigwedge_{i=1}^{k}(0\leq s_i<n_i)\wedge no\mathcal{X}\{n_i,s_i\}_{i=1}^{k}\\ \qquad\quad \wedge no\mathcal{X}\{a.min,a.max\}\wedge a.min\leq r\leq a.max,\cdots\end{array}$$

$$\begin{array}{l}\texttt{assign}(\texttt{Int}^a[\texttt{Int}^{n_1},..,\texttt{Int}^{n_k}]\ \texttt{a},\texttt{Int}^{s_1}\ \texttt{v}_1,..,\texttt{Int}^{s_k}\ \texttt{v}_k,\texttt{Int}^v\ \texttt{v})\\ \text{where}\ \bigwedge_{i=1}^{k}(0\leq s_i<n_i)\wedge no\mathcal{X}\{n_i,s_i\}_{i=1}^{k}\wedge\\ \qquad a.min'=min(a.min,v)\wedge a.max'=max(a.max,v),\cdots\end{array}$$

$$\begin{array}{l}\texttt{Int}^a[\texttt{Int}^{n_1},..,\texttt{Int}^{n_k}]\ \texttt{newArr}(\texttt{Int}^{s_1}\ \texttt{v}_1,..,\texttt{Int}^{s_k}\ \texttt{v}_k,\texttt{Int}^v\ \texttt{v})\\ \text{where}\ \bigwedge_{i=1}^{k}(0<s_i\wedge n_i=s_i)\wedge no\mathcal{X}\{s_i\}_{i=1}^{k}\wedge a.min'=a.max'=v),..\end{array}$$

Our approach for handling array indirection through size relations (in types inferred) relies entirely on compile-time analysis. Compared to a recent solution for indirection arrays[15], we do not incur any run-time overheads.

A.2 Converting WHILE Loops

We have also provided direct support for the `while`-loop construct. The reason for doing so is that method declaration is unable to handle certain loops due to the use of the pass-by-value parameter mechanism. Our solution is to analyse each `while` loop using its corresponding tail-recursive function but with the use of *pass-by-reference* mechanism instead. We achieve this through a preprocessing phase which converts each while-loop into an equivalent tail-recursive function. This conversion is for the purpose of modular inference only, since program execution is still based on loops. We realise this conversion through the following set of rewriting rules:

$$e\hookrightarrow_W e_1,W \qquad meth\hookrightarrow_W meth_1,W \qquad P\hookrightarrow_W P_1,W$$

These rules generate a special set of tail-recursive functions (where pass-by-reference is assumed) in the W output set. Most of the rewriting rules preserve the structure of the original expressions, except for the following:

$$\frac{\begin{array}{c}v^*{=}FV(e_1)\cup FV(e_2)\quad e_1\hookrightarrow_W e_3,W_1\quad e_2\hookrightarrow_W e_4,W_2\\ m{=}fresh()\quad W_0{=}\{m(v^*)\equiv\texttt{if}\ e_3\ \texttt{then}\ e_4;m(v^*)\ \texttt{else}\ ()\}\end{array}}{\texttt{while}\ e_1\ \texttt{do}\ e_2\hookrightarrow_W m(v^*),W_1\cup W_2\cup W_0}$$

Note that $FV(e)$ returns the set of free variables from expression e.

B. TYPE SOUNDNESS

In this section, both static semantics (i.e. type checking rules) and dynamic semantics are defined. We prove the soundness of the type checking with respect to the dynamic semantics of the language. We also prove the soundness and completeness of the type inference rules with respect to the type checking rules.

B.1 Type Checking for $\text{I}\scriptstyle{\text{MP}}_T$

The type checking rules of the form:

$$\Gamma;\Delta_1\vdash e::\hat{\tau},\Delta_2$$

state the judgment that an expression has type $\hat{\tau},\Delta_2$, where $\hat{\tau}$ is an annotated type and Δ_2 is a post-condition, under an environment Γ and a pre-condition satisfying Δ_1.

A program accepted by the type checker conforms to the syntax of $\text{I}\scriptstyle{\text{MP}}_T$. A typed-checked program guarantees that all array access are never out of range, even when some of the array access are *not* protected by the rudimentary array-bound checks. In the latter case, the type checker infers from the context that such array access is safe.

First, we define the notion of subtyping as follows:

Definition 1 (Subtyping). *Given two post-conditions Δ_1 and Δ_2 with the same set of free variables, if $\Delta_1\Rightarrow\Delta_2$ then $\Delta<:\Delta_2$.*

Type-checking rules are defined in Figure 12. Most of the rules are replica of the corresponding type-inference rules depicted in Figure 6, except the rules for method call and method declaration.

The main objective of the method call rule $\underline{[\textbf{C}-\textbf{Call}]}$ is to verify that the pre-conditions associated with the method can be satisfied under the calling context. This verification must hold for a program that has been specialized to avoid invalid array bound checks.

The method-declaration rule $\underline{[\textbf{C}-\textbf{Mtd}-\textbf{Declare}]}$ checks the type of the method body under the context that the pre-conditions Φ are satisfied. This is necessary so that accessing an array without array checks does not cause unnecessary alarm when the range of the array index value can be deduced the pre-conditions.

B.2 Dynamic Semantics for $\text{I}\scriptstyle{\text{MP}}_T$

The dynamic semantics is described in small step. Notations used are defined as follows.

Locations :	ι	$\in$	*Location*
primitive values :	k	$\in$	$prim =\ Float\uplus Int\uplus Bool\uplus Void$
Values :	δ	$\in$	$Value = prim\uplus Location$
Store :	ϖ	$\in$	$Store = Location\rightarrow_{fin} ArrVal$
Variable Env :	Π	$\in$	$VEnv = Var\rightarrow_{sfin} Value$
Array values :	η	$\in$	$ArrVal =$
			$Int\times Int^n\times(Int^n\rightarrow_{fin} prim)$

The only object values in our store are of array type. An object value comprises of three components: a number indicating the dimension of the array; a tuple containing the size of each of its dimensions, and a function mapping (tuple of) array indices to array elements.

The domain of the store, $dom(\varpi)$, represents all global locations that have been allocated at a given moment.

Note that $f: A\rightarrow_{sfin} B$ denotes a *stackable* mapping from A to B. Such a stackable mapping is defined as:

$$A\rightarrow_{sfin} B =_{df} (A^{\texttt{int}}\rightarrow_{fin} B,\ \texttt{int})$$

where `int` denotes the current frame number of the mapping, while $A^{\texttt{int}}$ denotes the domain that has been marked

$$\frac{\Gamma;\Delta_0 \vdash e :: \hat{\tau},\Delta_1 \quad \Delta_1 <: \Delta_2}{\Gamma;\Delta_0 \vdash e :: \hat{\tau},\Delta_2}\;[\mathbf{Sub1}](\text{Covariance})$$

$$\frac{\Gamma;\Delta_1 \vdash e :: \hat{\tau},\Delta_2 \quad \Delta_0 <: \Delta_1}{\Gamma;\Delta_0 \vdash e :: \hat{\tau},\Delta_2}\;[\mathbf{Sub2}](\text{Contravariance})$$

$$\frac{s = \mathit{fresh}() \quad \Delta_1 = \Delta \wedge (s = c_{Int})}{\Gamma;\Delta \vdash c_{Int} :: \mathtt{Int}^s,\Delta_1}\;[\mathbf{C-Const-Int}]$$

$$\frac{s = \mathit{fresh}() \quad \psi{=}(s{=}1 \lhd c_{Bool}{=}\mathtt{true} \rhd s{=}0)}{\Gamma;\Delta \vdash c_{Bool} :: \mathtt{Bool}^s,\Delta{\wedge}\psi}\;[\mathbf{C-Const-Bool}]$$

$$\frac{c{=}()|c_{Float} \quad t{=}(\mathtt{Void} \lhd c{=}() \rhd \mathtt{Float})}{\Gamma;\Delta \vdash c :: t,\Delta}\;[\mathbf{C-Const-Void-Float}]$$

$$\frac{\Gamma(v) = \hat{\tau} \quad \hat{\tau}_1 = \mathit{fresh}\ \hat{\tau} \quad \phi = \mathit{equate}(\hat{\tau}_1, \mathit{prime}(\hat{\tau}))}{\Gamma;\Delta \vdash v :: \hat{\tau}_1,\Delta \wedge \phi}\;[\mathbf{C-Var}]$$

$$\frac{\Gamma;\Delta \vdash e :: \hat{\tau}_1,\Delta_1 \quad \Gamma(v) = \hat{\tau} \quad X = FSV(\hat{\tau}_1) \quad \phi = \mathit{equate}(\mathit{prime}(\hat{\tau}), \hat{\tau}_1) \quad \Delta_2 = \exists X.\Delta_1 \circ_{FSV(\hat{\tau})} \phi}{\Gamma;\Delta \vdash (v = e) :: \mathtt{Void},\Delta_2}\;[\mathbf{C-Var-Assign}]$$

$$\frac{\begin{array}{c}\Gamma(v) = \mathtt{Bool}^b \quad \Gamma;\Delta \wedge (b' = 1) \vdash e_1 :: \hat{\tau}_1,\Delta_1 \\ \hat{\tau} = \mathit{fresh}\ \hat{\tau}_1 \quad \Gamma;\Delta \wedge (b' = 0) \vdash e_2 :: \hat{\tau}_2,\Delta_2 \\ \rho_i = \mathit{rename}(\hat{\tau}_i, \hat{\tau}) \quad \forall i \in \{1,2\} \quad \Delta_3 = \vee_{i=1}^2(\rho_i\Delta_i)\end{array}}{\Gamma;\Delta \vdash (\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) :: \hat{\tau},\Delta_1}\;[\mathbf{C-If}]$$

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash e_1 :: \hat{\tau}_1,\Delta_1 \quad \Delta_2 = \Delta_1 \wedge \mathit{init}(\hat{\tau}) \wedge \mathit{equate}(\hat{\tau}_1, \mathit{prime}(\hat{\tau})) \\ (\Gamma + \{v :: \hat{\tau}\}); \exists FSV(\hat{\tau}_1) \cdot \Delta_2 \vdash e_2 :: \hat{\tau}_2,\Delta_3 \\ \hat{\tau} = \mathit{fresh}(\tau) \quad X = FSV(\hat{\tau}) \cup \mathit{prime}(\hat{\tau}) \quad \Delta_4 = \exists X.\Delta_3\end{array}}{\Gamma;\Delta \vdash (\tau\ v\ =\ e_1;\ e_2) :: \hat{\tau}_2,\Delta_4}\;[\mathbf{C-Block}]$$

$$\frac{\begin{array}{c}\Gamma(v_i) = \tau_i\ \forall i \in 1..n \\ \tau =\ \mathit{fresh}\ \hat{\tau} \quad U = \bigcup_{i=1}^k FSV(\tau_i) \\ (\hat{\tau}\ m(\hat{\tau}_1\ x_1,\ldots,\hat{\tau}_n\ x_n)\ \mathtt{where}\ \Delta_m,\ \Phi_m,\cdots) \in P \cup P_m \\ \rho = \mathit{rename}(\hat{\tau}, \tau) \uplus \Sigma_{i=1}^n\{\mathit{rename}(\hat{\tau}_i, \tau_i)\} \\ (\Delta \not\approx_U \rho(\phi_i))\downarrow_V \quad \forall i \in 1..k\end{array}}{\Gamma;\Delta \vdash (\ell : m(v_1,\ldots,v_n)) :: \tau,\Delta \circ_U \rho(\Delta_m)}\;[\mathbf{C-Call}]$$

$$\frac{\begin{array}{c}\Gamma = \{v_1 : \hat{\tau}_1,\ldots,v_k : \hat{\tau}_k\} \\ V = \bigcup_{i=1}^k FSV(\hat{\tau}_i) \quad W = V \cup FSV(\hat{\mathtt{t}}) \\ \rho = \mathit{rename}(\tau, \hat{\tau}) \\ \Delta_1 = \mathit{init}(\Gamma) \wedge \phi_{pre} \\ \Gamma;\Delta_1 \vdash e :: \tau,\Delta_2 \quad \Delta = \rho(\Delta_2)\end{array}}{\vdash_C \hat{\tau}\ m(\hat{\tau}_1\ v_1,\ldots\hat{\tau}_n v_k)\ \mathtt{where}\ \Delta,\ \phi_{pre}\ \{e\}}\;[\mathbf{C-Mtd-Declare}]$$

$$\frac{\vdash_C md_i \quad \forall i = \{1,\ldots,n\}}{\vdash_C md_1;\cdots;md_n}\;[\mathbf{C-Program}]$$

Figure 12: Type Checking Rules

with frame numbers. Each time a new frame is created, the current frame number is increased by 1. Frame numbers are useful in making all variables distinct in the stack without having to resort to variable renaming. Main operators are:

Start a new frame :
$\quad \mathit{newframe}(f, n) = (f, n{+}1)$
Push variables into current frame :
$\quad (f, n) + \{(v \mapsto \nu)^*\} = (f + \{(v^n \mapsto \nu)^*\},\ n)$
Pop variables from the topmost frame :
$\quad (f, n) - \{v^*\} =$
$\quad\quad \mathtt{let}\ \hat{f} = f \backslash \{(v^n)^*\}\ \mathtt{in}\ (\hat{f},\ \mathit{max}\ \{m \mid \exists u \cdot u^m \in \mathit{dom}(\hat{f})\})$

The variable environment Π is such a stackable mapping. We write $\Pi[\nu/v]$ to denote an update of the value of the latest variable v in Π to ν. We write $\Pi + \{v \mapsto \nu\}$ to denote an extension of Π to include a binding of ν to v, while $\Pi - \{v^*\}$ removes a subset of the mappings. Similar notations are used for the update and enhancement of object values and stores.

We require some intermediate expressions for the dynamic semantics to follow through. Our syntaax is thus extended from the original expression syntax as follows:

$$e\ ::=\ \cdots \mid \iota \mid \eta \mid \mathtt{ret}(v^*, \rho, e) \mid \mathtt{error} \mid \mathtt{BndChkErr}$$

The expressions $\mathtt{ret}(v^*, \rho, e)$ is used to capture the result of evaluating a local block, or the result of a method invocation. The list of variables associated with ret is the local variables declared and used byt he block. The set of variables is popped from the stack at the end of the block's evaluation. We also supply ρ to capture the mapping of formal parameters to actual arguments for method call. This is used by the soundness proof.

The dynamic evaluation rules are of the following form:

$$\langle \Pi, \varpi \rangle\ [e] \hookrightarrow \langle \Pi, \varpi \rangle\ [e']$$

Fig 13 provides the specification of dynamic semantics.

Primitive calls (for array operations) are dealt with differently from user-defined methods in the dynamic semantics. Precondition associated with the call is verified. If it is satisfied, the call proceeds; otherwise, a run-time bound-check error (by the rule $[\mathbf{D-Prim-Err}]$) is raised. Checking of preconditions requires determining the size of the operation arguments. This is done by szC, which is defined as follows:

$$\begin{array}{lll} szC_{\langle \Pi, \varpi \rangle}(d) \ =_{df}\ \ \mathit{case}\ d\ \mathit{of} & & \\ \quad (\mathtt{Int}^i\ v) & \rightarrow & (i = \Pi(v)) \\ \quad (\mathtt{Bool}^b\ v) & \rightarrow & (b = \Pi(v)) \\ \quad (\mathtt{Float}\ v) & \rightarrow & \mathit{True} \\ \quad (\hat{\mathtt{t}}[\mathtt{Int}^{n_1},\ldots,\mathtt{Int}^{n_k}]\ v) & \rightarrow & \mathit{let}\ (k, (s_1,\ldots,s_k), f) = \varpi(\Pi(v)) \\ & & \mathit{in}\ \bigwedge_{i=1}^k \{(n_i = s_i)\} \end{array}$$

Lastly, array primitive may change the store and this shall be captured by the function primOp, as shown in the rule below.

$$\frac{\Pi(v) = \delta}{\langle \Pi, \varpi \rangle\, [v] \hookrightarrow \langle \Pi, \varpi \rangle\, [\delta]}\,[\mathbf{D-Var}] \qquad \frac{\langle \Pi, \varpi \rangle\, [e] \hookrightarrow \langle \Pi', \varpi' \rangle\, [e']}{\langle \Pi, \varpi \rangle\, [v = e] \hookrightarrow \langle \Pi', \varpi' \rangle\, [v = e']}\,[\mathbf{D-Assign-1}] \qquad \frac{\Pi(v) = k \quad \Pi' = \Pi\{v \mapsto k\}}{\langle \Pi, \varpi \rangle\, [v = k] \hookrightarrow \langle \Pi', \varpi \rangle\, [()]}\,[\mathbf{D-Assign-2}]$$

$$\frac{\langle \Pi, \varpi \rangle\, [e_1] \hookrightarrow \langle \Pi', \varpi' \rangle\, [e_1']}{\langle \Pi, \varpi \rangle\, [\{t\ v = e_1; e_2\}] \hookrightarrow \langle \Pi', \varpi' \rangle\, [\{t\ v = e_1'; e_2\}]}\,[\mathbf{D-Block-1}] \qquad \frac{\Pi' = \Pi + \{v \mapsto k\}}{\langle \Pi, \varpi \rangle\, [\{\mathtt{t}\ v = k; e_2\}] \hookrightarrow \langle \Pi', \varpi \rangle\, [\mathtt{ret}(v, [\,], e_2)]}\,[\mathbf{D-Block-2}]$$

$$\frac{\langle \Pi, \varpi \rangle\, [e] \hookrightarrow \langle \Pi', \varpi' \rangle\, [e']}{\langle \Pi, \varpi \rangle\, [\mathtt{ret}(v, \rho, e)] \hookrightarrow \langle \Pi', \varpi' \rangle\, [\mathtt{ret}(v, \rho, e')]}\,[\mathbf{D-Ret-1}] \qquad \frac{\Pi' = \Pi - \{v^*\}}{\langle \Pi, \varpi \rangle\, [\mathtt{ret}(v, \rho, \delta)] \hookrightarrow \langle \Pi', \varpi' \rangle\, [\delta]}\,[\mathbf{D-Ret-2}]$$

$$\frac{\Pi(v) = \mathtt{true}}{\langle \Pi, \varpi \rangle\, [(\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2)] \hookrightarrow \langle \Pi, \varpi \rangle\, [e_1]}\,[\mathbf{D-If-1}] \qquad \frac{\Pi(v) = \mathtt{false}}{\langle \Pi, \varpi \rangle\, [(\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2)] \hookrightarrow \langle \Pi, \varpi \rangle\, [e_2]}\,[\mathbf{D-If-2}]$$

$$\frac{\begin{array}{c}(\tau\ m(\tau_1\ x_1, \ldots, \tau_p\ x_p)\ \mathtt{where}\ \Delta,\ \phi_{pre}) \in P_m \\ \bigwedge_{i=1}^{p}\{szC_{\langle \Pi, \varpi \rangle}(\tau_i\ v_i)\} \ \Rightarrow\ \phi_{pre} \\ (\varpi', k) = primOp(m, [\Pi(v_1), \ldots, \Pi(v_p)], \varpi)\end{array}}{\langle \Pi, \varpi \rangle\, [m(v_1, \ldots, v_p)] \hookrightarrow \langle \Pi, \varpi' \rangle\, [k]}\,[\mathbf{D-PrimCall}] \qquad \frac{\begin{array}{c}(\tau\ m(\tau_1\ x_1, \ldots, \tau_p\ x_p)\ \mathtt{where}\ \Delta,\ \phi_{pre}) \in P_m \\ \bigwedge_{i=1}^{p}\{szC_{\langle \Pi, \varpi \rangle}(\tau_i\ v_i)\} \ \not\Rightarrow\ \phi_{pre}\end{array}}{\langle \Pi, \varpi \rangle\, [m(v_1, \ldots, v_p)] \hookrightarrow \mathtt{BndChkErr}}\,[\mathbf{D-Prim-Err}]$$

$$\frac{\begin{array}{c}(\tau\ m(\tau_1\ x_1, \ldots, \tau_p\ x_p)\ \mathtt{where}\ \Delta, \phi_{pre}\ \{e\}) \in P \\ \rho = \{x_i \mapsto v_i\}_{i=1}^{p} \quad \Pi' = \Pi + \{x_i \mapsto \Pi(v_i)\}_{i=1}^{p}\end{array}}{\langle \Pi, \varpi \rangle\, [m(v_1, \ldots, v_p)] \hookrightarrow \langle \Pi', \varpi \rangle\, [\mathbf{ret}([x_1..x_p], \rho, \{e\})]}\,[\mathbf{D-Call}]$$

Figure 13: Dynamic Semantics

B.3 Soundness of Type Checking

In this section, we show the type-checking rules are sound with respect to the dynamic semantics. This means a well-typed $\text{I}\textsc{mp}_T$ program will not raise a `BndChkErr` error.

We extend the static semantics of $\text{I}\textsc{mp}_T$ to include the intermediate expressions,

$$e \ ::= \ \cdots \mid \iota \mid \eta \mid \mathtt{ret}(v^*, \rho, e)$$

In the process, we require introduction of a *store typing* to describe the type of each location. This ensures that objects created in the store during run-time are type-wise consistent with that captured by the static semantics. Store typing is conventionally used to link static and dynamic semantics. In our case, it is denoted by:

$$\Sigma \in \mathit{StoreType} = \mathit{Location} \rightarrow_{\mathit{fin}} \mathit{Type}$$

Judgements in the static semantics will be extended with store typing, as follows:

$$\Gamma; \Sigma; \Delta_1 \vdash e :: t, \Delta_2$$

The static semantics for the new intermediate expressions are shown in Fig 14.

The soundness of our type-checking rules relies on the following consistency relationship between the static and dynamic semantics.

$$\frac{}{\Gamma; \Sigma; \Delta \vdash \iota :: \Sigma(\iota), \Delta}\,[\mathbf{Loc}]$$

$$\frac{\begin{array}{c}\mathit{fresh}\ s_1, \ldots, s_n, \hat{\mathtt{t}} \quad f :: \mathtt{Int} \ldots \mathtt{Int} \rightarrow\ \mathtt{t} \\ \Delta_1 = \Delta \wedge \bigwedge_{i=1}^{n}(s_i' = k_i)\end{array}}{\Gamma; \Sigma; \Delta \vdash (n, [k_1, \ldots, k_n], f) :: \mathtt{t}[\mathtt{Int}^{s_1}, \ldots, \mathtt{Int}^{s_n}], \Delta_1}\,[\mathbf{Array}]$$

$$\frac{\begin{array}{c}\{v_1, .., v_n\} \subseteq dom(\Gamma) \quad \Gamma; \Sigma; \Delta \vdash e :: \tau, \Delta_1 \\ \tau_i = \Gamma(v_i) \quad X_i = FSV(\tau_i)\ \forall i = 1..n \\ Y = \cup_{i=1}^{n}\{X_i \cup prime(X_i)\} \\ \rho_1 = \uplus_{i=1}^{n}\{rename(\tau_i, \hat{\tau}_i) \mid \hat{\tau}_i = \Gamma(\rho(v_i)), i = 1..n\}\end{array}}{\Gamma; \Sigma; \Delta \vdash \mathtt{ret}([v_1, .., v_n], \rho, e) :: \hat{\tau}, \exists Y \cdot (prime(\rho_1)(\Delta_1))}\,[\mathbf{Ret}]$$

Figure 14: Type Rules for Intermediates

$$\frac{\begin{array}{c}dom(\Pi) = dom(\Gamma) \quad dom(\Sigma) = dom(\varpi) \quad X = \mathcal{I}(\Gamma) \cup \mathcal{I}(\Sigma) \\ \forall v \in dom(\Pi) \cdot \Gamma; \Sigma; \mathtt{true} \vdash \Pi(v) : \hat{\mathtt{t}}_v, \Delta_v \\ \forall \iota \in dom(\varpi) \cdot \Gamma; \Sigma; \mathtt{true} \vdash \varpi(\iota) : \hat{\tau}_\iota, \Delta_\iota \\ C_1 = \bigwedge_{v \in dom(\Pi)}(\Delta_v \wedge equate(prime(\Gamma(v)), \hat{\mathtt{t}}_v) \\ C_2 = \bigwedge_{\iota \in dom(\varpi)}(\Delta_\iota \wedge equate(prime(\Sigma(\iota)), \hat{\tau}_\iota) \\ Y = \bigcup_{v \in dom(\Gamma)} FSV(\hat{\mathtt{t}}_v) \ \cup\ \bigcup_{\iota \in dom(\Sigma)} FSV(\hat{\tau}_\iota) \\ \exists Y.C_1 \wedge C_2 \implies \exists X.\Delta\end{array}}{\Gamma, \Sigma, \Delta \vDash \langle \Pi, \varpi \rangle}$$

The following main theorem states that each well-typed expression preserves its type under reduction with a run-time environment and a store which are consistent with the compile-time counterparts. Furthermore, the final size constraint is consistent with the value obtained on termination.

THEOREM 2 (PRESERVATION).

(a) (Value) If

$$\Gamma; \Sigma; \Delta \vdash \delta :: \mathtt{t}, \Delta_1$$
$$\Gamma; \Sigma; \Delta \vDash \langle \Pi, \varpi \rangle$$

then the following holds:

$$\Gamma + \{x :: t\}; \Sigma; \Delta_2 \vDash \langle \Pi + \{x \mapsto \delta\}, \varpi \rangle$$

where $x = \mathit{fresh}()$; $\Delta_2 = [v \mapsto v']_{v \in FSV(t)} \Delta_1$.

(b) (Expression) If

$$\Gamma; \Sigma; \Delta \vdash e :: \tau, \Delta_1$$
$$\Gamma; \Sigma; \Delta \vDash \langle \Pi, \varpi \rangle$$
$$\langle \Pi, \Sigma \rangle [e] \hookrightarrow \langle \Pi', \varpi' \rangle [e']$$

then there exists Σ', Γ' *and* Δ' *such that*

$$\Gamma - \mathit{diff}(e, e') = \Gamma' - \mathit{diff}(e', e)$$
$$\Sigma' \supseteq \Sigma$$
$$\Gamma'; \Sigma'; \Delta' \vdash e' :: t, \Delta_1$$
$$\Gamma'; \Sigma'; \Delta' \vDash \langle \Pi', \varpi' \rangle.$$

Function $\mathit{diff}(e, e')$ returns a list of local variables that appears in e but not e':

$$\begin{array}{lllll} \mathit{diff}(e, e') =_{df} & \mathit{let} & \mathit{lst} & = & \mathit{local}(e) \\ & & \mathit{lst1} & = & \mathit{local}(e') \\ & & n & = & \mathit{length}(\mathit{lst}) - \mathit{length}(\mathit{lst1}) \\ & \mathit{in} & \multicolumn{3}{l}{(\mathit{take}(n, \mathit{lst}) \lhd n \geq 0 \rhd [\,])} \end{array}$$
$$\mathit{take}(n, \mathit{lst}) =_{df} ([\,] \lhd n \leq 0 \rhd [\mathit{hd}(\mathit{lst})] \mathbin{+\!\!+} \mathit{take}(n-1, \mathit{tl}(\mathit{lst})))$$

Function $\mathit{local}(e)$ returns a list of sets of local variables. It is defined as follows:

$$\begin{array}{lll} \mathit{local}(e) =_{df} \ \mathit{case}\ e\ \mathit{of} & & \\ \quad \mathtt{ret}(v^*, \rho, e) & \rightarrow & \mathit{local}(e) \mathbin{+\!\!+} [\{v^*\}] \\ \quad w = e & \rightarrow & \mathit{local}(e) \\ \quad (t\ v = e_1; e_2) & \rightarrow & \mathit{local}(e_1) \\ \quad \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 & & \\ \quad \delta \mid w \mid \mathtt{new}\ cn(v^*) \mid mn(v^*) & \rightarrow & \emptyset \end{array}$$

Note that $\Gamma - [\,] =_{df} \Gamma$, $\Gamma - ([s] \mathbin{+\!\!+} S) =_{df} (\Gamma - s) - S$.

Proof: (a) This can be proved by establishing that the resulted post-conditions from [C−Const−Int], [C−Const−Bool] and [C−Const−Void−Float] are consistent with the run-time stack and store.

(b) By induction over the depth of type derivationfor expression e that will be evaluated. At each step of the induction, we assume that the desired preservation property holds for each induction step, and proceed by case analysis on the next derivation step. Our induction is carried out on the expected forms of well-typed expressions, as follows: □

THEOREM 3 (PROGRESS). *If*

$$\Gamma; \Delta; \Sigma \vdash e :: \tau, \Delta_1 \ \mathit{and}\ \Gamma, \Delta, \Sigma \vDash \langle \Pi, \varpi \rangle,$$

then e *is either a value, an* `error` *or there exists* Π', ϖ' *such that* $\langle \Pi, \varpi \rangle [e] \hookrightarrow \langle \Pi', \varpi' \rangle [e']$.

Proof: By induction on the height of the type derivation of and expression e.

C. SPECIALIZATION

In this section, we elaborate on two techniques for achieving specialization.

C.1 Monovariant Specialization

In monovariant specialization, we aim to create *at most one* specialized method for each method definition. As this specialized method will be used by all calls in place of the original methods, we can directly modify the original method definition to attain the specialized method. Furthermore, the specialization has to be conservative in order to ensure safe array operation at run-time. That is, *an array check will be residualized in a specialized method if the check has been marked by one or more run-time tests in the program.*

Our approach gathers all run-time tests in the program, and place each of these tests just before the respective array operations. Matching a run-time test to an array operation is easy, as a run-time test is labelled by a sequence of labels, the rightmost of which indicate the location of the corresponding array operation.

Formally, a set of label sequences is defined as follows:

$$\varsigma \ \in \ \mathbf{LSet} \ = \ \mathcal{P}(\mathbf{Label}^+)$$

Label sequences for all the run-time tests can be gathered from an inferred program by the function rtt. Furthermore, we provide a function prim defined as follows:

$$\mathit{prim}(\ell^+) = \ell_1.c \ \text{ where } \ \exists \ell_2^* : (\ell^+ = \ell_2^*.\ell_1.c)$$

We naturally extend prim to operate on a set of label sequences. The specialization of method declaration is now specified as follows:

$$\frac{\begin{array}{c}\varsigma = \mathit{rtt}(\mathtt{P}) \\ \forall\, md_i \in \mathtt{P} \ : \ (\mathtt{P},\, \varsigma \rhd^m_{mono} \ md_i \looparrowright md_i'')\end{array}}{\rhd_{mono}\ \mathtt{P} \looparrowright \bigcup_{md_i \in \mathtt{P}} \{md_i''\}}$$

$$\frac{\begin{array}{c} md = (\hat{\tau}\ m(\hat{\tau}_1\ v_1, \ldots, \hat{\tau}_k\ v_k)\ \mathtt{where}\ \Delta, \Phi, \Upsilon\{e\}) \\ \phi_{pre} = \bigwedge\{\phi \mid (\ell^+ : \phi) \in \Phi \wedge \mathit{prim}(\ell^+) \notin \mathit{prim}(\varsigma)\} \\ \mathtt{P},\, \varsigma \rhd^e_{mono}\ e \looparrowright e_1 \\ md'' = (\hat{\tau}\ m_1(\hat{\tau}_1\ v_1, \ldots, \hat{\tau}_k\ v_k)\ \mathtt{where}\ \Delta, \phi_{pre}\{e_1\}) \end{array}}{\mathtt{P},\, \varsigma \rhd^m_{mono}\ md \ \looparrowright\ md''}$$

Specialization of expression simply recurses down the syntactic structure until a primitive array operation is reached, at which state, the following rule is applied:

$$\frac{\begin{array}{c} \hat{\tau}\ m(\hat{\tau}_1\ x_1, \ldots, \hat{\tau}_n\ x_n) \ \text{ where } \Delta, \Phi, C \ \in \ P_m \\ \rho = [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \\ e_1 = \bigwedge\{\rho\ e \mid \ \ \ell.c \in \mathit{prim}(\varsigma) \wedge (c : e) \in C\} \\ e_2 = \mathtt{if}\ e_1\ \mathtt{then}\ m(v_1, \ldots, v_k)\ \mathtt{else}\ \mathtt{error} \\ e_3 = m(v_1, \ldots, v_k) \lhd (e_1 = \mathtt{true}) \rhd e_2 \end{array}}{\mathtt{P},\, \varsigma \rhd^e_{mono}\ (\ell : m(v_1, \ldots, v_k)) \looparrowright e_3}$$

For array operations, we identify that each check labelled with $\ell.c$ need be residualized when it falls in the set ς. The code for array check located at a primitive method declaration is written in terms of the primitive's parameters. To use it, we substitute these parameters with the arguments of the call. All codes generated are combined as a conjunct, named e_1, which is then wrapped as a runtime test for the primitive call to m. If the runtime set is empty – signified by e_1 being `true`, no conditional will be wrapped for the m call.

The correctness of monovariant specialization can be expressed as follows:

LEMMA 1. *Let $\triangleright_{mono}\ P_I \looparrowright P_T$ and e_I and e_T be the respective main expressions in P_I and P_T, then*

$$\textit{if}\ \langle \Pi_{init}, \varpi_{init}\rangle\,[e_T] \hookrightarrow^{*} \texttt{BndChkErr},$$
$$\textit{then}\ \langle \Pi_{init}, \varpi_{init}\rangle\,[e_I] \hookrightarrow^{*} \texttt{BndChkErr}.$$

C.2 Polyvariant Specialization

In polyvariant specialization, we aim to create *multiple* specialized methods for each method definition, such that different specialized versions of a method have different sets of array checks being eliminated. The challenge here is to ensure that correct specialized versions are being called at various call sites so that no unsafe array operation can occur.

Specializing a method requires information about the set of run-time tests to which calls in the method body may lead. Thus, a specialized method can be identified by a triple comprising the original method name, a set of label sequences associated with relevant run-time tests, and a new method name uniquely defined by the first two components of the triple. We call such a triple a *specialization signature* (or *signature* in short), and a set containing such signatures a *specialization cache* (or *cache* in short). Generation of a method name from an original name and a set of label sequences is defined by a function *genName*.

$$\begin{array}{lcl} (m, \varsigma, m_1) & \in & \mathbf{SSig}\ =\ \mathbf{MName} \times \mathbf{LSet} \times \mathbf{MName} \\ \sigma, \sigma_Y, \sigma_N & \in & \mathbf{SCache}\ =\ \mathcal{P}(\mathbf{SSig}) \end{array}$$

During specialization, two caches are maintained: σ_Y contains all signatures of the specialized codes that have been generated. σ_N contains those from the specialized codes that have not been generated. The specialization begins by putting the signatures for all methods in the program in σ_N. The rules defined in Fig 15 show how specialized codes are generated while signatures are moved from σ_N to σ_Y. Specialization process completes when the content of σ_N is exhausted.

Array operation is specialized in [$\mathbf{Spec{-}Prim}$] by calling the respective primitive method without array checks under the condition that the combined run-time checks for this operation, e_1, is true. This rule is similar to that described in the monovariant specialization. On the other hand, non-array operations, such as user-defined methods, may be specialized with respect to the set of run-time tests (represented by their label sequences ς) to be imposed ([$\mathbf{Spec{-}Call_1}$]). It produces a signature for this specialized method, if the latter has not been recorded in the current cache. Otherwise, it reuses the specialised method that has been recorded previously, as specified in the next rule.

The correctness of polyvariant specialization can be expressed similarly to the monvariant counterpart:

LEMMA 2. *Let $\triangleright_{poly}\ P_I \looparrowright P_T$ and e_I and e_T be the respective main expressions in P_I and P_T, then*

$$\textit{if}\ \langle \Pi_{init}, \varpi_{init}\rangle\,[e_T] \hookrightarrow^{*} \texttt{BndChkErr},$$
$$\textit{then}\ \langle \Pi_{init}, \varpi_{init}\rangle\,[e_I] \hookrightarrow^{*} \texttt{BndChkErr}.$$

D. SOUNDNESS OF TYPE INFERENCE

We rely on the correctness of type-checking rules (defined in Appendix B) and the correctness of specialization (defined in Appendix C) to attain the soundness of our type inference system (defined in Sec 3). Theorem 1 is the consequence of the following lemma:

LEMMA 3 (SOUNDNESS OF TYPE INFERENCE). *Let P be a program such that $P_m \vdash_I\ P \leadsto P_I \mid \mathcal{Q}$ under type inference, and $\triangleright_{mono}\ P_I \looparrowright P_T$ under monovariant specialization. For any triple of method declarations of m in P, P_I and P_T respectively:*

$$\begin{array}{c} md = (\tau\ m(\tau_1\ v_1, \ldots \tau_k v_k)\{e\}) \in P \\ md_I = (\hat{\tau}\ m(\hat{\tau}_1\ v_1, \ldots \hat{\tau}_k v_k)\ \texttt{where}\ m\langle W\rangle, \Phi, \Upsilon\ \{e_I\} \mid \mathcal{Q}) \in P_I \\ md_T = (\hat{\tau}\ m(\hat{\tau}_1\ v_1, \ldots \hat{\tau}_k v_k)\ \texttt{where}\ \Delta,\ \phi_{pre}\ \{e_T\}) \in P_T \end{array}$$

where $\mathcal{Q} = \{m\langle W\rangle = \Delta\}$. If $\vdash_I md \leadsto md_I$, then

$$\vdash_C\ \hat{\tau}\ m(\hat{\tau}_1\ v_1, \ldots \hat{\tau}_n v_k)\ \texttt{where}\ \Delta,\ \phi_{pre}\ \{e_T\}$$

Proof: By induction on the length of type inference derivation of the method body.

[MAIN−POLY]

$$\frac{\begin{array}{c} md_i = (\hat{\tau}_i\ m_i(\hat{\tau}_{i1}\ v_{i1}, \ldots \hat{\tau}_{ik} v_{ik})\ \mathtt{where}\ \Delta_i, \Phi_i, \Upsilon_i\{e_i\}) \quad \forall i \\ \sigma_N = \bigcup_{i=1}^{n} \{(m_i, \Upsilon_i, m_i'') \mid m_i'' = genName(m_i, \Upsilon_i)\} \\ \mathtt{P} = md_1 \cdots md_n \quad \mathtt{P},\ \emptyset,\ \sigma_N\ \rhd^{p}_{poly}\ \emptyset \nrightarrow S \end{array}}{\rhd_{poly}\ \mathtt{P} \nrightarrow S}$$

[Program₂]

$$\frac{\begin{array}{c} md = (\hat{\tau}\ m(\hat{\tau}_1\ v_1, \ldots \hat{\tau}_k\ v_k)\ \mathtt{where}\ \cdots) \in \mathtt{P} \\ sig = (m, \varsigma, m'') \in \sigma_N \quad \mathtt{P},\ \sigma_Y \cup \sigma_N,\ sig\ \rhd^{m}_{poly}\ md\ \nrightarrow\ md'',\ \sigma_N'' \\ \mathtt{P},\ (\{sig\} \cup \sigma_Y),\ (\sigma_N'' \cup \sigma_N \backslash sig) \rhd^{p}_{poly} \{md''\} \cup S \nrightarrow S'' \end{array}}{\mathtt{P},\ \sigma_Y,\ \sigma_N\ \rhd^{p}_{poly}\ S \nrightarrow S''}$$

[Program₁]

$$\frac{\sigma_N = \emptyset}{\mathtt{P},\ \sigma_Y,\ \sigma_N\ \rhd^{p}_{poly}\ S \nrightarrow S}$$

[Spec−Cst−Var]

$$\frac{cv \in \mathbf{Int} \cup \mathbf{Float} \cup \mathbf{Bool} \cup \mathbf{Var}}{\mathtt{P},\ \sigma,\ \varsigma\ \rhd^{e}_{poly}\ cv \nrightarrow cv,\ \emptyset}$$

[Spec−Assign]

$$\frac{\mathtt{P},\ \sigma,\ \varsigma\ \rhd^{e}_{poly}\ e \nrightarrow e',\ \sigma_N}{\mathtt{P},\ \sigma,\ \varsigma\ \rhd^{e}_{poly}\ (v = e) \nrightarrow (v = e'),\ \sigma_N}$$

[Spec−Mtd]

$$\frac{\begin{array}{c} md = (\hat{\tau}\ m(\hat{\tau}_1\ v_1, \ldots, \hat{\tau}_k\ v_k)\ \mathtt{where}\ m\langle W\rangle, \Phi, \Upsilon\ \{e\} \mid \mathcal{Q}) \\ \mathtt{P},\ \sigma,\ \varsigma\ \rhd^{e}_{poly}\ e \nrightarrow e'',\ \sigma_N \quad \phi_{pre} = \{\phi \mid (\ell^{+} : \phi) \in \Phi \wedge \ell^{+} \notin \varsigma\} \\ \mathcal{Q} = \{m\langle W\rangle = \Delta\} \quad md'' = (\hat{\tau}\ m''(\hat{\tau}_1\ v_1, \ldots, \hat{\tau}_k\ v_k)\ \mathtt{where}\ \Delta, \phi_{pre}\{e''\}) \end{array}}{\mathtt{P},\ \sigma,\ (m, \varsigma, m'')\ \rhd^{m}_{poly}\ md\ \nrightarrow\ md'',\ \sigma_N}$$

[Spec−If]

$$\frac{\begin{array}{c} \mathtt{P},\ \sigma,\ \varsigma\ \rhd^{e}_{poly}\ e_1 \nrightarrow e_3,\ \sigma_{N1} \\ \mathtt{P},\ (\sigma \cup \sigma_N'),\ \varsigma\ \rhd^{e}_{poly}\ e_2 \nrightarrow e_4,\ \sigma_{N2} \end{array}}{\begin{array}{l} \mathtt{P},\ \sigma,\ \varsigma\ \rhd^{e}_{poly}\ (\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) \\ \qquad \nrightarrow\ (\mathtt{if}\ v\ \mathtt{then}\ e_3\ \mathtt{else}\ e_3),\ \sigma_{N1} \cup \sigma_{N2} \end{array}}$$

[Spec−Prim]

$$\frac{\begin{array}{c} \hat{\tau}\ m(\hat{\tau}_1\ x_1, \ldots, \hat{\tau}_n\ x_n)\ \ \text{where}\ \Delta, \Phi, C\ \in\ P_m \\ \rho = [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \quad e_1 = \bigwedge\{\rho\ e \mid\ \ \ell.c \in (\varsigma) \wedge (c : e) \in C\} \\ e_2 = \mathtt{if}\ e_1\ \mathtt{then}\ m(v_1, \ldots, v_k)\ \mathtt{else\ error} \\ e_3 = m(v_1, \ldots, v_k) \lhd (e_1 = \mathtt{true}) \rhd e_2 \end{array}}{\mathtt{P},\ \sigma,\ \varsigma\ \rhd^{e}_{poly}\ (\ell : m(v_1, \ldots, v_k)) \nrightarrow e_3,\ \emptyset}$$

[Spec−Call₁]

$$\frac{\begin{array}{c} (\hat{\tau}\ m(\hat{\tau}_1\ x_1, \ldots \hat{\tau}_k x_k)\ \mathtt{where}\ \Delta, \Phi, \Upsilon\ \{e\}) \in \mathtt{P} \\ \varsigma_1 = \{\ell^{+} \mid \ell_1.\ell^{+} \in \varsigma\} \cup \Upsilon \\ (m, \varsigma_1, \_) \notin \sigma \quad m_s = genName(m, \varsigma_1) \end{array}}{\begin{array}{l} \mathtt{P},\ \sigma,\ \varsigma\ \rhd^{e}_{poly}\ (\ell_1 : m(v_1, \ldots, v_k)) \\ \qquad \nrightarrow\ m_s(v_1, \ldots, v_k),\ \{(m, \varsigma_1, m_s)\} \end{array}}$$

[Spec−Call₂]

$$\frac{\begin{array}{c} (\hat{\tau}\ m(\hat{\tau}_1\ x_1, \ldots \hat{\tau}_k x_k)\ \mathtt{where}\ \Delta, \Phi, \Upsilon\{e\}) \in \mathtt{P} \\ \varsigma_1 = \{\ell^{+} \mid \ell_1.\ell^{+} \in \varsigma\} \cup \Upsilon \qquad (m, \varsigma_1, m_s) \in \sigma \end{array}}{\mathtt{P},\ \sigma,\ \varsigma\ \rhd^{e}_{poly}\ (\ell_1 : m(v_1, \ldots, v_k))\ \nrightarrow\ m_s(v_1, \ldots, v_k),\ \emptyset}$$

[Spec−Block]

$$\frac{\begin{array}{c} \mathtt{P},\ \sigma,\ \varsigma\ \rhd^{e}_{poly}\ e_1 \nrightarrow e_3,\ \sigma_{N1} \\ \mathtt{P},\ (\sigma \cup \sigma_{N1}),\ \varsigma\ \rhd^{e}_{poly}\ e_2 \nrightarrow e_4,\ \sigma_{N2} \quad \sigma_{N3} = \sigma_{N1} \cup \sigma_{N2} \end{array}}{\mathtt{P},\ \sigma,\ \varsigma\ \rhd^{e}_{poly}\ \{\hat{\tau}\ v\ =\ e_1;\ e_2\} \nrightarrow \{\hat{\tau}\ v\ =\ e_3;\ e_4\},\ \sigma_{N3}}$$

Figure 15: Rules for Polyvariant Specialization