Stateful Manifest Contracts

Abstract

We integrate static and dynamic contract-based verification of stateful programs. Although contracts are useful to reason about programs, naive introduction of mutable references makes the reasoning difficult because state-dependent contracts that have been successfully checked can be invalidated by state mutation. The problem is more acute in manifest contract systems, in which contracts are part of static types, in that the naive introduction of assignments would break their important soundness properties.

We address this problem by designing a manifest contract system where specifications of pure code fragments are represented by refinement types and specifications of impure ones are by *computational Hoare types*—a variant of Nanevski et al's Hoare types—which express pre- and postconditions of stateful computations in the same language as programs. To prevent troubles caused by abusing references in contracts, we introduce a region-based effect system, which allows contracts in refinement types and Hoare types to manipulate references as long as they are *observationally* pure and read-only, respectively. We show that our calculus has the same kind of soundness property as existing pure manifest calculi and, furthermore, stateful computations satisfy their postconditions if they terminate without dynamic contract violation.

Following Belo et al., static verification in this work is "post facto", that is, we define the manifest contract system so that all contracts are checked at run time, formalize what dynamic checks can be eliminated safely, and show that programs with and without such checks are contextually equivalent—intuitively, checks of contracts can be eliminated when their satisfaction is derived from other, already established contracts. We also apply the idea of post facto verification to *region-based* local reasoning, which shows that satisfaction of preconditions which do not refer to references mutated by a computation is preserved even after executing the computation.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Programming by contracts; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: [Assertions]

General Terms Languages, Design, Theory, Verification

Keywords contracts, refinement types, computational effects, dynamic verification, assertion

[Copyright notice will appear here once 'preprint' option is removed.]

1. Introduction

1.1 Software Contracts with States

In many programming languages, such as C/C++, Java, Python, JavaScript, ML, etc., data can be stored to and retrieved from mutable memory cells and such imperative programming features are crucial to design many efficient data structures and algorithms. Stateful (and even stateless) data structures and algorithms often rest on certain specifications to guarantee correctness of their behavior—e.g., the binary search algorithm on arrays demands that the input array be sorted to find a correct answer. Although static verification methods for stateful programs have been extensively studied, verification of large-scale practical programs with states is still very challenging. Thus, in practice, software development often uses dynamic verification.

Software contracts are a common tool used broadly for dynamic verification of stateful programs. Roughly speaking, contracts are program specifications written in the same programming language as the target program and support a mechanism to check given specifications at run time. Most languages with imperative features provide contract systems as a standard library or a dedicated feature: the C language supports the `assert` macro, which checks a given Boolean expression and raises a run-time error if it fails; Eiffel, which advocates "Design by Contracts" [28], can check pre- and postconditions of methods and class invariants; Racket provides various advanced contracts such as higher-order contracts [14].

Software contracts are useful, especially in pure languages, to reason about program behavior—e.g., one would use an integer as a divisor if it has been checked successfully by the contract that accepts only positive integers—but naive introduction of mutable references makes such reasoning difficult because state-dependent contracts that have been successfully checked can be invalidated by assignment afterwards and, thus, it is often hard to grasp which contracts are still valid and which are not at a given program point. To see the problem, consider a string table, implemented by a mutable reference to a string list whose elements are pairwise distinct, and a function `add` to add a new string to a given table. A contract for `add` would specify that (1) the given table points to a list of unique strings; (2) the given string is not a member of the old table; and (3) after calling `add`, the added string really becomes a member of the updated table. It is easy to check these conditions at run time: (1) and (2) can be checked at the beginning of `add` as preconditions and (3) can be checked at the end as a postcondition. When (3) has been checked successfully, one may reason about the table by using the contract information that the table contains some string. For example, the table may be expected to be nonempty. However, it does not always hold because (3) will be invalidated if, say, the table is cleaned, whereas some contracts, e.g., (1), are still valid after the clean.

To make the problem more acute, we introduce manifest contract systems [3, 15, 19, 25, 38, 39], in which contracts are part of static types. Manifest contract systems are equipped with refinement types of the form `{x:T|e}`, which means a value `v` (of type `T`) satisfying the Boolean expression, also called a contract or refine-

```
type tbl = { l:string list | uniq l } ref

val add : t:tbl ->
          s:{s:string|not (mem !t s)} ->
          {u:unit|mem !t s}
val size  : t:tbl -> {x:int|x = length !t}
val clear : tbl -> unit

let t : tbl = ...
let s : {s:string|not (mem !t s)} = ...
let x : {u:unit|mem !t s} = add t s

let _ = f ()
let rate = 1 / (size t)
```

Figure 1. A program using functions for string table

ment, `e`, i.e., `{v/x}e` reduces to `true`. For example, if `x` is given `{x:int| x > 0}`, then `x` is positive *everywhere it is referenced*. However, naive introduction of assignments would break this important property of a manifest contract system.

We rephrase the problematic scenario above by using the code in Figure 1, which is written in an ML-like language with manifest contracts. The code generates a string table, calls `add` discussed above with the table, and produces the reciprocal of the size of the table after computing something by calling f. From the interpretation of refinement types above, it is not hard to see the meaning of `add`'s type agrees with the contract for `add`. Notice that `add` is given a dependent function type—dependent function types `x:T1->T2` mean functions which, when taking values `v` of `T1`, return values of `{v/x}T2`. One should expect from a sound manifest contract system that `mem !t s` evaluates to `true` everywhere in the scope of `x` and the last division will always succeed since the table would be nonempty and `size` would return the size of the table. However, it is not quite true here, because `f` may call `clear`! Even worse, manifest contracts with such an inconsistent contract system, where values may not satisfy refinements of their types, would be unsound [18, 39].

A lesson from the discussion above is that state-dependent contracts are necessary for specifying behavior of stateful computations but they may be invalidated during program execution and should not appear in refinement types. Although indeed there is extensive work which studies contracts for imperative programs [11, 12, 16, 20, 44], their work is not very satisfactory because they (1) focus on how to enforce contracts for references but not on abuse of references in contracts, or (2) restrict contracts excessively. For example, Flanagan, Freund, and Tomb [16] have addressed the problem by allowing only pure expressions as refinements, but contracts for operations on mutable data structures such as `add` would not be given because they are usually state-dependent.

1.2 Our Work

The goal of this paper is to introduce a sound yet expressive manifest contract calculus with mutable states, in particular, to deal with specifications about states, e.g., the property of `add` as discussed above. To express such specifications, we develop *computational Hoare types*, a variant of Hoare types in Nanevski et al.'s Hoare Type Theory (HTT) [31, 32]—unlike HTT, the specifications in computational Hoare types are contracts, that is, *executable* computations—while state-independent specifications can be represented by refinement types as usual. For short, computational Hoare types are simply called Hoare types if it is clear from the context that the computational version is meant.

We introduce a manifest contract calculus $\lambda^{\mathrm{H}}_{\mathsf{ref}}$, where, using Hoare types, the type system tracks what state-dependent contracts hold and what may have been invalidated. This characteristic of the type system solves the problem in Section 1.1 because the type system finds out that the contract `mem !t s` may be invalidated after calling `f` and so the dynamic check to verify whether the size of `t` is nonzero is needed before calculating the reciprocal, unless `f` ensures that `t` still contains some strings after executing it.

What follows describes our contributions in more detail.

Hoare types with state-dependent contracts Hoare types are a kind of dependent types inspired by Hoare triples in Hoare logic [21]. A Hoare triple $\{P\}\,c\,\{Q\}$ means that stateful computation c demands that precondition P hold and, after computing c, guarantees that postcondition Q holds. Similarly, Hoare types, written as $\{A_1\}x{:}T\{A_2\}$ in this work, denote stateful computations which assume precondition A_1 before their execution, return values of T, and ensure postcondition A_2 after their execution; variable x represents the return values of the computations in A_2. A_1 and A_2 are possibly *state-dependent* contracts and program states have to satisfy these contracts before and after executing computations of the Hoare type. From a denotational point of view, computations of the Hoare type can be interpreted as functions over states such that they take states satisfying A_1 and return states satisfying A_2 with some value x of T. Using Hoare types, for example, the specification given to `add` in Figure 1 can be presented as:

$$t{:}\mathsf{tbl} \rightarrow s{:}\mathsf{string} \rightarrow \{\mathsf{not}\,(\mathsf{mem}\,!t\;s)\}x{:}\mathsf{unit}\{\mathsf{mem}\,!t\;s\},$$

which describes that `add` accepts a string table t and a string s and insists that s is not a member of t in the state in calling `add`; when these all specifications are met, `add` computes something to result in the state where t contains s. Following HTT, $\lambda^{\mathrm{H}}_{\mathsf{ref}}$ classifies programs into pure and impure fragments in the monadic style [30, 50], and contracts for pure computations are represented by refinement types as in prior manifest calculi and ones for impure computations are by Hoare types.

Dynamic checking All contracts in $\lambda^{\mathrm{H}}_{\mathsf{ref}}$ are checked at run time by using either of two mechanisms: *casts* (called also type conversion), a customary means in manifest contracts to check refinements in pure fragments, and *assertions*, a new means to check pre- and postconditions of stateful computations—in both mechanisms, if a dynamic check fails, an uncatchable exception (called *blame* [14]) will be raised to notify contract violation. We extend the cast-based mechanism to reference types and Hoare types—casts for these types produce wrappers which check properties about states when they evaluate in impure fragments. For reference types, we use the idea of views and guards in the earlier work [11, 16, 20, 44]. $\lambda^{\mathrm{H}}_{\mathsf{ref}}$ also provides an assertion-based checking mechanism to check pre- and postconditions at run time. When computation c needs some preconditions to be satisfied, we can check them at run time before executing c; if they hold, the remaining computation proceeds; otherwise, an uncatchable exception is raised. On the contrary, when c is required to ensure postconditions, we can also check them at run time with the execution result of c; if they hold, the whole computation returns the result of c; otherwise, an uncatchable exception is raised.

Type-and-effect system We design a type system of $\lambda^{\mathrm{H}}_{\mathsf{ref}}$ to not only ensure that values of refinement types satisfy their refinements, which is a key property in manifest contracts [18, 38, 39], but also identify how long state-dependent contracts that have been checked remain true. We hope that contracts are as expressive as possible, but computations with assignments to references in programs should not be accepted as contracts because, intuitively, contracts are specifications and so should not affect the program behavior and, technically, type soundness needs that run-time contract checking be recalculatable, at least immediately after ending the check, but such assignments would make it impossible. To prevent troubles caused by assignments in contracts, we introduce a region-based effect system [7, 26, 47], which allows contracts in refinement types and Hoare types to manipulate references as long as they are *observationally* pure and read-only, respectively—that is, refinements can manipulate only locally allocated references and pre- and postconditions can dereference any memory cell in addition to arbitrary manipulation of locally allocated references. In this paper, we simply call such contracts pure and read-only.

Static verification of state-dependent contracts The effect system is also useful for static verification of state-dependent contracts. Following Belo et al., this work studies "post facto" static verification, that is, we define $\lambda^{\mathrm{H}}_{\mathsf{ref}}$ so that all contracts are checked at run time, formalize what dynamic checks can be eliminated safely, and show that programs with and without such checks are contextually equivalent. Intuitively, dynamic checks of contracts can be eliminated when their satisfaction is derived from other, already established contracts. For example, computations ensuring postcondition $\texttt{mem}\,!t\,\texttt{"foo"}$ also guarantee another postcondition $(\texttt{length}\,!t) > 0$ without additional assertions *if it is proven that the former implies the latter*—although what it says would seem trivial, showing soundness of static verification in manifest contracts is usually not easy [3, 25, 39].

We also apply post facto verification to *region-based* local reasoning, which is inspired by the frame rule in Separation Logic [34, 36] and means that satisfaction of preconditions which do not refer to references mutated by a computation is preserved even after executing the computation. The local reasoning would enable modular verification because we can verify subcomponents of a program without having to know the entire contract of the program.

Although, unlike the original work of manifest contracts [3, 15, 25], this work does not study static verification of refinements due to the difficulty from higher-order types, we expect that our work would be a foundation for such verification.

Organization The rest of this paper is organized as follows: we describe an overview of $\lambda^{\mathrm{H}}_{\mathsf{ref}}$ in Section 2; Sections 3, 4, and 5 offer the program syntax, the type system, and the operational semantics of $\lambda^{\mathrm{H}}_{\mathsf{ref}}$, respectively, and Section 6 shows type soundness after extending the type system with run-time typing rules. After showing static verification in Section 7, we discuss related work in Section 8 and conclude in Section 9.

We omit the proofs of our technical development from this paper; interested readers can refer to the supplementary material.

2. Overview of $\lambda^{\mathrm{H}}_{\mathsf{ref}}$

This section gives an overview of $\lambda^{\mathrm{H}}_{\mathsf{ref}}$, focusing on four key ideas: first, classification of programs to pure and impure fragments in the monadic style; second, computational Hoare types; third, a region-based effect system; and finally, an assertion-based mechanism to check pre- and postconditions of Hoare types at run time.

2.1 Terms and Computations in Monadic Style

Following HTT, the program syntax of our calculus consists of two syntax classes: terms, which are considered as "almost" pure program fragments and can be typed at refinement types, and computations, which are considered as impure program fragments and can be typed at Hoare types. Terms do not manipulate references but include cast applications, so they raise exceptions if some cast fails; this is the reason why terms are "almost" pure. To classify programs into terms and computations, we adopt the monadic style [30, 50], which is a well-known syntactic discipline to distinguish between program fragments with and without computational effects.

In fact, we would meet difficulties without any mechanism to distinguish pure and impure computation. One issue is how to deal with dependent types. Let us consider function application $e_1\ e_2$ where e_1 and e_2 are typed at dependent function type $x{:}T_1 \rightarrow T_2$ and type T_1. We expect the result type of $e_1\ e_2$ to be return type T_2, but T_2 is dependent on the argument variable x of T_1. Thus, a standard typing rule [43] gives $e_1\ e_2$ type $[\,e_2/x\,]\ T_2$ with substitution of e_2 for x. However, this approach is unsound if term e involves stateful effects. This problem has been addressed by using existential quantifier to hide information of substituted terms [13, 24]; however, this remedy cannot be directly applied to our calculus—hidden information cannot be recovered, unlike Knowles and Flanagan [24], due to effects in terms, and, even worse, it is very difficult (possibly impossible) to check specifications with existential quantifier at run time in an algorithmic way.

The monadic style solves this problem. Since arguments to functions must be pure, we can adopt the standard substitution-based rule. It is also convenient in formalizing typing rules, thanks to the fact that intermediate results during computation are all named. Another advantage is that it is easy to analyze when state-dependent contracts may be invalidated since the monadic style sequentializes effects, in particular, assignment, which is the only effect that can invalidate them.

Terms Terms e are mostly from the lambda calculus and manifest contracts. The most distinguishing construct from manifest contracts is casts $\langle T_1 \Leftarrow T_2 \rangle^{\ell}$, which check that, when applied to values of source type T_2, they can behave as target type T_1. For example, since 3 is positive, the cast application $\langle \{x{:}\mathsf{int} \mid x{>}0\} \Leftarrow \mathsf{int}\rangle^{\ell}\, 3$ succeeds and returns 3 whereas, since 0 is not positive, $\langle \{x{:}\mathsf{int} \mid x > 0\} \Leftarrow \mathsf{int}\rangle^{\ell}\, 0$ gives rise to an uncatchable exception $\Uparrow\ell$ with label ℓ to notify contract violation (the label ℓ is used to identify the program point with the cast). Terms also contain thunks, an usual construct in monadic languages. Thunks, written as $\mathsf{do}\ c$ using computation c, are introduced to deal with computations in the context of terms. Using thunks, for example, we can write functions with stateful effects. Thunks will be executed when they are connected with the top-level computation.

Computations Computations, denoted by c, are constructed as usual by two constructs: return and bind. The return construct $\mathsf{return}\ e$ returns the evaluation result of term e as a computation result. The bind construct $x \leftarrow e_1; c_2$ evaluates term e_1 to thunk $\mathsf{do}\ c_1$, and then computes c_1 and c_2 sequentially. The computation c_2 can refer to the result of c_1 as x. Moreover, since our interest is in stateful programs, computations can deal with operations on mutable references. Such operations are written as

$$x \Leftarrow \mathsf{ref}\ e_1; c_2 \qquad x \Leftarrow !e_1; c_2 \qquad x \Leftarrow e_1 := e_1'; c_2$$

which represent memory allocation, dereference, and assignment, respectively. After doing the corresponding action, they compute c_2 by binding x to the result of the action. The assignment action returns the unit value.

Example Let us consider a function that takes a reference that points to an integer value, increments the contents, and then returns the old value of the reference. Such a function can be written as:

$$f \stackrel{\mathrm{def}}{=} \lambda x{:}\mathsf{Ref\ int}.\mathsf{do}\ y \Leftarrow !x;\ \_ \Leftarrow x := y + 1; \mathsf{return}\ y$$

Here, $\mathsf{Ref\ int}$ is the type of integer references and "_" means an unused variable. We can call f by passing a reference to an integer:

$$x \Leftarrow \mathsf{ref}\ 1; y \leftarrow f\ x; z \Leftarrow !x; \mathsf{return}\ y + z$$

This computation returns integer 3 because the contents of x are updated to 2 and f returns the old value 1.

2.2 Hoare Types

Hoare types have pre- and postconditions of stateful computations. In our work, these conditions are sequences A of Boolean computations. Intuitively, condition A means that the conjunction of all contracts in A; since the empty sequence means the contract which always hold, we write $\top$ for it. Formalizing pre- and postconditions as sequences of computations simplifies the metatheory of our calculus—e.g., it allows strengthening preconditions and weakening postconditions in a natural way. Computations of Hoare type $\{A_1\}x{:}T\{A_2\}$ demand A_1, produces values v of type T (if any), and ensures $[\,v/x\,]\,A_2$.

Example Let us consider a contract that a computation requires reference x to point to an integer list each of which is a prime number, add new prime numbers to x, and returns the length of the result list. Using Hoare types, the contract is given as follows:

$$\begin{array}{l}\{y \Leftarrow !x; \mathsf{return}\,(\mathtt{for\_all\,prime?}\,y)\}\\ \quad z{:}\mathsf{int}\\ \{y \Leftarrow !x; \mathsf{return}\,(\mathtt{for\_all\,prime?}\,y)\ \mathtt{\&}\ (\mathtt{length}\,y = z)\}\end{array}$$

where $\mathtt{for\_all\,prime?}\,y$ returns whether each element in integer list y is a prime number and $\mathtt{length}$ returns the length of the argument list. The pre- and postcondition are allowed to read an integer list from reference x since contracts of Hoare types can be state-dependent, unlike refinement types. Similarly, add in Figure 1 can be given:

$$\begin{array}{l}t{:}\mathsf{tbl} \rightarrow s{:}\mathsf{string} \rightarrow\\ \{x \Leftarrow !t; \mathsf{return\,not}\,(\mathtt{mem}\,x\,s)\}y{:}\mathsf{unit}\{x \Leftarrow !t; \mathsf{return}\,(\mathtt{mem}\,x\,s)\}.\end{array}$$

2.3 Region-Based Effect System

As mentioned in Section 1.2, we have to restrict refinements to pure computations and pre- and postconditions to read-only computations. One naive approach to it is to restrict refinements to terms and pre- and postconditions to computations involving with only return and dereference constructs (binds may introduce assignments as thunks). However, it limits expressive power of contracts excessively. For example, it would disallow contracts to reuse imperative libraries, e.g., hash tables and regular expression matching [27].

Our solution to reconciling restriction and expressive power of contracts is to design an effect system with locally scoped *regions* [26, 47]. The effect system accepts pure computations, which manipulate only locally allocated references, as refinements and read-only computations, which do not apply assignment to references allocated in programs, as pre- and postconditions.

To observe whether contracts are independent of references in a program, our effect system tracks which program point references are allocated at, using a region, which intuitively identifies a program point, in a standard manner [7, 26, 47]. Regions, denoted by r, are introduced by let-region $\nu r.\, c$ [7, 47], which computes c under the local region r. If c manipulates only references allocated in r, it appears to be pure for a client of the let-region. Similarly, if it does not write data to references allocated in other regions, it appears to be read-only. Which regions references are allocated at is embedded into reference types, as usual in the work on region calculi with states [7, 26]. Reference types $\mathsf{Ref}_r\,T$ denote references which point to contents of T and are allocated at r.

To ensure that refinements are pure and pre- and postconditions are read-only, we track what operations contracts apply to references as effects and check that forbidden effects are not involved. Since all contracts are computations, effects in contracts are given to Hoare types. In the effect system, Hoare types take the form $\{A_1\}x{:}T\{A_2\}^{\langle\gamma_\mathtt{r},\gamma_\mathtt{w}\rangle}$, where $\gamma_\mathtt{r}$ and $\gamma_\mathtt{w}$ are sets of regions whose references are readable and writable in computations of the Hoare type, respectively. Refinements have to be typed at $\{\top\}x{:}\mathsf{bool}\{\top\}^{\langle\emptyset,\emptyset\rangle}$, where: $\gamma_\mathtt{r} = \gamma_\mathtt{w} = \emptyset$ means that the refinements have to be pure; $A_1 = \top$ that the refinements cannot suppose any precondition, since values of the refinement type must be copiable to any contexts; and $A_2 = \top$ that it is not needed to guarantee any postcondition. Pre- and postconditions have to be typed at $\{A_1\}x{:}\mathsf{bool}\{\top\}^{\langle\gamma_\mathtt{r},\emptyset\rangle}$ for some A_1 and $\gamma_\mathtt{r}$, where: $\gamma_\mathtt{w} = \emptyset$ means that the conditions have to be read-only—in other words, they are allowed to read data from references in regions $\gamma_\mathtt{r}$; and the conditions can assume some A_1 established before checking them.

Moreover, to enhance reusability of program components, we introduce region abstractions [26, 47], which are abstracted over region variables, so they can be used in any contracts. Region polymorphic types $\forall r.\, T$ are types for region abstractions.

Example The effect system allows many efficient algorithms and data structures with mutable references to be reused in contracts as well as programs. For example, using the let-region construct, an efficient implementation of regular expression matching, which often rests on mutable references, would be given the following type because it would rest on only locally allocated references:

$$\mathsf{string} \rightarrow \mathsf{string} \rightarrow \{\top\}x{:}\mathsf{bool}\{\top\}^{\langle\emptyset,\emptyset\rangle},$$

where the first and second arguments mean regular expressions and target strings, respectively. The Hoare type means that the regular expression matching is pure, so contracts can use it.

Moreover, the effect system enables contracts to mention "hidden states" of an abstracted data structure via interface functions for it. For example, functions $\mathtt{mem}$ and $\mathtt{add}$ for an abstracted version of tbl would be given the following types for some r:

$$\begin{array}{lcl}\mathtt{mem} & : & \mathsf{tbl} \rightarrow \mathsf{string} \rightarrow \{\top\}x{:}\mathsf{bool}\{\top\}^{\langle\{r\},\emptyset\rangle}\\ \mathtt{add} & : & t{:}\mathsf{tbl} \rightarrow s{:}\mathsf{string} \rightarrow \{y \leftarrow \mathtt{mem}\,t\,s; \mathsf{return\,not}\,y\}\\ & & \qquad x{:}\mathsf{unit}\\ & & \qquad \{y \leftarrow \mathtt{mem}\,t\,s; \mathsf{return}\,y\}^{\langle\{r\},\{r\}\rangle}\end{array}$$

where read and write effects in $\mathtt{mem}$ are $\{r\}$ and $\emptyset$, respectively, because it would dereference a string table pointer at r but would not update it, and both of read and write effects in $\mathtt{add}$ are $\{r\}$ because it would add a string to a string table after dereference. Under the naive syntactic restriction, this $\mathtt{add}$ would be rejected because the contracts for it use bind constructs.

2.4 Run-time Checking of Pre- and Postconditions

To check pre- and postconditions of Hoare types, we provide a new computation construct, assertions. Assertion $\mathsf{assert}\,(c_1)^\ell; c_2$ first checks contract c_1 and then: if the contract checking succeeds, i.e., c_1 returns true, the remaining computation c_2 will be executed; otherwise, if it fails, an uncatchable exception $\Uparrow\ell$ will be raised. Perhaps assertions might appear to be able to check only preconditions, but they can also be used to check postconditions. For example, we can write a computation that first executes c_1 and then check postcondition c_2, using assertions, as $x \leftarrow \mathsf{do}\,c_1; \mathsf{assert}\,(c_2)^\ell; \mathsf{return}\,x$. For short, we write this computation as $c_1; \lambda x.\mathsf{assert}\,(c_2)^\ell$.

Example Let us consider whether a computation c satisfies the first contract in Section 2.2, that is, when reference x points to a prime number list, after computing it, the contents of x are still a prime number list and the computation returns the length of the list referred to by x. Using assertions, it is checked as follows:

$$\begin{array}{l}\mathsf{assert}\,(y \Leftarrow !x; \mathsf{return}\,(\mathtt{for\_all\,prime?}\,y))^{\ell_1};\\ c;\\ \lambda z.\mathsf{assert}\,(y \Leftarrow !x;\\ \qquad\qquad \mathsf{return}\,(\mathtt{for\_all\,prime?}\,y)\ \mathtt{\&}\ (\mathtt{length}\,y = z))^{\ell_2}\end{array}$$

Note that the contracts checked by these assertions match the pre- and postcondition of the Hoare type given in Section 2.2. When this computation is executed, first of all the precondition is checked; if the reference x does not refer to a prime number list, exception $\Uparrow\ell_1$ is raised; otherwise, the computation c is performed. If c terminates and returns a value, then, as a postcondition, whether reference x still points to a prime number list and whether the integer result of c is the length of the prime number list are checked. If the checking succeeds, the result of c is returned as the result of the whole computation; otherwise, $\Uparrow\ell_2$ is raised.

As a more interesting example, we consider an implementation of add, the Hoare type of which is given in Section 2.2:

$$\begin{array}{l}\lambda t{:}\mathsf{tbl}.\lambda s{:}\mathsf{string}.\ \mathsf{do}\\ \quad l \Leftarrow\ !t;\\ \quad \mathsf{let}\ l' = \langle\{l'{:}\mathsf{string\ list} \mid \mathtt{uniq}\ l'\} \Leftarrow \mathsf{string\ list}\rangle^{\ell_1}\ (s :: l);\\ \quad \_ \Leftarrow t := l';\\ \quad \mathsf{return}\ ();\\ \quad \lambda\_.\mathsf{assert}\ (x \Leftarrow\ !t; \mathsf{return}\ (\mathtt{mem}\ x\ s))^{\ell_2}\end{array}$$

Here, $\mathsf{let}\ x = e_1; c_2$ is an abbreviation of $x \leftarrow \mathsf{do\ return}\ e_1; c_2$. This function accepts a string table t and a string s and returns a thunk that adds s to t. The thunk first dereferences t and obtains a list l whose elements are distinct from each other. Then, it checks with a cast that the new string s is fresh for l, and updates the string table with the new list if the check succeeds. Finally, the postcondition is checked by the assertion, in which $x \Leftarrow\ !t; \mathsf{return}\ (\mathtt{mem}\ x\ s)$ states that t contains s. This function involves two run-time checks: the cast $\langle\{l'{:}\mathsf{string\ list} \mid \mathtt{uniq}\ l'\} \Leftarrow \mathsf{string\ list}\rangle^{\ell_1}$ and the assertion $\mathsf{assert}\ (x \Leftarrow\ !t; \mathsf{return}\ (\mathtt{mem}\ x\ s))^{\ell_2}$. It is obvious that the latter check always succeeds while whether the former succeeds rests on whether string s is not contained in table t before calling add: if s is *not* in t, the check succeeds; otherwise, it fails. Since s is passed by clients of add, we require them to pass strings which do not occur in t.[1] As a result, the Hoare type of add is given as in Section 2.2.

Contrary to the fact that add ensures the postcondition by assertions, its client does the precondition. For example, a function that extends a given string table with the string "foo" is written as:

$$\begin{array}{l}\lambda t{:}\mathsf{tbl}.\ \mathsf{do\ assert}\ (x \Leftarrow\ !t; \mathsf{return\ not}\ (\mathtt{mem}\ x\ \texttt{"foo"}))^{\ell};\\ \qquad\qquad \_ \leftarrow \mathtt{add}\ t\ \texttt{"foo"};\ \ \mathsf{return}\ ()\end{array}$$

where the precondition of add is checked before calling it.

Fortunately, we do not need the precondition check if it is ensured by the preceding computation. For example, let fresh_str be a function which takes a string table and returns some string not contained in the table. It would be typed at

$$t{:}\mathsf{tbl} \rightarrow\ \ \{\top\}s{:}\mathsf{string}\{x \Leftarrow\ !t; \mathsf{return\ not}\ (\mathtt{mem}\ x\ s)\}.$$

After calling fresh_str, we can omit checking the precondition of add because it is ensured by (the postcondition of) fresh_str:

$$\lambda t{:}\mathsf{tbl}.\ \mathsf{do}\ s \leftarrow \mathtt{fresh\_str}\ t; \_ \leftarrow \mathtt{add}\ t\ s; \mathsf{return}\ ()$$

3. Syntax

We show the program syntax of $\lambda^{\mathrm{H}}_{\mathsf{ref}}$ in Figure 2. The syntax uses various metavariables: B ranges over base types, A lists of pre- and postconditions of computations, T types, k constants, e terms, d commands, c computations. We use x, y, z, etc. as term variables, r and s as region variables, γ as sets of region variables, and ϱ as pairs of region sets. We write $\varrho_1 \cup \varrho_2$ for the element-wise union of ϱ_1 and ϱ_2, $\langle\gamma_{\mathtt{r}}, \gamma_{\mathtt{w}}\rangle \uplus \gamma$ for $\langle\gamma_{\mathtt{r}} \uplus \gamma, \gamma_{\mathtt{w}} \uplus \gamma\rangle$, where $\uplus$ is the union operation defined only when the given two region sets are disjoint, and γ, r for $\gamma \uplus \{r\}$.

Types offer base types, dependent function types, and refinement types from usual manifest calculi, in addition to reference types and Hoare types, which are already described in Section 2.2 in detail. We do not fix base types but assume bool and unit at least

[1] The cast is still left in well-typed programs because static verification of contracts are "post facto" [3].

Variables

x, y, z ::= term variables $\qquad r, s$ ::= region variables
γ ::= finite sets of region variables $\qquad \varrho ::= \langle\gamma_{\mathtt{r}}, \gamma_{\mathtt{w}}\rangle$

Types

$B ::= \mathsf{bool} \mid \mathsf{unit} \mid \dots \qquad A ::= \top \mid A, c$
$T ::= B \mid x{:}T_1 \rightarrow T_2 \mid \{x{:}T \mid c\} \mid \mathsf{Ref}_r\, T \mid \{A_1\}x{:}T\{A_2\}^{\varrho} \mid \forall r.\, T$

Constants, Terms, Commands, and Computations

$k ::= \mathsf{true} \mid \mathsf{false} \mid () \mid \dots$
$e ::= x \mid k \mid op(e_1, \dots, e_n) \mid \lambda x{:}T.e \mid \langle T_1 \Leftarrow T_2\rangle^{\ell} \mid e_1\, e_2 \mid e_1 == e_2 \mid \mathsf{do}\ c \mid \lambda r.e \mid e\{r\}$
$d ::= \mathsf{ref}_r\, e \mid\ !e \mid e_1 := e_2$
$c ::= \mathsf{return}\ e \mid x \leftarrow e_1; c_2 \mid x \Leftarrow d_1; c_2 \mid \nu r.\, c \mid \mathsf{assert}\ (c_1)^{\ell}; c_2$

Figure 2. Program syntax in $\lambda^{\mathrm{H}}_{\mathsf{ref}}$.

for contracts and assignment. Function types $x{:}T \rightarrow T'$, refinement types $\{x{:}T \mid c\}$, and Hoare types $\{A_1\}x{:}T\{A_2\}^{\varrho}$ bind variable x in T', c, and A_2, respectively. Region polymorphic types $\forall r.\, T$ bind region r in T.

Terms are usual lambda terms with constants, casts, pointer equality tests, thunks, region abstractions and applications. Constants include, at least, Boolean values true and false and the unit value (). A pointer equality test expression $e_1 == e_2$ returns whether two pointers e_1 and e_2 are equal. $\lambda r.e$, where r is bound in e, abstracts regions and $e\{r\}$ applies region abstraction e to region r.

Computations consist of return, bind, operations on references, let-region, and assertion. Region r in $\mathsf{ref}_r\, e$ specifies where a memory cell storing e is allocated. $x \leftarrow e_1; c$ and $x \Leftarrow d_1; c$ bind variable x in c; $\nu r.\, c$ binds r in c.

Finally, we introduce usual notations. We write $[\,e'/x\,]\,e$ for capture avoiding substitution of e' for x in e. α-equivalent terms are identified and a term without free term variables is said to be closed. These notions are applied to other syntactic categories such as computations and types. We use function $frv\ (c)$, which returns the set of free region variables in computation c. As shorthand, we write: $T_1 \rightarrow T_2$ for $x{:}T_1 \rightarrow T_2$ when x does not occur free in T_2; and $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ for $(\lambda x{:}T.e_2)\ e_1$ where T is an adequate type.

4. Type System

This section introduces a type system for programs in $\lambda^{\mathrm{H}}_{\mathsf{ref}}$. The goal of the type system is to guarantee that well-typed programs can't go wrong *except for contract violations*, i.e., they evaluate to values, raise exceptions by cast or assertion failure, or diverge. The type system is not strong enough to exclude possible contract violations; however, it does guarantee that result values and result stores of well-typed programs satisfy the contracts on their types. The type system has five judgments defined mutually recursively by rules in Figure 3: typing context well-formedness $\gamma \vdash \Gamma$, type well-formedness $\gamma; \Gamma \vdash T$, assertion well-formedness $\gamma; \Gamma \vdash^{\varrho} A$ (where ϱ stands for effects which may occur in computations of A), term typing judgment $\gamma; \Gamma \vdash e : T$, and computation typing judgment $\gamma; \Gamma \vdash c : \{A_1\}x{:}T\{A_2\}^{\varrho}$. Typing contexts Γ, sequences of term and region variable declarations, are defined in a standard manner:

$$\Gamma ::= \emptyset \mid \Gamma, x{:}T \mid \Gamma, r$$

Region variables declared in Γ are introduced by region abstraction, whereas those in γ are by let-region. The two kinds of region variables are distinguished to reflect the fact that ν-bound variables

$\boxed{\gamma \vdash \Gamma}$ $\boxed{\gamma; \Gamma \vdash T}$ $\boxed{\gamma; \Gamma \vdash^{\varrho} A}$ **Well-Formedness Rules for Typing Contexts, Types, and Assertions**

$$\frac{}{\gamma \vdash \emptyset}\ \text{WF\_EMPTY} \qquad \frac{\gamma \vdash \Gamma \quad \gamma; \Gamma \vdash T}{\gamma \vdash \Gamma, x{:}T}\ \text{WF\_EXTENDVAR} \qquad \frac{\gamma \vdash \Gamma \quad r \notin \gamma}{\gamma \vdash \Gamma, r}\ \text{WF\_EXTENDREGION} \qquad \frac{\gamma \vdash \Gamma}{\gamma; \Gamma \vdash B}\ \text{WF\_BASE}$$

$$\frac{\gamma; \Gamma \vdash T_1 \quad \gamma; \Gamma, x{:}T_1 \vdash T_2}{\gamma; \Gamma \vdash x{:}T_1 \to T_2}\ \text{WF\_FUN} \qquad \frac{\gamma; \Gamma \vdash T \quad \gamma; \Gamma, x{:}T \vdash c \ :\ \{\top\} y{:}\mathsf{bool}\{\top\}^{\langle \emptyset, \emptyset \rangle}}{\gamma; \Gamma \vdash \{x{:}T \mid c\}}\ \text{WF\_REFINE}$$

$$\frac{\gamma; \Gamma \vdash T \quad r \in \gamma \cup \mathit{regions}\,(\Gamma)}{\gamma; \Gamma \vdash \mathsf{Ref}_r\, T}\ \text{WF\_REF} \qquad \frac{\gamma; \Gamma \vdash^{\varrho} A_1 \quad \gamma; \Gamma \vdash T \quad \gamma; \Gamma, x{:}T \vdash^{\varrho} A_2}{\gamma; \Gamma \vdash \{A_1\} x{:}T \{A_2\}^{\varrho}}\ \text{WF\_HOARE} \qquad \frac{\gamma; \Gamma, r \vdash T}{\gamma; \Gamma \vdash \forall r. T}\ \text{WF\_RFUN}$$

$$\frac{\gamma_{\mathtt{r}} \cup \gamma_{\mathtt{w}} \subseteq \gamma \cup \mathit{regions}\,(\Gamma)}{\gamma; \Gamma \vdash^{\langle \gamma_{\mathtt{r}}, \gamma_{\mathtt{w}} \rangle} \top}\ \text{WF\_EMPTYASSERT} \qquad \frac{\gamma; \Gamma \vdash^{\langle \gamma_{\mathtt{r}}, \gamma_{\mathtt{w}} \rangle} A \quad \gamma; \Gamma \vdash c \ :\ \{A\} x{:}\mathsf{bool}\{\top\}^{\langle \gamma_{\mathtt{r}}, \emptyset \rangle}}{\gamma; \Gamma \vdash^{\langle \gamma_{\mathtt{r}}, \gamma_{\mathtt{w}} \rangle} A, c}\ \text{WF\_EXTENDASSERT}$$

$\boxed{\gamma; \Gamma \vdash e \ :\ T}$ **Term Typing Rules**

$$\frac{\gamma \vdash \Gamma \quad x{:}T \in \Gamma}{\gamma; \Gamma \vdash x \ :\ T}\ \text{T\_VAR} \qquad \frac{\gamma \vdash \Gamma}{\gamma; \Gamma \vdash k \ :\ \mathit{ty}\,(k)}\ \text{T\_CONST} \qquad \frac{\begin{array}{c}\gamma \vdash \Gamma \quad \mathit{ty}\,(\mathit{op}) \ =\ x_1{:}T_1 \to \ldots \to x_n{:}T_n \to T \\ \forall\, i \in \{1, \ldots, n\}. \gamma; \Gamma \vdash e_i \ :\ [\, e_1/x_1, \ \ldots, e_{i-1}/x_{i-1} \,]\, T_i\end{array}}{\gamma; \Gamma \vdash \mathit{op}(e_1, \ldots, e_n) \ :\ [\, e_1/x_1, \ \ldots, e_n/x_n \,]\, T}\ \text{T\_OP}$$

$$\frac{\gamma; \Gamma, x{:}T_1 \vdash e \ :\ T_2}{\gamma; \Gamma \vdash \lambda x{:}T_1.e \ :\ x{:}T_1 \to T_2}\ \text{T\_ABS} \qquad \frac{\gamma; \Gamma \vdash T_1 \quad \gamma; \Gamma \vdash T_2 \quad T_1 \parallel T_2}{\gamma; \Gamma \vdash \langle T_1 \Leftarrow T_2 \rangle^{\ell} \ :\ T_2 \to T_1}\ \text{T\_CAST} \qquad \frac{\gamma; \Gamma \vdash c \ :\ \{A_1\} x{:}T \{A_2\}^{\varrho}}{\gamma; \Gamma \vdash \mathsf{do}\, c \ :\ \{A_1\} x{:}T \{A_2\}^{\varrho}}\ \text{T\_DO}$$

$$\frac{\gamma; \Gamma \vdash e_1 \ :\ x{:}T_1 \to T_2 \quad \gamma; \Gamma \vdash e_2 \ :\ T_1}{\gamma; \Gamma \vdash e_1\, e_2 \ :\ [\, e_2/x \,]\, T_2}\ \text{T\_APP} \qquad \frac{\gamma; \Gamma \vdash e_1 \ :\ \mathsf{Ref}_r\, T_1 \quad \gamma; \Gamma \vdash e_2 \ :\ \mathsf{Ref}_s\, T_2 \quad T_1 \parallel T_2}{\gamma; \Gamma \vdash e_1 \mathrel{==} e_2 \ :\ \mathsf{bool}}\ \text{T\_EQ}$$

$$\frac{\gamma; \Gamma, r \vdash e \ :\ T}{\gamma; \Gamma \vdash \lambda r.e \ :\ \forall r. T}\ \text{T\_RABS} \qquad \frac{\gamma; \Gamma \vdash e \ :\ \forall s. T \quad r \in \gamma \cup \mathit{regions}\,(\Gamma)}{\gamma; \Gamma \vdash e\{r\} \ :\ [\, r/s \,]\, T}\ \text{T\_RAPP}$$

$\boxed{\gamma; \Gamma \vdash c \ :\ \{A_1\} x{:}T \{A_2\}^{\varrho}}$ **Computation Typing Rules**

$$\frac{\gamma; \Gamma \vdash e \ :\ T \quad \gamma; \Gamma, x{:}T \vdash^{\varrho} A}{\gamma; \Gamma \vdash \mathsf{return}\, e \ :\ \{[\, e/x \,]\, A\} x{:}T \{A\}^{\varrho}}\ \text{CT\_RETURN} \qquad \frac{\begin{array}{c}\gamma; \Gamma \vdash e_1 \ :\ \{A_1\} y{:}T_1 \{A_3\}^{\varrho_1} \quad \gamma; \Gamma \vdash \{A_1\} x{:}T_2 \{A_2\}^{\varrho_1 \cup \varrho_2} \\ \gamma; \Gamma, y{:}T_1 \vdash c_2 \ :\ \{A_3\} x{:}T_2 \{A_2\}^{\varrho_2}\end{array}}{\gamma; \Gamma \vdash y \leftarrow e_1; c_2 \ :\ \{A_1\} x{:}T_2 \{A_2\}^{\varrho_1 \cup \varrho_2}}\ \text{CT\_BIND}$$

$$\frac{\begin{array}{c}\gamma; \Gamma \vdash e \ :\ T' \quad \gamma; \Gamma \vdash \{A_1\} x{:}T \{A_2\}^{\langle \gamma_{\mathtt{r}}, \gamma_{\mathtt{w}} \cup \{r\} \rangle} \\ \gamma; \Gamma, y{:}\mathsf{Ref}_r\, T' \vdash c \ :\ \{A_1\} x{:}T \{A_2\}^{\langle \gamma_{\mathtt{r}}, \gamma_{\mathtt{w}} \rangle}\end{array}}{\gamma; \Gamma \vdash y \Leftarrow \mathsf{ref}_r e; c \ :\ \{A_1\} x{:}T \{A_2\}^{\langle \gamma_{\mathtt{r}}, \gamma_{\mathtt{w}} \cup \{r\} \rangle}}\ \text{CT\_NEW} \qquad \frac{\begin{array}{c}\gamma; \Gamma \vdash e \ :\ \mathsf{Ref}_r\, T' \quad \gamma; \Gamma \vdash \{A_1\} x{:}T \{A_2\}^{\langle \gamma_{\mathtt{r}} \cup \{r\}, \gamma_{\mathtt{w}} \rangle} \\ \gamma; \Gamma, y{:}T' \vdash c \ :\ \{A_1\} x{:}T \{A_2\}^{\langle \gamma_{\mathtt{r}}, \gamma_{\mathtt{w}} \rangle}\end{array}}{\gamma; \Gamma \vdash y \Leftarrow\, !e; c \ :\ \{A_1\} x{:}T \{A_2\}^{\langle \gamma_{\mathtt{r}} \cup \{r\}, \gamma_{\mathtt{w}} \rangle}}\ \text{CT\_DEREF}$$

$$\frac{\begin{array}{c}\gamma; \Gamma \vdash e_1 \ :\ \mathsf{Ref}_r\, T' \quad \gamma; \Gamma \vdash \{A_1\} x{:}T \{A_2\}^{\langle \gamma_{\mathtt{r}}, \gamma_{\mathtt{w}} \cup \{r\} \rangle} \\ \gamma; \Gamma \vdash e_2 \ :\ T' \quad \gamma; \Gamma, y{:}\mathsf{unit} \vdash c_3 \ :\ \{\top\} x{:}T \{A_2\}^{\langle \gamma_{\mathtt{r}}, \gamma_{\mathtt{w}} \rangle}\end{array}}{\gamma; \Gamma \vdash y \Leftarrow e_1 \mathrel{:=} e_2; c_3 \ :\ \{A_1\} x{:}T \{A_2\}^{\langle \gamma_{\mathtt{r}}, \gamma_{\mathtt{w}} \cup \{r\} \rangle}}\ \text{CT\_ASSIGN} \qquad \frac{\gamma; \Gamma \vdash c_2 \ :\ \{A_1, c_1\} x{:}T \{A_2\}^{\varrho}}{\gamma; \Gamma \vdash \mathsf{assert}\,(c_1)^{\ell}; c_2 \ :\ \{A_1\} x{:}T \{A_2\}^{\varrho}}\ \text{CT\_ASSERT}$$

$$\frac{\begin{array}{c}\gamma; \Gamma \vdash \{A_1\} x{:}T \{A_2\}^{\varrho} \\ \gamma, r; \Gamma \vdash c \ :\ \{A_1\} x{:}T \{A_2\}^{\varrho \uplus \{r\}}\end{array}}{\gamma; \Gamma \vdash \nu r.\, c \ :\ \{A_1\} x{:}T \{A_2\}^{\varrho}}\ \text{CT\_LETREGION} \qquad \frac{\begin{array}{c}\gamma; \Gamma \vdash c \ :\ \{A_1'\} x{:}T \{A_2'\}^{\varrho} \quad A_1' \subseteq A_1 \\ A_2 \subseteq A_2' \quad \gamma; \Gamma \vdash \{A_1\} x{:}T \{A_2\}^{\varrho}\end{array}}{\gamma; \Gamma \vdash c \ :\ \{A_1\} x{:}T \{A_2\}^{\varrho}}\ \text{CT\_WEAK}$$

Figure 3. Type system for $\lambda^{\mathrm{H}}_{\mathsf{ref}}$.

are never replaced by substitution. We assume that term and region variables declared in typing contexts are distinct and write *regions* (Γ) for the set of region variables declared in Γ.

Inference rules for typing context and type well-formedness judgments are standard [3, 18, 38, 39] or straightforward except (WF_REFINE) and (WF_HOARE). The rule (WF_REFINE) means that refinements must be pure ($\langle \emptyset, \emptyset \rangle$) and cannot assume but do not have to ensure anything. The rule (WF_HOARE) states that the pre- and postcondition of a Hoare type with effect ϱ should be no more effectful than ϱ.

Assertion well-formedness is derived by two rules. In the rule (WF_EXTENDASSERT), which is applied to append another condition c to a sequence A, the second premise means that: the appended condition c can assume that the preceding conditions A hold, has to be read-only, and does not have to ensure anything.

Typing rules for terms are syntax-directed and almost standard. The rules (T_CONST) and (T_OP) use metafunction ty, which returns the type of each constant and each operator. The rule (T_APP) substitutes an argument term e_2 for variable x bound in return type T_2. This substitution causes no problems because terms in our calculus are almost pure—i.e., they return values or raise exceptions—and it is known that some computational effects, such as raise of uncatchable exceptions and nontermination, do not cause problems in manifest contracts [3, 39]. The rule (T_CAST) requires the source and target types of a well-typed cast to be well formed and *compatible* [19, 25]. Intuitively, a type T_1 is compatible with another type T_2 when they are identified after dropping all contracts and region information. Formally, compatibility $\parallel$ is the congruence satisfying $\{x{:}T \mid c\} \parallel T$, $\mathsf{Ref}_r\, T \parallel \mathsf{Ref}_s\, T$, and $\{A_{11}\} x{:}T \{A_{12}\}^{\varrho_1} \parallel \{A_{21}\} x{:}T \{A_{22}\}^{\varrho_2}$. The rule (T_EQ) allows terms typed at compatible reference types to be compared because the same pointer can be cast to different (reference) types; note that $T_1 \parallel T_2$ iff $\mathsf{Ref}_r\, T_1 \parallel \mathsf{Ref}_s\, T_2$ for any r and s. The compatibility check in

(T_CAST) and (T_EQ) reports casts that always fail and equality tests that always return false without evaluating.

Computation typing rules are more interesting. The typing rule (CT_RETURN) gives return e the Hoare type $\{[\,e/x\,]\,A\}x\!:\!T\{A\}^{\varrho}$ where T is the type of e and ϱ is effects which may occur in A. If $[\,e/x\,]\,A$ is satisfied, return e results in a store satisfying A because x in A denotes the value of e and return e does not manipulate stores. The rule (CT_BIND) for $x \leftarrow e_1; c_2$ requires e_1 to evaluate to a thunk and allows the remaining computation c_2 to refer to, by x, the result of computing the thunk. Since $x \leftarrow e_1; c_2$ involves effects of the thunk and c_2, the effect of the result type is the union $\varrho_1 \cup \varrho_2$ of effects of e_1 and c_2. Satisfaction of the precondition of c_2 has to be promised by the postcondition of the thunk of e_1 because c_2 will be computed immediately after executing the thunk. Moreover, (CT_BIND) demands that the result type $\{A_1\}x\!:\!T_2\{A_2\}^{\varrho_1 \cup \varrho_2}$ be well formed under the typing context Γ without y since y is a variable locally bound by bind—other typing rules need similar conditions. The typing rules (CT_NEW), (CT_DEREF), and (CT_ASSIGN) are applied to a computation with memory allocation, dereference, and assignment, respectively. In these rules, the corresponding effect is added to the result Hoare type. The first two rules are not surprising. The rule (CT_ASSIGN) says that the remaining computation c_3 cannot assume anything (hence the empty condition) because the assignment could invalidate the precondition A_1—for example, the condition $x \Leftarrow !e; \mathsf{return}\, x = 0$ does not hold after executing $e := 2$. In general, we do not know which conditions still hold and which conditions do not, so we suppose the worst-case scenario, that is, that all conditions are invalidated. In fact, we can do better because if references manipulated by assignment are allocated at r, conditions which do not involve effects including r will not be invalidated—we discuss this recovery of preconditions of the remaining computation in Section 7. Finally, the result of assignment is the unit value, so the typing context in the third premise of (CT_ASSIGN) includes y:unit. The rule (CT_LETREGION) is applied to let-region $\nu r.\, c$ and requires the body c to be well typed under the region set including region r. The well-formedness of the result Hoare type under the region set without r ensures that access to r is not observed by clients. An assertion $\mathsf{assert}\,(c_1)^{\ell}; c_2$ is typed by (CT_ASSERT), which allows the remaining computation c_2 to assume that the condition c_1 holds since its satisfaction is ensured at run time; well typedness of c_1 is ensured by the preceding typing derivation. The last rule (CT_WEAK) allows strengthening preconditions, weakening postconditions, and permuting them. The partial order $A_1 \subseteq A_2$ over conditions means that, for any c in A_1, there exist some A and A' such that $A_2 = A, c, A'$. Although (CT_WEAK) manipulates conditions syntactically, static verification in Section 7 enables more flexible manipulation. For example, the technique in that section allows computations of $\{\top\}t\!:\!\mathsf{tbl}\{x \Leftarrow !t; \mathsf{return}\,(\mathtt{mem}\,x\,\mathtt{"foo"})\}^{\varrho}$ to be regarded as ones of $\{\top\}t\!:\!\mathsf{tbl}\{x \Leftarrow !t; \mathsf{return}\,(\mathtt{length}\,x > 0)\}^{\varrho}$ (if it can be proven that the former postcondition implies the latter), which cannot be derived from (CT_WEAK). Finally, we make a remark about effect weakening—well typed computations can be given a Hoare type with more effects. Although there is no rule for effect weakening, it is admissible:

Lemma 1 (Effect Weakening)**.** *If* $\langle \gamma_{\mathtt{r}}, \gamma_{\mathtt{w}} \rangle \subseteq \langle \gamma_{\mathtt{r}}', \gamma_{\mathtt{w}}' \rangle$ *and* $\gamma_{\mathtt{r}}', \gamma_{\mathtt{w}}' \subseteq \gamma \cup \mathit{regions}\,(\Gamma)$ *and* $\gamma; \Gamma \vdash c : \{A_1\}x\!:\!T\{A_2\}^{\langle \gamma_{\mathtt{r}}, \gamma_{\mathtt{w}} \rangle}$, *then* $\gamma; \Gamma \vdash c : \{A_1\}x\!:\!T\{A_2\}^{\langle \gamma_{\mathtt{r}}', \gamma_{\mathtt{w}}' \rangle}$.

5. Semantics

In this section, we define small-step call-by-value operational semantics for $\lambda^{\mathrm{H}}_{\mathsf{ref}}$. The semantics mainly consists of two relations, reduction relation $\rightsquigarrow$ for terms and computation relation $\longrightarrow$ for computations. In what follows, we begin with describing intuition about run-time checking introduced in this work—casts for reference types and Hoare types, and contract checks with local regions and local stores. Then, we formalize the semantics after extending the program syntax with run-time terms and computations.

5.1 Run-time Checking for References and Thunks

In this section, we outline how casts for reference types and Hoare types work. Casts for other types are similar to the previous work [3, 15, 25, 38, 39].

Casts between reference types with the same region generate *reference guards* $T_1 \Leftarrow^{\ell} T_2 : v$, which are a key construct in the earlier work on dynamic checking with references [11, 16, 20, 44]:

$$\langle \mathsf{Ref}_r\, T_1 \Leftarrow \mathsf{Ref}_r\, T_2 \rangle^{\ell}\, v \rightsquigarrow T_1 \Leftarrow^{\ell} T_2 : v \qquad \text{R\_REF}$$

Otherwise, if the regions are different, the cast fails:

$$\langle \mathsf{Ref}_r\, T_1 \Leftarrow \mathsf{Ref}_s\, T_2 \rangle^{\ell}\, v \rightsquigarrow \Uparrow\!\ell \quad (\text{where } r \neq s) \quad \text{R\_REFFAIL}$$

Reference guards are “proxies” to monitor dereference and assignment operations at run time so that they behave as the target reference type. When reference guard $T_1 \Leftarrow^{\ell} T_2 : v$ is dereferenced, the run-time system checks that the contents of v can work as T_1:

$$x \Leftarrow !(T_1 \Leftarrow^{\ell} T_2 : v); c \longrightarrow y \Leftarrow !v; \mathsf{let}\, x = \langle T_1 \Leftarrow T_2 \rangle^{\ell}\, y; c.$$

When it is assigned a value v' of T_1, it is checked that v' can be assigned to v:

$$x \Leftarrow (T_1 \Leftarrow^{\ell} T_2 : v) := v'; c \longrightarrow x \Leftarrow v := \langle T_2 \Leftarrow T_1 \rangle^{\ell}\, v'; c.$$

Casts between Hoare types are similar to casts between function types [14, 15] in the sense that computations are functions over states. Since a cast is a term-level construct, the result of cast application $\langle \{A_{11}\}x\!:\!T_1\{A_{12}\}^{\varrho_1} \Leftarrow \{A_{21}\}x\!:\!T_2\{A_{22}\}^{\varrho_2} \rangle^{\ell}\, v$ (where v's type is $\{A_{21}\}x\!:\!T_2\{A_{22}\}^{\varrho_2}$) is a thunk, which triggers execution of v with additional checks: it is ensured that the thunk does not hide the effects of v; the precondition A_{21} of v is checked before its execution; the result of v is cast back to T_1 from T_2; and, finally, the postcondition A_{12} is checked. Formally, the cast application reduces as follows:

$$\begin{array}{l} \langle \{A_{11}\}x\!:\!T_1\{A_{12}\}^{\varrho_1} \Leftarrow \{A_{21}\}x\!:\!T_2\{A_{22}\}^{\varrho_2} \rangle^{\ell}\, v \rightsquigarrow \\ \quad \mathsf{do}\, \mathsf{assert}\,(A_{21})^{\ell}; y \leftarrow v; \mathsf{let}\, x = \langle T_1 \Leftarrow T_2 \rangle^{\ell}\, y; \\ \qquad \mathsf{assert}\,(A_{12})^{\ell}; \mathsf{return}\, x \quad (\text{where } \varrho_2 \subseteq \varrho_1) \quad \text{R\_HOARE} \end{array}$$

where y is a fresh variable and notation $\mathsf{assert}\,(A)^{\ell}; c$ means, for $A = c_1, ..., c_n$, $\mathsf{assert}\,(c_1)^{\ell}; ...; \mathsf{assert}\,(c_n)^{\ell}; c$. If ϱ_1 is not more effectful than ϱ_2, the cast fails:

$$\begin{array}{l} \langle \{A_{11}\}x\!:\!T_1\{A_{12}\}^{\varrho_1} \Leftarrow \{A_{21}\}x\!:\!T_2\{A_{22}\}^{\varrho_2} \rangle^{\ell}\, v \rightsquigarrow \Uparrow\!\ell \\ \qquad\qquad\qquad (\text{where } \varrho_2 \not\subseteq \varrho_1) \quad \text{R\_HOAREFAIL} \end{array}$$

5.2 Contract Checking with Local Stores

Effects for references at locally introduced regions in contracts must not be observed by programs, which is important especially for showing correctness of static verification in Section 7. Unfortunately, this request makes it difficult to apply the small-step operational semantics of the previous work on a region calculus [7], where reduction of let-regions changes program stores, to our work because it is unclear how to distinguish changes to program stores by programs and those by contracts.

Our approach to the request is to introduce *local* stores to intermediate states of contract checking and design a semantics where memory allocation and assignment operations with respect to locally introduced regions in contracts are applied to the local stores. Thanks to local stores, during contract checking, program stores do not change and, as a result, effects involved by contracts are not observed.

Stores, Values, Terms, Computations, and Checking States

$$
\begin{array}{rcl}
a, b & ::= & \text{memory addresses} \qquad \mu ::= \{a_i@r_i \mapsto v_i\}_i \\
v & ::= & k \mid \lambda x{:}T.e \mid \langle T_1 \Leftarrow T_2 \rangle^\ell \mid \mathsf{do}\, c \mid \lambda r.e \mid \\
 & & a@r \mid T_1 \Leftarrow^\ell T_2 : v \\
e & ::= & ... \mid a@r \mid T_1 \Leftarrow^\ell T_2 : v \mid \\
 & & \Uparrow\ell \mid \langle\!\langle \{x{:}T \mid c\}, e \rangle\!\rangle^\ell \mid \langle \{x{:}T \mid c\}, p, v \rangle^\ell \\
c & ::= & ... \mid \Uparrow\ell \mid \langle \mathsf{assert}\,(c_1), p_2 \rangle^\ell; c_3 \qquad p ::= \nu\gamma.\langle \mu \mid c \rangle
\end{array}
$$

Figure 4. Run-time syntax in $\lambda^{\mathsf{H}}_{\mathsf{ref}}$.

Formally, we introduce an expression of the form $\nu\gamma.\langle\mu \mid c\rangle$ (called checking state), where γ, μ, and c are a set of local regions, a local store, and a computation, respectively, to express a program state during contract checking. Starting with checking a contract c, the run-time system generates an initial checking state $\nu\emptyset.\langle\emptyset \mid c\rangle$, where there are no locally introduced regions and no locally allocated references, and then starts computing c. During the check, newly introduced regions are added to the local regions, new memory cells are allocated to the local store, and dereference and assignment for memory cells at the local regions are applied to the local store. If c results in $\mathsf{return\,true}$, the check succeeds; if c results in $\mathsf{return\,false}$, it fails.

5.3 Definition

5.3.1 Run-time Syntax

We show the run-time syntax in Figure 4. We use a and b to denote memory addresses. The definition of values v is not surprising except that memory addresses are given regions to indicate which region they are allocated at. Stores, ranged over by μ, are finite mappings from pairs of a memory address and a region to closed values. We write $\mu_1 \uplus \mu_2$ for the concatenation of μ_1 and μ_2 with disjoint domains. Checking states $\nu\gamma.\langle\mu \mid c\rangle$, denoted by p, bind γ in μ and c.

The forms of run-time terms are straightforward except for the last two constructs, which represent intermediate states of refinement checking. A waiting check $\langle\!\langle \{x{:}T \mid c\}, e \rangle\!\rangle^\ell$, introduced by Sekiyama et al. [38] to prove a critical property of a manifest contract calculus with Belo et al.'s approach [3, 18], waits for evaluation of e before starting a check that the value of e satisfies the contract c. An active check $\langle\{x{:}T \mid c\}, p, v\rangle^\ell$ is verifying that the value v satisfies the refinement c; p is an intermediate state of a check which has been started by running $[\,v/x\,]\,c$. If the intermediate computation of p results in true, the active check evaluates to v; otherwise, if it results in false, $\Uparrow\ell$ will be raised.

Run-time computations have two additional constructs: exception $\Uparrow\ell$ and assertion intermediate state $\langle\mathsf{assert}\,(c_1), p_2\rangle^\ell; c_3$ of a check of contract c_1: if the check succeeds, the remaining computation c_3 will be executed, and if it fails, $\Uparrow\ell$ will be raised.

5.3.2 Reduction

The reduction $\rightsquigarrow$, defined over closed run-time terms, is given by using rules shown at the top of Figure 5, together with rules presented in Section 5.1. The first three rules are standard in lambda calculi with call-by-value semantics, or straightforward. $[\![-]\!]$ in (R_OP) assigns a function over base type values to an operation op. Reduction of pointer equality tests uses $ungrd$ to peel off all reference guards:

$$ungrd\,(a@r) = a@r \quad ungrd\,(T_1 \Leftarrow^\ell T_2 : v) = ungrd\,(v)$$

The next several rules are cast reduction rules, some of which are similar to the ones in the previous work [38, 39]. The rule (R_RFUN) generates a region abstraction that wraps the target value, like cast reduction for type abstractions [3, 39]. The rule (R_FUN) produces a "function proxy" [5, 19], which applies the contravariant cast on argument types to an argument, passes its result to the original function, and applies the covariant cast on return types to the value returned by the original function. To avoid capture of variable x bound in the source return type, it is renamed with a fresh variable y. By (R_FORGET) and (R_PRECHECK), a cast application for refinement types first forgets all refinements in the source type and then reduces to a waiting check which verifies that the target value satisfies the outermost contract after checks of inner contracts. The side condition in (R_PRECHECK) makes the semantics deterministic. After checks of inner contracts, (R_CHECK) produces an active check to verify the outermost contract. The rule (R_CHECKING) reduces the active check by evaluating the contract checking state under the empty store—we see how checking states reduce later—and exceptions that happen during the check are lifted up by (R_BLAME). If the checking succeeds, the active check returns the target value (R_OK); otherwise, if it fails, the check raises $\Uparrow\ell$ (R_FAIL).

5.3.3 Computation

The computation $\longrightarrow$, defined over pairs of a store and a closed run-time computation, is given by rules at the bottom of Figure 5 and auxiliary rules in the middle of Figure 5 to execute commands.

The rules (C_RED) and (C_COMPUT) are applied to reduce subterms and subcomputations of a computation, using computation contexts on terms, ranged over by C^{e}:

$$
\begin{array}{rcl}
E & ::= & [\,] \mid op(v_1, \ldots, v_n, E, e_1, \ldots, e_n) \mid E\ e_2 \mid v_1\ E \mid \\
 & & E == e_2 \mid v_1 == E \mid E\{r\} \mid \langle\!\langle \{x{:}T \mid c_1\}, E \rangle\!\rangle^\ell \\
D & ::= & \mathsf{ref}_r E \mid {!E} \mid E := e_2 \mid v_1 := E \\
C^{\mathsf{e}} & ::= & \mathsf{return}\,E \mid x \leftarrow E; c_2 \mid x \Leftarrow D; c_2
\end{array}
$$

The definition above means that subterms reduce from left to right.

Exceptions raised by subterms and subcomputations are lifted up by (C_RBLAME) and (C_CBLAME), respectively. The rule (C_CBLAME) also lifts up exceptions raised by contracts, using computation contexts on exceptions.

$$C^{1} \quad ::= \quad x \leftarrow \mathsf{do}\,[\,]\,; c_2 \mid \langle \mathsf{assert}\,(c_1),\, \nu\gamma.\langle\mu \mid [\,]\rangle\rangle^\ell; c_3$$

The rule (C_RETURN) performs, when term e_1 of bind $x \leftarrow e_1; c_2$ reduces to a thunk (by (C_RED)) and the thunk returns a value v (by (C_COMPUT)), the remaining computation $[\,v/x\,]\,c_2$. Let-regions are lifted up to checking states by (C_REGION) in order to propagate newly created regions. For example, checking state $\nu\gamma.\langle\mu \mid ...\nu r.\ c'...\rangle$, where the let-region will evaluate, go on to $\nu\gamma.\langle\mu \mid \nu r.\ ...c'...\rangle$ by (C_REGION). The rule (C_COMMAND), applied to execute commands, rests on the command relation $\rightarrowtail$, which transforms a command to a computation with an adequate action by rules shown in the middle of Figure 5. The first three command rules are standard except for use of regions. The last two rules are used for dereference from and assignment to reference guards, as described in Section 5.1; there variable x can be arbitrarily chosen since commands are closed.

Other computation rules are applied to check contracts with assertion. The rule (C_ASSERT) produces an assertion intermediate state, which proceeds with the program store by (C_CHECKING) and: if the checking succeeds, the remaining computation c_2 starts (C_OK); otherwise, if it fails, $\Uparrow\ell$ is raised (C_FAIL).

Finally, computation of checking state $\nu\gamma.\langle\mu_1 \mid c_1\rangle$ under global store μ proceeds by two rules. (P_COMPUT) computes c_1 under the concatenation of the program store and the local store. The result store $\mu \uplus \mu_2$ means that the global store remains the same and, although dereference from the global is possible, memory allocation and assignment cannot take place only on it. The rule

$\boxed{e_1 \rightsquigarrow e_2}$ **Reduction Rules**

$$op(k_1, \ldots, k_n) \rightsquigarrow [\![op]\!](k_1, \ldots, k_n) \quad \text{R\_OP} \qquad (\lambda x{:}T.e)\, v \rightsquigarrow [\, v/x \,]\, e \quad \text{R\_BETA} \qquad (\lambda r.e)\{s\} \rightsquigarrow [\, s/r \,]\, e \quad \text{R\_RBETA}$$

$$v_1 == v_2 \rightsquigarrow \mathsf{true} \quad (\text{where } \mathit{ungrd}\,(v_1) = \mathit{ungrd}\,(v_2)) \quad \text{R\_EQ} \qquad v_1 == v_2 \rightsquigarrow \mathsf{false} \quad (\text{where } \mathit{ungrd}\,(v_1) \neq \mathit{ungrd}\,(v_2)) \quad \text{R\_NEQ}$$

$$\langle B \Leftarrow B \rangle^{\ell}\, v \rightsquigarrow v \quad \text{R\_BASE} \qquad \langle \forall r.T_1 \Leftarrow \forall r.T_2 \rangle^{\ell}\, v \rightsquigarrow \lambda r.\langle T_1 \Leftarrow T_2 \rangle^{\ell}\, (v\{r\}) \quad \text{R\_RFUN}$$

$$\langle x{:}T_{11} \rightarrow T_{12} \Leftarrow x{:}T_{21} \rightarrow T_{22} \rangle^{\ell}\, v \rightsquigarrow \lambda x{:}T_{11}.\mathsf{let}\ y = \langle T_{21} \Leftarrow T_{11} \rangle^{\ell}\, x\ \mathsf{in}\ (\langle T_{12} \Leftarrow [\, y/x \,]\, T_{22} \rangle^{\ell}\, (v\ y)) \quad (\text{where } y \text{ is fresh}) \quad \text{R\_FUN}$$

$$\langle T_1 \Leftarrow \{x{:}T_2 \mid c_2\} \rangle^{\ell}\, v \rightsquigarrow \langle T_1 \Leftarrow T_2 \rangle^{\ell}\, v \quad \text{R\_FORGET} \qquad \langle \{x{:}T_1 \mid c_1\} \Leftarrow T_2 \rangle^{\ell}\, v \rightsquigarrow \langle\!\langle \{x{:}T_1 \mid c_1\}, \langle T_1 \Leftarrow T_2 \rangle^{\ell}\, v \rangle\!\rangle^{\ell}$$
$$(\text{where } T_2 \neq \{y{:}T \mid c\} \text{ for any } y,\, T, \text{ and } c) \quad \text{R\_PRECHECK}$$

$$\langle\!\langle \{x{:}T \mid c\}, v \rangle\!\rangle^{\ell} \rightsquigarrow \langle \{x{:}T \mid c\}, \nu\emptyset.\langle \emptyset \mid [\, v/x \,]\, c \rangle, v \rangle^{\ell} \quad \text{R\_CHECK} \qquad \langle \{x{:}T \mid c\}, \nu\gamma.\langle \mu \mid \Uparrow\ell' \rangle, v \rangle^{\ell} \rightsquigarrow \Uparrow\ell' \quad \text{R\_BLAME}$$

$$\langle \{x{:}T \mid c\}, \nu\gamma.\langle \mu \mid \mathsf{return\,true} \rangle, v \rangle^{\ell} \rightsquigarrow v \quad \text{R\_OK} \qquad \langle \{x{:}T \mid c\}, \nu\gamma.\langle \mu \mid \mathsf{return\,false} \rangle, v \rangle^{\ell} \rightsquigarrow \Uparrow\ell \quad \text{R\_FAIL}$$

$$\langle \{x{:}T \mid c\}, p, v \rangle^{\ell} \rightsquigarrow \langle \{x{:}T \mid c\}, p', v \rangle^{\ell} \quad (\text{where } \emptyset \mid p \hookrightarrow p') \quad \text{R\_CHECKING}$$

$\boxed{\mu_1 \mid d_1 \rightarrowtail \mu_2 \mid c_2}$ **Command Rules**

$$\mu \mid \mathsf{ref}_r v \rightarrowtail \mu \uplus \{a@r \mapsto v\} \mid \mathsf{return}\ a@r \quad \text{C\_NEW} \qquad \mu \uplus \{a@r \mapsto v'\} \mid a@r \coloneqq v \rightarrowtail \mu \uplus \{a@r \mapsto v\} \mid \mathsf{return}\,() \quad \text{C\_ASSIGN}$$

$$\mu \mid\, !a@r \rightarrowtail \mu \mid \mathsf{return}\ \mu(a@r) \quad \text{C\_DEREF} \qquad \mu \mid\, !(T_1 \Leftarrow^{\ell} T_2 : v) \rightarrowtail \mu \mid x \Leftarrow !v; \mathsf{return}\,(\langle T_1 \Leftarrow T_2 \rangle^{\ell}\, x) \quad \text{C\_GDEREF}$$

$$\mu \mid (T_1 \Leftarrow^{\ell} T_2 : v_2) \coloneqq v_1 \rightarrowtail \mu \mid x \Leftarrow v_2 \coloneqq (\langle T_2 \Leftarrow T_1 \rangle^{\ell}\, v_1); \mathsf{return}\,() \quad \text{C\_GASSIGN}$$

$\boxed{\mu_1 \mid c_1 \longrightarrow \mu_2 \mid c_2}$ **Computation Rules**

$$\frac{e_1 \rightsquigarrow e_2}{\mu \mid C^{\mathsf{e}}\,[\, e_1 \,] \longrightarrow \mu \mid C^{\mathsf{e}}\,[\, e_2 \,]}\ \text{C\_RED} \qquad \frac{\mu_1 \mid c_1 \longrightarrow \mu_2 \mid c_1'}{\mu_1 \mid x \leftarrow \mathsf{do}\, c_1; c_2 \longrightarrow \mu_2 \mid x \leftarrow \mathsf{do}\, c_1'; c_2}\ \text{C\_COMPUT}$$

$$\mu \mid C^{\mathsf{e}}\,[\Uparrow\ell] \longrightarrow \mu \mid \Uparrow\ell \quad \text{C\_RBLAME} \qquad \mu \mid x \leftarrow \mathsf{do\,return}\ v_1; c_2 \longrightarrow \mu \mid [\, v_1/x \,]\, c_2 \quad \text{C\_RETURN}$$

$$\mu \mid C^{1}\,[\Uparrow\ell] \longrightarrow \mu \mid \Uparrow\ell \quad \text{C\_CBLAME} \qquad \mu \mid x \leftarrow (\mathsf{do}\, \nu r.\, c_1); c_2 \longrightarrow \mu \mid \nu r.\,(x \leftarrow \mathsf{do}\, c_1; c_2) \ \ (\text{where } r \notin \mathit{frv}\,(c_2)) \quad \text{C\_REGION}$$

$$\frac{\mu_1 \mid d_1 \rightarrowtail \mu_2 \mid c_1}{\mu_1 \mid x \Leftarrow d_1; c_2 \longrightarrow \mu_2 \mid x \leftarrow \mathsf{do}\, c_1; c_2}\ \text{C\_COMMAND} \qquad \frac{\mu \mid p_1 \hookrightarrow p_2}{\mu \mid \langle \mathsf{assert}\,(c_1), p_1 \rangle^{\ell}; c_2 \longrightarrow \mu \mid \langle \mathsf{assert}\,(c_1), p_2 \rangle^{\ell}; c_2}\ \text{C\_CHECKING}$$

$$\mu \mid \mathsf{assert}\,(c_1)^{\ell}; c_2 \longrightarrow \mu \mid \langle \mathsf{assert}\,(c_1), \nu\emptyset.\langle \emptyset \mid c_1 \rangle \rangle^{\ell}; c_2 \quad \text{C\_ASSERT} \qquad \mu \mid \langle \mathsf{assert}\,(c_1), \nu\gamma.\langle \mu \mid \mathsf{return\,true} \rangle \rangle^{\ell}; c_2 \longrightarrow \mu \mid c_2 \quad \text{C\_OK}$$

$$\mu \mid \langle \mathsf{assert}\,(c_1), \nu\gamma.\langle \mu \mid \mathsf{return\,false} \rangle \rangle^{\ell}; c_2 \longrightarrow \mu \mid \Uparrow\ell \quad \text{C\_FAIL}$$

$\boxed{\mu \mid p_1 \hookrightarrow p_2}$ **Checking State Computation Rules**

$$\frac{\mu \uplus \mu_1 \mid c_1 \longrightarrow \mu \uplus \mu_2 \mid c_2}{\mu \mid \nu\gamma.\langle \mu_1 \mid c_1 \rangle \hookrightarrow \nu\gamma.\langle \mu_2 \mid c_2 \rangle}\ \text{P\_COMPUT} \qquad \mu \mid \nu\gamma.\langle \mu' \mid \nu r.\, c \rangle \hookrightarrow \nu(\gamma, r).\langle \mu' \mid c \rangle \quad \text{P\_REGION}$$

Figure 5. Operational semantics.

(P_REGION) adds regions bubbled up by (C_REGION) to the local regions.

Top-level programs are executed in the form of checking states. We call computations c such that $\{r\}; \emptyset \vdash c : \{\top\}x{:}T\{A_2\}^{\langle\{r\},\{r\}\rangle}$ *programs*. A program c with designated region r, which stands for the global store, is executed by starting computation from $\nu\emptyset.\langle \emptyset \mid \nu r.\, c \rangle$ and program execution is performed by evaluation of checking states $\mu \mid p_1 \hookrightarrow^* p_2$, which means that there are $p_1', \ldots, p_n'$ such that $\mu \mid p_1 \hookrightarrow p_1', \mu \mid p_1' \hookrightarrow p_2', \ldots, \mu \mid p_n' \hookrightarrow p_2$. Thus, the program evaluation that the program c results in v is denoted by $\emptyset \mid \nu\emptyset.\langle \emptyset \mid \nu r.\, c \rangle \hookrightarrow^* \nu\gamma.\langle \mu \mid \mathsf{return}\ v \rangle$, where μ is the result global store.

6. Type Soundness

Following Belo et al. [3], which avoided semantic type soundness proofs, we show type soundness of $\lambda^{\mathrm{H}}_{\mathsf{ref}}$ via standard syntactic approaches (namely, progress and preservation [51]). In particular, we show that (1) a well-typed program returns a value, raises an exception, or diverges and (2) the result value and the result store of a well-typed program satisfies contracts on its type.

6.1 Run-time Type System

To prove type soundness, we extend the type system in Section 4 to deal with run-time terms and computations. As usual, we introduce store typing contexts, ranged over by Σ, to record the type of the value that each reference points to. They are defined as follows:

$$\Sigma ::= \emptyset \mid \Sigma, a@r{:}T$$

We assume that references declared in store typing contexts are distinct and write Σ, Σ' for the concatenation of Σ and Σ'.

As shown in Figure 6, the run-time type system consists of extensions of judgments in Section 4 with store typing contexts—in addition, the computation typing judgment $\mu; \Sigma; \gamma; \Gamma \vdash c : \{A_1\}x{:}T\{A_2\}^{\varrho}$ refers to μ denoting current stores—and two new judgments for checking states $\mu; \Sigma; \gamma \vdash p : T^{\gamma'}$, which means that the computation of p may refer to memory cells at γ' and returns a value of T (if any) under the store μ together with the local store of p, and stores $\gamma \vdash \mu : \Sigma^{\gamma'}$, which means that all memory cells in store μ are allocated at γ' and their contents are assigned types by Σ. The computation and checking state typing judgments need a current store μ for simulating contract checks in the type system; see (CT_CHECK) below for details. In what follow, we write $\mu \models A$ when, for any $c \in A$, $\mu \mid \nu\emptyset.\langle \emptyset \mid c \rangle \hookrightarrow^* \nu\gamma'.\langle \mu' \mid \mathsf{return\,true} \rangle$.

Figure 6 shows selected rules for well-formedness and typing judgments. The rules for typing context well-formedness, type well-formedness, and assertion well-formedness are similar to the rules in Figure 3 except for use of store typing contexts. Since contracts are specifications, refinements, pre- and postconditions should not depend on a current store, so they must be well typed under the empty store.

$\boxed{\Sigma;\gamma \vdash \Gamma}$ $\boxed{\Sigma;\gamma;\Gamma \vdash T}$ $\boxed{\Sigma;\gamma;\Gamma \vdash^{\varrho} A}$ $\boxed{\Sigma;\gamma;\Gamma \vdash e \,:\, T}$ **Run-time Term Typing Rules (Selected)**

$$\frac{\Sigma;\gamma \vdash \Gamma \quad a@r{:}T \in \Sigma \quad \Sigma;\gamma;\emptyset \vdash \mathsf{Ref}_r\, T}{\Sigma;\gamma;\Gamma \vdash a@r \,:\, \mathsf{Ref}_r\, T}\ \textrm{T\_ADDRESS} \qquad \frac{\Sigma;\gamma \vdash \Gamma \quad \Sigma;\gamma;\emptyset \vdash v \,:\, \mathsf{Ref}_r\, T_2 \quad T_1 \parallel T_2 \quad \Sigma;\gamma;\emptyset \vdash \mathsf{Ref}_r\, T_1}{\Sigma;\gamma;\Gamma \vdash T_1 \Leftarrow^{\ell} T_2 : v \,:\, \mathsf{Ref}_r\, T_1}\ \textrm{T\_GUARD}$$

$$\frac{\Sigma;\gamma \vdash \Gamma \quad \Sigma;\gamma;\emptyset \vdash T}{\Sigma;\gamma;\Gamma \vdash \Uparrow\!\ell \,:\, T}\ \textrm{T\_BLAME} \qquad \frac{\Sigma;\gamma \vdash \Gamma \quad \Sigma;\gamma;\emptyset \vdash \{x{:}T \mid c\} \quad \Sigma;\gamma;\emptyset \vdash e \,:\, T}{\Sigma;\gamma;\Gamma \vdash \langle\!\langle\, \{x{:}T \mid c\}, e \,\rangle\!\rangle^{\ell} \,:\, \{x{:}T \mid c\}}\ \textrm{T\_WCHECK}$$

$$\frac{\begin{array}{c}\Sigma;\gamma \vdash \Gamma \quad \Sigma;\gamma;\emptyset \vdash \{x{:}T \mid c\} \quad \Sigma;\gamma;\emptyset \vdash v \,:\, T \\ \emptyset;\Sigma;\gamma \vdash p \,:\, \mathsf{bool}^{\emptyset} \quad \emptyset \mid \nu\emptyset.\langle \emptyset \mid [\,v/x\,]\, c\rangle \hookrightarrow^{*} p\end{array}}{\Sigma;\gamma;\Gamma \vdash \langle \{x{:}T \mid c\}, p, v\rangle^{\ell} \,:\, \{x{:}T \mid c\}}\ \textrm{T\_ACHECK} \qquad \frac{\Sigma;\gamma \vdash \Gamma \quad \Sigma;\gamma;\emptyset \vdash \{x{:}T \mid c\} \quad \Sigma;\gamma;\emptyset \vdash v \,:\, T \quad \emptyset \models [\,v/x\,]\, c}{\Sigma;\gamma;\Gamma \vdash v \,:\, \{x{:}T \mid c\}}\ \textrm{T\_EXACT}$$

$$\frac{\Sigma;\gamma \vdash \Gamma \quad \Sigma;\gamma;\emptyset \vdash v \,:\, \{x{:}T \mid c\}}{\Sigma;\gamma;\Gamma \vdash v \,:\, T}\ \textrm{T\_FORGET} \qquad \frac{\Sigma;\gamma \vdash \Gamma \quad \Sigma;\gamma;\emptyset \vdash e \,:\, T_1 \quad T_1 \equiv T_2 \quad \Sigma;\gamma;\emptyset \vdash T_2}{\Sigma;\gamma;\Gamma \vdash e \,:\, T_2}\ \textrm{T\_CONV}$$

$\boxed{\mu;\Sigma;\gamma;\Gamma \vdash c \,:\, \{A_1\}x{:}T\{A_2\}^{\varrho}}$ **Run-time Computation Typing Rules (Selected)**

$$\frac{\Sigma;\gamma \vdash \Gamma \quad \Sigma;\gamma;\emptyset \vdash \{A_1\}x{:}T\{A_2\}^{\varrho}}{\mu;\Sigma;\gamma;\Gamma \vdash \Uparrow\!\ell \,:\, \{A_1\}x{:}T\{A_2\}^{\varrho}}\ \textrm{CT\_BLAME} \qquad \frac{\begin{array}{c}\Sigma;\gamma \vdash \Gamma \quad \mu;\Sigma;\gamma;\emptyset \vdash c_1 \,:\, \{A_1\}y{:}T_1\{A_3\}^{\varrho_1} \\ \emptyset;\Sigma;\gamma;y{:}T_1 \vdash c_2 \,:\, \{A_3\}x{:}T_2\{A_2\}^{\varrho_2} \quad \Sigma;\gamma;\emptyset \vdash \{A_1\}x{:}T_2\{A_3\}^{\varrho_1 \cup \varrho_2}\end{array}}{\mu;\Sigma;\gamma;\Gamma \vdash y \leftarrow \mathsf{do}\, c_1; c_2 \,:\, \{A_1\}x{:}T_2\{A_2\}^{\varrho_1 \cup \varrho_2}}\ \textrm{CT\_CBIND}$$

$$\frac{\Sigma;\gamma \vdash \Gamma \quad \emptyset;\Sigma;\gamma;\emptyset \vdash c_3 \,:\, \{A_1, c_1\}x{:}T\{A_2\}^{\langle \gamma_{\mathtt{r}}, \gamma_{\mathtt{w}}\rangle} \quad \mu;\Sigma;\gamma \vdash p_2 \,:\, \mathsf{bool}^{\gamma_{\mathtt{r}}} \quad \mu \mid \nu\emptyset.\langle \emptyset \mid c_1\rangle \hookrightarrow^{*} p_2}{\mu;\Sigma;\gamma;\Gamma \vdash \langle \mathsf{assert}\,(c_1), p_2\rangle^{\ell}; c_3 \,:\, \{A_1\}x{:}T\{A_2\}^{\langle \gamma_{\mathtt{r}}, \gamma_{\mathtt{w}}\rangle}}\ \textrm{CT\_CHECK}$$

$$\frac{\Sigma;\gamma \vdash \Gamma \quad \mu;\Sigma;\gamma;\emptyset \vdash c \,:\, \{A_{11}\}x{:}T_1\{A_{12}\}^{\varrho} \quad \{A_{11}\}x{:}T_1\{A_{12}\}^{\varrho} \equiv \{A_{21}\}x{:}T_2\{A_{22}\}^{\varrho} \quad \Sigma;\gamma;\emptyset \vdash \{A_{21}\}x{:}T_2\{A_{22}\}^{\varrho}}{\mu;\Sigma;\gamma;\Gamma \vdash c \,:\, \{A_{21}\}x{:}T_2\{A_{22}\}^{\varrho}}\ \textrm{CT\_CONV}$$

$\boxed{\mu;\Sigma;\gamma \vdash p \,:\, T^{\gamma'}}$ $\boxed{\gamma \vdash \mu \,:\, \Sigma^{\gamma'}}$ **Checking State Typing and Store Well-Formedness Rules**

$$\frac{\gamma,\gamma' \vdash \mu' \,:\, (\Sigma,\Sigma')^{\gamma'} \quad \mathit{dom}\,(\mu') = \mathit{dom}\,(\Sigma') \quad \mu \uplus \mu';\Sigma,\Sigma';\gamma,\gamma';\emptyset \vdash c' \,:\, \{A_1\}x{:}T\{\top\}^{\langle \gamma' \cup \gamma'', \gamma'\rangle} \quad \mu \uplus \mu' \models A_1}{\mu;\Sigma;\gamma \vdash \nu\gamma'.\langle \mu' \mid c'\rangle \,:\, T^{\gamma''}}\ \textrm{PT}$$

$$\frac{\mathit{dom}\,(\mu) = \mathit{dom}\,(\Sigma|_{\gamma'}) \quad \forall\, a@r \in \mathit{dom}\,(\mu).\, \Sigma;\gamma;\emptyset \vdash \mu(a@r) \,:\, \Sigma(a@r)}{\gamma \vdash \mu \,:\, \Sigma^{\gamma'}}\ \textrm{WF\_STORE}$$

Figure 6. Run-time typing rules.

There are several additional typing rules for run-time terms. In these typing rules, typing contexts in their premises are empty since run-time terms are closed; however, the conclusions allow nonempty typing contexts because run-time terms can be put under binders by substitution in (T_APP). The first five typing rules are syntax-directed. The rule (T_GUARD) requires the contents types of a reference guard to be compatible since the reference guard uses casts between these types when dereference and assignment are applied. Exceptions can be typed at any well-formed type by (T_BLAME)—it is important for showing preservation. The rule (T_ACHECK) requires p in an active check to return Boolean values (if any) in the fourth premise and to be an intermediate state of the checking in the last premise. The rule (T_EXACT) allows values of T satisfying a contract c to be typed at the refinement type $\{x{:}T \mid c\}$. By the rule (T_FORGET), which corresponds with (R_FORGET), we can peel off the outermost contract of a refinement type. The final rule (T_CONV) is introduced by Belo et al. [3] to show subject reduction in manifest contracts with dependent function types. To see its motivation, let us consider well typed term application $v_1\, e_2$. From (T_APP), the type of $v_1\, e_2$ would be $[\,e_2/x\,]\, T_2$ for some x and T_2. If e_2 reduces to term e_2', the type of the application changes to $[\,e_2'/x\,]\, T_2$, which is different from $[\,e_2/x\,]\, T_2$ in general. Thus, subject reduction would not hold if there are no ways to connect $[\,e_2/x\,]\, T$ with $[\,e_2'/x\,]\, T$. In fact, the rule (T_CONV) does connect these two types by allowing terms to be retyped at different, but equivalent types. The type equivalence, denoted by $\equiv$, is given as follows:

Definition 1 (Type Equivalence)**.** *$T_1 \Rrightarrow T_2$ iff there exist some T, x, E, e_1, and e_2 such that $T_1 = [\,E[e_1]/x\,]\, T$, $T_2 = [\,E[e_2]/x\,]\, T$, and $e_1 \rightsquigarrow e_2$. Type equivalence $\equiv$ is the symmetric and transitive closure of $\Rrightarrow$.*

Computation typing rules are also added. The rule (CT_CBIND), which looks similar to (CT_BIND), accepts bind constructs $x \leftarrow \mathsf{do}\, c_1; c_2$ where c_1 and c_2 are typed under the current store μ and the empty store, respectively; the differences of stores in c_1 and c_2 stems from the fact that c_1 will be executed under μ but c_2 may or may not (since c_1 can mutate the store). Similarly, remaining computations in other rules are also required to be typed under the empty store. The rule (CT_CHECK), applied to assertion checks, requires the checking state p to be well typed under the current store μ in the third premise and be an actual intermediate state in the last premise. The final rule (CT_CONV) is needed because thunks can be typed at equivalent types by (T_CONV).

Store well-formedness is derived by (WF_STORE), which demands that all memory cells in well-formed stores be allocated at given regions γ' and their contents have types given by store typing contexts; $\Sigma|_{\gamma'}$ is the same store typing context as Σ except that its domain is restricted to addresses with regions in γ'.

Finally, using a local store typing context Σ', the rule (PT) gives type T to checking state $\nu\gamma'.\langle \mu' \mid c'\rangle$ if and only if the followings hold. First, the local store μ' maps only references at the local regions γ' and each contents in it have the type assigned by Σ'. Second, the domains of μ' and Σ' coincide. Third, using the local information μ', Σ', and γ', c' can be given a Hoare type whose result type is T, allowed to read data from references at regions γ''. Finally, the precondition of c' holds under the concatenation of the program store and the local store.

6.2 Type Soundness

We show type soundness, which says that, given a well typed program c of $\{\top\}x{:}T\{A_2\}^{\langle\{r\},\{r\}\rangle}$, $p = \nu\emptyset.\langle\emptyset \mid \nu r.\, c\rangle$ results in a value, raises an exception, or diverges and, moreover, if p terminates at value v with store μ, v satisfies refinements in T and $\mu \models [\,v/x\,]\,A_2$ holds. Although the proof of type soundness follows progress and preservation as in the previous work [38], this paper states just their simplified versions; the full statements are shown in the supplementary material. We assume that $ty\,(op)$ and $[\![op]\!]$ agree in a certain sense; also see the supplementary material for details.

Lemma 2 (Value Inversion)**.** *If $\Sigma;\gamma;\emptyset \vdash v : \{x{:}T \mid c\}$, then $\emptyset \models [\,v/x\,]\,c$.*

Lemma 3 (Progress)**.** *If $\emptyset;\Sigma;\gamma \vdash p : T^{\emptyset}$, then: (1) $\emptyset \mid p \hookrightarrow p'$ for some p'; (2) $p = \nu\gamma'.\langle\mu' \mid \mathsf{return}\; v'\rangle$ for some γ', μ', and v'; or (3) $p = \nu\gamma'.\langle\mu' \mid \Uparrow\ell'\rangle$ for some γ', μ', and ℓ'.*

Lemma 4 (Preservation)**.** *If $\emptyset;\Sigma;\gamma \vdash p : T^{\emptyset}$ and $\emptyset \mid p \hookrightarrow p'$, then $\emptyset;\Sigma;\gamma \vdash p' : T^{\emptyset}$.*

Lemma 5 (Postcondition Satisfaction)**.** *If (1) $\mu;\Sigma;\gamma;\emptyset \vdash c : \{A_1\}x{:}T\{A_2\}^{\langle\gamma,\gamma\rangle}$, (2) $\gamma \vdash \mu : \Sigma^{\gamma}$, (3) $\mu \models A_1$, and (4) $\emptyset \mid \nu\gamma.\langle\mu \mid c\rangle \hookrightarrow^{*} \nu\gamma'.\langle\mu' \mid \mathsf{return}\; v'\rangle$, then $\mu' \models [\,v'/x\,]\,A_2$.*

Theorem 1 (Type Soundness)**.** *Suppose that $\emptyset;\emptyset;\{r\};\emptyset \vdash c : \{\top\}x{:}T\{A_2\}^{\langle\{r\},\{r\}\rangle}$. Let $p = \nu\emptyset.\langle\emptyset \mid \nu r.\, c\rangle$. Then, one of the followings hold: (1) $\emptyset \mid p \hookrightarrow^{*} \nu\gamma.\langle\mu \mid \mathsf{return}\; v\rangle$ for some γ, μ, and v; (2) $\emptyset \mid p \hookrightarrow^{*} \nu\gamma.\langle\mu \mid \Uparrow\ell\rangle$ for some γ, μ, and ℓ; or (3) there is an infinite sequence of computation $\emptyset \mid p \hookrightarrow p_1, \emptyset \mid p_1 \hookrightarrow p_2, \cdots$. Moreover, if (1) holds, then: (a) if $T = \{x{:}T_0 \mid c_0\}$, then $\emptyset \models [\,v/x\,]\,c_0$; and (b) $\mu \models [\,v/x\,]\,A_2$.*

Proof. By Progress and Preservation. The properties (a) and (b) are shown by Lemmas 2 and 5. □

7. Static Contract Verification

This work studies "post facto" static verification of state-dependent contracts—more precisely, we identify assertions such that programs with and without them are contextually equivalent [3]. This paper focuses on two verification techniques: elimination of assertions for pre- and postconditions and region-based local reasoning. Note that, although we are interested in, this paper is not concerned about specific verification algorithms; instead, we study what static checking can verify as in the earlier work on manifest contracts [3, 15, 25]. We do not present the formal definition of contextual equivalence here; interested readers are referred to the supplementary material.

7.1 Elimination of Pre- and Postcondition Assertions

Intuitively, a precondition of a computation always holds if it is implied from the other, already established preconditions of the computation. For example, for string table t and string s in Figure 1, the contract $x \Leftarrow\, !t; \mathsf{return}\,(\mathtt{length}\, x \neq 0)$, which is needed to calculate the reciprocal of t's size, would be implied from $x \Leftarrow\, !t; \mathsf{return}\,(\mathtt{mem}\, x\; s)$ with the aid of a theorem prover (here, $!t$ can be seen as an uninterpreted function which returns a list with s). To formalize this "implication", we consider closing substitutions, which give interpretations to free variables, and possible stores under which contracts evaluate. In what follows, σ denotes mappings from term variables to values and from region variables to region variables and write $\sigma(\gamma)$ for the image of γ under σ.

Definition 2 (Closing Substitution and Possible Store)**.** *We write $\Sigma;\gamma;\Gamma \vdash \langle\mu,\sigma\rangle^{\langle\gamma_{\mathtt{r}},\gamma_{\mathtt{w}}\rangle}$ when there exist some Σ' and γ' such that: (1) $\Sigma \subseteq \Sigma'$; (2) $\gamma \subseteq \gamma'$; (3) for any $r \in \gamma$, $r \notin dom\,(\sigma)$; (4) for any $r \in \Gamma$, $\sigma(r) \in \gamma'$; (5) for any $x{:}T \in \Gamma$, $\Sigma';\gamma';\emptyset \vdash \sigma(x) : \sigma(T)$; and (6) $\gamma' \vdash \mu : \Sigma'^{\sigma(\gamma_{\mathtt{r}})\cup\sigma(\gamma_{\mathtt{w}})}$. We write $\Sigma;\gamma;\Gamma \vdash \sigma$ for $\Sigma;\gamma;\Gamma \vdash \langle\emptyset,\sigma\rangle^{\langle\emptyset,\emptyset\rangle}$.*

What is meant by "contract A' is implied from A" is that, for any interpretation of free variables and store such that A results in true, so does A'. We write A, A' for the concatenation of A and A'.

Definition 3 (Contract Implication)**.** *Suppose that $\gamma;\Gamma \vdash^{\varrho} A, A'$. Then, A' is implied from A if, for any μ and σ such that $\emptyset;\gamma;\Gamma \vdash \langle\mu,\sigma\rangle^{\varrho}$ and $\mu \models \sigma(A)$, $\mu \models \sigma(A')$ holds.*

Lemma 6 (Precondition Assertion Elimination)**.** *Suppose that $\gamma;\Gamma \vdash c : \{A_1, c_1\}x{:}T\{A_2\}^{\varrho}$. If c_1 is implied from A, $\mathsf{assert}\,(c_1)^{\ell}; c$ and c are contextually equivalent.*

Since postcondition checks are a derived form using assertions, we can show elimination of postcondition assertions as a corollary.

Corollary 1 (Postcondition Assertion Elimination)**.** *Suppose that $\gamma;\Gamma \vdash c : \{A_1\}x{:}T\{A_2\}^{\varrho}$. For any c_2, if $\gamma;\Gamma,x{:}T \vdash^{\varrho} A_2, c_2$ and c_2 is implied from A_2, then $c;\lambda x.\mathsf{assert}\,(c_2)^{\ell}$ and c are contextually equivalent.*

These elimination techniques of assertions enable us to strengthen preconditions and weaken postconditions semantically.

Corollary 2 (Semantic Weakening)**.** *Suppose that $\gamma;\Gamma \vdash c : \{A_1\}x{:}T\{A_2\}^{\varrho}$. For any A_1' and A_2', if $\gamma;\Gamma \vdash^{\varrho} A_1'$ and $\gamma;\Gamma,x{:}T \vdash^{\varrho} A_2'$ and A_1 is implied from A_1' and A_2' is implied from A_2, then $\mathsf{assert}\,(A_1)^{\ell_1}; c;\lambda x.\mathsf{assert}\,(A_2')^{\ell_2}$ and c are contextually equivalent at $\{A_1'\}x{:}T\{A_2'\}^{\varrho}$.*

7.2 Region-Based Local Reasoning

Local reasoning allows applying verification methods to subcomponents of a program locally, so it is important to scale up verification to large programs. We achieve *region-based* local reasoning in the form similar to Separation logic [34, 36], where the so-called frame rule plays an important role: given a computation requiring precondition P, which specifies a heap before the computation, and guaranteeing postcondition Q, which specifies the heap after the computation, the rule allows the computation to require a condition R separated from P and Q and guarantee the same condition. It is represented in the form of Hoare triples as follows:

$$\frac{\{P\}\,c\,\{Q\}}{\{P * R\}\,c\,\{Q * R\}}$$

where $P * R$ is the separating conjunction of P and R and states that P and R specify disjoint heaps. In our context, computation c can require and guarantee any condition A if A never mentions references mutated by c. The effect system in Section 4 is useful to verity whether A mentions such references because it can analyze what references computations manipulate with the help of regions. To formalize the local reasoning, we first introduce the notion of "disjointness" of regions—region sets γ_1 and γ_2 are disjoint if the results of application of any region substitutions are disjoint.

Definition 4 (Disjoint Regions)**.** *We write $\Sigma;\gamma;\Gamma \vdash \gamma_1\; \mathsf{disj}\; \gamma_2$ when, for any σ such that $\Sigma;\gamma;\Gamma \vdash \sigma$, $\sigma(\gamma_1) \cap \sigma(\gamma_2) = \emptyset$.*

Lemma 7 (Local Reasoning)**.** *Suppose $\gamma;\Gamma \vdash c : \{A_1\}x{:}T\{A_2\}^{\langle\gamma_{\mathtt{r}},\gamma_{\mathtt{w}}\rangle}$. For any A and ${\gamma_{\mathtt{r}}}' \subseteq \gamma_{\mathtt{r}}$, if $\gamma;\Gamma \vdash^{\langle{\gamma_{\mathtt{r}}}',\emptyset\rangle} A$ and $\emptyset;\gamma;\Gamma \vdash {\gamma_{\mathtt{r}}}'\; \mathsf{disj}\; \gamma_{\mathtt{w}}$, then, for any fresh y, $c;\lambda y.\mathsf{assert}\,(A)^{\ell}$ and c are contextually equivalent at $\{A_1, A\}x{:}T\{A_2, A\}^{\langle\gamma_{\mathtt{r}},\gamma_{\mathtt{w}}\rangle}$*

The local reasoning enables us to recover contract information lost by assignment.

Corollary 3. *Suppose that (1)* $\gamma; \Gamma \vdash e_1 : \mathsf{Ref}_r\, T'$*; (2)* $\gamma; \Gamma \vdash e_2 : T'$*; (3)* $\gamma; \Gamma \vdash^{\langle \gamma_{\mathrm{r}}, \emptyset \rangle} A$*; and (4)* $\emptyset; \gamma; \Gamma \vdash \gamma_{\mathrm{r}}\, \mathsf{disj}\, \{r\}$*. Then,* $x \Leftarrow e_1 := e_2; \mathsf{return}\,(); \lambda x.\mathsf{assert}\,(A)^{\ell}$ *and* $x \Leftarrow e_1 := e_2; \mathsf{return}\,()$ *are contextually equivalent at* $\{A\}x{:}T\{A\}^{\langle \gamma_{\mathrm{r}}, \{r\} \rangle}$.

Although the local reasoning would be a useful technique, its applicability rests on how many regions are judged to be disjoint. Unfortunately, $\lambda^{\mathrm{H}}_{\mathsf{ref}}$ does not have a very strong power for this judgment—nonempty region sets are disjoint only if they contain no region variables introduced by region abstraction, which is not very satisfactory because region abstractions are a key feature to promote program reuse and so would be often used. We could address this issue by adding operations on region variables. For example, consider an extension of $\lambda^{\mathrm{H}}_{\mathsf{ref}}$ with an equality operator $r == s$ on regions, which behaves as follows:

$$r == r \rightsquigarrow \mathsf{true} \qquad r == s \rightsquigarrow \mathsf{false} \quad (\text{if } r \neq s)$$

Using this equality, we can state that region r is different from s as a contract and then $\{r\}$ and $\{s\}$ are judged to be disjoint even if either or both of them are abstracted regions because closing substitutions must respect the contract.

8. Related Work

Hoare Type Theory Hoare Type Theory [31, 32] is a theoretical framework to verify stateful programs with higher-order functions statically, incorporating the core ideas of Hoare logic (and Separation logic) into a type system with dependent types. The key idea of HTT is to introduce Hoare types, where pre- and postconditions are written in classical multi-sorted first-order logic with predicates to specify heaps. Computational Hoare types in this work are a variant of HTT's Hoare types and allow pre- and postconditions to be computational so that their dynamic checks are possible. On one hand, in addition to dynamic checking, the computability of pre- and postconditions enables programmers to give natural specifications using program functions defined by them, as the last example in Section 2.3, which uses interface functions to mention internal states of an abstract type. Although Nanevski et al. extended HTT to deal with internal states of functions [33], the extension does not allow use of user-defined functions in specifications. Another benefit of our work is that we can reuse program components in specifications easily. On the other hand, the computability restricts the expressive power of specifications—for example, unlike HTT, it appears to be difficult to work well with existential quantifier and specify the relationship between stores before and after computation—though expressive powers of computational and noncomputational Hoare types are incomparable generally because the former accepts nonterminating contracts whereas the latter does not. It is left as future work to give a remedy for the defect of computational Hoare types.

Other work on static verification of stateful programs Other than HTT, there are researches for static verification of stateful programs with dependent type systems. Dependent ML [53, 54] and Applied Type System [52] are programming languages with support for static verification of stateful programs. Specifications in these languages are neither state-dependent nor computational. Vekris et al. [49] study a refinement type system for static verification of TypeScript [29] programs, which are functional, object-oriented, and imperative. Their system supports imperative features, such as variable assignment and objects with mutable fields, from TypeScript and is able to specify not only refinements but also class invariants. Although specifications in the system can refer to variables and object fields, Vekris et al. deal with only state-*in*dependent specifications by transforming code with variable assignment to a static single assignment form [1, 37] and restricting fields accessible from specifications to immutable ones (immutability are annotated in field declarations). Gordon et al. [17] proposed *rely-guarantee references*, where a reference is augmented with a guarantee relation, which describes possible actions through the reference, and a rely relation, which describes possible actions through aliases, and developed a framework with rely-guarantee references to verify that assignment to a reference does not invalidate predicates with respect to aliases of the reference. Applying their approach to dynamic checking is interesting, but it would need a run-time mechanism to monitor guarantee and rely relations. Swamy et al. [45, 46] developed Dijkstra monads, a variant of Nanevski et al.'s Hoare types, to verify effectful programs automatically. We expect that their technique can be applied to our work for automatic verification.

In objected-oriented languages, many techniques—e.g., ownership types [8–10], variants of Separation logic [4, 35, 42], dynamic frames [22, 23], implicit dynamic frames [41], regional logic [2], etc.—have been studied to address the frame problem [6], which is a common theme in verification of programs with pointer aliasing. We address the frame problem with Hoare types and a region-based effect system, but the earlier work above would inspire us to refine our approach and, furthermore, to investigate better approaches.

Contracts for references Flanagan, Freund, and Tomb [16] studied combination of static and dynamic approaches to checking contracts of (im)mutable objects. Although their goal is similar to ours, they allow only pure contracts which never depend on even locally allocated references whereas we accept even state-dependent contracts. That work also proposed reference guards, which have been a usual approach to dynamic checking for references [11, 20, 44] (there is also another method [40], though). Perhaps, one might consider that state-dependent contracts could be embedded into reference guards, that is, properties with respect to references could be represented by refining contents of reference types, as type `tbl` in Figure 1. Unfortunately, this approach is not satisfactory because it would not be possible to relate results of two or more stateful computations (e.g., the contract of `add` would be disallowed) or to abstract implementation types of mutable data structures as in the last example of Section 2.3.

Tob and Pucella [48] integrated programs in two languages—one has a conventional type system and the other has an affine type system—by using stateful contracts with assignment. Their system uses contracts to monitor that conventional programs use affine values just once and does not use them as specifications of program components. Disney, Flanagan, and McCarthy [12] proposed a higher-order temporal contract system to monitor temporal behavior of stateful programs by specifying orders in which functions of modules should be called. Though they dealt with predicate contracts with imperative features, the issue of state-dependent contracts is not in their interests.

9. Conclusion

We address the issue of state-dependent manifest contracts by introducing a region-based effect system with computational Hoare types. To formalize our ideas, we define $\lambda^{\mathrm{H}}_{\mathsf{ref}}$, where refinements are checked with casts and pre- and postconditions of Hoare types are checked with assertions, and show its type soundness. We also study "post facto" static verification, in particular, elimination of assertions for pre- and postconditions and region-based local reasoing. This work is a stepping stone for integrating static and dynamic verification of state-dependent contracts and we have many directions of future work. For example, it is interesting to investigate how we can strengthen our contract language so that relationships between heaps before and after computation can be expressed. Implementation of our calculus is also left for future work.

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proc. of ACM POPL*, pages 1–11, 1988.

[2] A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *Proc. of ECOOP*, pages 387–411, 2008.

[3] J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. Polymorphic contracts. In *Proc. of ESOP*, pages 18–37, 2011.

[4] J. Bengtson, J. B. Jensen, F. Sieczkowski, and L. Birkedal. Verifying object-oriented programs with higher-order Separation logic in Coq. In *Proc. of Interactive Theorem Proving*, pages 22–38, 2011.

[5] M. Blume and D. A. McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 16(4-5):375–414, 2006.

[6] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Trans. Software Eng.*, 21(10):785–798, 1995.

[7] C. Calcagno, S. Helsen, and P. Thiemann. Syntactic type soundness results for the region calculus. *Information and Computation*, 173(2): 199–221, 2002.

[8] D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 15–58. 2013. .

[9] D. G. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proc. of ACM OOPSLA*, pages 48–64, 1998.

[10] W. Dietl and P. Müller. Object ownership in program verification. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 289–318. 2013. .

[11] C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Complete monitors for behavioral contracts. In *Proc. of ESOP*, pages 214–233, 2012.

[12] T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. In *Proc. of ACM ICFP*, pages 176–188, 2011.

[13] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *Proc. of ACM POPL*, pages 236–249, 2003.

[14] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proc. of ACM ICFP*, pages 48–59, 2002.

[15] C. Flanagan. Hybrid type checking. In *Proc. of ACM POPL*, pages 245–256, 2006.

[16] C. Flanagan, S. N. Freund, and A. Tomb. Hybrid types, invariants, and refinements for imperative objects. In *ACM FOOL/WOOD*, 2006.

[17] C. S. Gordon, M. D. Ernst, and D. Grossman. Rely-guarantee references for refinement types over aliased mutable data. In *Proc. of ACM PLDI*, pages 73–84, 2013.

[18] M. Greenberg. *Manifest Contracts*. PhD thesis, University of Pennsylvania, 2013.

[19] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *Proc. of ACM POPL*, pages 353–364, 2010.

[20] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, pages 1–18, 2007.

[21] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[22] I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Proc. of Formal Methods*, pages 268–283, 2006.

[23] I. T. Kassios. The dynamic frames theory. *Formal Asp. Comput.*, 23 (3):267–288, 2011.

[24] K. Knowles and C. Flanagan. Compositional reasoning and decidable checking for dependent contract types. In *Proc. of ACM PLPV*, pages 27–38, 2009.

[25] K. Knowles and C. Flanagan. Hybrid type checking. *ACM TOPLAS*, 32(2:6), 2010.

[26] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proc. of ACM POPL*, pages 47–57, 1988.

[27] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Trans. Electron. Comput.*, 9:39–47, 1960.

[28] B. Meyer. *Object-Oriented Software Construction, 1st Edition*. Prentice-Hall, 1988. ISBN 0-13-629031-0.

[29] Microsoft Corporation. TypeScript language specification. URL `https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md`. Accessed on 2016-06-06.

[30] E. Moggi. Computational lambda-calculus and monads. In *Proc. of LICS*, pages 14–23, 1989.

[31] A. Nanevski and G. Morrisett. Dependent type theory of stateful higher-order functions. Technical Report TR-24-05, Harvard University, 2005.

[32] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *Proc. of ACM ICFP*, pages 62–73, 2006.

[33] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in Hoare type theory. In *Proc. of ESOP*, pages 189–204, 2007.

[34] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proc. of CSL*, pages 1–19, 2001.

[35] M. J. Parkinson and G. M. Bierman. Separation logic for object-oriented programming. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 366–406. 2013. .

[36] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS*, pages 55–74, 2002.

[37] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proc. of ACM POPL*, pages 12–27, 1988.

[38] T. Sekiyama, Y. Nishida, and A. Igarashi. Manifest contracts for datatypes. In *Proc. of ACM POPL*, pages 195–207, 2015.

[39] T. Sekiyama, A. Igarashi, and M. Greenberg. Polymorphic manifest contracts, revised and resolved, 2016. Submitted for publication.

[40] J. G. Siek, M. M. Vitousek, M. Cimini, S. Tobin-Hochstadt, and R. Garcia. Monotonic references for efficient gradual typing. In *Proc. of ESOP*, pages 432–456, 2015.

[41] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and Separation logic. In *Proc. of ECOOP*, pages 148–172, 2009.

[42] J. Smans, B. Jacobs, and F. Piessens. VeriFast for Java: A tutorial. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 407–442. 2013. .

[43] M. H. Sørensen and U. Pawel. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. Elsevier, New York, NY, USA, 2006. ISBN 0444520775.

[44] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: Run-time support for reasonable interposition. In *Proc. of ACM SPLASH/OOPSLA*, pages 943–962, 2012.

[45] N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the dijkstra monad. In *Proc. of ACM PLDI*, pages 387–398, 2013.

[46] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Z. Béguelin. Dependent types and multi-monadic effects in F*. In *Proc. of ACM POPL*, pages 256–270, 2016.

[47] M. Tofte and J.-P. Talpin. Implementation of the type call-by-value λ-calculus using a stack of regions. In *Proc. of ACM POPL*, pages 188–201, 1994.

[48] J. A. Tov and R. Pucella. Stateful contracts for affine types. In *Proc. of ESOP*, pages 550–569, 2010.

[49] P. Vekris, B. Cosman, and R. Jhala. Refinement types for typescript. In *Proc. of ACM PLDI*, pages 310–325, 2016.

[50] P. Wadler. The essence of functional programming. In *Proc. of ACM POPL*, pages 1–14, 1992.

[51] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[52] H. Xi. Applied type system: Extended abstract. In *TYPES*, pages 394–408, 2003.

[53] H. Xi. Dependent ML An approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.

[54] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. of ACM POPL*, pages 214–227, 1999.