BOSTON UNIVERSITY
GRADUATE SCHOOL OF ARTS AND SCIENCES

Thesis

TYPED STATEFUL PROGRAMMING

by

DENGPING ZHU

B.E., Huazhong University of Science and Technology, China, 1999

Submitted in partial fulfillment of the requirements for the degree of Master of Arts 2004
Approved by

First Reader

Hongwei Xi, Ph.D.
Assistant Professor of Computer Science.

Second Reader

Assaf Kfoury, Ph.D.
Professor of Computer Science.
Acknowledgments

I am very grateful to my advisor Hongwei Xi for suggesting to me this wonderful topic and giving me appropriate guidance and freedom during my work. I simply cannot imagine where I would be without his encouragement, kindness and all-around smarts. His guidance was instrumental to me in focusing and formulating the problem. His critical feedbacks and engaging arguments contributed immensely to this thesis.

I express my gratitude to Prof. Assaf Kfoury for his insightful comments, and for his kindness in general, which I shall always look up to. His genuine good nature and smile made my days brighter.
Although the resemblance between Floyd-Hoare logic and type system has been observed for a long time, they still reside in two different worlds and little, if there is any, effort has been made to build a bridge between them. In this thesis, we present an approach to support Floyd-Hoare logic-like reasoning via a type system. By enriching a type system with a restricted form of dependent types and state assertions, we are able to enforce more safety properties as well as type safety. We make use of dependent types to capture run-time information statically. On the other hand, incorporating state assertions into type system enables a type checker to track ephemeral program properties at compile time. We formalize a generic type system, presenting both static and dynamic semantics and then establishing its soundness. We present concrete examples to demonstrate the expressiveness and advantages of this type system especially when stateful programming is involved. In particular, we show how Floyd-Hoare logic can be readily encoded in this type system, presenting a natural integration of Floyd-Hoare logic into type theory. On the other hand, this encoding provides an evidence for the reasoning ability of our type system on states.
# Contents

1 Introduction ........................................... 1
   1.1 Introductory Examples .................................. 3
   1.2 Related Work ........................................ 7
       1.2.1 Model Checking .................................... 8
       1.2.2 Program Analysis .................................. 9
       1.2.3 Automated Deduction .............................. 9
       1.2.4 Type Checking .................................... 10
   1.3 Outline ............................................... 13

2 Formal Development ................................. 15
   2.1 Statics .............................................. 15
   2.2 Dynamics ............................................ 20
   2.3 Erasure .............................................. 23
   2.4 Address Polymorphism ............................... 25
   2.5 State Polymorphism .................................. 25
   2.6 Aliasing Control .................................... 26

3 Soundness ............................................. 29

4 Encoding Floyd-Hoare Logic ....................... 46

5 Programming with Typed States ..................... 52
6 Conclusion and Future Work
List of Figures

1.1 Factorial .................................................. 3
1.2 Binary Search ............................................. 4
1.3 Two Short Examples on File Access ......................... 7

2.1 The syntax for statics ....................................... 16
2.2 The rules for sub-typing .................................... 19
2.3 The Syntax for dynamics .................................... 20
2.4 The typing rules for dynamics ............................... 24

4.1 The syntax for SIPL ......................................... 46
4.2 The proof rules for Hoare logic ............................. 47
4.3 Statement translation ....................................... 48

5.1 Reversing Lists ............................................ 53
5.2 Swapping .................................................. 54
Chapter 1

Introduction

A fundamental question in computer science is how to develop reliable software. While hardware improves every day, software is entangled in complexity and programming is error-prone and expensive in terms of time and cost. Most innovations in software industry are confined to finding new things to do with software based on existing technology, and have not give us much surprise in finding new ways to produce software. Now computers are increasingly being introduced into safety-critical systems, such as nuclear control systems. As a consequence, computers have been involved in accidents. Tragedies caused by software failure are painfully clear from Leveson and Turner’s investigation [21].

Creating correct and reliable software consumes vast amounts of human time and energy. Humans propose high-level requirements, including safety properties, for software and translate them into low-level requirements to drive the steps of software design, implementation and testing. Although certain aspects of reliability can be achieved by careful system design and the use of current techniques such as type checking and fault tolerance, program errors have a significant effect on software reliability. A lot of efforts have been made to detect and prevent certain classes of errors in programs.

One practical approach to enforce program safety properties is type checking. Types play an important role in the design and implementation of programming languages. A
well-designed type system can significantly alleviates the burden of testing and debugging by capturing numerous program errors at compile-time. After H. Xi and F. Pfenning successfully made dependent types available in practical programming [43, 35], a type system can even enable a programmer to catch some computation errors statically. For instance, array index out-of-bound can be statically captured by the type system of DML [37]. However, those safety properties a type system can enforce are limited in the sense that it can only guarantee type safety. Some other safety specifications, such as program correctness, are beyond the ability of an ordinary type system such as that of ML. In addition, when stateful programming feature is involved, a pure type system is too weak to capture state changes.

Along another line of research, program reasoning and verification shed some lights on enforcing safety properties about program behavior. Most of existing approaches to program verification are based on Floyd-Hoare Logic [16, 18]. A programmer can make assertions about state changes through a specification language and then use inference rules to check the correctness of those assertions. While Floyd-Hoare logic has been studied for at least three decades, its actual use in general software practice is rare. A compelling reason for this is briefly explained in the following quote.

Although types and assertions may be semantically similar, the actual development of type systems for programming languages has been quite separate from the development of approaches to specification such as Hoare Logic, refinement calculi, or model checking.

-J.C.Reynolds. *Separation Logic: A Logic for Shared Mutable Data Structures* [29]

In this thesis we build a bridge to unify type theory with Floyd-Hoare logic. We call it ATS/ST, which stands for ATS (Applied Type System) with State Types. In the rest of this chapter we use concrete examples to illustrate the practicality of our approach. We also describe the context in which this thesis exists, and then outline the rest of this thesis.
1.1 Introductory Examples

In this section we present several introductory examples to illustrate the expressiveness and advantages of the type system we are going to formulate and study. In order to understand and appreciate those examples in this section, the reader should have a basic knowledge of dependent types, the interested reader is referred to [43] and [35].

fun fact {n:int | n >= 0} (num: int(n)): Int = 
    if num = 0 then 1 else num * fact (num - 1)

Figure 1.1: Factorial

let us start with a very simple example. In Figure 1.1, a program, which implements a factorial function in ML-like syntax. The header in the definition of this function suggests that the following type is assigned to fact:

\[ \forall n : \text{int}\n \geq 0 \supset (\text{int}(n) \rightarrow \text{Int}) \]

Notice the difference between \text{int}, \text{int} and \text{Int}. \text{int} is a sort for static terms. \text{int} is a type constructor which take a static term, say \(n\), of sort \text{int} to form a type \text{int}(n), which is a singleton type containing a dynamic term whose value is \(n\). \text{Int} is the abbreviation for \(\exists n : \text{int}.\text{int}(n)\). A guarded type with the form \(A \supset B\), read as “\(A\) guards \(B\)”, means that \(A\) is a type guard for \(B\). If we want to use a term of a guarded type, we should first prove that the type guard holds. For instance, if a piece of program needs to apply \text{fact} to an integer \(m\), it should provide a proof that persuades the type checker that \(m\) is greater than 0. Then the guard is revoked and the application is permitted. Otherwise, the type checker will complain and this application is forbidden. An obvious advantage of the above code is that we do not need to check whether \text{num} is negative at run time. Here, we call \(n \geq 0\) a proposition instead of an assertion to avoid the potential confusion since we are going to introduce state assertions later.
typedef array {a: type, n:int} = '{
    length: int(n);
    sub: {i: int | 0 <= i < n} int(i) -> a;
    update: (i: int | 0 <= i < n) int(i) * a -> unit
}

fun bsearch (n: int | n >= 0}
    (key: Int, vec: array(Int, n)): Int =
    newvar low, high, mid, x in
    let fun loop {i: int, j: int | 0 <= i <= j+1 <= n}
            ([low: int(i), high: int(j)]; /* none */): {Int} =
            /*none*/; Int) =
            if (!low <= !high) then
                mid := !low + !high / 2;
                x := vec.sub (!mid);
                if (key == !x) then !mid
                else if (key < !x) then
                    (high := !mid - 1; loop ())
                else (low := !mid + 1; loop ())
            else -1
            in
            low := 0; high := vec.length - 1; loop ()
    end
end

Figure 1.2: Binary Search
The next example in Figure 1.2 implements a binary search function on integer arrays. A type definition is first declared, stating that \texttt{array} is a type constructor: given a type $T$ and static integer term $I$, $\text{array}(T, I)$ is a type for records in which the labeled components are assigned the following types:

\begin{align*}
\text{length} & : \text{int}(I) \\
\text{sub} & : \forall a : \text{int}.(0 \leq a \land a \leq I) \supset (\text{int}(a) \rightarrow T) \\
\text{update} & : \forall a : \text{int}.(0 \leq a \land a \leq I) \supset (\text{int}(a) \ast T \rightarrow 1)
\end{align*}

The header in the definition of $bsearch$ means that the following type is assigned to $bsearch$:

$$\forall n : \text{int}. n \geq 0 \supset (\text{Int} \ast \text{array}(\text{Int}, n) \rightarrow \text{Int})$$

In addition, we assign the following type to the function $\text{loop}$,

$$\forall a_1 : \text{int}. \forall a_2 : \text{int}. P \supset (\text{SA} \supset 1 \rightarrow CT)$$

where we have $P = (0 \leq a_1) \land (a_1 \leq a_2 + 1) \land (a_2 + 1 \leq n)$, and $\text{SA} = [\text{low} : \text{int}(a_1), \text{high} : \text{int}(a_2), \text{mid} : \top, x : \top]$, and $CT = [\text{low} : \top, \text{high} : \top, \text{mid} : \top, x : \top] \land \text{Int}$. Some explanation is necessary to help the reader to understand these notations. $P$ is a proposition of static variables. $\text{SA}$ is a \textit{state assertion} which asserts the states of those available resources, such as mutable data structures. For instance, the above state assertion $\text{SA}$ asserts the following: $\text{low}$ and $\text{high}$ are mutable variables which hold two integer $a_1$ and $a_2$, respectively; $\text{mid}$ and $x$ are mutable variables whose value we don’t know or don’t care. The convention we use to form state assertions is simple: If an identifier is not mentioned, then it is mapped to the type $\top$ automatically. $CT$ is a \textit{computation type} which contains a state assertion $[\text{low} : \top, \text{high} : \top, \text{mid} : \top, x : \top]$ and the type $\text{Int}$. We call $A \land B$ an \textit{asserting type}, which says that the state assertions holds and a dynamic term has type $\text{Int}$. Note that $\text{SA} \supset T \rightarrow CT$ is also a guarded type which says that the current state must satisfy $\text{SA}$ in order to reach a term of type $CT$. We will come back to this point later. Similar to the first one, this example also exhibits some potential advantages of our type system in
compiler optimization. For an array of length $I$ and the type of each element in is $T$, the type guard of the function $sub$ precisely states that it returns an element of type $T$ when given an integer equal to $a$ such that $0 \leq a < I$ holds. Clearly, if the subscript function $sub$ is called, there is no need for run-time array bound checks in order to ensure memory safety. Not only does this enhance the robustness of the code, but also its efficiency. Both examples presented above demonstrate that we can gain efficiency without losing safety.

Now we would like to introduce another example which exhibits the potential usage of our system on access control. We present a classical example of file access. The protocol for file access requires that a file must be $open$ before being written or read and should be closed before being opened or the program terminates. Most approaches enforce this protocol by run-time checks. We would like to demonstrate how our system can verify those safety properties statically. The following interface defines the types of each operations over files.

\[
\begin{align*}
\text{fnew} & : \text{String} \rightarrow \exists f : \text{file.}(\text{closed}(f) \land \text{File}(f)) \\
\text{fopen} & : \forall f : \text{file}\text{.closed}(f) \supset (\text{File}(f) \rightarrow (\text{open}(f) \land 1)) \\
\text{fclose} & : \forall f : \text{file}\text{.open}(f) \supset (\text{File}(f) \rightarrow (\text{closed}(f) \land 1)) \\
\text{fwrite} & : \forall f : \text{file}\text{.open}(f) \supset ((\text{File}(f), \text{String}) \rightarrow (\text{open}(f) \land 1))
\end{align*}
\]

We demonstrate the ability of our type system with some short functions and show how our type system detects potential usages which violate the file access protocol.

The first example is not well-typed because $myFile$ is a new file whose state is $open$ according to the signature of $fnew$ and thus it can not be read or written, but the second line tries to write to $myFile$ and the current state can not satisfy the type guard $open(f)$ of $fwrite$. The second example solve this problem by changing the file state from $closed$ to $open$ before writing to it.
1: let myFile = fnew ("proof.tex") in
2: fwrite (myFile, "test") // type error: f is not open.

3: let myFile = fnew ("proof.tex") in
4: let _ = fopen (myFile) in
5: fwrite(myFile, "test") // writing allowed.

Figure 1.3: Two Short Examples on File Access

1.2 Related Work

In this section we describe the context in which this thesis exists. Due to the vastness of this field, it is certainly beyond reasonable hope to mention even a moderate part of the research work on program verification. We shall mention some efforts toward this direction and examine those which have a close connection to our work in details.

In general, it is not possible to come up with an algorithm to decide whether a program accomplished the intentions of its user because this kind of problems is undecidable. Therefore, many efforts have been made to prevent and detect obvious or potential errors in a program. In industry world testing is the most popular and practical way to eliminate errors (because there is no better choice). But, testing can only prove the presence of errors and not their absence. For a variety of reasons, some of which are to be explained in the following presentations, program verification was a fertile research field and its actual use in practical software development seems to be greatly limited. Now, this field is experiencing a resurgence. There are a lot of research areas toward this direction which can classified into four categories: Model Checking, Program Analysis, Automated Deduction and Type Checking.
1.2.1 Model Checking

Model checking [9] is the algorithmic exploration of the state spaces of finite state models of systems. It has been used in hardware verification for more than twenty years and it is proved to be useful in practice. The growing importance of model checking in hardware verification and the difficulty of producing reliable software are driving a growing interest in the application of model checking to software. Model checking focuses on the evolving behavior of a program over time and use some pre-defined algorithms to check whether or not the program meets a temporal property (for instance, there is no deadlock). Traditional model checking [2, 4] is based on the following paradigm: build an abstract finite state model, check the desired properties, if fails then refine the models and start over. Since both step 1 and step 2 are computationally hard problems, the method does not scale to large systems without additional optimizations. Model checking suffers from the state space explosion, which means that the size of the state space of a system can be exponential in the size of a system. In addition, the abstraction is highly non-trivial.

Currently there are several on-going projects in this area such as SLAM [3]. Some of them are very promising and being applied in safety-demanding software such as device drivers. In [2] T. Ball and S.K. Rajamani present an approach for validating temporal safety properties of software that uses a well-defined interface. They have applied this to a number of Windows NT and XP device drivers to validate critical safety properties such as correct locking behavior. A major expense in their process is to check the reachability of Error state (step 2), whose running time, in worst case, is $O(N(GL)^3)$, where $N$ is the size of the boolean program (which is the result of the abstraction step), $G$ is the number of global states , and $L$ is the maximum number of local states over all procedures. In the worst case, the number of states is exponential to the maximal number of variables in scope. Additionally, there is a substantial overhead in SLAM process since it have to iterate many times. Lazy abstraction [15] addresses this problem and provides a way to integrate and optimize the three phases of the abstract-check-refine loop.
1.2.2 Program Analysis

Program analysis detects the presence or absence of facts (invariants) in programs. In most cases, it uses those knowledge to rewrite the original program to improve its performance. The traditional application of program analysis is in compiler optimization. With performance improvement of optimized code being the main metric of program analysis, the program analysis community has to compete with the tempo of hardware according to Moore’s Law. Gordon Moore made his famous observation [24] in 1965 that the capacity of semiconductor ICs doubled every 18 to 24 months. This trend has been maintained and still holds true today. In 1998 Todd Proebsting made a less optimistic observation about optimizing compilers (known as Proebsting’s law) which states that advances in compiler optimization double computing power every 18 years.

However, program analysis can be useful in some other directions. Daikon [10] can assist a programmer in an evolution task by using a dynamic invariants detector. The on-going project Jiggetai [8] is to develop a tool that reads Java source code and infers generic types for classes that are used polymorphically.

1.2.3 Automated Deduction

One of the most important properties of a program we want to verify is whether or not it carries out its intended functionalities, which can be specified by making general assertions about the values of those relevant variables and program states. The use of assertions to specify and prove correctness of flowchart programs was developed independently by Naur [25] and Floyd [12]. In 1969 C.A.R.Hoare formalized this approach and constructed a partial correctness system [17] which is known as Hoare logic. A Hoare triple has the form \( P\{Q\}R \), where \( P \) is called pre-condition, \( Q \) is a program in consideration, and \( R \) is called post-condition. \( P\{Q\}R \) essentially means that if the program \( Q \) starts at a state satisfying \( P \) and \( Q \) terminates, then, after its execution, it shall reach some state which satisfies the
assertion $Q$. Later a lot of efforts have been made to extend and strengthen Hoare logic. In 1975 Dijkstra invented the notion of weakest preconditions [7] and explored Hoare logic in more details. In [28] the authors present an axiomatic method for proving a number of properties of parallel programs. Recently Separation Logic [29] was introduced to address the problem when shared mutable data structures are present. But, ever since Hoare showed us how to prove program correctness by using assertions, people have been thinking how such reasoning could be automated. Then Nelson provided a solution in his landmark paper by showing how automated decision procedures, such as theorem provers, could assist us by automating much of the logical reasoning about programs. With the evolution of theorem provers, it becomes possible to use them as black boxes in the verification progress. ESC/Java [6] is a programming tool for detecting errors in Java programs based on program verification technology. By using an underlying automatic theorem prover, ESC/Java can detect, at compile time, many common programming errors such as array index bound errors and nil dereference errors. But the trade-off is the programmer has to insert assertions about invariants that her Java code should maintain. In some worst cases, many assertions need to be inserted in order to get useful information from the analysis algorithm.

1.2.4 Type Checking

Types represent the most widespread use of program safety specifications and play a pivotal role in the design and implementation of programming languages. Not only can types document a program consistently, but also provide a way for compiler to detect and prevent a variety of program errors at compile time. The use of types to facilitate programming goes back to the early days of FORTRAN. After several decades of work on type theory, people gain a lot of knowledge and techniques on type system design. The widely-used programming language JAVA is an example. However, the fact that a program passes the type checker does not ensure that this program accomplish the user’s intention. For
instance, we can define a function `IncreaseByOne`, which should take an integer `n` and return `n + 1`, as `fun IncreaseByOne (n:int):int = n-1`. No doubt that the type checker will treat `IncreaseByOne` as a good program since it does not violate type safety. But, obviously, `IncreaseByOne` does not carry out the desired functionality. The reason is in that type checking can only enforce type safety which focuses on the compatibility of producers and consumers of data. In another word, types in programming languages such as ML and Java are of relatively limited expressive power when compare to Floyd-Hoare logic. In the recent two decades many efforts have been made to enrich type system so that it could capture more program invariants and enforce more safety properties.

In Maritin-l[of’s constructive type theory [23, 26], dependent types offer a precise means to capturing program properties; complex specifications can be expressed in terms of dependent types; if programs can be assigned such dependent types, they are guaranteed to meet the specifications. However, because of no separation between programs and types, that is, programs may be used to construct types, a language based on Martin-l[of’s type theory is often too pure and limited to be useful for practical purpose.

In Dependent ML (DML), a restrict form of dependent types is proposed that completely separates programs from types, this design makes it rather straightforward to support realistic programming features such as general recursion and effects in the presence of dependent types. Subsequently, this restricted form of dependent types is used in designing Xanadu [36] and DTAL [42] and attempts to reap similar benefits from dependent types in imperative programming. While it can be readily noticed that there is a close relation between Floyd-Hoare logic and the type system of Xanadu, such a relation is difficult to be made concrete given the rather ad hoc nature of the type system of Xanadu.

Along another line of research, a new notion of types called guarded recursive (g.r.) datatypes is recently introduced [41]. Noting the close resemblance between the restricted form of dependent types (developed in DML) and g.r. datatypes, we immediately initiate an effort to design a unified framework for both forms of types, leading to the formalization
There have been a great number of research activities on verifying program properties by tracking state changes. For instance, Cyclone [20] allows the programmer to specify safe stack and region memory allocation. In [22], the author proposes an effective theory of type refinements, where the aim is to develop a general theory for reasoning about the behavior of effectual programs. They develop an explicit two level system: the first level is an ordinary ML-style type system and the second level is the type refinements based on intuitionistic linear logic. But their framework is still not general enough to accommodate some states. For instance, they do not provide a way for programmers to create a new variable and track the states of this variable. The expression refinements $\eta$ has the form $\exists[\vec{b}](\phi; \psi)$, which can not capture some subtle relations when dependent types is present. For example, suppose that we define a function $f$ which takes an integer and returns a value with type $int(n)$ provided that $n \geq 5$, the type of $f$ is beyond the expressive power of $\eta$ since it can not reflect the witness $n \geq 5$ before constructing the return type for $f$. In addition, the world $\omega$ in [22] is growing during the life-cycle of a program. It’s unclear to us how to remove a state from $\omega$. This will lead to the efficiency problem especially when the program is very large. To the concern of Floyd-Hoare logic, we do not know how to use their framework to encode it because they do not introduce singleton types for those propositions and thus there seems no way to deal with the (while) rule.

There exist a lot of languages designed to verify particular safety properties. Vault [5, 11] has been applied to verification of safety conditions in device drivers. Its type system is derived from capability calculus, which is somewhat ad hoc, and alias types. In their later work [11], M. Fahndrich and R. Deline present a way to solve the conflict between linearity, which provides powerful reasoning about state changes, and aliasing, which is restricted when checking a protocol on an object. Although their system is capable of protocol checking and resource management, it is not strong enough to capture some program invariants and state changes involved with values. For instance, they can not statically verify whether
a specification for a function \( f \) which asserts that the value of \( x \) will greater than 100 after the execution of \( f \) holds. And thus it is not suitable for an encoding of Floy-Hoare logic. As far as we are aware, the semantics of Vault has not been fully formalized. Guarded types also appear in Vault, but it is much more restricted than ours. A guarded type in Vault has the form \( C \triangledown \tau \), where \( C \), called type guard, is either true or a conjunction of one or more of atomic predicates, which is whether a given key is in the held-key set. In our work, we provide two kinds of guarded types, \( SA \triangleright \tau \) and \( P \triangleright \tau \), where \( SA \) is a collection of state assertions and \( P \) is a set of propositions on index variables.

Separation Logic [29] is an extension of Floy-Hoare Logic that permits reasoning about low-level imperative programs that use shared mutable data structures. Assertions are extended by introducing separating conjunction, which asserts that its sub-formulas hold for disjoint parts of the heap, and separating implication. This extension significantly alleviates the complexity and poor scalability of reasoning about program properties with controlled sharing and thus makes the concise and flexible description of structures possible. Separation logic can provide great help to define specification languages. P.W.O’Hearn and H.Yang also made a lot of efforts on reasoning for stateful programming and shared data structures [44, 27, 30, 45]. However, they only provide a logical reasoning about program properties. None of them present a formal system, such as a type system, to check those properties.

1.3 Outline

The rest of the thesis is organized as follows. In the next chapter, we present the formal development of ATS/ST. We completely separate programs from types. Programs and types correspond to dynamics and statics in our formulation. We also introduce a typed imperative language and present its operational semantics. In Chapter 3 we prove various well-known properties of ATS/ST such as subject reduction and progress and thus establish
its soundness. We show that Floyd-Hoare logic can be readily encoded in ATS/ST in Chapter 4. In Chapter 5, we present some practical examples and demonstrate the expressive power of ATS/ST. Lastly, we conclude and point out some directions for future research.
Chapter 2

Formal Development

In this chapter we formulate a type system ATS/ST that extends a particularly chosen applied type system with a notion of states. However, we emphasize that the approach we take can be readily applied to a generic applied type systems as is formalized in [39, 40].

2.1 Statics

The static component of ATS/ST, or statics, is simply typed, and we use the name sort for a type in the statics. The syntax for the statics is given in Figure 2.1.

We assume that there are four sorts bool, int, type and addr for static terms $s$. We use $a$ for static variables, $i$ for integer constants such as $0, 1, -1, \cdots$, and $b$ for boolean values $tt$ (true) and $ff$ (false). Also, we can use some primitive functions on integers such as $+, -, \ast$ and $/$ and booleans such as $>, \geq$ and $<$ to form static integer terms $I$ and static boolean terms (or static propositions) $P$. We use $T$ to range over static type terms (or types, for short), which acts as types for dynamic terms that are to be introduced later.

We assume an infinite set of identifiers, which could, for instance, be represented as integers. An identifier belongs to the sort addr. That’s to say, we treat an identifier as an unique memory address. The type of $id$ is $\text{ptr}(id)$. A state assertion $SA$ is a finite mapping that maps identifiers to types. We use $[]$ for the empty mapping, and $SA[id : T]$ for the
The syntax for statics

mapping that extends $SA$ with a link from $id$ to $T$, where we assume that $id$ is not in the domain of $SA$ (denoted as $\text{dom}(SA)$). We may also write $[id_1 : T_1, \ldots, id_n : T_n]$ for a state assertion $SA$ such that $\text{dom}(SA) = \{id_1, \ldots, id_n\}$ and $SA(id_i) = T_i$ for $1 \leq i \leq n$, where $id_1, \ldots, id_n$ are assumed to be distinct identifiers. Note that if $SA(id) = T$, it means that the type of the content at address denoted by $id$ is $T$.

We use $\Sigma$ for static variable context, which are finite mappings from static variables to types, and $\vec{P}$ for a (possibly empty) sequence of propositions. Given a static term $s$, we write $\Sigma \vdash s : \sigma$ to mean that $s$ can be assigned the sort $\sigma$. All the standard sorting rules are omitted here. A static substitution $\Theta$ is a finite mapping from static variables to static terms, and we use $s[\Theta]$ for the result of applying $\Theta$ to $s$. Also, we write $\Sigma_0 \vdash \Theta : \Sigma$ to mean that $\Sigma_0 \vdash \Theta(a) : \Sigma(a)$ holds for every $a \in \text{dom}(\Theta) = \text{dom}(\Sigma)$.

**Definition 2.1.1** We write $\Sigma; \vec{P} \models P_0$ for a constraint relation, where we assume $\Sigma \vdash P_0 : \text{bool}$ holds and $\Sigma \vdash P : \text{bool}$ holds as well for each $P$ in $\vec{P}$. A constraint $\Sigma; \vec{P} \models P_0$ is satisfied if for each static substitution $\Theta$ such that $\emptyset \vdash \Theta : \Sigma$ holds, $P_0[\Theta] = \text{tt}$ whenever $P[\Theta] = \text{tt}$ for every $P$ in $\vec{P}$. 

We are not to discuss the issue of effectively determining whether a constraint is satisfied, for which we may simply assume the availability of an oracle.

We now present some intuitive explanation on some forms of types:

- \( \top \) is a top type, that is, every type is a subtype of \( \top \).

- \( \text{bool}(P) \) is a singleton type containing the only truth value equal to \( P \).

- \( \text{int}(I) \) is a singleton type containing the only integer equal to \( I \).

- \( \text{ptr}(id) \) is a type for a pointer to the address denoted by \( id \). Simply put, \( \text{ptr}(id) \) is the type for the address \( id \). This type is a singleton type: any pointer described by this type is a pointer to the one address \( id \) and to no other addresses. Note that \( id \) should have sort \( \text{addr} \). In implementation, we can treat an address as an integer and allow arithmetic operations such as addition on address. Notice that the \( id \) in \( \text{ptr}(id) \) is a static identifier, but \( id \) alone is a dynamic value. In another word, we can think the type of \( id \) is an unpacking of \( \exists id. \text{ptr}(id) \). In addition, note that in the state type \( SA \), \([id : T]\) means that the type of the contents at address \( id \) is \( T \), but \( id \) itself has type \( \text{ptr}(id) \).

- \( SA \supset (T \rightarrow CT) \) is a type for dynamic functions that can be applied to dynamic values of type \( T \) only if the current program state (when the application occurs) meets the state assertion \( SA \); such an application yields a computational dynamic term that can assigned the computational type \( CT \), which is of th form \( \exists \Sigma', \tilde{P}.SA' \land T' \). Clearly, \( SA \) is related to the notion of pre-condition while the state assertion \( SA' \) in \( CT \) corresponds to the the notion of post-condition. For simplicity, we may write \( SA \supset T \rightarrow CT \) for \( SA \supset (T \rightarrow CT) \) in the following presentation.

- \( P \supset T \) is called a guarded type and \( P \land T \) is called an asserting type. As an example, the following type is for a function from natural numbers to negative integers:

\[
\forall a : \text{int}.a \geq 0 \supset (\text{int}(a) \rightarrow \exists a' : \text{int}.a' < 0 \land \text{int}(a'))
\]
The guard \( a \geq 0 \) indicates that the function can only be applied to an integer that is greater than or equal to 0; the assertion \( a' < 0 \) means that the function only returns negative integers. As another example, the following type,

\[
\forall a : bool. bool(a) \rightarrow (a = tt) \land 1
\]

where \( 1 \) is the type for unit, indicates that a boolean value must be true if a function of this type called on the boolean values returns. Hence, we can assign this type to a function that verifies at run-time whether a given assertion (i.e., a boolean expression) holds (i.e., yields true).

A computation type \( CT \) is always of the form \( \exists \Sigma, \vec{P}. SA \land T \), and we may write \( SA \land T \) for \( \exists \emptyset, \emptyset. SA \land T \). Intuitively, \( \exists \Sigma, \vec{P}. SA \land T \) is for a dynamic term that evaluates to a value \( v \) at some state \( ST \) such that for some static substitution \( \Theta \), \( \emptyset \vdash \Theta : \Sigma \) holds, each proposition in \( \vec{P}[\Theta] \) is true, \( v \) is of type \( T[\Theta] \), and \( ST \) meets \( SA[\Theta] \) (for which the precise definition is given later).

As in \( ATS \), we introduce a sub-typing relation \( T_1 \leq_{tp} T_2 \) on static type terms. The rules for deriving sub-typing judgment’s of the form \( \Sigma; \vec{P} \models T_1 \leq_{tp} T_2 \) are given in Figure 2.2, where the obvious side conditions associated with certain rules are omitted.

**Proposition 2.1.2** We have the following:

1. \( \Sigma; \vec{P} \models T \leq_{tp} T \) for all types \( T \).

2. \( \Sigma; \vec{P} \models T_1 \leq_{tp} T_2 \) and \( \Sigma; \vec{P} \models T_2 \leq_{tp} T_3 \) imply \( \Sigma; \vec{P} \models T_1 \leq_{tp} T_3 \) for all types \( T_1, T_2 \) and \( T_3 \).

3. If \( \Sigma, a : \sigma \vdash T_0 : type \) and \( \Sigma \vdash s_0 : \sigma \), then \( \Sigma \vdash T_0[a \mapsto s_0] : type \) holds.

In other words, (1) and (2) mean the sub-typing relation is both reflexive and transitive.

The type equality \( T_1 =_{tp} T_2 \) can be defined as \( T_1 \leq_{tp} T_2 \) and \( T_2 \leq_{tp} T_1 \), though there is no need for it in our formulation.
\begin{align*}
\Sigma; \vec{P} \models T \leq_{tp} T \tag{sub-tp-top} \\
\Sigma; \vec{P} \models P_1 \equiv P_2 \tag{sub-tp-eq} \\
\Sigma; \vec{P} \models \text{bool}(P_1) \leq_{tp} \text{bool}(P_2) \tag{sub-tp-bool} \\
\Sigma; \vec{P} \models I_1 = I_2 \tag{sub-tp-int} \\
\Sigma; \vec{P} \models \text{int}(I_1) \leq_{tp} \text{int}(I_2) \tag{sub-tp-bool} \\
\Sigma; \vec{P} \models \text{SA}' \leq_{sa} \text{SA} \tag{sub-tp-fun} \\
\Sigma; \vec{P} \models T' \leq_{tp} T \tag{sub-tp-guard} \\
\Sigma; \vec{P} \models \text{int}(T) \leq_{tp} \text{int}(T) \tag{sub-tp-ass} \\
\Sigma; \vec{P} \models \forall a : \sigma. T \leq_{tp} \forall a : \sigma. T' \tag{sub-tp-\forall} \\
\Sigma; \vec{P} \models \exists a : \sigma. T \leq_{tp} \exists a : \sigma. T' \tag{sub-tp-\exists} \\
\Sigma; \vec{P} \models \text{SA}(id) \leq_{tp} \text{SA}'(id) \text{ for all } id \in \text{dom}(SA) = \text{dom}(SA') \tag{sub-sa} \\
\Sigma; \vec{P} \models \text{SA} \leq_{sa} \text{SA}' \tag{sub-ct} \\
\Sigma; \vec{P} \models \exists \Sigma_0, \vec{P}_0. \text{SA} \land T \leq_{ct} \exists \Sigma_0', \vec{P}_0'. \text{SA}' \land T' \tag{sub-tp-subst} \\
\Sigma; \vec{P} \models \exists \Sigma_0, \vec{P}_0. T[a \mapsto s_0] \leq_{tp} T'[a \mapsto s_0] \tag{sub-ct-subst} \\
\Sigma; \vec{P} \models \text{SA} \leq_{sa} \text{SA}' \tag{sub-sa-subst} \\
\Sigma; \vec{P} \models \text{SA}[a \mapsto s_0] \leq_{sa} \text{SA}'[a \mapsto s_0] \tag{sub-sa-subst}
\end{align*}

Figure 2.2: The rules for sub-typing
dynamic terms \[ d ::= \mathit{x} \mid \mathit{f} \mid \mathit{id} \mid \mathit{dc}[d_1, \cdots, d_n] \mid \mathit{if}(d_1, d_2, d_3) \mid \mathit{lam} \ x. \ d \]

\[ \mid \mathit{app}(d_1, d_2) \mid \mathit{fix} \ f. \ d \mid \mathit{⊃}^+(v) \mid \mathit{⊃}^-(d) \]

\[ \mid \mathit{let} \ \land (x) = d_1 \ \mathit{in} \ d_2 \mid \mathit{∀}^+(v) \mid \mathit{∀}^-(d) \mid \mathit{∃}(d) \]

\[ \mid \mathit{let} \ \exists(x) = d_1 \ \mathit{in} \ d_2 \mid \mathit{newvar} \ \mathit{id} \ \mathit{in} \ d \]

\[ \mid !d \mid d \leftarrow d \mid \mathit{let}_c x = d_1 \ \mathit{in} \ d_2 \mid (d)_{id} \]

values \[ v ::= \mathit{x} \mid \mathit{id} \mid \mathit{dcc}[v_1, \cdots, v_n] \mid \mathit{lam} \ x. \ d \mid \mathit{⊃}^+(v) \mid \land(v) \]

\[ \mid \mathit{∀}^+(v) \mid \mathit{∃}(v) \]

dynamic var. ctx. \[ \Delta ::= \emptyset \mid \Delta, x : T \]

**Figure 2.3:** The Syntax for dynamics

### 2.2 Dynamics

The dynamic component, or dynamics, is a typed language, and a static type term \( T \) such as a type in the dynamics. The syntax for the dynamics is given in Figure 2.3. We use \( x \) for a \texttt{lam}-variable and \( f \) for a \texttt{fix}-variable, and \( xf \) for either a \texttt{lam}-variable or a \texttt{fix}-variable. A \texttt{lam}-variable is a value but a \texttt{fix}-variable is not. The markers \( \mathit{⊃}^+ \), \( \mathit{⊃}^- \), \( \land(\cdot) \), \( \mathit{∀}^+(\cdot) \), \( \mathit{∀}^-(\cdot) \), \( \mathit{∃}(\cdot) \) are mainly needed to facilitate inductive proofs on typing derivations. In the dynamic term \texttt{newvar id in d}, \texttt{newvar} is treated as a binder and the identifier \( id \) can be \( α \)-renamed. We use \( (d)_{id} \) for a special form of dynamic terms that are only supposed to occur during the evaluation of dynamic terms. There may be some pre-defined dynamic constants \( dc \), each of which is either a dynamic constant constructor \( dcc \) or a dynamic constant function \( dcf \). We write \( dc[d_1, \cdots, d_n] \) for applying \( dc \) to \( n \) arguments \( d_1, \cdots, d_n \), and may write \( dc \) for \( dc[] \). Each dynamic constant \( dc \) is assigned a dynamic constant type (or \( dc \)-type, for short) of the following form:

\[
∀a_1 : σ_1 \cdots ∀a_k : σ_k. P_1 ⊃ (\cdots (P_m ⊃ ([T_1, \cdots, T_n] ⇒ T)) \cdots)
\]
where $n$ indicates the arity of the dynamic constant $dc$. For instance, the division function on integers is assigned the following $dc$-type:

$$\forall a_1 : \text{int} \forall a_2 : \text{int} \ni a_2 \neq 0 \supset [\text{int}(a_1), \text{int}(a_2)] \Rightarrow \text{int}(a_1/a_2)$$

In particular, we assume formally that the unit constant $\langle \rangle$ is assigned the type $[] \Rightarrow 1$, each boolean constant $b$ is assigned the type $[] \Rightarrow \text{bool}(b)$ and each integer constant $i$ is assigned the type $[] \Rightarrow \text{int}(i)$.

In order to assign a call-by-value dynamic semantics to dynamic terms, we make use of evaluation contexts, which are defined as follows:

$$E ::= [] | dc[v_1, \ldots, v_i-1, E, d_{i+1}, \ldots, d_n] | \text{if}(E, d_1, d_2)$$
$$\text{app}(E, d) | \text{app}(v, E) | \supset^{-}(E) | \land(E) | \text{let} \land(x) = E \text{ in } d | \forall^{-}(E) | \exists(E) | \text{let} \exists(x) = E \text{ in } d | \text{id} \leftarrow E | (E)_{st}$$

We define redexes and their reductions as follows.

- $\text{app}(\text{lam } x. d, v)$ is a redex, and $d[x \mapsto v]$ is its reduction.
- $\text{fix } f. d$ is a redex, and $d[f \mapsto \text{fix } f. d]$ is its reduction.
- $\supset^{-}((\supset^{+}(v)))$ is a redex, and $v$ is its reduction.
- $\text{let} \land(x) = \land(v) \text{ in } d$ is a redex, and $d[x \mapsto v]$ is its reduction.
- $\forall^{-}((\forall^{+}(v)))$ is a redex, and $v$ is its reduction.
- $\text{let} \exists(x) = \exists(v) \text{ in } d$ is a redex, and $d[x \mapsto v]$ is its reduction.
- $dcf[v_1, \ldots, v_n]$ is a redex if it is defined to equal some value $v$, and $v$ is its reduction.

Given two dynamic terms $d_1$ and $d_2$ such that $d_1 = E[d]$ and $d_2 = E[d']$ for some redex $d$ and its reduction $d'$, we write $d_1 \rightarrow d_2$ and say that $d_1$ reduces to $d_2$ in one step. Let $\rightarrow^*$ be the reflexive and transitive closure of $\rightarrow$.

A state $ST$ is a mapping from a finite set of identifiers to values. We write $(ST_1, d_1) \rightarrow_{st} (ST_2, d_2)$ if

...
• \(d_1 \rightarrow d_2\) and \(ST_2 = ST_1\), or

• \(d_1 = E[\textbf{newvar} \ id \ in \ d]\) and \(ST_2 = ST_1[\idmap{\id}]{\varnothing}\) and \(d_2 = E[(d_1, d)]\), or

• \(d_1 = E[(v_1)_id]\) and \(ST_1 = ST_2[\idmap{\id}]{v_2}\) for some value \(v_2\) and \(d_2 = E[v_1]\).

• \(d_1 = E[\!\id]\) for some \(\id \in \text{dom}(ST_1)\) and \(ST_2 = ST_1\) and \(D_2 = E[ST_1(\id)]\), or

• \(d_1 = E[\idmap{\id}]{\varnothing}\) for some \(\id \in \text{dom}(ST_1)\) and \(ST_2 = ST_1\) and \(D_2 = E[\langle \rangle]\), where we use \(ST_1[\idmap{\id}]{\varnothing}\) for a finite mapping \(ST\) such that \(ST(\id) = v\) and \(ST(\id') = ST_1(\id')\) for every \(\id' \neq \id \in \text{dom}(ST) = \text{dom}(ST_1)\).

• \(d_1 = E[\textbf{let}_c \ x = v \ in \ d]\) and \(ST_2 = ST_1\) and \(d_2 = d[x \mapsto v]\).

The typing rules for the dynamics are given in Figure 2.4, where there are two forms of typing judgments: \(\Sigma; \vec{P}; \Delta \vdash d : T\) and \(\Sigma; \vec{P}; \Delta; SA \vdash d : CT\). Generally speaking, if \(\emptyset; \emptyset; \emptyset \vdash d : T\) is derivable, then the evaluation of \(d\) can start at any state \(ST\). On the other hand, if \(\emptyset; \emptyset; \emptyset; SA \vdash d : T\) is derivable, then the evaluation of \(d\) can only start at a state \(ST\) which can meet the state assertion \(SA\), that is, for every \(\id \in \text{dom}(SA)\), \(\id \in \text{dom}(ST)\) and \(\emptyset; \emptyset; \emptyset \vdash ST(\id) : SA(\id)\) is derivable.

Note that we have omitted the obvious side conditions associated with certain typing rules. For instance, we need to assume that \(\text{dom}(\Sigma)\) and \(\text{dom}(\Sigma')\) are disjoint in the rule (ty-ext); also, for both the rule (ty-\forall-intro) and the rule (ty-\exists-elim), we need to assume that the variable \(a\) has no free occurrence in \(\vec{P}\) or \(\Delta\). In addition, we need to assume that there are no occurrence of \(\id\) in \(SA'\) and \(T\) for the rule (ty-id-intro).

Most of the typing rules should be easy to understand. The idea behind the rule (ty-ext) is simple: if an identifier is not used during a function call, then the value associated with it stays the same when the function call returns. Clearly, this issue may also be addressed with the notion of row polymorphism [33] and frame rule [27]. With the rule (ty-ext), we can simply treat the usual function type \(T_1 \rightarrow T_2\) as a shorthand for \(\boxed{T_1 \rightarrow T_2}\).


\((T_1 \rightarrow \emptyset \land T_2)\). In addition, it is easy to show that the following rule is admissible.

\[
\frac{\Sigma; \overline{P} \models SA \leq_{sa} SA' \quad \Sigma; \overline{P}; \Delta; SA' \vdash d : CT}{\Sigma; \overline{P}; \Delta; SA \vdash d : CT} \quad \text{(ty-state-sub)}
\]

### 2.3 Erasure

In this section we define an erasure function that erases all the markers \(\sqsupset^+ (\cdot)\), \(\sqsupset^- (\cdot)\), \(\land (\cdot)\), \(\forall^+ (\cdot)\), \(\forall^- (\cdot)\), \(\exists(\cdot)\) in dynamic terms and prove that the erasure function preserves the dynamic semantics.

**Definition 2.3.1** The erasure of a dynamic term \(d\) is defined as follows:

\[
\begin{align*}
|x| &= x, & \text{let } \land (x) = d_1 \text{ in } d_2 | &= \text{let } x = |d_1| \text{ in } |d_2| \\
|f| &= f, & |\forall^+ (v)| &= |v| \\
|id| &= id, & |\forall^- (d)| &= |d| \\
|dc[d_1, \ldots, d_n]| &= dc[|d_1|, \ldots, |d_n|] \\
|\text{let } c x = d_1 \text{ in } d_2| &= \text{let } c x = |d_1| \text{ in } |d_2| \\
\text{let } c x = d_1 \text{ in } d_2| &= \text{let } c x = |d_1| \text{ in } |d_2| \\
\text{let } c x = d_1 \text{ in } d_2| &= \text{let } c x = |d_1| \text{ in } |d_2| \\
\text{let } c x = d_1 \text{ in } d_2| &= \text{let } c x = |d_1| \text{ in } |d_2| \\
\text{let } c x = d_1 \text{ in } d_2| &= \text{let } c x = |d_1| \text{ in } |d_2|
\end{align*}
\]

Similar to assigning dynamic semantics to dynamic terms, we can readily assign dynamic semantics to the erasure, which are just untyped \(\lambda\)-expressions. We write \(e_1 \leftrightarrow e_2\) to mean that \(e_1\) reduces \(e_2\) in one step, and use \(\leftrightarrow^*\) for the reflexive and transitive closure of \(\leftrightarrow\).

**Theorem 2.3.2** Assume that \(D :: \emptyset; \emptyset; \emptyset \vdash d : T\) or \(D :: \emptyset; \emptyset; \emptyset; SA \vdash d : CT\).

1. If \(d \leftrightarrow^* v\), then \(|d| \leftrightarrow^* |v|\).

2. If \(|d| \leftrightarrow^* w\), then there is a term \(d'\) such that \(d \leftrightarrow^* d'\) and \(|d'| = w\).
Figure 2.4: The typing rules for dynamics
PROOF. Straightforward induction on $D$.

Theorem 2.3.2 means that evaluation commutes with erasure in the sense that these operations can be performed in either order - we reach the same term by evaluating and then erasing as we do by erasing and then evaluating.

## 2.4 Address Polymorphism

In general, the particular address $id$ that contains an object is irrelevant to the program being executed. It is too restricted to require that an operation can only be applied to one specific address. This restriction makes most programs useless. For instance, if the assignment function could only apply to a specific memory block, then we have to implement one assignment function for every address. The information which does matter is the connection between the address $id$ and its content. Our calculus abstracts away from the concrete address by introducing address polymorphism. For instance, the $id$ in rules (ty-read) and (ty-write) could be any address. In another way, the dereference constructor $!$ and assignment constructor $←$ are assigned the following types:

$$
! : \forall id : addr. [id : T] \supset \text{ptr}(id) \rightarrow ([id : T] \land T) \\
← : \forall id : addr. [id : T] \supset (\text{ptr}(id), T') \rightarrow ([id : T'] \land 1)
$$

We notice that the type for dereference and assignment has a type guard $[id : T]$, which means that before accessing a memory location, we need to prove that it exists in the state (that’s to say, it is still alive). Therefore, our type system can prevent accessing dangling pointers.

## 2.5 State Polymorphism

Most routines only operate on a portion of the state, and thus its execution leave other parts of the state intact. In order to use those routines in multiple contexts, we abstract
irrelevant portions of the state by introducing state polymorphism, which is carried out by the rule (ty-ext).

\[
\begin{align*}
\Sigma; \bar{P}; \Delta \vdash d : SA \supset (T \to \exists \Sigma', \bar{P}'.SA' \land T') \\
\Sigma; \bar{P}; \Delta \vdash d : SA[id : T_0] \supset (T \to \exists \Sigma', \bar{P}'.SA'[id : T_0] \land T')
\end{align*}
\]

(\text{ty-ext})

The intuition behind (ty-ext) is that if a routine \( f \) starts in a state \( ST \) satisfying \( SA \) and ends in some state \( ST' \), then adding some fresh locations to the state \( ST \) will not affect the execution of \( f \). In addition, \( f \) will pass these irrelevant portions from beginning to end and leave them untouched. By state polymorphism, one can extend a local state, involving only the locations that are actually used by \( f \) (which \( \text{O'Hearn} \) calls the footprint of \( f \)) by adding arbitrary predicates or locations that are not modified or mutated by \( f \). In some sense, state polymorphism is similar to row polymorphism. In Separation Logic, there is also a counterpart called Frame Rule.

\[
\begin{align*}
\{ P \} \ c \ \{ q \} \\
\{ P * r \} \ c \ \{ q * r \}
\end{align*}
\]

The essence of state polymorphism can be addressed by the following quote.

To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the programs actually accesses. The value of any other cell will automatically remain unchanged.

- P. W. \( \text{O'Hearn} \), etc..  \textit{Local Reasoning about Programs that Alter Data Structures} [27]

### 2.6 Aliasing Control

One well-known principle for proving type safety is based on type-invariance of memory locations which states that, once allocated, a memory object should preserve its type during its life-cycle. This property forbids the type of a memory object be changed during evaluation. Once this property is maintained, it is straightforward to prove some significant
properties of a type system such as subject reduction which is crucial to establishing type soundness. Examples using this property to prove type soundness can be found in [14] and [34]. This type-invariance principle prevents most type-safe languages from allowing some operations on memory which could potentially violate this principle such as user-level initialization or explicit memory de-allocation. This property is too restricted and many modern language such as C++ and Java violate this principle.

Since Linear Logic [13] was introduced, it has been found to be promising for an important problem domain - reasoning with states. Type systems [32, 31] based on linear logic employ a different principle to achieve type-preservation property. The gist of a linear type system is that every memory location exactly has one reference, that’s to say, aliasing is not permitted in a linear setting. With this restriction, the type of a memory object can be changed during evaluation. In addition, explicit memory initialization and de-allocation can be performed. But the cost of aliasing restriction is too expensive: many common and efficient data structures that use sharing or involve cycles can not be implemented. For instance, it is hard or even impossible to implement in-place reversal of a linked list.

Our calculus can readily handle aliasing: aliasing restriction is lifted and thus a memory block can have many aliases and its type can be changed over time; on the other hand, we still retain powerful reasoning about state changes and keep the type system consistent and sound. We say that $id_1$ and $id_2$ are aliases if both point to the same memory location. That’s to say, $id_1$ and $id_2$ are aliases exactly when there is a state such that both $[id_1 : \text{ptr}(id)]$ and $[id_2 : \text{ptr}(id)]$ holds, where $id$ is a memory location. Simply put, we keep a unique identifier for each memory objects and all aliases of this memory object can only access it via that identifier. In some sense, $id$ acts as a unique handle for a memory object. For instance, suppose $id$ stands for a memory block, all aliases of this memory block should be assigned the type $\text{ptr}(id)$. This principle is enforced by the rule $\text{(ty-write)}$, which is the
only place where an alias can be created.

\[
\Gamma; \vec{P}; \Delta \vdash d : T \\
\vdash d : \text{SA}[^{\text{id}}; T'] \land 1
\]  

(\text{ty-write})

In (\text{ty-write}), if \(d\) is an identifier \(id'\), then the type \(T\) should be \(\text{ptr}(id')\). After the assignment, the type of \(id\) is changed from some \(T'\) to \(\text{ptr}(id')\). The only way it can access the content is through the pointer \(id'\). Simply put, to ensure type consistency, we assign the type of an address to all aliases, but not the content type of that address. In this way, the type of a memory object can be changed over time and all the aliases are aware of those changes.
Chapter 3

Soundness

In this chapter, we present and prove the soundness of our system.

Given a judgment $J$, we write $D :: J$ to indicate that $D$ is a derivation of $J$, that is, $D$ is a derivation whose conclusion is $J$.

Lemma 3.0.1 (SUBSTITUTION)

1. Assume that $D :: \Sigma, a : \sigma; \bar{P}; \Delta \vdash d : T$ and $D_1 :: \Sigma \vdash s_0 : \sigma$, then $\Sigma; \bar{P}[a \mapsto s_0]; \Delta[a \mapsto s_0] \vdash d : T[a \mapsto s_0]$ is derivable;
   
   Assume that $D :: \Sigma, a : \sigma; \bar{P}; \Delta; SA \vdash d : CT$ and $D_1 :: \Sigma \vdash s_0 : \sigma$, then $\Sigma; \bar{P}[a \mapsto s_0]; \Delta[a \mapsto s_0]; SA[a \mapsto s_0] \vdash d : CT[a \mapsto s_0]$ is derivable.

2. Assume that $D :: \Sigma; \bar{P}, \Delta \vdash d : T$ and $D_1 :: \Sigma; \bar{P} \models P$, then $\Sigma; \bar{P}; \Delta \vdash d : T$ is derivable;
   
   Assume that $D :: \Sigma; \bar{P}, \Delta; SA \vdash d : CT$ and $D_1 :: \Sigma; \bar{P} \models P$, then $\Sigma; \bar{P}; \Delta; SA \vdash d : CT$ is derivable.

3. Assume that $D :: \Sigma; \bar{P}; \Delta, x : T \vdash d : T'$ and $D_1 :: \Sigma; \bar{P}; \Delta \vdash v : T$, then $\Sigma; \bar{P}; \Delta \vdash d[x \mapsto v] : T'$ is derivable;
   
   Assume that $D :: \Sigma; \bar{P}; \Delta, x : T; SA \vdash d : CT$ and $D_1 :: \Sigma; \bar{P}; \Delta \vdash v : T$, then $\Sigma; \bar{P}; \Delta; SA \vdash d[x \mapsto v] : CT$ is derivable.
PROOF. We can prove (1),(2) and (3) by structural induction on $\mathcal{D}$. Following gives the proof for (1).

- **(ty-sub)**

  \[
  \begin{array}{c}
  \mathcal{D}': \Sigma, a : \sigma; \bar{P}; \Delta \vdash d : T \\
  \mathcal{D}'' : \Sigma, a : \sigma; \bar{P} \vdash T \leq_{tp} T'
  \end{array}
  \]

  By induction hypothesis on $\mathcal{D}'$, we have $\mathcal{D}''' : \Sigma; \bar{P}[a \mapsto s_0]; \Delta[a \mapsto s_0] \vdash d : T[a \mapsto s_0]$. In addition, applying rule (sub-tp-subst) to $\mathcal{D}''$ and $\mathcal{D}_1$ yields a derivation $\mathcal{D}''' : \Sigma; \bar{P}[a \mapsto s_0] = T[a \mapsto s_0] \leq_{tp} T'[a \mapsto s_0]$. Applying rule (ty-sub) to $\mathcal{D}'''$ and $\mathcal{D}'''$ gives us the derivation $\Sigma; \bar{P}[a \mapsto s_0]; \Delta[a \mapsto s_0] \vdash d : T'[a \mapsto s_0]$.

- **(ty-sub-c)**

  \[
  \begin{array}{c}
  \mathcal{D}': \Sigma, a : \sigma; \bar{P}; \Delta; \textit{SA} \vdash d : CT \\
  \mathcal{D}'' : \Sigma, a : \sigma; \bar{P} \vdash CT \leq_{ct} CT'
  \end{array}
  \]

  By induction hypothesis on $\mathcal{D}'$, we have $\mathcal{D}''' : \Sigma; \bar{P}[a \mapsto s_0]; \Delta[a \mapsto s_0]; \textit{SA}[a \mapsto s_0] \vdash d : CT[a \mapsto s_0]$. In addition, applying rule (sub-ct-subst) to $\mathcal{D}''$ and $\mathcal{D}_1$ yields a derivation $\mathcal{D}''' : \Sigma; \bar{P}[a \mapsto s_0] = CT[a \mapsto s_0] \leq_{tp} CT'[a \mapsto s_0]$. Applying rule (ty-sub) to $\mathcal{D}'''$ and $\mathcal{D}'''$ gives us the derivation $\Sigma; \bar{P}[a \mapsto s_0]; \Delta[a \mapsto s_0]; \textit{SA}[a \mapsto s_0] \vdash d : CT'[a \mapsto s_0]$.

- **(ty-state)**

  \[
  \begin{array}{c}
  \mathcal{D}': \Sigma, a : \sigma; \bar{P}; \Delta \vdash d : T \\
  \mathcal{D}'' : \Sigma, a : \sigma; \bar{P} \vdash \text{SA} \wedge T
  \end{array}
  \]

  By induction hypothesis on $\mathcal{D}'$, we have $\mathcal{D}'' : \Sigma; \bar{P}[a \mapsto s_0]; \Delta[a \mapsto s_0] \vdash d : T[a \mapsto s_0]$. Applying rule (ty-state) to $\mathcal{D}''$ yields the derivation $\Sigma; \bar{P}[a \mapsto s_0]; \Delta[a \mapsto s_0]; \textit{SA}[a \mapsto s_0] \vdash d : \text{SA}[a \mapsto s_0] \wedge T[a \mapsto s_0]$. 
• (ty-ext)

\[ D_0 :: \Sigma, a : \sigma \vdash \text{id : addr} \]
\[ D' :: \Sigma, a : \sigma \vdash T_0 : \text{type} \]
\[ D'' :: \Sigma, a : \sigma; \bar{P}; \Delta \vdash d : \text{SA} \supset (T \rightarrow \exists \Sigma', \bar{P}', \text{SA}' \land T') \]
\[ \Sigma, a : \sigma; \bar{P}; \Delta \vdash d : \text{SA}[\text{id : T}_0] \supset (T \rightarrow \exists \Sigma', \bar{P}', \text{SA}'[\text{id : T}_0] \land T') \]

Applying proposition 2.1.2.(3) to \( D' \) and \( D_1 \) gives a derivation \( D''' :: \Sigma \vdash T_0[a \mapsto s_0] : \text{type} \). In addition, by induction hypothesis on \( D'' \), we have \( D' :: \Sigma \vdash \text{id[a \mapsto s_0]} : \text{addr} \). Furthermore, it is straightforward that \( D_0 :: \Sigma \vdash \text{id[a \mapsto s_0]} : \text{addr} \) holds from \( D_0 \). Applying (ty-ext) to \( D', D'' \) and \( D''' \) yields the derivation \( \Sigma; \bar{P}[a \mapsto s_0]; \Delta \vdash \text{lam x.d} : \text{CT} \).

• (ty-lam)

\[ D' :: \Sigma, a : \sigma; \bar{P}; \Delta, x : T; \text{SA} \vdash d : \text{CT} \]
\[ \Sigma, a : \sigma; \bar{P}; \Delta \vdash \text{lam x.d} : \text{SA} \supset T \rightarrow \text{CT} \]

By induction hypothesis on \( D' \), we have \( \Sigma; \bar{P}[a \mapsto s_0]; \Delta[a \mapsto s_0], x : T[a \mapsto s_0] ; \text{SA[a \mapsto s_0]} \vdash d : \text{CT} \). Applying rule (ty-lam) again yields the derivation \( \Sigma; \bar{P}[a \mapsto s_0]; \Delta[a \mapsto s_0] \vdash d : \text{SA[a \mapsto s_0]} \supset T[a \mapsto s_0] \rightarrow \text{CT} \).

• (ty-fix). Similar to the previous case.

• (ty-app). Then \( D \) is of the following form.

\[ D' :: \Sigma, a : \sigma; \bar{P} \vdash \text{SA} \leq \text{sa} \ \text{SA}' \]
\[ D'' :: \Sigma, a : \sigma; \bar{P}; \Delta \vdash d_1 : \text{SA}' \supset T \rightarrow \text{CT} \]
\[ D''' :: \Sigma, a : \sigma; \bar{P}; \Delta \vdash d_2 : T \]
\[ \Sigma, a : \sigma; \bar{P}; \Delta ; \text{SA} \vdash \text{app}(d_1, d_2) : \text{CT} \]

By induction hypothesis on \( D'' \) and \( D''' \), we have \( D_2 :: \Sigma; \bar{P}[a \mapsto s_0]; \Delta[a \mapsto s_0] \vdash d_1 : (\text{SA}' \supset T \rightarrow \text{CT})[a \mapsto s_0] \) and \( D_3 :: \Sigma; \bar{P}[a \mapsto s_0]; \Delta[a \mapsto s_0] \vdash d_2 : T[a \mapsto s_0] \rightarrow \text{CT} \).
s_0], respectively. In addition, applying rule \((\text{sub-sa-subst})\) to \(\mathcal{D}'\) and \(\mathcal{D}_1\) yields a derivation \(\mathcal{D}_4 :: \Sigma; \bar{P}[a \mapsto s_0] \vdash SA[a \mapsto s_0] \leq_{sa} SA'[a \mapsto s_0].\) Then applying rule \((\text{ty-app})\) to \(\mathcal{D}_2, \mathcal{D}_3\) and \(\mathcal{D}_4\) gives the derivation \(\mathcal{D}_5 :: \Sigma; vecP[a \mapsto s_0]; \Delta[a \mapsto s_0]; SA[a \mapsto s_0] \vdash app(d_1, d_2) : CT[a \mapsto s_0],\) which is what we need to prove.

All other cases for (1) can be proved similarly.

In the following, we present some proof cases for (2).

- **(ty-lam)**. Then \(\mathcal{D}\) is of the following form,

\[
\begin{array}{c}
\mathcal{D}' :: \Sigma; \bar{P}, P; \Delta, x : T; SA \vdash d : CT \\
\Sigma; \bar{P}, P; \Delta \vdash \text{lam} x.d : SA \supset T \rightarrow CT
\end{array}
\]

\((\text{ty-lam})\)

Then by induction hypothesis on \(\mathcal{D}'\), we have a derivation \(\mathcal{D}' :: \Sigma; \bar{P}; \Delta, x : T; SA \vdash d : CT.\) Applying rule \((\text{ty-lam})\) to \(\mathcal{D}'\) yields a derivation \(\Sigma; \bar{P}; \Delta \vdash \text{lam} x.d : SA \supset T \rightarrow CT.\)

- **(ty-\forall\text{-intro})**. Then \(\mathcal{D}\) is of the following form,

\[
\begin{array}{c}
\mathcal{D}' :: \Sigma, a : \sigma; \bar{P}, P; \Delta \vdash v : T \\
\Sigma; \bar{P}, P; \Delta \vdash \forall^+(v) : \forall a : \sigma.T
\end{array}
\]

\((\text{ty-\forall\text{-intro}})\)

Then by induction hypothesis on \(\mathcal{D}'\), we have a derivation \(\mathcal{D}'' :: \Sigma, a : \sigma; \bar{P}; \Delta \vdash v : T.\) Applying rule \((\text{ty-\forall\text{-intro}})\) to \(\mathcal{D}''\) yields a derivation \(\Sigma; \bar{P}; \Delta \vdash \forall^+(v) : \forall a : \sigma.T.\)

- **(ty-\forall\text{-intro})**. Then \(\mathcal{D}\) is of the following form,

\[
\begin{array}{c}
\mathcal{D}' :: \Sigma; \bar{P}, P; \Delta \vdash d : \forall a : \sigma.T \\
\mathcal{D}'' :: \Sigma \vdash s : \sigma \\
\Sigma; \bar{P}, P; \Delta \vdash \forall^-(d) : T[a \mapsto s]
\end{array}
\]

\((\text{ty-\forall\text{-elim}})\)

Then by induction hypothesis on \(\mathcal{D}'\), we have a derivation \(\mathcal{D}''' :: \Sigma; \bar{P}; \Delta \vdash d : \forall a : \sigma.T.\) Applying rule \((\text{ty-\forall\text{-elim}})\) to \(\mathcal{D}''\) and \(\mathcal{D}'''\) yields a derivation \(\Sigma; \bar{P}; \Delta \vdash \forall^-(d) : T[a \mapsto s].\)

All other cases for (2) can be proved similarly.

Some proof cases for (3) are given as follows.
• (ty-lam). Then \( D \) is of the following form,

\[
\frac{\mathcal{D} \:: \; \Sigma; \bar{P}; \Delta, x : T, y : T'; SA \vdash d : CT}{\Sigma; \bar{P}; \Delta, x : T \vdash \text{lam} \ y. d : SA \supset T' \rightarrow CT} \quad (\text{ty-lam})
\]

Then by induction hypothesis on \( \mathcal{D}' \), we have a derivation \( \mathcal{D}' \:: \; \Sigma; \bar{P}; \Delta, y : T' ; SA \vdash d[x \mapsto v] : CT \). Applying rule (ty-lam) to \( \mathcal{D}' \) yields a derivation \( \Sigma; \bar{P}; \Delta \vdash \text{lam} \ y. (d[x \mapsto v]) : SA \supset T \rightarrow CT \). Then we are done since \( (\text{lam}yd)[x \mapsto v] = \text{lam} \ y. (d[x \mapsto v]) \) (because \( x \) and \( y \) are distinct variables).

• (ty-\( \wedge \)-intro). Then \( D \) is of the following form,

\[
\frac{\mathcal{D}' \:: \; \Sigma; \bar{P} \models P \quad \mathcal{D}'' \:: \; \Sigma; \bar{P}; \Delta, x : T \vdash v' : T'}{\Sigma; \bar{P}; \Delta, x : T \vdash \land^+(v') : P \land T'} \quad (\text{ty-\( \wedge \)-intro})
\]

Then by induction hypothesis on \( \mathcal{D}'' \), we have a derivation \( \mathcal{D}'' \:: \; \Sigma; \bar{P}; \Delta \vdash v'[x \mapsto v] : T' \). Applying rule (ty-\( \wedge \)-intro) to \( \mathcal{D}' \) and \( \mathcal{D}'' \) yields a derivation \( \Sigma; \bar{P}; \Delta \vdash \land^+(v'[x \mapsto v]) : P \land T' \).

• (ty-\( \wedge \)-elim). Then \( D \) is of the following form,

\[
\frac{\mathcal{D}' \:: \; \Sigma; \bar{P}; \Delta, x : T \vdash d_1 : P \land T_1}{\Sigma; \bar{P}; \Delta, x : T \vdash \text{let} \ \land (y) = d_1 \ \text{in} \ d_2 : T_2} \quad (\text{ty-\( \wedge \)-elim})
\]

Then by induction hypothesis on \( \mathcal{D}' \), we have a derivation \( \mathcal{D}''' \:: \; \Sigma; \bar{P}; \Delta \vdash d_1[x \mapsto v] : P \land T_1 \). Similarly, by induction hypothesis on \( \mathcal{D}'' \), we can get a derivation \( \mathcal{D}'''' \:: \; \Sigma; \bar{P}, P; \Delta, y : T_1 \vdash d_2[x \mapsto v] : T_2 \). Applying rule (ty-\( \wedge \)-elim) to \( \mathcal{D}''' \) and \( \mathcal{D}'''' \) yields a derivation \( \Sigma; \bar{P}; \Delta \vdash \text{let} \ \land (y) = d_1[x \mapsto v] \ \text{in} \ d_2[x \mapsto v] : T_2 \).

All other cases for (3) can be proved similarly.

This completes the proof.

**Lemma 3.0.2 (Canonical Forms).** Assume \( D :: \emptyset; \emptyset; \emptyset \vdash v : T \). Then we have the following:
1. If $T = SA \supset T' \rightarrow CT$, then $v$ is of the form $\text{lam} \ x.\ d$.

2. If $T = P \supset T'$, then $v$ is of the form $\supset^+(v_0)$.

3. If $T = \forall a : \sigma.T'$, then $v$ is of the form $\forall^+(v_0)$.

4. If $T = P \land T'$, then $v$ is of the form $\land^+(v_0)$.

5. If $T = \exists a : \sigma.T'$, then $v$ is of the form $\exists(v_0)$.

PROOF. The lemma follows from structural induction on $\mathcal{D}$ immediately.

Given a derivation $\mathcal{D}$, we use $h(\mathcal{D})$ for the height of $\mathcal{D}$, which can be defined in a standard manner.

Lemma 3.0.3 (Inversion). Assume that $\mathcal{D} :: \Sigma; \vec{P}; \Delta \vdash d : T$.

1. If $d = \text{lam} \ x.\ d_1$ and $T = SA \supset T' \rightarrow CT$, then there exists a derivation $\mathcal{D}' :: \Sigma; \vec{P}; \Delta \vdash d : T$ such that $h(\mathcal{D}') \leq h(\mathcal{D})$ and the last rule applied in $\mathcal{D}'$ is not (ty-sub) or (ty-ext).

2. If $d = \forall^+(d_1)$ and $T = \forall a : \sigma.T'$, then there exists a derivation $\mathcal{D}' :: \Sigma; \vec{P}; \Delta \vdash d : T$ such that $h(\mathcal{D}') \leq h(\mathcal{D})$ and the last rule applied in $\mathcal{D}'$ is not (ty-sub).

3. If $d = \supset^+(d_1)$ and $T = P \supset T'$, then there exists a derivation $\mathcal{D}' :: \Sigma; \vec{P}; \Delta \vdash d : T$ such that $h(\mathcal{D}') \leq h(\mathcal{D})$ and the last rule applied in $\mathcal{D}'$ is not (ty-sub).

4. If $d = \exists(d_1)$ and $T = \exists a : \sigma.T'$, then there exists a derivation $\mathcal{D}' :: \Sigma; \vec{P}; \Delta \vdash d : T$ such that $h(\mathcal{D}') \leq h(\mathcal{D})$ and the last rule applied in $\mathcal{D}'$ is not (ty-sub).

5. If $d = \land^+(d_1)$ and $T = P \land T'$, then there exists a derivation $\mathcal{D}' :: \Sigma; \vec{P}; \Delta \vdash d : T$ such that $h(\mathcal{D}') \leq h(\mathcal{D})$ and the last rule applied in $\mathcal{D}'$ is not (ty-sub).
PROOF. Proof proceeds by induction on \( h(D) \). We only prove case (1) and (2) in the following.

1. \( d = \text{lam } x, d_1 \) and \( T = SA \supset T' \rightarrow CT \). Let \( D' \) be \( D \) if \( D \) does not end with an application of the rule \((\text{ty-sub})\) or \((\text{ty-ext})\). Hence, in the rest of the proof, we prove two cases.

   - The last applied rule in \( D \) is \((\text{ty-sub})\), that is , \( D \) is of the following form:

     \[
     \begin{align*}
     &D_1 :: \Sigma; \bar{P}; \Delta \vdash \text{lam } x, d_1 : SA_1 \supset T_1 \rightarrow CT_1 \\
     &D_2 :: \Sigma; \bar{P} \vdash (SA_1 \supset T_1 \rightarrow CT_1) \leq_{tp} (SA_2 \supset T_2 \rightarrow CT_2) \\
     &\Sigma; \bar{P}; \Delta; SA \vdash \text{lam } x, d_1 : SA_2 \supset T_2 \rightarrow CT_2
     \end{align*}
     \]

     By induction hypothesis, we can find a derivation \( D'_1 \) which does not end with \((\text{ty-sub})\) or \((\text{ty-ext})\). Therefore, we only need to consider the case where the last rule applied in \( D'_1 \) is \((\text{ty-lam})\). In addition, using the rules for sub-tying, we can rewrite \( D \) as

     \[
     \begin{align*}
     &\Sigma; \bar{P}; \Delta, x : T_1; \Sigma; \bar{P} \vdash T_1 \leq_{tp} T_1 \\
     &\Sigma; \bar{P} \vdash CT_1 \leq_{ct} CT_2 \\
     &\Sigma; \bar{P}; \Delta; SA \vdash \text{lam } x, d_1 : SA_2 \supset T_2 \rightarrow CT_2
     \end{align*}
     \]

     It is easy to show that rule \((\text{reg-sub})\) is admissible.

     \[
     \begin{align*}
     &\Sigma; \bar{P}; \Delta, x : T_1; \Sigma; \bar{P} \vdash T_2 \leq_{tp} T_1 \\
     &\Sigma; \bar{P} \vdash CT_1 \leq_{ct} CT_2 \\
     &\Sigma; \bar{P}; \Delta; SA \vdash \text{lam } x, d_1 : CT_2
     \end{align*}
     \]

     Let \( D' \) be the following derivation,

     \[
     \begin{align*}
     &\Sigma; \bar{P}; \Delta, x : T_1; \Sigma; \bar{P} \vdash T_2 \leq_{tp} T_1 \\
     &\Sigma; \bar{P} \vdash CT_1 \leq_{ct} CT_2 \\
     &\Sigma; \bar{P}; \Delta; SA \vdash \text{lam } x, d_1 : CT_2
     \end{align*}
     \]

     and we are done since \( h(D') = 1 + 1 + h(D_3) = 1 + h(D'_1) \leq 1 + h(gd_1) = h(D) \).

   - The last applied rule in \( D \) is \((\text{ty-ext})\), that is , \( D \) is of the following form:

     \[
     \begin{align*}
     &D_0 :: \Sigma \vdash id : \text{addr} \\
     &D_1 :: \Sigma \vdash T_0 : \text{type} \\
     &D_2 :: \Sigma; \bar{P}; \Delta \vdash \text{lam } x, d_1 : SA \supset (T \rightarrow \exists \Sigma', \bar{P}', SA' \land T') \\
     &\Sigma; \bar{P}; \Delta \vdash \text{lam } x, d_1 : SA[id \mapsto T_0] \supset (T \rightarrow \exists \Sigma', \bar{P}', SA'[id \mapsto T_0] \land T')
     \end{align*}
     \]
By induction hypothesis on \( D_2 \), we can find a derivation \( D'_2 \) which does not end with (\texttt{ty-sub}) or (\texttt{ty-ext}). Therefore, we only need to consider the case where the last applied rule in \( D'_2 \) is (\texttt{ty-lam}), that is, \( D_2 \) is of the following form,

\[
\frac{D_3 :: \Sigma; \bar{P}; \Delta, x : T; \SA \vdash d_1 : \exists \Sigma', \bar{P}', \ SA' \land T'}{
\Sigma; \bar{P}; \Delta \vdash \text{lam } x. d_1 : \SA \supset (T \rightarrow \exists \Sigma', \bar{P}', \ SA' \land T')}
\]

It is easy to show that rule (\texttt{state-ext}) is admissible.

\[
\frac{\Sigma \vdash id : \text{addr} \quad \Sigma \vdash T_0 : \text{type} \quad \Sigma; \bar{P}; \Delta, x : T; \SA \vdash d_1 : \exists \Sigma', \bar{P}', \ SA' \land T'}{
\Sigma; \bar{P}; \Delta, x : T; \SA[id : T_0] \vdash d_1 : \exists \Sigma', \bar{P}', \ SA'[id : T_0] \land T'}
\]

Let \( D' \) be the following derivation,

\[
\frac{\Sigma \vdash id : \text{addr} \quad \Sigma \vdash T_0 : \text{type} \quad D_3 :: \Sigma; \bar{P}; \Delta, x : T; \SA \vdash d_1 : \exists \Sigma', \bar{P}', \ SA' \land T'}{
\Sigma; \bar{P}; \Delta, x : T; \SA[id : T_0] \vdash d_1 : \exists \Sigma', \bar{P}', \ SA'[id : T_0] \land T'}
\]

and we are done since \( h(D') = 1 + 1 + h(D_3) = 1 + h(D'_2) \leq 1 + h(D_2) = h(D) \).

2. \( d = \forall^+(d_1) \) and \( T = \forall a : \sigma.T' \). In this case, we know that \( d_1 \) is some value \( v \). Let \( D' \) be \( D \) if \( D \) does not end with an application of the rule (\texttt{ty-sub}). Hence, in the rest of the proof, we assume that the last rule applied in \( D \) is (\texttt{ty-sub}), that is, \( D \) is of the following form,

\[
\frac{D_1 :: \Sigma; \bar{P}; \Delta \vdash v : \forall a : \sigma.T_1 \quad D_2 :: \Sigma; \bar{P} \models \forall a : \sigma.T_1 \leq_{tp} \forall a : \sigma.T_2}{
\Sigma; \bar{P}; \Delta \vdash \forall^+(v) : \forall a : \sigma.T_2}
\]

By induction hypothesis, we can find a derivation \( D'_1 \) which does not end with (\texttt{ty-sub}). Therefore, we only need to consider the case where the last rule applied in \( D_1 \) is (\texttt{ty-\forall-intro}). In addition, by using the rules for sub-typing, we can rewrite \( D \) as

\[
\frac{D_3 :: \Sigma, a : \sigma; \bar{P}; \Delta \vdash v : T_1 \quad D_4 :: \Sigma, a : \sigma; \bar{P} \models T_1 \leq_{tp} T_2}{
\Sigma; \bar{P}; \Delta \vdash v : \forall a : \sigma.T_1 \quad \Sigma; \bar{P} \models \forall a : \sigma.T_1 \leq_{tp} \forall a : \sigma.T_2}
\]

Then \( h(D) = 1 + h(D_1) \), and \( h(D'_1) = 1 + h(D_3) \leq h(D_1) \). Let \( D' \) be the following

---

derivation,

\[
\begin{align*}
\frac{D_3 :: \Sigma, a : \sigma; \vec{P}; \Delta \vdash v : T_1 \quad D_4 :: \Sigma, a : \sigma; \vec{P} \models T_1 \leq_{tp} T_2}{\Sigma, a : \sigma; \vec{P}; \Delta \vdash v : T_2} \\
\frac{\Sigma, a : \sigma; \vec{P}; \Delta \vdash v : T_2}{\Sigma; \vec{P}; \Delta \vdash \forall^+(v) : \forall a : \sigma.T_2}
\end{align*}
\]

and we are done since \( h(D') = 1 + 1 + h(D_3) = 1 + h(D_1) \leq 1 + h(D_1) = h(D) \).

All other cases can be proved similarly.

This completes the proof.

**Theorem 3.0.4 (Subject Reduction)**

1. Assume that \( \emptyset; \emptyset; \emptyset \vdash d : T \) is derivable and \( d \rightarrow d' \) holds. Then \( \emptyset; \emptyset; \emptyset \vdash d' : T \) is also derivable.

2. Assume that \( \emptyset; \emptyset; \emptyset; SA \vdash d : CT \) is derivable and \( (ST, d) \rightarrow (ST', d') \) for some state \( ST \) such that \( ST : SA \) holds. Then \( \emptyset; \emptyset; \emptyset; SA' \vdash d' : CT \) is also derivable for some \( SA' \) such that \( ST' : SA' \) holds.

**PROOF.** We prove the above two theorems simultaneously. Assume that \( d := E[d_0] \) and \( d' = E[d'_0] \), where \( d_0 \) is a redex and \( (ST_1, d_0) \rightarrow_{st} (ST_2, d_0) \). The proof proceeds by structural induction on \( h(D) \). At first we deal with the case where the last applied rule is \( (\text{ty-sub}) \).

Assume that the last applied rule in \( D \) is \( (\text{ty-sub}) \), that is, \( D \) is of the following form:

\[
\begin{align*}
D_1 :: \emptyset; \emptyset; \emptyset \vdash d : T & \quad D_2 :: \emptyset; \emptyset \models T \leq_{tp} T' \\
\emptyset; \emptyset; \emptyset \vdash d : T' 
\end{align*}
\]

By induction hypothesis on \( D_1 \), we have \( D_3 :: \emptyset; \emptyset; \emptyset \vdash d' : T \). Applying rule \( (\text{ty-sub}) \) to \( D_3 \) and \( D_2 \) yields the derivation \( \emptyset; \emptyset; \emptyset \vdash d' : T' \).

From now on, we assume that the last applied in \( D \) is not \( (\text{ty-sub}) \) and proceed by structural induction on the evaluation context \( E \). We present the most interesting case where \( E = [] \). In this case, \( d = d_0 \) and \( d' = d'_0 \).

- \( d = \text{app}(\text{lam } x.d_1, v) \) and \( d' = d_1[x \mapsto v] \). By Lemma 3.0.3, there exist three sub-cases:
– The last applied rule in \( D \) is (\text{ty-app}). Then the derivation \( D \) is of the following form,

\[
\begin{align*}
D_1 &: \emptyset; \emptyset; \emptyset; x: T'; SA \vdash d_1 : CT \\
D_2 &: \emptyset; \emptyset; \emptyset; \emptyset; \vdash \text{lam } x. d_1 : SA' \supset T' \rightarrow CT \\
D_3 &: \emptyset; \emptyset; \emptyset; \vdash \text{app}(\text{lam } x. d_1, v) : CT
\end{align*}
\]

By Lemma 3.0.1, we know that \( D_4 \vdash \emptyset; \emptyset; \emptyset; SA' \vdash d_1[x \mapsto v] : CT \) is derivable.

Applying (\text{ty-state-sub}) to \( D_1 \) and \( D_4 \) yields the derivation \( \emptyset; \emptyset; \emptyset; \vdash d_1[x \mapsto v] : CT \).

– The last applied rule in \( D \) is (\text{ty-sub_c}). Then \( D \) is of the following form,

\[
D_1 :: \Sigma; \vec{P}; \Delta; SA \vdash d : CT \\
D_2 :: \Sigma; \vec{P} \vdash CT \rightarrow \Sigma' \\
SA \vdash d' : \Sigma' \rightarrow T \\
\Sigma; \vec{P}; \Delta; SA \vdash d' : CT'
\]

By induction hypothesis on \( D_1 \), we have a derivation \( D_1' :: \Sigma; \vec{P}; \Delta; SA \vdash d_1[x \mapsto v] : CT \). Then applying rule (\text{ty-sub_c}) to \( D_1' \) and \( D_2 \) yields a derivation \( \Sigma; \vec{P}; \Delta; SA \vdash d_1[x \mapsto v] : CT' \).

– The last applied rule in \( D \) is (\text{ty-\exists_c-intro}). Then \( D \) is of the following form,

\[
D_1 :: \Sigma \vdash \Theta : \Sigma' \\
D_2 :: \Sigma; \vec{P} \vdash \vec{P}'[\Theta] \\
D_3 :: \Sigma; \vec{P}; \Delta; SA \vdash d : SA'\Theta \land T[\Theta] \\
\Sigma; \vec{P}; \Delta; SA \vdash d' : \exists \Sigma', \vec{P}'.SA' \land T
\]

By induction hypothesis on \( D_3 \), we have a derivation \( D_3' :: \Sigma; \vec{P}; \Delta; SA \vdash d' : \Sigma' \land T[\Theta] \). Then applying rule (\text{ty-\exists_c-intro}) to \( D_1, D_2 \) and \( D_3' \) yields a derivation \( \Sigma; \vec{P}; \Delta; SA \vdash d' : \exists \Sigma', \vec{P}'.SA' \land T \).

\begin{itemize}
  \item \( d = \text{fix } f. d_1 \) and \( d' = d_1[f \mapsto \text{fix } f. d_1] \). We may assume that the derivation \( D \) is of the following form,

\[
\emptyset; \emptyset; \emptyset; f : T_1 \vdash d_1 : T_1 \\
\emptyset; \emptyset; \emptyset; \vdash \text{fix } f. d_1 : T_1
\]

By Lemma 3.0.1, we can readily verify that \( \emptyset; \emptyset; \emptyset; \emptyset; \vdash d_1[f \mapsto \text{fix } f. d_1] : T_1 \) is derivable.

\item \( d = \exists (\exists^+ (v)) \) and \( d' = v \). Hence, by Lemma 3.0.3, we may assume that the
derivation $\mathcal{D}$ is of the following form:

\[
\begin{array}{c}
\emptyset; \emptyset, P; \emptyset \vdash v : T \\
\emptyset; \emptyset, \emptyset \vdash \top^+(v) : P \supset T \\
\emptyset; \emptyset \vdash \top^-(\top^+(v)) : T
\end{array}
\]

By Lemma 3.0.1, we know that $\emptyset; \emptyset; \emptyset \vdash v : T$.

- $d = \forall^-(\forall^+(v))$ and $d' = v$. Similar to the previous case.

- $d = \operatorname{let} \exists(x) = \exists(v) \in d_1$ and $d' = d_1[x \mapsto v]$. Hence, by Lemma 3.0.3, we may assume that the derivation $\mathcal{D}$ is of the following form:

\[
\begin{array}{c}
\mathcal{D}_1 :: \emptyset \vdash s : \sigma \\
\mathcal{D}_2 :: \emptyset; \emptyset; \emptyset \vdash v : T_1[a \mapsto s] \\
\mathcal{D}_3 :: \emptyset, a : \sigma; \emptyset, x : T_1 \vdash d_1 : T_2
\end{array}
\]

\[
\emptyset; \emptyset; \emptyset \vdash \exists(x) = \exists(v) \in d_1 : T_2
\]

Given derivations $\mathcal{D}_1$ and $\mathcal{D}_3$, by Lemma 3.0.1, we have the derivation $\mathcal{D}_4 :: \emptyset; \emptyset; \emptyset, x : T_1[a \mapsto s] \vdash d_1 : T_2[a \mapsto s]$. In addition, we can deduce the derivation $\mathcal{D}_5 :: \emptyset; \emptyset; \emptyset \vdash d_1[x \mapsto v] : T_2[a \mapsto s]$. Since $\emptyset; \emptyset; \emptyset \vdash \operatorname{let} \exists(x) = \exists(v) \in d_1 : T_2$, we know that $T_2$ does not contain any free static variables. Therefore, $T_2[a \mapsto s] = T_2$. Then we can rewrite $\mathcal{D}_5$ as $\emptyset; \emptyset; \emptyset \vdash d_1[x \mapsto v] : T_2$, which is what need to prove in this case.

- $d = \operatorname{let} \land (x) = \land(v) \in d_1$ and $d' = d_1[x \mapsto v]$. Similar to the previous case.

- $d = \operatorname{newvar} \ id \ in \ d$ and $d' = (d)_{id}$. Now $ST' = ST[id \mapsto \emptyset]$. Hence, by Lemma 3.0.3, we may assume that the derivation $\mathcal{D}$ is of the following form:

\[
\begin{array}{c}
\emptyset; \emptyset; \emptyset, SA_1[id : \mathbf{1}] \vdash d : SA_2[id : T_1] \land T \\
\emptyset; \emptyset; \emptyset, SA_1[id : \mathbf{1}] \vdash (d)_{id} : SA_2 \land T
\end{array}
\]

\[
\emptyset; \emptyset; \emptyset, SA_1 \vdash \text{newvar} \ id \ in \ d_1 : SA_2 \land T
\]

In this case, $SA = SA_1$, $SA' = SA_1[id : \mathbf{1}]$ and $CT = SA_2 \land T$. Obviously, we have $\emptyset; \emptyset; \emptyset, SA' \vdash d' : CT$ is derivable. In addiction, since $ST : SA$ holds, $ST : SA'$ also holds.
• $d = \text{id}$ and $d' = ST(id) = v$. Hence, by Lemma 3.0.3, we may assume that the derivation $D$ is of the following form:

$$\begin{align*}
SA_1(id) &= T \\
\emptyset; \emptyset; \emptyset \vdash id : SA_1 \land T
\end{align*}$$

Obviously, we have $\emptyset; \emptyset; \emptyset \vdash v : T$. Applying rule $(\text{ty-state})$ yields the derivation $\emptyset; \emptyset; \emptyset; SA_1 \vdash v : SA_1 \land T$. Here, $SA = SA_1 = SA'$ and $ST = ST'$.

• $d = \text{id} \leftarrow v$ and $d' = \langle \rangle$. Hence, by Lemma 3.0.3, we may assume that the derivation $D$ is of the following form:

$$\begin{align*}
\emptyset; \emptyset; \emptyset \vdash v : T \\
\emptyset; \emptyset; \emptyset; SA_1 \vdash \text{id} \leftarrow v : SA_1[id : T] \land 1
\end{align*}$$

It is trivial that $\emptyset; \emptyset; \emptyset \vdash d' : 1$. Applying rule $(\text{ty-state})$ yields the derivation $\emptyset; \emptyset; \emptyset; SA_1[id : T] \vdash d' : SA_1[id := T]$. In this case, $SA = SA_1$ and $SA' = SA_1[id := T]$.

In addition, $ST' = ST[\text{id} := v]$. And thus, $ST' : SA'$ holds if $ST : SA$ holds.

• $d = \text{let}_{\epsilon} x = v \text{ in } d_1$ and $d' = d_1[x \mapsto v]$. There exist three sub-cases:

  - The last rule applied in $D$ is $(\text{ty-}\exists_{\epsilon}-\text{elim})$. Then $D$ is of the following form,

$$\begin{align*}
D_1 :: \emptyset; \emptyset; \emptyset \vdash v : \exists \Sigma' \cdot \vec{P}' \cdot (SA' \land T) \\
D_2 :: \emptyset; \emptyset; \emptyset \vdash \sem{x : T; SA' \vdash d_1 : CT}
\end{align*}$$

Since $v$ is a value, we know that both $\Sigma'$ and $\vec{P}'$ are empty. Now we can rewrite $D$ as follows,

$$\begin{align*}
\emptyset \vdash \Theta : \emptyset \quad \emptyset; \emptyset \vdash \emptyset[\Theta] \\
D_3 :: \emptyset; \emptyset; \emptyset \vdash v : SA'[\Theta] \land T[\Theta] \\
\emptyset; \emptyset; \emptyset \vdash \exists \emptyset, \emptyset, \emptyset[\Theta \land T] \\
D_2 :: \emptyset; \emptyset; \emptyset \vdash \emptyset. x : T; SA' \vdash d_1 : CT \\
\emptyset; \emptyset; \emptyset \vdash \text{let}_{\epsilon} x = v \text{ in } d_1 : CT
\end{align*}$$

In addition, we can infer that the last applied rule in $D_3$ must be $(\text{ty-state})$ according to the fact that $v$ is a value. Hence, $SA = SA'$. We also know that the substitution $\Theta$ is empty. Therefore, by Lemma 3.0.1, we can get a derivation $\emptyset; \emptyset; \emptyset \vdash d_1[x \mapsto v] : CT$. 
The last applied rule in \( \mathcal{D} \) is \((ty-sub_c)\). Then \( \mathcal{D} \) is of the following form,

\[
\begin{align*}
\mathcal{D}_1 &:: \Sigma; \vec{P}; \Delta; SA \vdash d : CT & \mathcal{D}_2 &:: \Sigma; \vec{P} \vdash CT \leq_{ty} CT' \\
\Sigma; \vec{P}; \Delta; SA &\vdash d : CT'
\end{align*}
\]

By induction hypothesis on \( \mathcal{D}_1 \), we have a derivation \( \mathcal{D}'_1 : \Sigma; \vec{P}; \Delta; SA \vdash d' : CT' \).
Then applying rule \((ty-sub_c)\) to \( \mathcal{D}'_1 \) and \( \mathcal{D}_2 \) yields a derivation \( \Sigma; \vec{P}; \Delta; SA \vdash d' : CT' \).

The last applied rule in \( \mathcal{D} \) is \((ty-\exists_c\text{-intro})\). Then \( \mathcal{D} \) is of the following form,

\[
\begin{align*}
\mathcal{D}_1 &:: \Sigma \vdash \Theta : \Sigma' & \mathcal{D}_2 &:: \Sigma; \vec{P} \models \vec{P}'[\Theta] & \mathcal{D}_3 &:: \Sigma; \vec{P}; \Delta; SA \vdash d : SA'[\Theta] \land T[\Theta] \\
\Sigma; \vec{P}; \Delta; SA &\vdash d : \exists \Sigma', \vec{P}' . SA' \land T
\end{align*}
\]

By induction hypothesis on \( \mathcal{D}_3 \), we have a derivation \( \mathcal{D}'_3 : \Sigma; \vec{P}; \Delta; SA \vdash d' : SA'[\Theta] \land T[\Theta] \).
Then applying rule \((ty-\exists_c\text{-intro})\) to \( \mathcal{D}_1 \), \( \mathcal{D}_2 \) and \( \mathcal{D}'_3 \) yields a derivation \( \Sigma; \vec{P}; \Delta; SA \vdash d' : \exists \Sigma', \vec{P}' . SA' \land T \).

This completes the proof.

**Theorem 3.0.5 (Progress)**

1. Assume that \( \emptyset ; \emptyset ; \emptyset \vdash d : T \) is derivable. Then either \( d \) is a value or \( d \rightarrow d' \) holds for some \( d' \).

2. Assume that \( \emptyset ; \emptyset ; \emptyset ; SA \vdash d : CT \) is derivable and \( ST : SA \) holds. Then either \( d \) is a value or \( (ST, d) \rightarrow (ST', d') \) for \( ST' \) and \( d' \).

**Proof.** We prove the above two theorems simultaneously. The proof follows from structural induction on \( \mathcal{D} \).

- We present some cases and the other cases can be handled similarly.
  - **(ty-lam)**. Then \( d = \text{lam} \ x. \ d_1 \). This case is trivial since \( d \) is a value.
  - **(ty-fix)**. Then \( d = \text{fix} \ x. \ d_1 \). It is straightforward since \( d \rightarrow d' = d_1[f \mapsto \text{fix} \ x. \ d_1] \).
• (ty-app). Then $D$ is of the following form.

$$
\begin{align*}
D_1 &:: \emptyset; \emptyset \vdash A \leq_s A' \\
D_2 &:: \emptyset; \emptyset \vdash d_1 : A' \supset T \rightarrow CT \\
D_3 &:: \emptyset; \emptyset; \emptyset \vdash \text{app}(d_1, d_2) : CT
\end{align*}
$$

Here, $d = \text{app}(d_1, d_2)$. There are three sub-cases:

- $d_1$ is not a value. Then $d_1 \rightarrow d'_1$ holds by induction hypothesis. Therefore, we can readily verify that $\text{app}(d_1, d_2) \rightarrow \text{app}(d'_1, d_2)$.

- $d_1$ is a value and $d_2$ is not value. Then $d_1 = v_1$, and $d_2 \rightarrow d'_2$ holds by induction hypothesis. Therefore, we can readily verify that $\text{app}(d_1, d_2) \rightarrow \text{app}(v_1, d'_2)$.

- Both $d_1$ and $d_2$ are values. Then $d_1 = v_1$ and $d_2 = v_2$. Since $v_1$ is a value with type $SA' \supset T \rightarrow CT$, from Lemma 3.0.2, we know that $v_1$ has the form $\text{lam } x. d_3$. Therefore, $d \rightarrow d_3[x \mapsto v_2]$ holds.

• (ty-∀-intro). It is trivial since $d$ is a value.

• (ty-∀-elim). Then $D$ is of the following form.

$$
\begin{align*}
\emptyset; \emptyset \vdash d_1 &:: \forall a : \sigma. T \\
\emptyset; \emptyset \vdash s &:: \sigma \\
\emptyset; \emptyset \vdash \forall^{-}(d_1) &:: T[a \mapsto s]
\end{align*}
$$

Here, $d = \forall^{-}(d_1)$. There are two sub-cases:

- $d_1 \rightarrow d'_1$ holds. Then we can readily verify that $d \rightarrow d' = \forall^{-}(d'_1)$.

- $d_1$ is a value. Then $d = \forall^{-}(d_1)$ is also a value.

• (ty-⊃-intro). Then $D$ is of the following form.

$$
\begin{align*}
\emptyset; \emptyset; P; \emptyset \vdash d_1 &:: T \\
\emptyset; \emptyset; \emptyset &\vdash \supset^{+}(d_1) : P \supset T
\end{align*}
$$

Here, $d = \supset^{+}(d_1)$. There are two sub-cases:

- $d_1 \rightarrow d'_1$. Then we can readily verify that $d \rightarrow d^{+} = \supset^{+}(d'_1)$.

- $d_1$ is a value. Then $d = \supset^{+}(d_1)$ is also a value.
(ty-⊃-elim). Similar to case (ty-∀-elim).

(ty-∃-intro). Similar to case (ty-⊃-intro).

(ty-∃-elim). Then $D$ is of the following form:

$$
\frac{
\emptyset; \emptyset; \emptyset \vdash d_1 : \exists a : \sigma. T_1 \quad \emptyset; a : \sigma; \emptyset \vdash d_2 : T_2
}{\emptyset; \emptyset; \emptyset \vdash \text{let } \exists(x) = d_1 \text{ in } d_2 : T_2}
$$

Here, $d = \text{let } \exists(x) = d_1 \text{ in } d_2$. There are two sub-cases.

- $d_1 \rightarrow d_1'$. Then we can readily verify that $d \rightarrow d' = \text{let } \exists(x) = d_1' \text{ in } d_2$.
- $d_1$ is a value. Since $d_1$ is a value with type $\exists a : \sigma. T_1$, by Lemma 3.0.2, $d_1 = \exists(v)$ for some $v$. Then $d \rightarrow d_2[x \mapsto v]$.

(ty-∧-intro). It is trivial since $d$ is a value.

(ty-∧-elim). Similar to case (ty-∃-elim).

(ty-id-intro). Then $D$ is of the following form:

$$
\frac{
D_1 :: \emptyset; \emptyset; \emptyset; \mathit{SA}[id : T_1] \vdash d_1 : \mathit{SA}'[id : T_2] \land T
}{\emptyset; \emptyset; \emptyset; \mathit{SA}[id : T_1] \vdash (d_1)_id : \mathit{SA}' \land T}
$$

Here, $d = (d_1)_id$. There are two sub-cases:

- $d_1$ is not a value. By induction hypothesis, $(ST, d_1) \rightarrow (ST', d_1')$. Then $(ST, d) \rightarrow (ST', (d_1)_id)$.
- $d_1$ is a value. Then $(ST, d) \rightarrow (ST', d_1)$, where $ST = ST'[id \mapsto v]$ for some value $v'$.

(ty-id-elim). Then $D$ is of the following form.

$$
\frac{
\emptyset; \emptyset; \emptyset; \mathit{SA}[id : \mathbf{1}] \vdash (d_1)_id : \mathit{SA}' \land T
}{\emptyset; \emptyset; \emptyset; \mathit{SA} \vdash \text{newvar } id \text{ in } d_1 : \mathit{SA}' \land T}
$$

Here, $d = \text{newvar } id \text{ in } d_1$. Assume that $ST : \mathit{SA}$ holds. It is trivial that $(ST, d) \rightarrow (ST', (d_1)_id)$ for some $ST'$ and $ST' : \mathit{SA}[id : \mathbf{1}]$ holds. Furthermore, $ST' = ST[id \mapsto \langle \rangle]$.
• **(ty-read)**. Then $D$ is of the following form.

\[
\begin{align*}
\text{SA}(id) &= T \\
\emptyset; \emptyset; \emptyset; \text{SA} &\vdash !id : \text{SA} \land T
\end{align*}
\]

Here, $d = !id$. Assume that $ST : \text{SA}$ holds. It is trivial that $(ST, d) \rightarrow (ST', ST(id))$ and $ST' = ST$.

• **(ty-write)**. Then $D$ is of the following form.

\[
\begin{align*}
\emptyset; \emptyset; \emptyset &\vdash d_1 : T \\
\emptyset; \emptyset; \emptyset; \text{SA}[id : T'] &\vdash id \leftarrow d_1 : \text{SA}[id : T] \land T
\end{align*}
\]

Here, $d = id \leftarrow d_1$. There are two sub-cases:

- $d_1 \rightarrow d'_1$. Then it is trivial that $(ST, d) \rightarrow (ST', id \leftarrow d'_1)$ and $ST' = ST$.

- $d_1$ is a value. Assume that $ST : \text{SA}$ holds. We know that $(ST, d) \rightarrow (ST', \langle \rangle)$, where $ST' = ST[id := d_1]$.

• **(ty-if)**. Then $D$ is of the following form.

\[
\begin{align*}
\emptyset; \emptyset; \emptyset &\vdash \text{bool}(P) \\
\emptyset; \emptyset; P; \emptyset; \text{SA} &\vdash d_2 : CT \\
\emptyset; \emptyset; P; \emptyset; \text{SA} &\vdash d_3 : CT
\end{align*}
\]

Here, $d = \text{if}(d_1, d_2, d_3)$. There are three sub-cases:

- $d_1 \rightarrow d'_1$. It is trivial that $(ST, d) \rightarrow (ST', \text{if}(d'_1, d_2, d_3))$, where $ST' = ST$.

- $d_1$ is a value and $P$ holds. Then $(ST, d) \rightarrow (ST', d_2)$, where $ST' = ST$.

- $d_1$ is a value and $\neg P$ holds. Then $(ST, d) \rightarrow (ST', d_3)$, where $ST' = ST$.

• **(ty-∃c-elim)**. Then $D$ is of the following form.

\[
\begin{align*}
\emptyset; \emptyset; \emptyset &\vdash \exists \Sigma', \vec{P}'. (\text{SA}' \land T) \\
\emptyset; \Sigma'; \vec{P}', \emptyset; x : T; \text{SA}' &\vdash d_2 : CT
\end{align*}
\]

Here, $d = \text{let}_c x = d_1 \text{ in } d_2$. There are two sub-cases:

- $d_1 \rightarrow d'_1$. It is trivial that $(ST, d) \rightarrow (ST', \text{let}_c x = d_1 \text{ in } d_2)$, where $ST' = ST$. 
– $d_1$ is a value. Then $(ST, d) \rightarrow (ST', d_2[x \mapsto d_1])$, where $ST' = ST$.

This completes the proof.

With Theorem 3.0.4 and 3.0.5, we have shown that ATS/ST is sound.
Chapter 4

Encoding Floyd-Hoare Logic

Floyd-Hoare logic [17, 19] provides an axiomatic basis for reasoning about procedural programs. In this chapter, we present a direct encoding of Floyd-Hoare logic in ATS/ST. Since Floyd-Hoare logic was introduced in 1960’s, it has been successfully extended, in literally hundreds of papers, to handle other programming constructs. It’s beyond our reasonable hope to support those various extensions. Instead, we start with the original idea of C.A. Hoare and focus on the essence of Hoare logic to avoid the complications involved which would obscure the central issues.

We first introduce a simple imperative programming language SIPL and its proof theory based on Floyd-Hoare logic. The syntax of the language SIPL is presented in Figure 4.1. We use $x_N$ for (integer) reference variables, that is, variables storing integers, $e_N$ for integer

\[
\begin{align*}
\text{int. exp.} \quad e_N &::= \quad i \mid x_N \mid e^1_N + e^2_N \mid e^1_N - e^2_N \mid e^1_N \ast e^2_N \mid e^1_N/e^2_N \mid \cdots \\
\text{bool exp.} \quad e_B &::= \quad b \mid e^1_N < e^2_N \mid e^1_N \leq e^2_N \mid e^1_N > e^2_N \mid e^1_N \geq e^2_N \\
&\quad \mid e^1_N = e^2_N \mid e^1_N \neq e^2_N \mid \cdots \\
\text{statements} \quad S &::= \quad x_N := e_N \mid S_1; S_2 \mid \text{if } e_B \text{ then } S_1 \text{ else } S_2 \text{ fi} \\
&\quad \mid \text{while } e_B \text{ do } S \text{ od}
\end{align*}
\]

Figure 4.1: The syntax for SIPL
expressions, $e_B$ for boolean expressions, and $S$ for statements. A statement is also called a command. To facilitate the following presentation, we assume there is total ordering < on reference variables, that is, $x_1 > x_2$ or $x_1 < x_2$ for any two reference variables $x_1$ and $x_2$.

A state $st$ in SIPL is a finite mapping from integer reference variables to integer constants. Let $S$ be a statement containing at most $n$ reference variables $x_1^N, \cdots, x_n^N$ and $st$ be a state with domain $\text{dom}(st) = \{x_1^N, \cdots, x_n^N\}$. It is straightforward to define a relation $(st, S) \downarrow st'$ to mean the execution of $S$ starts at the state $st$ and terminates at the state $st'$. We omit the related details, which, for instance, can be found in [1].

We say $p$ is an assertion if $p$ is a boolean expression $e_B$ or $p$ is $\neg p_1, p_1 \land p_2, p_1 \lor p_2$ or $p_1 \supset p_2$, where $p_1$ and $p_2$ are assertions. A Hoare triple is a formula of the form $\{p_1\}S\{p_2\}$, where $p_1$ and $p_2$ are called the pre-condition and the post-condition of this formula. Intuitively, this formula asserts that if the statement $S$ is executed in a state satisfying $p_1$ and the execution terminates, then a state satisfying $p_2$ can be reached. The inference rules for deriving Hoare triples are given in Figure 4.2.

Given an integer expression $e_N$ containing at most $n$ reference variables $x_1^N < \cdots < x_n^N$, we write $\hat{e}_N[a_1, \cdots, a_n] (\hat{e}_N[id_1, \cdots, id_n])$ for the static (dynamic) term $I(d)$ obtained from replacing $x_i^N$ with $a_i(id_i)$ for $1 \leq i \leq n$. We also have similar notations for boolean

\[
\begin{align*}
\{p[x_N \mapsto e]\}x_N := e_N\{p\} & \quad \text{(assign)} \\
\{p_1\}S_1\{p_2\} & \quad \{p_2\}S_2\{p_3\} & \quad \text{(seq)} \\
\{p_1 \land e_B\}S_1\{p_2\} & \quad \{p_1 \land \neg e_B\}S_2\{p_2\} & \quad \{p_1\}\text{if } e_B \text{ then } S_1 \text{ else } S_2 \{p_2\} & \quad \text{(if)} \\
\{p \land e_B\}S\{p\} & \quad \{p\}\text{while } e_B \text{ do } S \text{ od}\{p \land \neg e_B\} & \quad \text{(while)} \\
p_1 \supset p_2 & \quad \{p_2\}S\{p_3\} & \quad p_3 \supset p_4 & \quad \{p_1\}S\{p_4\} & \quad \text{(conseq)}
\end{align*}
\]

Figure 4.2: The proof rules for Hoare logic
expression $e_B$ and assertion $p$. For a statement $S$ containing at most $n$ reference variables $x_N^1 < \cdots < x_N^n$, we define $\hat{S}[id_1, \cdots, id_n]$ in Figure 4.3. Note that we use $\forall_n^+(\cdot)(\forall_n^-(\cdot))$ for $\forall^+(\cdots(\forall^+(\cdot))\cdots)$, where there are $n$ occurrences of $\forall^+(\forall^-)$.

The following theorem indicates that the statement translation in Figure 4.3 preserves the dynamic semantics of a statement.

**Theorem 4.0.6** Let $S$ be a statement in SIPL containing at most $n$ reference variables $x_N^1 < \cdots < x_N^n$ and $st$ be a state such that $\text{dom}(st) = \{x_N^1, \cdots, x_N^n\}$. Also, let $id_1, \cdots, id_n$ be $n$ distinct identifiers and $ST$ be a state such that $ST(id_i) = st(x_N^i)$. If $(st, S) \downarrow st'$ for some state $st'$, that is, the execution of $S$ starts at $st$ and terminates at $st'$, then $(ST, (\hat{S})[id_1, \cdots, id_n]) \rightarrow^* (ST', \langle\rangle)$ holds for some state $ST'$ such that $ST'(id_i) = st'(x_N^i)$ holds for each $1 \leq i \leq n$.

We now describe an approach to translating Hoare triples into typing judgments in our system. Given a Hoare triple $\{p_1\}S\{p_2\}$ containing at most $n$ reference variables, let us choose some distinct static variables $a_1, \cdots, a_n, a'_1, \cdots, a'_n$ and some distinct identifiers
\[id_1, \ldots, id_n\] and assume:

\[
\Sigma = \emptyset, a_1 : \text{int}, \ldots, a_n : \text{int}\\
\Sigma' = \emptyset, a'_1 : \text{int}, \ldots, a'_n : \text{int}\\
SA = [id_1 : \text{int}(a_1), \ldots, id_n : \text{int}(a_n)]\\
SA' = [id_1 : \text{int}(a'_1), \ldots, id_n : \text{int}(a'_n)]\\
CT = \exists \Sigma', \hat{p}_2[a'_1, \ldots, a'_n].SA' \land 1
\]

Then the translation of \(\{p_1\}S\{p_2\}\) is \(\Sigma; \hat{p}_1[a_1, \ldots, a_n]; \emptyset; SA \vdash \hat{S}[id_1, \ldots, id_n] : CT\).

**Theorem 4.0.7** If a Hoare triple \(\{p_1\}S\{p_2\}\) is derivable, then its translation, which is a typing judgment in ATS/ST, is also derivable.

**PROOF.** Assume that there are at most \(n\) reference variables \(x^1_N, \ldots, x^n_N\) in the derivation of the Hoare triple \(\{p_1\}S\{p_2\}\), and we pick distinct static variables \(a_1, \ldots, a_n, a'_1, \ldots, a'_n\) and distinct identifiers \(id_1, \ldots, id_n\). We may write \(\vec{a}\) for \(a_1, \ldots, a_n\), \(\vec{a'}\) for \(a'_1, \ldots, a'_n\) and \(\vec{id}\) for \(id_1, \ldots, id_n\), respectively. All we need is to show that each proof rules in Figure 4.2 in translated into an admissible typing rule in ATS/ST. let us assume:

\[
\Sigma = \emptyset, a_1 : \text{int}, \ldots, a_n : \text{int}\\
\Sigma' = \emptyset, a'_1 : \text{int}, \ldots, a'_n : \text{int}\\
SA = [id_1 : \text{int}(a_1), \ldots, id_n : \text{int}(a_n)]\\
SA' = [id_1 : \text{int}(a'_1), \ldots, id_n : \text{int}(a'_n)]\\
\]

- As for the rule (assign), let \(p' = p[x_N \mapsto e_N]\), where \(x_N\) is some \(x^k_N\). The translation of \(\{p'\}x_N := e_N\{p\}\) is

\[
\Sigma; \hat{p}'[a_1, \ldots, a_n]; \emptyset; SA \vdash d : CT
\]

where we have \(d = id_k \leftarrow \hat{e}_N[id_1, \ldots, id_n]\) and \(CT = \exists \Sigma', \hat{p}[a'_1, \ldots, a'_n].SA' \land 1\).

According to the dy-types assigned to the functions \(+, -, *, /,\) etc., we know that the following typing judgment is derivable:

\[
\Sigma; \hat{p}'[\vec{a}]; \emptyset \vdash \hat{e}_N[\vec{id}] : \text{int}(\hat{e}_N[\vec{a}])
\]
Hence, the following typing judgment is also derivable:

\[
\Sigma; \hat{\nu}'[\vec{a}]; \emptyset; SA \vdash id_k \leftarrow \hat{e}_N[\vec{i}d] : SA[id_k : \text{int}(\hat{e}_N[\vec{a}])] \land 1
\]

Let Θ be the static substitution that maps \(a'_k\) to \(\hat{e}_N[\vec{a}]\) and \(a'_i\) to \(a_i\) for each \(1 \leq i \neq k \leq n\). Then we have \((\hat{p}[\vec{a}'])[\Theta] = \hat{\nu}'[\vec{a}]\) and \(SA'[\Theta] = SA\). By the typing rule \(\text{(ty-∃c-intro)}\), \(\Sigma; \hat{\nu}'[\vec{a}]; SA \vdash d : CT\) is derivable.

- **Rule (seq).** The statement \(S_1; S_2\) is translated into \(\text{let}_c x = \hat{S}_1[\vec{i}d] \text{ in } \hat{S}_2[\vec{i}d]\). Assume that
  
  1. \(\Sigma; \hat{\nu}_1[\vec{a}]; \emptyset; SA \vdash \hat{S}_1[\vec{i}d] : \exists \Sigma', \hat{\nu}_2[\vec{a}'].SA' \land 1\) (from \(\{p_1\}S_1\{p_2\}\)), and
  2. \(\Sigma'; \hat{\nu}_2[\vec{a}']; \emptyset; SA' \vdash \hat{S}_2[\vec{i}d] : \exists \Sigma'', \hat{\nu}_3[\vec{a}''].SA'' \land 1\) (from \(\{p_2\}S_2\{p_3\}\)), where \(\Sigma'' = \emptyset, a'_1 : \text{int}, \ldots, a'_n : \text{int}\) and \(SA'' = [id_1 : \text{int}(a'_1), \ldots, id_n : \text{int}(a'_n)]\).

In addition, we know that the \(\text{dom}(\Sigma)\) and \(\text{dom}(\Sigma')\) are distinct. By the rule \(\text{(ty-∃c-elim)}\), the following typing judgment is derivable:

\[
\Sigma; \hat{\nu}_1[\vec{a}]; \emptyset; SA \vdash \text{let}_c x = \hat{S}_1[\vec{i}d] \text{ in } \hat{S}_2[\vec{i}d] : CT
\]

where \(CT = \exists \Sigma'', \hat{\nu}_3[\vec{a}''].SA'' \land 1\).

- **Rule (if).** The statement is translated into \(\text{if}(\hat{e}_B[\vec{i}d], \hat{S}_1[\vec{i}d], \hat{S}_2[\vec{i}d])\). We have the following assumption based on the two premises of rule (if):
  
  1. \(\Sigma; \hat{\nu}_1[\vec{a}] \land \hat{e}_B[\vec{a}]; \emptyset; SA \vdash \hat{S}_1[\vec{i}d] : CT\), and
  2. \(\Sigma; \hat{\nu}_1[\vec{a}] \land \neg \hat{e}_B[\vec{a}]; \emptyset; SA \vdash \hat{S}_2[\vec{i}d] : CT\).

where \(CT = \exists \Sigma', \hat{\nu}_2[\vec{a}'].SA' \land 1\). Then by typing rule \(\text{(ty-if)}\), it is trivial that \(\Sigma; \hat{\nu}_1[\vec{a}]; \emptyset; SA \vdash \text{if}(\hat{e}_B[\vec{i}d], \hat{S}_1[\vec{i}d], \hat{S}_2[\vec{i}d]) : CT\) is derivable.

- **Rule (while).** The statement \(\text{while } e_B \text{ do } S \text{ od}\) is translated into \(d(\langle \rangle)\), where \(d\) is the following dynamic term:

\[
\text{fix } f. \forall^+ \text{lam } x. \text{if}(\hat{e}_B[\vec{i}d], \text{let}_c x_2 = \hat{S}_1[\vec{i}d] \text{ in } \forall^- (f)(x), \langle \rangle)\)
Assume that \( \Sigma; \hat{p}[\vec{a}] \land \hat{e}_{B}[\vec{a}]; SA \vdash \hat{S}[i\vec{d}] : \exists \Sigma', \hat{p}[\vec{a}'].SA' \land 1 \) is derivable. Then it can be readily verified that the following typing judgment is derivable:

\[
\emptyset; \emptyset; \emptyset \vdash : \forall a_1 : int \cdots \forall a_n : int. \hat{p}[\vec{a}] \supset (SA \supset (1 \rightarrow CT))
\]

where \( CT = \exists \Sigma', \hat{p}[\vec{a}'] \land \neg \hat{e}_{B}[\vec{a}'].SA' \land 1. \)

- Rule \textit{(conseq)}. \( p_1 \supset p_2 \) is translated into \( \Sigma; \hat{p}_1[\vec{a}] \models \hat{p}_2[\vec{a}] \) in ATS/ST. It is trivial to justify that the following rule holds in ATS/ST,

\[
\begin{array}{c}
\Sigma; \vec{P}_1 \models \vec{P}_2 \\
\Sigma; \vec{P}_2; \Delta; SA \vdash d : CT
\end{array}
\]

\[
\Sigma; \vec{P}_1; \Delta; SA \vdash d : CT
\]

and thus we have \( \Sigma; \hat{p}_1[\vec{a}]; \emptyset; SA \vdash \hat{S}[i\vec{d}] : CT \), where \( CT = \exists \Sigma', \hat{p}_3[\vec{a}'].SA' \land 1. \) Because \( p_3 \supset p_4 \), it is easy to deduce that \( CT \leq_{ct} CT' \) by rule \textit{(sub-ct)} in Figure 2.2, where \( CT' = \exists \Sigma', \hat{p}_4[\vec{a}'].SA' \land 1. \) Hence, it is easy to verify that \( \Sigma; \hat{p}_1[\vec{a}]; \emptyset; SA \vdash \hat{S}[i\vec{d}] : CT' \) by rule \textit{(ty-sub)} in Figure 2.4.

This completes the proof.

By Theorem 4.0.6 and Theorem 4.0.7, we have formally justified the encoding of Floyd-Hoare logic in ATS/ST.
Chapter 5

Programming with Typed States

We have presented some examples in the introduction. In this chapter we show more examples to illustrate the potential practical use of ATS/ST. Although we have not actually tested these examples in an implementation of ATS/ST, we can reasonably expect that these examples will pass such a test in the future as similar corresponding examples have already been successfully tested in a prototype implementation of Xanadu [38, 36].

The program in Figure 5.1 implements the function that reverse a given list. The datatype declaration introduces type constructors list that takes a type $T$ and a static integer term $I$ to form a type $\text{list}(T,I)$. The two value constructors associated with list are assigned the following types:

- $\text{nil} : \forall a : \text{type}. \text{list}(a,0)$
- $\text{cons} : \forall a_1 : \text{type}. \forall a_2 : \text{int}. a_2 \geq 0 \supset a_1 * \text{list}(a_1,a_2) \rightarrow \text{list}(a_1,a_2 + 1)$

The header in the definition of the function $\text{reverse}$ means that the following type is assigned to $\text{reverse}$:

$$\forall a_1 : \text{type}. \forall a_2 : \text{type}. a_2 \geq 0 \supset \text{list}(a_1,a_2) \rightarrow \text{list}(a_1,a_2)$$

That is, $\text{reverse}$ is a length-preserving function on lists. The pattern matching here needs some explanation: if a value of the form $\text{cons}(v_1, v_2)$ matches the pattern $\text{cons}(&x', &xs')$, 

52
datatype list (type, int) =
    {a: type} nil of list (a, 0) |
    {a: type} cons (a, n+1) of a * list (a, n)

fun reverse {a:type, n:int | n >= 0}
(xs: list(a, n)): list(a, n) =
newvar x', xs', ys' in
  let
    fun loop {p:int, q:int | n = p+q}
      ([xs: list(a, p), ys': list(a, q)]; /*none*/): (/*none*/; list(a, n)) =
      case !xs' of
        | nil => ys'
        | cons (&x', &xs') =>
          (ys' := cons (!x', !ys'); loop ())
      in
      (xs' := xs; ys' := nil; loop ())
  end

Figure 5.1: Reversing Lists
fun swap {id: addr, id': addr, T: type, T': type}
    ([:id: T, id': T'; x: ptr(id), y: ptr(id')]:
    ([:id: T', id': T]; /*none*/) =
        let temp = !x
        in
        (x := !y; y := temp)
end

Figure 5.2: Swapping

then $v_1$ and $v_2$ are assigned to $x'$ and $xs'$, respectively. We use $\&$ to mean that it is an
in-place update.

In the above example, we feed nothing to the inner function loop and simply rely on
the imperative features of this language to mutate the contents of those variable declared
outside. In fact, a function can take pointers as arguments. Figure 5.2 implements a
swapping function which takes two pointers and swaps their contents. We can observe the
invariants from state assertions. The header in the definition of the function swap means
that the following type is assigned to swap:

$$\forall id : addr. \forall id' : addr. \forall T : type. \forall T' : type.
    [id : T, id' : T'] \supset (ptr(id), ptr(id')) \rightarrow ([id : T', id' : T] \land 1)$$

The type of swap means that, initially, the memory location id contains a value of type $T$
and id' stores a value of type $T'$, and after the execution of swap, the types of the contents
at id and id' are $T'$ and $T$, respectively.
Chapter 6

Conclusion and Future Work

We have presented an approach to unify type theory with Floyd-Hoare logic. By doing so, Floyd-Hoare logic-like reasoning can be supported in a practical programming language. On the other hand, the consequential type system is able to capture more run-time information and state changes statically and thus enforce more safety properties as well as type safety at compile time. Furthermore, this type system is suitable for stateful programming when imperative features such as assignment and pointers are involved. In the future we are going to handle recursive data structures such as linked list. In addition, we are interested in supporting some extensions of Floyd-Hoare logic such as Separation logic and enforcing more safety properties at compile time.
Bibliography


[37] Hongwei Xi. Dependent ML. Available at
[38] Hongwei Xi. Xanadu: Imperative Programming with Dependent Types, 2001. Available at:

[39] Hongwei Xi. Applied type system, July 2003. Available at:
http://www.cs.bu.edu/~hwxi/ATS.


