Writing an Operating System with a Strictly Typed Assembly Language

Toshiyuki Maeda

**Submitted to Department of Computer Science,
Graduate School of Information Science and Technology,
The University of Tokyo on December 16, 2005
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy**

**Thesis Supervisor: Akinori Yonezawa
Professor**

Abstract

The recent advances in static program analysis, especially in type theory, have been widely accepted as a fundamental technology for software development. For example, many real-world applications are written in strictly-typed programming languages (e.g., Java, C# and Objective Caml). One strength of the strictly-typed programming languages is that programs written in them never go wrong, that is, the programs cause no unexpected errors, thanks to type safety ensured by their type checkers.

However, there is one kind of software that does not fully utilize the type theory: operating systems (OS). Traditional and current OS kernels have been written in the C programming language (weakly-typed unsafe programming language) and untyped assembly languages. Therefore, it is very hard to ensure and verify safety of the OS kernels.

One of the reasons why OS kernels have not been written in strictly-typed programming languages is that it has been believed that it is hard or impossible because strictly-typed programming languages do not seem to provide means to implement important mechanisms of OS kernels: memory management and multi-thread management mechanisms.

This thesis breaks the mistaken believes about types and operating systems by showing how to write an OS kernel in a strictly-typed programming language. More specifically, this thesis proposes a new strictly-typed assembly language which is flexible and expressive enough to implement OS kernels. The key point of the language is that its type system supports variable-length arrays, explicit alias tracking and integer constraints between variables. Therefore, practical memory management (i.e., malloc/free) and multi-thread management mechanisms can be implemented in the language and fully type-checked. This thesis also introduces a prototype OS kernel implementation written in the language.

The safety ensured by the type checker of the language is memory safety and control-flow safety. The memory safety means that a program accesses only memory which the program is permitted to access, while the control-

flow safety means that a program jumps to only valid code which the program is permitted to execute. More sophisticated and complicated safety (e.g., deadlock freedom and resource usage safety) can be ensured by extending the type system, but it is out of the scope of this thesis. In addition, this thesis also discusses the previous works in the area of formal verification of OS.

Acknowledgment

I am deeply indebted to my supervisor Prof. Akinori Yonezawa, for leading me to this very exciting area in computer science, programming language design, implementation, and type theory. He provided me numerous enlightening suggestions and comments. He also always mentally encouraged me in my work, and gave me numerous chances to go abroad and meet many foreign researchers.

I am grateful to Dr. Yoshihiro Oyama whose stimulating suggestions helped me to refine my thoughts. He supported me with his deep knowledge about operating systems and software security.

I am more than thankful to Dr. Eijiro Sumii. He first mentioned the intuitive idea of applying type theory to operating systems. He enlightened me by his comprehensive knowledge about type theory. He also suggested the direction of my work and supported me with his bright ideas.

I would also like to thank Dr. David Nowak, for his insightful comments to the theoretical aspects of my work. He pointed out numerous problems in the type system of early stages.

Prof. Takashi Masuda and Dr. Kenji Kono have also been great mentors since I was an undergraduate student. They taught me basics of operating systems.

I would also like to thank colleagues of Yonezawa group. They have stimulated me in positive and important ways. In particular, I give my appreciation to Mr. Daisuke Shimamoto for helping me to write this acknowledgment.

I would like to give my special thanks to the members of the fan club for Amin Okada, an excellent manga artist. They encouraged me in special and unique ways.

I am also especially indebted to the Ushio foundation for their financial assistance. This thesis would not have been possible without their help.

Finally, I am thankful to the members of the thesis committee for their insightful comments and criticism.

Contents

List of Figures

Chapter 1

Introduction

Today, computers (e.g., PCs, cell-phones and electronic wallets) are widely used in the world and their network becomes one of the indispensable social infrastructures. Therefore, the importance of ensuring safety of software is commonly recognized. For example, many programs come to be written in strictly-typed languages (e.g., OCaml [17], C# [59], Java [60] and so on [88, 66, 29, 97, 87, 64]), because the program that is written in a strictly-typed language raises no unexpected errors (e.g., segmentation faults and stack overflows) at runtime. As of writing, according to SourceForge.net [84], which is the world's largest open source software development web site, 6 out of 10 applications are written with strictly-typed languages.

However, there is one kind of programs that have not been written in strictly-typed languages: operating system (OS). For example, today's commonly-used OSes (e.g., Linux [8], FreeBSD [67], Solaris [65] and Windows XP [80]) are written in weakly-typed languages (C [57] and assembly languages). In addition, programs that have similar functionality as OSes (e.g., standard runtime libraries [36], language interpreters [97, 87, 64] and Internet servers [35, 19, 56]) are also written in the weakly-typed languages.

Therefore, it seems very hard to ensure and/or verify safety of OSes. One approach for ensuring the safety is to use model checking [51]. Model checking is a method of formally verifying finite models of programs according to specified formal specifications. One problem of model-checking is that it does not scale with large programs because of the state explosion problem. Thus, ensuring the safety with model checking is a difficult task.

Another approach is to directly prove the safety with proof assistants [11, 10]. One problem of this approach is that proof assistants are still too

difficult to use for average programmers. In addition, in this approach, programs and proofs are constructed independently. Therefore, if the programs are modified, we need to prove the safety again. This doubles the cost of building OSes.

Compared to the above two approaches, writing an OS with a strictly typed language is more effective and efficient. If an OS is written in a strictly typed language, safety of the OS is ensured by the type checker of the language automatically. In addition, programmers need not to prove the safety separately from the OS. Thus, programmers can concentrate on building the OS.

One of the reasons why OSes have not been written in strictly typed languages is that it is believed that important OS facilities, such as memory management (i.e., malloc/free), multi-thread management and device drivers, cannot be written in them. In fact, exiting strictly typed languages typically depend on external memory management mechanisms, such as garbage collection.

To solve the problem, we have designed the strictly and statically typed assembly language (named TALK) that is powerful enough to implement memory management and multi-thread management, and actually implemented them in TALK. The memory management and multi-thread management are considered as the core of all kinds of OSes, and the other facilities are built on the core. Thus, writing the core in strictly and statically typed languages is a significant step for establishing a practical way to ensure and verify the safety of OSes.

The safety properties ensured by the type system of TALK is the basic type safety: memory safety and control-flow safety. The memory safety means that a program accesses only memory which the program is permitted to access, while the control-flow safety means that a program jumps to only valid code which the program is permitted to execute. It might be argued that the safety ensured by the type system of TALK is too simple for OSes. In fact, there are many sophisticated and valuable safety properties that should be satisfied by OSes, such as the deadlock and/or livelock freedom, the multi-thread safety on SMP, the correctness of the file system, the safety of resource usage, the secure information flow, and so on. However, we still believe that TALK has made a significant step toward the goal of ensuring the safety of OSes, because the sophisticated safety properties are typically built on the basis of the basic type safety. For example, the basic type safety is assumed by the type system of [33] that ensures the deadlock and race freedom, the type system of [54] that ensures the resource usage safety, and the type system of [72] that ensures the secure information flow.

We believe that TALK can be extended with the type systems for ensuring the more sophisticated safety properties, but it is out of the scope of this thesis.

More specifically, the contribution of this thesis is that we have designed the new type system that is flexible and powerful enough to implement memory management code that satisfies the memory safety and the control-flow safety. The key of the type system is that it integrates the following four mechanisms: (1) the support for variable-length arrays as language primitives, (2) integer constraints between variables, (3) explicit alias tracking between pointers, and (4) split and concatenation of the variable-length arrays. Although the integer constraints and the explicit alias tracking have been separately studied by different research areas, we have first realized that integrating them with the variable-length arrays and their split/concatenation enables us to write practical memory management code in strictly-typed languages.

A statically typed language is employed because programs written in a statically typed language are type-checked at compile time, that is, we can prevent runtime errors. Dynamically typed languages can detect runtime errors, but it is sometimes useless when implementing OSes. For example, let us think of a program which switches contexts of threads. Detecting errors in the program at runtime is insufficient, because no other threads can be executed.

We chose the typed assembly language as a base of TALK for two reasons. The first is that the typed assembly language can express low-level operations (e.g., register manipulation) because it is an ordinary assembly language (except for begin typed, of course). The low-level operations are essential for implementing memory management and multi-thread management mechanisms.

The second is that programs built with the typed assembly language can be type-checked at the level of binary executables by annotating the executables with type information. This means that the type safety of OSes can be verified without their source code. In addition, we can keep the trusted computing base small because only the type checker is to be trusted. Of course, writing the whole OS directly in the assembly language is very difficult. Therefore, in practice, we need high-level languages and their compilers that can be compiled to TALK, but it is out of the scope of this thesis.

1.1 Scope of this thesis

To clarify the scope of this thesis and its effectiveness, we explain what kind of safety properties are ensured by the approach of this thesis. (Note that the safety properties shown below are only examples, that is, we are not claiming that the properties that are not mentioned in this section cannot be ensured by TALK.)

In sum, the type system of TALK is able to ensure memory safety and control-flow safety of programs. In addition, based on the memory and control-flow safeties, race freedom on single CPU machines can be ensured easily. On the other hand, more complex and sophisticated safeties (e.g., deadlock and livelock freedom, race freedom on multi-CPU machines and full-fledged resource usage safety) cannot be ensured directly by the type system of TALK. The limitations of the TALK type system are discussed in Section 2.8.

1.1.1 Safety properties ensured by TALK

Memory safety

Memory safety is the property that programs never perform illegal memory accesses. More specifically, memory-safe programs never access the memory regions that they are not permitted to access. One strong point of the type system of TALK is that it is able to ensure the memory safety and expressive enough to implement memory management mechanism such as malloc/free.

The memory safety is the most fundamental and important safety property in the sense that, usually, verification of more complex safety properties is performed under the assumption that the memory safety is ensured. For example, existing model checkers assume that the memory safety is satisfied by some means and do not (in fact, cannot) verify it by itself (see Section 4.2). In addition, type systems that can ensure more complex safety properties also assume the memory safety in order to simplify formal arguments.

One naive way to ensure the memory safety is to ban explicit memory access in programs. For example, if a program is written in a programming language which does not have pointers or references, the program is apparently memory-safe. However, it cannot be used to write OSes because memory management code cannot be implemented in it.

The typical way to ensure the memory safety in languages with pointers

or references is to preserve type invariance property by their type system. Type invariance property is the property that once a memory region initialized as a certain type, the type of the memory region never changes. For example, in the Java programming language [60], once a memory region is allocated and initialized with a certain class (e.g., `new Object();`), the memory region cannot be used or initialized as other classes. It is only accessible as the class specified at its initialization (e.g., `Object`).

If the type invariance property is satisfied, it is easy to ensure the memory safety of programs because all we have to do is just to assign appropriate types to memory regions when the regions are initialized, and check the types when accessing the regions. This approach works because once a memory region is initialized with a valid pointer, the memory region always holds valid pointers.

However, the type invariance property is too restrictive to implement memory management mechanisms. For example, consider a memory allocation function. What the function does is to find a free memory region and initializes it as a certain type. Here the point is that the type specified to the function is almost always different from that of the free memory region. That is, the function must change the type of the free memory region. In sum, the function cannot be written in the language that satisfy the type invariance property.

Therefore, the type system of TALK adopts a more relaxed but strong enough property. The property is that all the pointers that point to a memory region have the same type. It is allowed to change the type of the memory region, but the change must be reflected to all the pointers that point to the region.

If the property is satisfied, it is easy to ensure the memory safety of programs because we can easily detect invalid pointers in the programs. For example, suppose that there exists a memory region that holds a valid pointer and two pointers (X and Y) that point to the memory region. By the property, the pointers X and Y have the same pointer type that indicates that the region pointed by the pointers contains a valid pointer. Next, suppose a program updates the memory region with an invalid pointer (say NULL) thorough the pointer X (e.g., `*X = NULL`). Then, the type of the pointer X changes to the type that indicates that the region pointed by the pointer contains an invalid pointer. The point is that, by the property again, the type of the pointer Y is also updated to the same type as the pointer X. Therefore, it is impossible to make an illegal memory access through the pointer Y, because the type of the pointer Y prohibits the access (see Chapter 2 for details).

Control-flow safety

Control-flow safety is the property that programs never perform illegal execution. More specifically, control-flow safe programs never execute operations that the programs are not allowed to execute. Note that the control-flow safety and the memory safety is closely related (to see why, substitute "access" for "execute" and "memory regions" for "operations"). Unlike the memory safety, however, it is easy to ensure the control-flow safety because TALK prohibits overwriting program code at runtime.

In TALK, the control-flow safety is ensured as follows. First, every basic block in a TALK program is annotated with its label type by programmers. The label type of a basic block represents the precondition that must be satisfied when a control-flow reaches the basic block. Then, the TALK type checker verifies the following two properties for every basic block. First, if there is a jump instruction, its target address must be a head of a basic block. This is easily accomplished by checking whether the target operand of the jump instruction has a label type. Second, the precondition specified by the label type of the target basic block must be satisfied at the jump instruction (see Section 2.2 for details).

One of the good points of the control-flow safety is that, along with the memory safety, it can prevent notorious buffer-overflow attacks (e.g., stack-overflow and heap-overflow attacks). The buffer-overflow attack is an attack which overwrites the important value (e.g., return addresses) by exploiting bugs in memory access code. If the attack successes, the attacker may take full control of the system.

In TALK, the buffer-overflow attacks never success because the buggy memory access code never be type-checked in TALK. In TALK, each buffer is usually represented as a single memory region. Therefore, the type checker of TALK can reject the code that accesses outside of the buffer. Moreover, even if the buffer and some important values coexist in a single memory region, the TALK type check still rejects buggy code. For example, consider a routine which is permitted to access a memory region of size 3 (bytes) and suppose that the first two bytes represent an integer buffer and the last one byte holds a return address, that is, the types of the first two bytes are integer and that of the last one byte is a label type. Now suppose that a buggy operation in the routine may accidentally overwrite the last one byte of the memory region by an arbitrary integer value. Then, when type-checking the operation, the type checker notices that the type of the last one byte may change to integer. Therefore, a jump instruction which takes the last one byte of the memory region as its target operand cannot pass

the type check of TALK, because the operand may not have a label type. Thus, the TALK type checker rejects the buggy routine that may allow the buffer-overflow attack.

Race freedom on single CPU machines

The type system of TALK is expressive enough to implement multi-thread management (see Section 2.5.3). Therefore, even in multi-thread environment, the memory safety and the control-flow safety is ensured without problems, as long as they are executed on single CPU machines.

However, race conditions that do not affect the memory safety and the control-flow safety cannot be prevented directly by the type system of TALK. For example, let us suppose that we run two threads that execute the function shown in Figure 1.1 (for readability, examples are shown in pseudo C-like language instead of TALK in this chapter). The function increments a shared integer value `x`. Here assume that the initial value of `x` is 0. The function itself is memory-safe and control-flow safe, so it passes the type check of TALK. However, there is a race condition, that is, the result value of `x` may be `1` after the two threads exit.

```
1 void Inc(void) {
2    x = x + 1;
3 }
```

Figure 1.1: A function that increments a shared integer

To prevent the race conditions, programmers must synchronize accesses to shared data by using synchronization primitives. The advantage of the TALK type system is that it is able to express the synchronization primitives and their semantics. For example, spin locks can be implemented in TALK by exploiting existential types, variant types and dependent types. Moreover, the critical sections created with the spin locks are enforced by the type system of TALK (see Section 3.2 for details). In sum, the type system of TALK is able to prevent the race conditions indirectly, with a little help from programmers.

Note that the above arguments are only applicable to single CPU machines. On multi-CPU machines, the memory safety, the control-flow safety and the race freedom cannot be ensured by the current type system of TALK.

1.2 Organization of this thesis

The rest of this thesis is organized as follows. First, we introduce TALK in Chapter 2. We also show various examples that exhibit the expressiveness of TALK in the chapter. Next, Chapter 3 describes a prototype OS kernel implementation written in TALK. Then, Chapter 4 discusses related work. Last, we conclude this thesis in Chapter 5. The formal arguments of TALK are presented in Appendixes A and B.

Chapter 2

Typed Assembly Language for Kernel

In this chapter, we propose a new typed assembly language which is flexible and powerful enough to implement practical memory management (i.e., malloc/free) and multi-thread management. First, we show what is needed for realizing practical memory management. More specifically, we show that the support for variable-length arrays, explicit alias tracking and integer constraints is suffice to implement malloc/free. Next, we introduce the formal definition of TALK. More specifically, we give the syntax, the operational semantics and the type system for a virtual RISC CPU architecture. We also give examples in order to show the expressiveness of TALK. Then, we describe how to implement TALK for the real IA-32 architecture [21]. Last, we discuss the limitations of the proposed TALK type system.

2.1 Requisites for implementing memory management

2.1.1 Variable-length arrays

First of all, memory management code must be able to handle memory. Typically, the memory consists of memory regions. At the lowest level, the memory region is just an array of bytes. From the viewpoint of type theory, one important point is that the array is a variable-length array because its size is not known until runtime. For example, let us think of the program that is executed just after a system boots. Apparently, the program cannot make any assumption about the size of the memory, because the amount

of the available memory varies from system to system. Therefore, the type system is required to support variable-length arrays. Otherwise we cannot even know the size of the available memory, rather than implement memory management.

2.1.2 Integer constraints (singleton type and dependent type)

The simplest representation of the variable-length array is a pair of the array and its size. In addition, we introduce special functions for accessing the pairs and do not allow programmers to directly access the pair. With this representation, the type system needs not to maintain the size of the arrays.

However, this approach has a big problem: we cannot implement an allocation function of the pair (the variable-length array) in the type system. That is, we must trust the external allocation function. This contradicts the goal, designing the statically typed programming language that is able to implement memory management without external trusted memory management facilities.

To solve the problem, we need to introduce the idea of the dependent type [102, 101] to the type system. In the dependent type system, types can depend on the value which is known only at runtime. For example, the type of a variable-length array of integers can be represented as $\texttt{int}\,[\alpha]$. Here $[\alpha]$ indicates the size of the array, but the exact value, α, is not known. The type is a kind of dependent type because it depends on the integer value α.

In addition, we need to handle all integers with the dependent type, as well as the variable-length arrays, because the type information is removed and not available at runtime. For example, let us think of a program which accesses an element of the above array. Let us also suppose that a variable (say `x`) holds an integer index to the array. To ensure memory safety, the type system must be able to check whether the access is safe or not. That is, the type system must be able to check whether the value of `x` is smaller than α.

To achieve this, the type system needs to keep track of the value of `x` explicitly. For example, the type of `x` must be $\texttt{int}(\beta)$, instead of `int`. Here $\texttt{int}(\beta)$ is called a singleton type. β indicates the value of `x`, but the exact value is not known to the type system.

In addition, the type system also needs to keep track of integer constraints (the constraint between α and β in this case). For example, if the type system knows that $\alpha > \beta$, the array access is safe. Otherwise, the

access may be dangerous. Such integer constraints are generated by usual branch operations. For example, if the type of a variable `y` is $int(\alpha)$, the branch operation `if y > x` {...} `else` {...} generates the constraint $\alpha > \beta$ for the taken branch and $\alpha \leq \beta$ for the other branch. Here α is the size of the variable-length arrays. Therefore, the type system knows that it is safe to access the array by the index `x` in the taken branch.

2.1.3 Alias tracking

In the previous section, we argued how to represent memory in the type system with the variable-length arrays. As the next step, this section discusses how to manage the memory. From the viewpoint of type theory, memory management is almost the same as changing types of memory regions. For example, changing a type of a memory region from a pointer type to an integer type can be viewed as freeing the memory region that contains a pointer and reusing it for holding an integer. Figure 2.1 is an example C code which performs this *memory reuse*. The function reuses the memory region pointed by pointer `x` (in line 3).

```
1 void pointer_to_int(int** x)
2 {
3     int* y = (int*)x;
4     *y = 1;
5 }
```

Figure 2.1: Example of C code that reuses a memory region

However, existing strictly and statically typed programming languages do not allow programmers to change types of memory regions. This is because memory safety cannot be ensured. Therefore, memory management cannot be implemented in the existing strictly and statically typed languages and they depend on external memory management mechanisms, such as garbage collection. To see how changing types of memory regions violates the memory safety, consider the function in Figure 2.2 that uses the function of Figure 2.1. The function passes the type check of C, but it is apparently unsafe because it tries to dereference an integer which is no longer a pointer (in line 4).

The essential problem is that the type system does not know that `y` in the function `pointer_to_int` and the argument `x` of function `dangerous_func`

```
1 void dangerous_func(int** x)
2 {
3     pointer_to_int(x);
4     **x;
5     ...
```

Figure 2.2: Example of C code that breaches memory safety

alias, that is, point to the same memory location.

To solve the problem, we need to introduce the idea of the alias type [96] to the type system in order to keep track of aliases explicitly. The basic idea of the alias type is to change the representation of pointer types. In usual type systems, the type of a pointer is represented as the type of the memory region pointed by the pointer. In the alias type system, on the other hand, the type of the pointer is just the address of the memory region. The type of the memory region is separately maintained as memory type. The memory type is a map from addresses to the types of the memory regions at the addresses.

For example, based on the idea of the alias type, the code in Figure 2.1 can be rewritten as in Figure 2.3. First, the type of the argument `x` is changed from `int**` to `ptr(p)` (in line 2). The type `ptr(p)` indicates that `x` is a pointer which points to the address `p`. Next, the declaration surrounded by "[" and "]" represents the memory type. The memory type added before the function indicates the state of the memory before the function is called (in line 1). The other memory type added after the function indicates the state of the memory after the function is executed (in line 6). Here the memory type before the function indicates that the memory region at the address `p` has pointer type `ptr(q)` and the memory region at the address `q` has the integer type. Thus, the type system knows that `x` is a pointer to a pointer to an integer. Then, the function stores an integer to the memory region at the address `p` (in line 4). Therefore, the memory type after the function indicates that the memory region at the address `p` is an integer. Note that the alias type system ensures that `p` and `q` are different integers, because the memory type is a map.

In addition, the code of Figure 2.2 can be rewritten as Figure 2.4. The type check of the alias type system rejects the rewritten code in line 5, because after the function call (`pointer_to_int`, in line 4), the type of the memory region is changed from a pointer type (`ptr(q)`) to the integer type

```
1 [ p --> ptr(q), q --> int]
2 void pointer_to_int(ptr(p) x)
3 {
4     *x = 1;
5 }
6 [ p --> int, q --> int]
```

Figure 2.3: Example of pseudo code based on the idea of alias type

(`int`). Thus, the alias type system allows programmers to reuse memory regions explicitly because it keeps track of aliases in the memory type.

```
1 [ p --> ptr(q), q --> int]
2 void dangerous_func(ptr(p) x)
3 {
4     pointer_to_int(x);
5     **x;
6     ...
```

Figure 2.4: Example of pseudo code that curses a type error

2.1.4 Split and concatenation of arrays

As described above, we can handle memory with the variable-length arrays and the integer constraints and reuse regions in the memory by explicitly tracking pointer aliases with alias type. However, we need one more mechanism in the type system to implement practical memory management. For example, let us suppose that free memory (not in use memory) is represented as an array of integers. Then, its memory type is represented as $\texttt{int}\,[a]$ (here we assume that $a > 0$). Now, let us think of the code in Figure 2.5 that allocates one element from the top of the free memory and reuses it as a pointer to integer (in line 4). The access to the array is obviously safe because $a > 0$. However, there is a problem in how to represent the memory type of the memory after the memory reuse. More specifically, the problem is that it is difficult to represent the variable-length array whose elements are integers except for its first element.

To solve the problem, we introduce a notion of split (and concatenation)

```
1 [ p --> int[a], q --> int] where a > 0
2 void alloc_and_reuse(ptr(p) x, ptr(q) y)
3 {
4     x[0] = y;
5     ...
```

Figure 2.5: Example of pseudo code that allocates a pointer to integer from free memory (incomplete)

of arrays to the type system. For example, memory type `[ p --> int[a] ]`, which indicates that there is an array of size `a` at address `p`, can be split to memory type `[ p --> int[a1], p2 --> int[a2] ]`, which indicates that there is one array of size `a1` at address `p` and the other array of size `a2` at address `p2` (here, $\texttt{a} = \texttt{a1} + \texttt{a2}$ and $\texttt{p2} = \texttt{p} + \texttt{a1}$). In addition, the latter memory type can be concatenated back to the former memory type.

With the notion of the split of the arrays, the code of Figure 2.5 can be rewritten as in Figure 2.6. The `split` operation (in line 4) splits the free memory into new array of size 1 at `p` and the rest of the free memory at `p + 1`. In this case, we can naturally represent the memory type of the free memory after line 5 as `[ p --> ptr(q), (p + 1) --> int[a - 1] ]`.

```
1 [ p --> int[a], q --> int] where a > 0
2 void alloc_and_reuse(ptr(p) x, ptr(q) y)
3 {
4     split p, 1;
5     x[0] = y;
6     ...
```

Figure 2.6: Example of pseudo code that allocates a pointer to integer from free memory (complete)

Note that split and concatenation of variable-length arrays just change the view of memory, not change the memory itself. That is, they can be ignored at runtime. Therefore, they do not introduce any overhead at runtime. Arrays are split and concatenated only at type-checking time.

2.2 Formal definition

This section introduces the TALK language. Although the language explained in this section is based on a virtual CPU architecture, the actual implementation is based on the IA-32 [21] assembly language. For the details, refer Section 2.7.

In this section, we first explain its abstract machine and types. Then, its operational semantics and typing rules are introduced. The syntax of TALK is shown in Figures 2.7 and 2.8.

2.2.1 Abstract machine

The abstract machine of TALK is based on an ordinary three operands RISC architecture (see Figure 2.7).

$$
\begin{array}{lllll}
(\mathit{state}) & S & ::= & (P, M, R, I) \\
(\mathit{prog.}) & P & ::= & \cdot \mid \{l \mapsto I\}P \\
(\mathit{memory}) & M & ::= & \cdot \mid \{n \mapsto a\}M \\
(\mathit{regs.}) & R & ::= & \{\mathtt{r1} \mapsto v_1, \ldots, \mathtt{rn} \mapsto v_n\} \\
(\mathit{register}) & r & ::= & \mathtt{r1} \mid \ldots \mid \mathtt{rn} \\
(\mathit{array}) & a & ::= & \langle t_1, \ldots, t_n \rangle \\
(\mathit{tuple}) & t & ::= & \langle v_1, \ldots, v_n \rangle \mid \mathtt{roll}\,(t) \\
 & & & \mid \mathtt{pack}_{[c_1,\ldots,c_n \mid M]}\,(t) \\
(\mathit{value}) & v & ::= & n \mid l\,[c_1, \ldots, c_n/\Delta] \\
(\mathit{integer}) & n & & \\
(\mathit{label}) & l & & \\
(\mathit{insts.}) & I & ::= & \mathtt{ld}\,[r_s + n], r_d; I \mid \mathtt{st}\; r_s, [r_d + n]; I \\
 & \multicolumn{4}{l}{\mid \mathtt{mov}\; r_s, r_d; I \mid \mathtt{movi}\; v, r_d; I \mid \mathtt{add}\; r_{s1}, r_{s2}, r_d; I} \\
 & \multicolumn{4}{l}{\mid \mathtt{sub}\; r_{s1}, r_{s2}, r_d; I \mid \mathtt{mul}\; r_{s1}, r_{s2}, r_d; I} \\
 & \multicolumn{4}{l}{\mid \mathtt{beq}\; r_{s1}, r_{s2}, r_d; I \mid \mathtt{ble}\; r_{s1}, r_{s2}, r_d; I} \\
 & \multicolumn{4}{l}{\mid \mathtt{jmp}\; r_d \mid \mathtt{apply}\; r\,[c_1, \ldots, c_n/\Delta]\,; I} \\
 & \multicolumn{4}{l}{\mid \mathtt{roll}_{\mu\eta[\Delta].\tau(c_1,\ldots,c_n)}\; i; I \mid \mathtt{unroll}\; i; I} \\
 & \multicolumn{4}{l}{\mid \mathtt{pack}_{[c_1,\ldots,c_n \mid \Sigma_1] as \exists\Delta.\mid C \mid [\Sigma_2].\tau}\; i; I \mid \mathtt{unpack}\; i\; \mathtt{with}\; \Delta; I} \\
 & \multicolumn{4}{l}{\mid \mathtt{split}\; i_1, i_2; I \mid \mathtt{concat}\; i_1, i_2, i_3; I} \\
 & \multicolumn{4}{l}{\mid \mathtt{tuple\_split}\; i_1, n_2; I \mid \mathtt{tuple\_concat}\; i_1, i_2; I}
\end{array}
$$

Figure 2.7: Syntax of abstract machine

A state S of the abstract machine consists of program P, memory M, registers R and instructions I. The instructions I in the state of the abstract machine represent an instruction sequence that is ready to execute on the abstract machine. As the first instruction of the instruction sequence is executed, the state of the abstract machine is updated according to the operational semantics of the instructions that are explained in Section 2.2.3.

The program P represents all the basic blocks (the instruction sequences) that exist in the abstract machine. Specifically, P is a map from label l to the instructions I. The label l represents the address of the basic block. The jump/branch instructions take the labels as the jump destination. For example, the following program contains three basic blocks at the label l_1, l_2 and l_3.

$$\{l_1 \mapsto \texttt{movi}\ l_2,\ \texttt{r3}; \texttt{jmp}\ \texttt{r3}\}$$
$$\{l_2 \mapsto \texttt{movi}\ l_3,\ \texttt{r3}; \texttt{jmp}\ \texttt{r3}\}$$
$$\{l_3 \mapsto \texttt{movi}\ l_1,\ \texttt{r3}; \texttt{jmp}\ \texttt{r3}\}$$

As shown in the above program, each basic block is surrounded by $\{$ and $\}$, and the operator $\mapsto$ indicates that the basic block represented by the instructions I resides in the address represented by the label l ($\{l \mapsto I\}$). Note that we do not consider the syntactic order between elements in a map. For example, we do not distinguish the program $\{l_1 \mapsto I_1\}\{l_2 \mapsto I_2\}$ and the program $\{l_2 \mapsto I_2\}\{l_1 \mapsto I_1\}$.

The registers R literally represents registers of the abstract machine. Specifically, the registers R is just a map from register r to value v. For example, the registers $\{\texttt{r1} \mapsto 123, \texttt{r2} \mapsto l\}$ indicates that the register `r1` holds the integer value 123, and the register `r2` holds the label l.

The memory M represents all the available memory in the abstract machine. Specifically, M is a map from an integer constant n, which represents an address of the memory, to an array a, which represents the memory region that resides at the address. For example, the following memory indicates that the array a_1 resides at the address `0x42`, and the array a_2 resides at the address `0x123`.

$$\{\texttt{0x42} \mapsto a_1\}\{\texttt{0x123} \mapsto a_2\}$$

As shown in the above memory, each memory region is surrounded by $\{$ and $\}$, and the operator $\mapsto$ indicates that the memory region represented by the array a resides in the address represented by the integer constant n ($\{n \mapsto a\}$).

The array a consists of tuples t, the tuple consists of values v, and the value v is the integer constant n or the label l. (Note that the tuples `roll` (t)

and $\texttt{pack}_{[c_1,\ldots,c_n|M]}\,(t)$ are introduced only for formal arguments of recursive types and existential types. In addition, the suffix of the label l, that is, the substitution $[c_1,\ldots,c_n/\Delta]$ is required only by the type checker.)

2.2.2 Types

The types of TALK are defined naturally corresponding to the abstract machine states (see Figure 2.8).

$$
\begin{array}{lcll}
(\textit{label type}) & lt & ::= & \forall\Delta.\,|C|\,[\Sigma]\,(\Gamma) \\
(\textit{small type}) & \sigma & ::= & i \mid lt \\
(\textit{integer type}) & i & ::= & n \mid \alpha \mid i_1\ aop\ i_2 \\
(\textit{type var.}) & \delta & ::= & \alpha, \epsilon \\
(\textit{type vars.}) & \Delta & ::= & \cdot \mid \delta, \Delta \\
(\textit{type}) & \tau & ::= & \langle\sigma_1,\ldots,\sigma_n\rangle \mid \exists\Delta.\,|C|\,[\Sigma]\,\tau \\
 & & & \mid \rho\,(c_1,\ldots,c_n) \\
(\textit{type scheme}) & \rho & ::= & \eta \mid \mu\eta\,[\Delta]\,.\tau \\
(\textit{array type}) & at & ::= & \tau\,[i] \mid \tau\ (\equiv \tau\,[1]) \\
(\textit{memory type}) & \Sigma & ::= & \cdot \mid \Sigma \otimes \{i \mapsto at\} \mid \Sigma \otimes \epsilon \\
(\textit{regs. type}) & \Gamma & ::= & \cdot \mid \{r \mapsto \sigma\}\Gamma \\
(\textit{prog. type}) & \Phi & ::= & \cdot \mid \{l \mapsto lt\}\Phi \\
(\textit{constructor}) & c & ::= & i \mid \Sigma \mid st \\
(\textit{constraints}) & C & ::= & \cdot \mid i_1\ cop\ i_2 \\
 & & & \mid C \wedge C \mid C \vee C \mid \neg C \\
(\textit{compareop.}) & cop & ::= & = \mid < \mid \leq \mid > \mid \geq \\
(\textit{arithop.}) & aop & ::= & + \mid - \mid *
\end{array}
$$

Figure 2.8: Syntax of types

The type of integers is represented by i. The integer type i is integer constants n, type variables α or the result of integer arithmetic operations $i_1\ aop\ i_2$. For example, if a certain register r has the integer type 3, the register r holds the value 3. In addition, if two registers r_1 and r_2 have the same type α, we know that $r_1 = r_2$, though the exact values of r_1 and r_2 is not known.

The type of memory is represented as Σ. The memory type Σ is represented as mappings from the integer type i to array type at, and type variables ϵ. For example, the following memory type indicates that there is

only one array of the type at at the address $\mathtt{0x}c0345810$, and there are no other memory regions in the memory.

$$\{\mathtt{0x}c0345810 \mapsto at\}$$

In addition, the following memory type indicates that there are three arrays of the type at_1, at_2 and at_3 at the addresses $\mathtt{0x}c0345810$, α and β, respectively, and there are no other memory regions in the memory.

$$\{\mathtt{0x}c0345810 \mapsto at_1\} \otimes \{\alpha \mapsto at_2\} \otimes \{\beta \mapsto at_3\}$$

In the above memory type, the operator $\otimes$ indicates that the three addresses specified in the memory type are different each other, that is, $\mathtt{0x}c0345810 \neq \alpha$, $\mathtt{0x}c0345810 \neq \beta$ and $\alpha \neq \beta$, if the sizes indicated by the array types are greater than 0. (Thus, strictly speaking, the memory type is not just a map, because it may map one address to several array types in order to support the arrays of size 0.)

Further, the following memory type indicates that there is only one array of the type at at the address $\mathtt{0x}c0345810$, and there may exist other memory regions in the memory.

$$\{\mathtt{0x}c0345810 \mapsto at\} \otimes \epsilon$$

In the above memory type, the type variable (memory type variable) ϵ indicates that there may exist other memory regions in the memory.

The array type at is written as $\tau\,[i]$. This represents an array whose elements have the type τ and whose size is i. Because we can use type variables for representing sizes of arrays, we can deal with arrays whose sizes is not known until runtime.

The type of elements of arrays is the tuple type τ. There are three kinds of the tuple type. $\langle \sigma_1, \ldots, \sigma_n \rangle$ represents a type of ordinary tuples whose elements have types σ_i. $\exists \Delta.\, |C|\, [\Sigma]\, .\tau$ represents the type of a tuple which is packed as an existential type. (The details of existential types are explained later.) $\rho\,(c_1, \ldots, c_n)$ is a (parametric) recursive type for recursive data structures. The type of elements of tuples, σ, can be the integer type i or label type lt.

The label type is written as $\forall \Delta.\, |C|\, [\Sigma]\, (\Gamma)$. It indicates a constraint condition that must be satisfied whenever a control flow reaches the label. First, Δ represents a set of type variables. This means that the instructions of the label are polymorphic over the type variables. Next, C represents integer constraints. The instructions of the label are type-checked

under the assumption that the constraints are satisfied, because the typing rules ensure that the constraints are satisfied at all the points of jumping to the label. Then, Σ is the memory type described above. As with the integer constraints, the instructions of the label are type-checked under the assumption that the memory has the memory type Σ, because the typing rules ensure that the memory has the type Σ at all the points of jumping to the label. Last, the registers type Γ indicates the condition for registers that must be satisfied whenever execution reaches the label. For example, the following label type

$$\forall\alpha, \beta, \epsilon.\, |\beta \leq 128|\, [\epsilon \otimes \{\alpha \mapsto \langle 0\rangle\, [\beta]\}]\, (\texttt{r1} : \alpha)$$

represents instructions that take a pointer (register `r1`) to an array whose size is not greater than 128 and whose elements are 0.

Additionally, the existential type $\exists\Delta.\, |C|\, [\Sigma]\, \tau$ represents tuples that have the type τ, and indicates that the integer constraints C are satisfied and there exists memory whose type is Σ. For example, the existential tuple type

$$\exists\alpha, \beta.\, |\beta \leq 128|\, [\{\alpha \mapsto \langle 0\rangle\, [\beta]\}]\, \langle\alpha\rangle$$

represents a tuple whose only element is a pointer to an array whose size is not greater than 128, and ensures that the array exists surely.

The program type Φ represents the type of program P. It is a map from the label l to the label type lt.

2.2.3 Instructions and operational semantics

This section describes the meaning of instructions of the TALK abstract machine. There are two kinds of instructions in TALK. One is the ordinary instructions that update the state of the abstract machine. The other is the coerce instructions that update only the type information when type-checking. Figures 2.9 and 2.10 represent their operational semantics. The operational semantics are defined as the relation ($\mapsto_S$) between the abstract machine states. Note that, in this thesis, $e\,[b/a]$ represents a capture-avoiding substitution of b for free variable a in e. In addition, $e\,[b_1, b_2/a_1, a_2]$ is an abbreviation of $e\,[b_1/a_1, b_2/a_2]$.

Ordinary instructions

There are ten ordinary instructions in TALK. $\texttt{ld}\ [r_s + n], r_d$ is a memory load instruction which loads nth element of a tuple which resides in the

address specified by the register r_s and stores the element to the register r_d. `st` $r_s, [r_d + n]$ is a memory store instruction which stores the value of the register r_s into nth element of a tuple which resides in the address specified by the register r_d.

`mov` r_s, r_d is a register-copy instruction which just copies the value of the register r_s to the register r_d. `movi` v, r_d is a constant-load instruction which loads the value v to the register r_d.

`add` r_{s1}, r_{s2}, r_d is an add instruction which stores the sum of r_{s1} and r_{s2} into the register r_d. `sub` and `mul` is a subtraction and multiplication instruction, respectively. In TALK, there is no reference types or pointer types. Memory addresses are only integers. Therefore, the address calculation are performed with these arithmetic instructions.

`beq` r_{s1}, r_{s2}, r_d is a branch instruction which jumps to the label specified by the register r_d if $r_{s1} = r_{s2}$. `ble` r_{s1}, r_{s2}, r_d is the other branch instruction which jumps to the label specified by the register r_d if $r_{s1} \leq r_{s2}$.

`jmp` r_d is a jump instruction which jumps to the label specified by the register r_d and executes the instructions of the label.

Coerce instructions

There are nine coerce instructions for manipulating type information when type-checking. The instructions incur no runtime overhead because they are interpreted only by type checkers and not executed at runtime.

`apply` $r\,[c_1, \ldots, c_n/\Delta]$ is a type application instruction which instantiates the type of the label before jumping to the instructions of the label, typically. Specifically, it substitutes $c_1, \ldots, c_n$ for the type variables Δ that are bound by the type of the label specified by the register r. A type variable (δ) is an integer type variable (α) or a memory type variable (ϵ).

$\texttt{roll}_{\mu\eta[\Delta].\tau(c_1,\ldots,c_n)}\ i$ and `unroll` i are instructions for recursive types which unroll a recursive type once (`unroll`) and vice versa (`roll`).

$\texttt{pack}_{[c_1,\ldots,c_n|\Sigma]as\ \tau}\ i$ and `unpack` i `with` Δ are instructions for existential types which pack the type of the tuple that resides in the address i into an existential type (`pack`) and vice versa (`unpack`). As in the alias type system [96], we can hide part of memory in existential types. The encapsulated memory cannot be accessed unless the existential type is unpacked.

`split` i_1, i_2 and `concat` i_1, i_2, i_3 are instructions for the arrays. As mentioned in Section 2.1.4, `split` splits an array into two adjacent arrays and `concat` concatenates two adjacent arrays into one array. These instructions are used to access an element of an array (see Section 2.2.4 for details). In addition, they are useful for implementing memory manage-

$$
\begin{array}{lcl}
(P, M\{R(r_s) \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\}, & \mapsto_S & (P, M\{R(r_s) \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\}, \\
\quad R, \texttt{ld}\,[r_s + n'], r_d; I) & & \quad R\{r_d \mapsto v_{n'}\}, I) \\
(P, M\{R(r_d) \mapsto \langle\langle \ldots, v_{n'}, \ldots\rangle\rangle\}, & \mapsto_S & (P, M\{R(r_d) \mapsto \langle\langle \ldots, R(r_s), \ldots\rangle\rangle\}, \\
\quad R, \texttt{st}\ r_s, [r_d + n']; I) & & \quad R, I) \\
(P, M, R, \texttt{mov}\ r_s, r_d; I) & \mapsto_S & (P, M, R\{r_d \mapsto R(r_s)\}, I) \\
(P, M, R, \texttt{movi}\ v, r_d; I) & \mapsto_S & (P, M, R\{r_d \mapsto v\}, I) \\
(P, M, R, \texttt{add}\ r_{s1}, r_{s2}, r_d; I) & \mapsto_S & (P, M, R\{r_d \mapsto R(r_{s2}) + R(r_{s1})\}, I) \\
(P, M, R, \texttt{sub}\ r_{s1}, r_{s2}, r_d; I) & \mapsto_S & (P, M, R\{r_d \mapsto R(r_{s2}) - R(r_{s1})\}, I) \\
(P, M, R, \texttt{mul}\ r_{s1}, r_{s2}, r_d; I) & \mapsto_S & (P, M, R\{r_d \mapsto R(r_{s2}) * R(r_{s1})\}, I) \\
(P, M, R, \texttt{beq}\ r_{s1}, r_{s2}, r_d; I) & \mapsto_S & \textit{if } R(r_{s1}) = R(r_{s2}) \textit{ then} \\
 & & \quad (P, M, R, P(l)\,[c_1, \ldots, c_n/\Delta])) \\
 & & \textit{else } (P, M, R, I) \\
 & & \textit{where } R(r_d) = l\,[c_1, \ldots, c_n/\Delta] \\
(P, M, R, \texttt{ble}\ r_{s1}, r_{s2}, r_d; I) & \mapsto_S & \textit{if } R(r_{s1}) \leq R(r_{s2}) \textit{ then} \\
 & & \quad (P, M, R, P(l)\,[c_1, \ldots, c_n/\Delta])) \\
 & & \textit{else } (P, M, R, I) \\
 & & \textit{where } R(r_d) = l\,[c_1, \ldots, c_n/\Delta] \\
(P, M, R, \texttt{jmp}\ r_d) & \mapsto_S & (P, M, R, P(l)\,[c_1, \ldots, c_n/\Delta])) \\
 & & \textit{where } R(r_d) = l\,[c_1, \ldots, c_n/\Delta]
\end{array}
$$

Figure 2.9: Operational semantics (instructions)

ment facilities (see Section 3.1 for details). The reason why the operational semantics of `split` and `concat` are complicated is that the type system treats the arrays of size 0.

`tuple_split` i_1, n_2 and `tuple_concat` i_1, i_2 resemble `split` and `concat`, but for tuple types. `tuple_concat` is used for creating a tuple type from adjacent arrays of size 1, and `tuple_split` is vice versa. In TALK, allocation of a tuple can be represented as follows. First, an array is obtained by `split` from one of the arrays that represent the free memory. Then, the obtained array is further split into adjacent arrays of size 1. Then the arrays are concatenated into a tuple by `tuple_concat` (see the examples of Sections 2.3 and 3.1 for details)

2.2.4 Typing rules

Typing rules are shown in Figures 2.11, 2.12 and 2.13. $\vdash S$ states that the abstract machine state S is well-formed. The well-formed abstract machine state causes no runtime error. More formally, the following theorem holds.

Theorem 2.1 (Type Soundness)
If $\vdash S$ and $S \mapsto_S^* S'$, then S' is not stuck.

Here the stuck state is defined as follows.

Definition 2.2 (Stuck State)
S is stuck if and only if there exists no abstract machine state S' such that $S \mapsto_S S'$.

The proof of the type soundness theorem is shown in Appendix A.

The judgement of abstract machine states consists of the judgement of program, memory, registers and instructions. Note that, in the following typing rules, we omit the program type Φ if not needed, because it is invariant while type-checking. In addition, the kind of type variables (α and ϵ) is not explicitly described in the typing rules. We implicitly assume that the type variable α is the integer type and ϵ is the memory type. Therefore, for example, the type substitutions $[3/\epsilon]$ and $[\{\texttt{0x1} \mapsto \langle 0 \rangle\}/\alpha]$ are rejected by the type checker, though not mentioned in the following typing rules.

Well-formedness of program

$\vdash P : \Phi$ states that the program P is well-formed (PROGRAM). The rule checks whether all the labels in the program are typed in the program type

$$
\begin{array}{lcl}
(P, M, R, \texttt{apply}\ r_d\,[c_1, \ldots, c_n/\Delta]\,; I) & \mapsto_S & (P, M, R\{r_d \mapsto R(r_d)\,[c_1, \ldots, c_n/\Delta]\}, I) \\
 & & \textit{where } R(r_d) = l\,[c'_1, \ldots, c'_m/\Delta'] \quad (l \in P) \\
(P, M\{n \mapsto \langle t \rangle\}, R, \texttt{roll}_\tau\ n; I) & \mapsto_S & (P, M\{n \mapsto \langle \texttt{roll}\,(t) \rangle\}, R, I) \\
(P, M\{n \mapsto \langle \texttt{roll}\,(t) \rangle\}, & & \\
\qquad R, \texttt{unroll}\ n; I) & \mapsto_S & (P, M\{n \mapsto \langle t \rangle\}, R, I) \\
(P, M\{n \mapsto \langle t \rangle\} M', & & \\
\qquad R, \texttt{pack}_{[c_1,\ldots,c_n|\Sigma] as\ \tau}\ n; I) & \mapsto_S & \left(P, M\{n \mapsto \left\langle \texttt{pack}_{[c_1,\ldots,c_n|M']}\,(t) \right\rangle\}, R, I\right) \\
 & & \textit{where } Dom(M') \subseteq Dom(\Sigma) \\
(P, M\{n \mapsto \left\langle \texttt{pack}_{[c_1,\ldots,c_n|M']}\,(t) \right\rangle\}, & & \\
\qquad R, \texttt{unpack}\ n\ \texttt{with}\ \Delta; I) & \mapsto_S & (P, MM'\{n \mapsto \langle t \rangle\}, R, [c1, \ldots, c_n/\Delta] I) \\
(P, M\{n_1 \mapsto \langle t_1, \ldots, t_n \rangle\}, & \mapsto_S & (P, M\{n_1 \mapsto \langle t_1, \ldots, t_{n_2} \rangle\} \\
\qquad R, \texttt{split}\ n_1, n_2; I) & & \qquad \{n'_1 \mapsto \langle t_{n_2+1}, \ldots, t_n \rangle\}, R, I) \\
 & & \textit{where } 0 < n_2 < n \textit{ and} \\
 & & \qquad n'_1 = n_1 + \sum_{i=1}^{n_2} \textit{sizeof}\ (t_i) \\
(P, M, R, \texttt{split}\ n_1, 0; I) & \mapsto_S & (P, M, R, I) \\
(P, M\{n_1 \mapsto \langle t_1, \ldots, t_n \rangle\}, & & \\
\qquad R, \texttt{split}\ n_1, n; I) & \mapsto_S & (P, M\{n_1 \mapsto \langle t_1, \ldots, t_n \rangle\}, R, I) \\
(P, M\{n_1 \mapsto \langle t_1, \ldots, t_n \rangle\} & & \\
\qquad \{n_2 \mapsto \langle t'_1, \ldots, t'_m \rangle\}, R, & & \\
\qquad \texttt{concat}\ n_1, n_2, m; I) & \mapsto_S & (P, M\{n_1 \mapsto \langle t_1, \ldots, t_n, t'_1, \ldots, t'_m \rangle\}, R; I) \\
 & & \textit{where } m > 0 \textit{ and } n_2 = n_1 + \sum_{i=1}^{n} \textit{sizeof}\ (t_i) \\
(P, M\{n_1 \mapsto \langle t_1, \ldots, t_n \rangle\}, & & \\
\qquad R, \texttt{concat}\ n_1, n_2, 0; I) & \mapsto_S & (P, M\{n_1 \mapsto \langle t_1, \ldots, t_n \rangle\}, R, I) \\
 & & \textit{where } n_2 = n_1 + \sum_{i=1}^{n} \textit{sizeof}\ (t_i) \\
(P, M\{n_1 \mapsto \langle t_1, \ldots, t_m \rangle\}, & & \\
\qquad R, \texttt{concat}\ n_1, n_1, m; I) & \mapsto_S & (P, M\{n_1 \mapsto \langle t_1, \ldots, t_m \rangle\}, R, I) \\
 & & \textit{where } m > 0 \\
(P, M, R, \texttt{concat}\ n, n, 0; I) & \mapsto_S & (P, M, R, I) \\
(P, M\{n_1 \mapsto \langle \langle v_1, \ldots, v_n \rangle \rangle\}, & \mapsto_S & (P, M\{n_1 \mapsto \langle \langle v_1, \ldots, v_{n_2} \rangle \rangle\} \\
\qquad R, \texttt{tuple\_split}\ n_1, n_2; I) & & \qquad \{n'_1 \mapsto \langle \langle v_{n_2+1}, \ldots, v_n \rangle \rangle\}, R, I) \\
 & & \textit{where } n'_1 = n_1 + n_2 \\
(P, M\{n_1 \mapsto \langle \langle v_1, \ldots, v_n \rangle \rangle\} & & \\
\qquad \{n_2 \mapsto \langle \langle v'_1, \ldots, v'_m \rangle \rangle\}, R, & & \\
\qquad \texttt{tuple\_concat}\ n_1, n_2; I) & \mapsto_S & (P, M\{n_1 \mapsto \langle \langle v_1, \ldots, v_n, v'_1, \ldots, v'_m \rangle \rangle\}, R; I) \\
 & & \textit{where } n_2 = n_1 + n
\end{array}
$$

Figure 2.10: Operational semantics (coerce)

Φ. It also checks whether each block of instructions in the program is well-formed according to its label type specified in Φ. The well-formedness of instructions are described in Section 2.2.4.

Well-formedness of registers

$\vdash R : \Gamma$ states that the registers R is well-formed (REGISTERS). The rule checks whether the value of each register has the small value type specified in the registers type Γ.

Well-formedness of memory

$\vdash M : \Sigma$ states that the memory M has the memory type Σ (MEMORY). The judgement rule checks whether all the arrays in the memory (including encapsulated memory regions inside existential packages) do not overlap each other (which is denoted as $\mathtt{GU}(M)$) in order to keep track of pointer aliases properly.

Before showing the formal definition of GU, we first show the definition of the global memory G.

Definition 2.3 (Global Memory)
$\mathtt{G}(M)$ is the multi-set of the memory regions defined as follows.

$$
\begin{aligned}
&\mathtt{G}(\{n_1 \mapsto a_1\} \ldots \{n_m \mapsto a_m\}) = \\
&\quad [n_1, n_1 + \mathit{sizeof}(a_1) - 1] \uplus \ldots \uplus [n_m, n_m + \mathit{sizeof}(a_m) - 1] \\
&\qquad \uplus \mathtt{G}(a_1) \uplus \ldots \uplus \mathtt{G}(a_m) \\
&\mathtt{G}(\langle t_1, \ldots, t_m \rangle) = \mathtt{G}(t_1) \uplus \ldots \uplus \mathtt{G}(t_m) \\
&\mathtt{G}(\mathtt{pack}_{[\ldots|M]}(t)) = \mathtt{G}(M) \\
&\mathtt{G}(\mathtt{roll}(t)) = \mathtt{G}(t) \\
&\mathtt{G}(\mathit{other}) = \emptyset
\end{aligned}
$$

Then, the formal definition of GU is as follows.

Definition 2.4 (Global Uniqueness)
$\mathtt{GU}(M)$ if and only if there are no duplicate memory regions in $G(M)$.

In addition, the judgement rule checks whether the domain of M is a subset of that of Σ. It also checks whether, for each address $n \in \mathit{Dom}(M)$, the array $M(n)$ has the array type $\Sigma(n)$. In addition, it checks whether the size of the array types $\Sigma(m)$ is equal to zero, for each $m \in \mathit{Dom}(\Sigma) \backslash \mathit{Dom}(M)$. For example, we have $\vdash \{\mathtt{0x12345679} \mapsto \langle\langle 0 \rangle\rangle\} : \{\mathtt{0x12345679} \mapsto \langle 0 \rangle\,[1]\} \otimes$

$\{\texttt{0x12345679} \mapsto \langle 1 \rangle\,[0]\}$. More specifically, this is checked by the equality rules of memory. The details are described in Section 2.2.5.

$\Delta; C \vdash a : at$ states that, with the type variables Δ and under the assumption that the integer constraints C are satisfied, the array a has the array type at. The typing rule ARRAY checks whether all the elements of the array have the same tuple type τ and the size of the array equals to the size specified in the array type. For example, $\Delta; C \vdash \langle t_1, t_2 \rangle : \tau\,[i]$ checks whether the tuples t_1 and t_2 have the type τ. It also checks whether $i = 2$ under the assumption C, using a constraint solver. We write this as $\Delta; C \models i = 2$. The formal definition of the relation $\models$ are defined as follows.

Definition 2.5 (Integer Constraints)
$\Delta; C \models C'$ if and only if C' is deduced from C, no matter how the integer variables in Δ are instantiated.

It is well-known that the problem of integer constraints solving is decidable if the constraints are linear. The only instruction that may introduce a non-linear constraint is the `mul` instruction.

$\Delta; C \vdash t : \tau$ states that the element t of an array has the type τ. There are three typing rules for tuples (TUPLE, TUPLEROLL and TUPLEPACK). The typing rule TUPLE checks whether each element (v_i) of an tuple has the type (σ_i) specified in the tuple type.

The typing rule TUPLEROLL check whether τ is a recursive type. Then, it recursively applies the typing rules for tuples in order to check the tuple inside t according to the tuple type obtained by unrolling the recursive type once.

The typing rule TUPLEPACK check whether τ is an existential type. In addition, it checks whether the encapsulated memory is well-formed according to the memory type specified in the existential type. It also verifies that the integer constraints specified in the existential type is satisfiable under the current assumption. Then, it recursively applies the typing rule for tuples in order to check the packed tuple.

$\Delta; C \vdash v : \sigma$ states that the value v has the type σ. There are two typing rules for integers (VALUEINTEGER) and labels (VALUELABEL). The VALUEINTEGER rule checks whether the integer n equals to the type i using a constraint solver ($\Delta; C \models n = i$). The VALUELABEL rule checks whether the type of the label l can be instantiated to the specified label type σ according to the substitution $[c_1, \ldots, c_n / \Delta'']$.

$$\frac{\vdash P : \Phi \qquad \vdash M : \Sigma \qquad \vdash R : \Gamma \qquad \cdot;\Gamma;\cdot;\Sigma \vdash I}{\vdash (P, M, R, I)} \quad \text{(STATE)}$$

$$\frac{\begin{array}{c} Dom(P) = Dom(\Phi) \\ \forall l \in Dom(P).\ \Delta;\Gamma;C;\Sigma \vdash P(l) \qquad \Phi(l) \equiv \forall\Delta.\,|C|\,[\Sigma]\,(\Gamma) \end{array}}{\vdash P : \Phi} \quad \text{(PROGRAM)}$$

$$\frac{\begin{array}{c} \texttt{GU}(M) \qquad M \equiv \{n_1 \mapsto a_1\} \ldots \{n_k \mapsto a_k\} \\ \cdot;\cdot \vdash \Sigma = \{n_1 \mapsto at_1\} \otimes \ldots \otimes \{n_k \mapsto at_k\} \\ \forall i.\ \cdot;\cdot \vdash a_i : at_i \end{array}}{\vdash M : \Sigma} \quad \text{(MEMORY)}$$

$$\frac{\forall r_i \in Dom(\Gamma).\ \cdot;\cdot \vdash R(r_i) : \Gamma(r_i)}{\vdash R : \Gamma} \quad \text{(REGISTERS)}$$

$$\frac{\Delta;C \vdash t_j : \tau \qquad \Delta;C \models n = i}{\Delta;C \vdash \langle t_1, \ldots, t_n \rangle : \tau\,[i]} \quad \text{(ARRAY)}$$

$$\frac{\Delta;C \vdash v_j : \sigma_j}{\Delta;C \vdash \langle v_1, \ldots, v_n \rangle : \langle \sigma_1, \ldots, \sigma_n \rangle} \quad \text{(TUPLE)}$$

$$\frac{\begin{array}{c} \tau \equiv \mu\eta\,[\Delta'].\tau'\,(c_1, \ldots, c_n) \\ \Delta;C \vdash t : \tau'\,[\mu\eta\,[\Delta'].\tau'/\eta]\,[c_1, \ldots, c_n/\Delta'] \end{array}}{\Delta;C \vdash \texttt{roll}\,(t) : \tau} \quad \text{(TUPLEROLL)}$$

$$\frac{\begin{array}{c} \Delta;C \vdash t : \tau'\,[c_1, \ldots, c_n/\Delta'] \qquad \tau \equiv \exists\Delta'.\,|C'|\,[\Sigma']\,\tau' \\ \vdash M : \Sigma'\,[c_1, \ldots, c_n/\Delta'] \qquad \Delta;C \models C'\,[c_1, \ldots, c_n/\Delta'] \end{array}}{\Delta;C \vdash \texttt{pack}_{[c_1,\ldots,c_n|M]}\,(t) : \tau} \quad \text{(TUPLEPACK)}$$

$$\frac{\Delta;C \models n = i}{\Delta;C \vdash n : i} \quad \text{(VALUEINTEGER)}$$

$$\frac{\begin{array}{c} \forall\Delta'.\,|C'|\,[\Sigma']\,(\Gamma') \equiv \Phi\,(l) \qquad \theta \equiv [c_1, \ldots, c_n/\Delta''] \\ C'' \equiv C'\theta \qquad \Sigma'' \equiv \Sigma'\theta \qquad \Gamma'' \equiv \Gamma'\theta \\ \Delta;C \vdash \sigma = \forall\Delta'\backslash\Delta''.\,|C''|\,[\Sigma'']\,(\Gamma'') \end{array}}{\Delta;C \vdash l\,[c_1, \ldots, c_n/\Delta''] : \sigma} \quad \text{(VALUELABEL)}$$

Figure 2.11: Typing rules (machine state)

Well-formedness of instructions

$\Delta; \Gamma; C; \Sigma \vdash I$ states that the instructions I is well-formed with the type variables Δ, with the registers that satisfies the registers type Γ and under the assumption that the integer constraints C are satisfied and the memory has the memory type Σ.

The typing rule LOAD is for type-checking the load instruction. First, the rule checks whether the value of the register r_s is a valid memory address in the memory type Σ and an array resides in the address. Then, it checks whether the size of the array equals to 1 and the size of the tuple that is only element of the array is larger than n. Finally, it checks the rest of instructions I under the new register type that is modified so that the register r_d has the type σ_n that represents the loaded value.

The typing rule STORE is for type-checking the store instruction. As with LOAD, it checks whether the value of the register r_d is a valid memory address in the memory type Σ and an array resides in the address. Then, it checks whether the size of the array that resides in the address equals to 1 and the size of the tuple that is only element of the array is larger than n. Finally, it checks the rest of instructions I under the modified memory type such that the nth element of the tuple that resides in the address is replaced with the type of r_s.

Note that LOAD and STORE only permit load/store operations for arrays whose size is 1. Therefore, to access an array whose size is greater than 1, it is required to clip out an array of size 1 from the array, with the `split` instruction. At first glance, this limitation seems to be pointless, but it is essential. For example, let us consider the type of an integer array. It can be represented as $\exists\alpha.\, \langle\alpha\rangle\, [\beta]$ (The integer constraints and the memory type are omitted). To load a value from the array, we must unpack one of its elements. However, it is difficult to express the type of the array whose all elements have the existential type, except for the one element. The same can be said for storing a value to the array (as mentioned in Section 2.1.4).

The equality of memory types ($\Delta; C \vdash \Sigma = \Sigma'$) is almost the same as the ordinary equality of maps. However, it takes into account the integer constraints between type variables. For example, $\alpha, \beta; \cdot \nvdash \{\alpha \mapsto \langle\alpha\rangle\} = \{\beta \mapsto \langle\beta\rangle\}$, but $\alpha, \beta; \alpha = \beta \vdash \{\alpha \mapsto \langle\alpha\rangle\} = \{\beta \mapsto \langle\beta\rangle\}$. In addition, arrays whose size is 0 can be ignored in the equality check. For example, $\alpha, \beta; \beta = 0 \vdash \{\alpha \mapsto \langle 0\rangle\, [\beta]\} = \cdot$. The details are explained in Section 2.2.5.

The typing rule MOVE does almost nothing but checks the rest of the instructions I with the modified registers type that indicates that the register r_d has the same type as the register r_s. The typing rule MOVEI type-checks

the constant-load instruction. First, it checks the type of the value to be loaded ($\Delta; C \vdash v : \sigma$). Then, it checks the following instructions I with the modified registers type that indicates that the register r_d has the type σ.

The typing rule ARITH type-checks the arithmetic instructions. The rule checks whether the operands have the integer types. Then, it type-checks the rest of instructions I with the modified register type that indicates that the register r_d has the result of the arithmetic operations.

The typing rule BRANCH is for type-checking the branch instructions. For the taken branch, it first checks whether the value of the register r_d has the label type. Then, it checks whether the condition specified in the label type is satisfied under the current context ($\Delta; C$) extended with the condition of the taken branch ($\Gamma\,(r_{s1})\,(=, \leq)\Gamma\,(r_{s2})$). The relation $\Delta; C \vdash \Gamma \leq \Gamma'$ means that the registers type Γ indicates a stronger condition than the registers type Γ'. For example, $\alpha; \cdot \vdash \{\mathtt{r1} \mapsto \alpha\} \leq \{\mathtt{r1} \mapsto \alpha\}$ and $\alpha; \cdot \vdash \{\mathtt{r1} \mapsto \alpha, \mathtt{r2} \mapsto 42\} \leq \{\mathtt{r1} \mapsto \alpha\}$, but $\alpha; \cdot \nvdash \{\mathtt{r1} \mapsto \alpha\} \leq \{\mathtt{r1} \mapsto \alpha, \mathtt{r2} \mapsto 42\}$. The details are explained in Section 2.2.6. For the non-taken branch, it checks the following instructions I under the extended context with $\Gamma\,(r_{s1})\,(\neq, >)\Gamma\,(r_{s2})$. Moreover, if C'' (for the taken branch) or the extended C (for the non-taken branch) contains a contradiction, the corresponding type-check could be omitted without breaking the type soundness, because the contradiction indicates that execution never reaches the branch.

The typing rule JUMP type-checks the jump instruction. The rule checks whether the value of the register r_d has the label type. Then, it checks whether the condition specified in the label type is satisfied under the current context.

Careful readers might notice that nonsense label types can be written in TALK. For example, the label type $\forall\alpha, \beta.|\alpha = \beta|[\{\alpha \mapsto \langle 0\rangle\} \otimes \{\beta \mapsto \langle 1\rangle\}].(\Gamma)$ is nonsense because the memory type indicates that the tuple at the address $\alpha(= \beta)$ has the integer value 0 and 1. Even if a block of instructions passes the type check of TALK according to the nonsense label type, it may raise a runtime error if it is executed. However, the nonsense label type does not break the type soundness of TALK because the block is never executed. For example, let us suppose that there exists a well-formed machine state (P, M, R, I), where the last instruction of I is the jump instruction and its target register (r_d) has a nonsense label type. From the JUMP typing rule, we know that the typing context $(\Delta; \Gamma; C; \Sigma)$ is also nonsense when type-checking the jump instruction. If I does not contain the branch instructions, it contradicts the well-formedness of the machine state because the initial typing context $(\cdot; \Gamma; \cdot; \Sigma)$ is valid (not nonsense) and the only typing rule that may generate a nonsense context from a valid context is the BRANCH

typing rule, more specifically, the non-taken branch of the rule. Therefore, there must exists at least one branch instruction which introduces a new integer constraint which conflicts with the typing context of the BRANCH rule. This means that no matter how we instantiate the type variables, the new constraint is never satisfied. That is, the branch is never taken at runtime. Thus, execution never reaches the jump instruction.

$$\frac{\Delta; C \vdash \Sigma = \Sigma' \otimes \{\Gamma\,(r_s) \mapsto \langle \ldots, \sigma_n, \ldots \rangle\} \quad \Delta; \Gamma\{r_d \mapsto \sigma_n\}; C; \Sigma \vdash I}{\Delta; \Gamma; C; \Sigma \vdash \texttt{ld}\ [r_s + n], r_d; I} \tag{LOAD}$$

$$\frac{\Delta; C \vdash \Sigma = \Sigma' \otimes \{\Gamma\,(r_d) \mapsto \langle \ldots, \sigma_n, \ldots \rangle\} \quad \Delta; \Gamma; C; \Sigma' \otimes \{\Gamma\,(r_d) \mapsto \langle \ldots, \Gamma\,(r_s), \ldots \rangle\} \vdash I}{\Delta; \Gamma; C; \Sigma \vdash \texttt{st}\ r_s, [r_d + n]; I} \tag{STORE}$$

$$\frac{\Delta; \Gamma\{r_d \mapsto \Gamma\,(r_s)\}; C; \Sigma \vdash I}{\Delta; \Gamma; C; \Sigma \vdash \texttt{mov}\ r_s, r_d; I} \tag{MOVE}$$

$$\frac{\Delta; C \vdash v : \sigma \qquad \Delta; \Gamma\{r_d \mapsto \sigma\}; C; \Sigma \vdash I}{\Delta; \Gamma; C; \Sigma \vdash \texttt{movi}\ v, r_d; I} \tag{MOVEI}$$

$$\frac{\Delta; \Gamma\{r_d \mapsto \Gamma\,(r_{s2})\,(+, -, *)\Gamma\,(r_{s1})\}; C; \Sigma \vdash I}{\Delta; \Gamma; C; \Sigma \vdash (\texttt{add}, \texttt{sub}, \texttt{mul})\ r_{s1}, r_{s2}, r_d; I} \tag{ARITH}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Gamma\,(r_d) = \forall.\,|C'|\,[\Sigma']\,(\Gamma') \\ C'' \equiv C \wedge \Gamma\,(r_{s1})\,(=, \leq)\Gamma\,(r_{s2}) \\ \Delta; C'' \models C' \qquad \Delta; C'' \vdash \Sigma = \Sigma' \qquad \Delta; C'' \vdash \Gamma \leq \Gamma' \\ \Delta; \Gamma; C \wedge \Gamma\,(r_{s1})\,(\neq, >)\Gamma\,(r_{s2})\,; \Sigma \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash (\texttt{beq}, \texttt{ble})\ r_{s1}, r_{s2}, r_d; I} \tag{BRANCH}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Gamma\,(r_d) = \forall.\,|C'|\,[\Sigma']\,(\Gamma') \\ \Delta; C \models C' \qquad \Delta; C \vdash \Sigma = \Sigma' \qquad \Delta; C \vdash \Gamma \leq \Gamma'\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{jmp}\ r_d} \tag{JUMP}$$

Figure 2.12: Typing rules (instructions)

The typing rule APPLY type-checks the type application instruction. The rule type-checks the rest of instructions I with the modified registers type that indicates that the register r has the instantiated type σ'_f.

Typing rules ROLL and UNROLL check whether the instructions for recursive types (`roll` and `unroll`) are well-formed. The rule ROLL checks

whether the type of the tuple at the address i can be rolled to the specified recursive type. Then, it checks the following instructions I with the new memory type modified so that the type is rolled. The rule UNROLL is vice versa. Typing rules PACK and UNPACK type-check the instructions for packing and unpacking the existential types (`pack` and `unpack`). The rule PACK first checks whether the tuple at the address i can be packed into the specified existential type. Next, it modifies the memory type so that a tuple is packed into an existential type, and removes the portion of the memory that is hidden into the existential type. Then, it checks the rest of the instructions I. The rule UNPACK is vice versa. The rules ROLL, UNROLL, PACK and UNPACK only allow arrays whose size is 1, as with LOAD and STORE.

Typing rules SPLIT and CONCAT check the well-formedness of the instructions for splitting/concatenating arrays (`split` and `concat`). The rule SPLIT checks whether the size (j_1) of the array to be split is greater than or equal to the required size ($\Delta; C \models i_2 \leq j_1$). Then, it splits the array into two arrays and extends the memory type with them. The rule CONCAT checks whether the given two arrays are adjacent ($\Delta; C \models j_1 = i_1 + \mathit{sizeof}\ (\tau) * i_2$). Then, it concatenates the two arrays into one and extends the memory type with it. Here $\mathit{sizeof}(\tau)$ is the size of the tuple represented by the type τ. If τ is a recursive type ($\mu \ldots \tau'$) or an existential type ($\exists \ldots \tau'$), $\mathit{sizeof}(\tau)$ is recursively applied to the inner tuple type τ'. Because $\mathit{sizeof}(\tau)$ is always a constant integer, rules SPLIT and CONCAT never generate non-linear constraints. Note that the `split` instruction can create the array of size 0 (if the second argument of the instruction is 0 or equals to the size of the array). This is because, without the array of size 0, special handling is required to access the first and the last element of the variable-length arrays. The arrays of size 0 do not affect the type soundness because they are never accessed and the equality check of the memory types absorbs them.

Typing rules TUPLESPLIT and TUPLECONCAT are almost the same as SPLIT and CONCAT, but they type-check the split and concatenation of tuples.

2.2.5 Equality of memory types

This section describes the formal definition of the equality checking rules of memory types. These rules are necessary because ordinary map equality is insufficient. For example, let us suppose two memory types $\{\alpha_1 \mapsto \tau\,[\beta_1]\}$ and $\{\alpha_2 \mapsto \tau\,[\beta_2]\}$. At first glance, they are not equal. However, they may be equal under some circumstances. For example, let us assume that $\alpha_1 =$

$$\frac{\begin{array}{c}\Gamma(r) \equiv \forall \Delta'.\, |C'|\, [\Sigma']\, (\Gamma') \\ \theta \equiv [c_1, \ldots, c_n/\Delta''] \quad C'' \equiv C'\theta \quad \Sigma'' \equiv \Sigma'\theta \quad \Gamma'' \equiv \Gamma'\theta \\ \sigma'_f \equiv \forall \Delta' \backslash \Delta''.\, |C''|\, [\Sigma'']\, (\Gamma'') \quad \Delta; \Gamma\{r \mapsto \sigma'_f\}; C; \Sigma \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{apply}\ r\, [c_1, \ldots, c_n/\Delta'']\, ; I} \text{ (APPLY)}$$

$$\frac{\begin{array}{c}\tau \equiv \mu\eta\, [\Delta'].\tau'\, (c_1, \ldots, c_n) \qquad \Delta; \Gamma; C; \Sigma' \otimes \{i \mapsto \tau\} \vdash I \\ \Delta; C \vdash \Sigma = \Sigma' \otimes \{i \mapsto \tau'\, [\mu\eta\, [\Delta'].\tau'/\eta]\, [c_1, \ldots, c_n/\Delta']\}\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{roll}_\tau\ i; I} \text{ (ROLL)}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Sigma = \Sigma' \otimes \{i \mapsto \mu\eta\, [\Delta'].\tau'\, (c_1, \ldots, c_n)\} \\ \Delta; \Gamma; C; \Sigma' \otimes \{i \mapsto \tau'\, [\mu\eta\, [\Delta'].\tau'/\eta]\, [c_1, \ldots, c_n/\Delta']\} \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{unroll}\ i; I} \text{ (UNROLL)}$$

$$\frac{\begin{array}{c}\theta \equiv [c_1, \ldots, c_n/\Delta'] \quad \Delta; C \vdash \Sigma = \Sigma'' \otimes \{i \mapsto \tau\theta\} \otimes \Sigma'\theta \\ \Delta; C \models C'\theta \qquad \Delta; \Gamma; C; \Sigma'' \otimes \{i \mapsto \exists \Delta'.\, |C'|\, [\Sigma']\, \tau\} \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{pack}_{[c_1, \ldots, c_n | \Sigma'[c_1, \ldots, c_n/\Delta']] as \exists \Delta'.|C'|[\Sigma']\tau}\ i; I} \text{ (PACK)}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Sigma = \Sigma'' \otimes \{i \mapsto \exists \Delta'.\, |C'|\, [\Sigma']\, \tau\} \quad \theta \equiv [\Delta''/\Delta'] \\ \Delta\Delta''; \Gamma; C \wedge C'\theta; \Sigma'' \otimes \{i \mapsto \tau\theta\} \otimes \Sigma'\theta \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{unpack}\ i\ \texttt{with}\ \Delta''; I} \text{ (UNPACK)}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \tau\, [j_1]\} \qquad \Delta; C \models 0 \leq i_2 \leq j_1 \\ k_1 \equiv i_1 + \mathit{sizeof}\ (\tau) * i_2 \qquad k_2 \equiv j_1 - i_2 \\ \Delta; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \tau\, [i_2]\} \otimes \{k_1 \mapsto \tau\, [k_2]\} \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{split}\ i_1, i_2; I} \text{ (SPLIT)}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \tau\, [i_2]\} \otimes \{j_1 \mapsto \tau\, [j_2]\} \\ \Delta; C \models j_1 = i_1 + \mathit{sizeof}\ (\tau) * i_2 \\ \Delta; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \tau\, [i_2 + j_2]\} \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{concat}\ i_1, j_1, j_2; I} \text{ (CONCAT)}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n \rangle\} \qquad \Delta; C \models 0 < n_2 < n \\ \Delta; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_{n_2} \rangle\} \otimes \{i_1 + n_2 \mapsto \langle \sigma_{n_2+1}, \ldots, \sigma_n \rangle\} \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{tuple\_split}\ i_1, n_2; I}$$

(TUPLESPLIT)

$$\frac{\begin{array}{c}\Delta; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n \rangle\} \otimes \{i_2 \mapsto \langle \sigma'_1, \ldots, \sigma'_m \rangle\} \\ \Delta; C \models i_2 = i_1 + n \qquad \Delta; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_m \rangle\} \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{tuple\_concat}\ i_1, i_2; I}$$

(TUPLECONCAT)

Figure 2.13: Typing rules (coerce instructions)

α_2 and $\beta_1 = \beta_2$. Then, the memory types should be considered to be equal. In addition, in order to handle the arrays of size 0, a special treatment is required in the type system.

The equality rules of memory types are shown in Figure 2.14. The rule EQMEMEMPTY states that the empty memory type is equal to itself.

The rule EQMEMLOC states that two memory types are equal if one mapping of one memory type is equal to another mapping of the other memory type, and the rest of the memory types are equal. The equality of the mappings are verified by checking the integers that represent the address are equal and the array types are equal. The equality rules of the array types are explained later.

The rule EQMEMVAR states that two memory types are equal if one memory type variable in one memory type is the same type variable as the other memory type, and the rest of the memory types are equal. That is, it checks the equality of the memory type variables almost syntactically. This is because the type system does not keep track of the equality constraints between memory type variables. Thus, this rule is simpler than the rule EQMEMLOC.

The rule EQMEMZEROARRAYL and EQMEMZEROARRAYR handle the arrays of size 0. These rules state that two memory types are equal even if the arrays of size 0 are ignored. The rule EQMEMZEROARRAYL ignores the arrays of size 0 in the memory type of the left side, and the rule EQMEMZEROARRAYR ignores the arrays of size 0 in the memory type of the right side. For example, $\alpha; \alpha = 0 \vdash \epsilon \otimes \{\texttt{0x12345789} \mapsto \tau\,[\alpha]\} = \epsilon$ by the rule EQMEMZEROARRAYL, and $\alpha; \alpha = 0 \vdash \epsilon = \epsilon \otimes \{\texttt{0x12345789} \mapsto \tau\,[\alpha]\}$ by the rule EQMEMZEROARRAYR.

$$\Delta; C \vdash \cdot = \cdot \qquad \text{(EQMEMEMPTY)}$$

$$\frac{\Delta; C \vdash \Sigma = \Sigma' \qquad \Delta; C \models i_1 = i_2 \qquad \Delta; C \vdash at_1 = at_2}{\Delta; C \vdash \Sigma \otimes \{i_1 \mapsto at_1\} = \Sigma' \otimes \{i_2 \mapsto at_2\}} \text{(EQMEMLOC)}$$

$$\frac{\Delta; C \vdash \Sigma = \Sigma'}{\Delta; C \vdash \Sigma \otimes \epsilon = \Sigma' \otimes \epsilon} \qquad \text{(EQMEMVAR)}$$

$$\frac{\Delta; C \vdash \Sigma = \Sigma' \qquad \Delta; C \models i_2 = 0}{\Delta; C \vdash \Sigma \otimes \{i_1 \mapsto \tau\,[i_2]\} = \Sigma'} \text{(EQMEMZEROARRAYL)}$$

$$\frac{\Delta; C \vdash \Sigma = \Sigma' \qquad \Delta; C \models i_2 = 0}{\Delta; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \tau\,[i_2]\}} \text{(EQMEMZEROARRAYR)}$$

Figure 2.14: Equality rules (memory types)

The equality rules for array types, tuples types and value types are shown in Figure 2.15. The rule EQARRAY states that two array types are equal if the types of the elements of the array types are equal and the size of the arrays are equal.

There are three equality rules for tuple types, EQTUPLE, EQREC and EQEX. The rule EQTUPLE checks the ordinary tuple types. It checks whether each element type of two tuples is equal. The rule EQREC checks the equality of the recursive types. It just checks the syntactic equality. The rule EQEX checks the equality of the existential types. First, it checks whether the type variables in the existential types are equal (modulo the alpha conversion). Next, it checks whether the integer constraints indicate the same conditions. Then, it checks whether the memory types packed in the existential types are equal under the extend assumption with the integer constraints. It also checks whether the tuple types in the existential types are equal under the same assumption. For example, we have $\cdot;\cdot \vdash \exists\alpha\beta.|\alpha = \beta|\,\langle\alpha\rangle = \exists\alpha\beta.|\alpha = \beta|\,\langle\beta\rangle$, though this may be useless in practice. The equality rules for integer constraints are described later.

The equality rules of values are EQINT and EQLABEL. The rule EQINT states that the two integer types are equal. It only checks whether the two integers are equal with a constraint solver. The rule EQLABEL states that the two label types are equal. First, it checks whether the type variables in the label types are equal (modulo the alpha conversion). Next, it checks whether the integer constraints indicate the same conditions. Then, it checks whether the memory types mentioned in the label types are equal under the extended assumption with the integer constraints. It also checks whether the registers types are equal under the same assumption. The equality rules for registers types are explained later.

The equality rules for integer constraints and registers types are shown in Figure 2.16. The rule EQCSTRT states the equality of integer constraints. It checks whether if one integer constraints are deduced from the other integer constraints and vice versa. For example, we have $\alpha, \beta; \cdot \vdash (\alpha = \beta) = (\beta = \alpha)$. In addition, we have $\alpha; \cdot \vdash (\alpha = \alpha) = (\cdot)$. We also have $\alpha, \beta; \alpha = \beta \vdash (\alpha = 42) = (\beta = 42)$.

The equality rules for registers types are EQREGSNULL and EQREGSREG. The rule EQREGSNULL simply states that the null registers type $(\cdot)$ is equal to itself. The rule EQREGSREG states that two registers types are equal if types of one register are equal, and the rest of the registers types are equal.

$$\frac{\Delta; C \vdash \tau = \tau' \qquad \Delta; C \models i = i'}{\Delta; C \vdash \tau\,[i] = \tau'\,[i']} \quad \text{(EqArray)}$$

$$\frac{\Delta; C \vdash \sigma_i = \sigma_i'}{\Delta; C \vdash \langle \sigma_1, \ldots, \sigma_n \rangle = \langle \sigma_1', \ldots, \sigma_n' \rangle} \quad \text{(EqTuple)}$$

$$\Delta; C \vdash \rho\,(c_1, \ldots, c_n) = \rho\,(c_1, \ldots, c_n) \quad \text{(EqRec)}$$

$$\frac{\begin{array}{c}\Delta\Delta'; C \vdash C_1 = C_2 \\ \Delta\Delta'; C \wedge C_1 \vdash \Sigma_1 = \Sigma_2 \qquad \Delta\Delta'; C \wedge C_1 \vdash \tau_1 = \tau_2\end{array}}{\Delta; C \vdash \exists \Delta'.\, |C_1|\,[\Sigma_1]\,\tau_1 = \exists \Delta'.\, |C_2|\,[\Sigma_2]\,\tau_2} \quad \text{(EqEx)}$$

$$\frac{\Delta; C \models i = i'}{\Delta; C \vdash i = i'} \quad \text{(EqInt)}$$

$$\frac{\begin{array}{c}\Delta\Delta'; C \vdash C_1 = C_2 \\ \Delta\Delta'; C \wedge C_1 \vdash \Sigma_1 = \Sigma_2 \qquad \Delta\Delta'; C \wedge C_1 \vdash \Gamma_1 = \Gamma_2\end{array}}{\Delta; C \vdash \forall \Delta'.\, |C_1|\,[\Sigma_1]\,(\Gamma_1) = \forall \Delta'.\, |C_2|\,[\Sigma_2]\,(\Gamma_2)} \quad \text{(EqLabel)}$$

Figure 2.15: Equality rules (types)

$$\frac{\Delta; C \wedge C_1 \models C_2 \qquad \Delta; C \wedge C_2 \models C_1}{\Delta; C \vdash C_1 = C_2} \quad \text{(EqCstrt)}$$

$$\Delta; C \vdash \cdot = \cdot \quad \text{(EqRegsNull)}$$

$$\frac{\Delta; C \vdash \Gamma = \Gamma' \qquad \Delta; C \vdash \sigma = \sigma'}{\Delta; C \vdash \Gamma\,\{r : \sigma\} = \Gamma'\,\{r : \sigma'\}} \quad \text{(EqRegsReg)}$$

Figure 2.16: Equality rules (miscellaneous)

2.2.6 Registers subtyping

This section explains the subtyping rules used in the typing rule BRANCH and JUMP. $\Delta; C \vdash \Gamma \leq \Gamma'$ indicates that the registers type Γ indicates a stronger condition than Γ'. That is, when type-checking a jump instruction, if the current registers type is Γ and the type of the jump target label specifies Γ', the jump instruction is safe because the precondition Γ' is satisfied.

Specifically, the rule SUBREGSNULL simply states that any registers type is a subtype of the null registers types $(\cdot)$. The rule SUBREGSREG states that one registers type is a subtype of another registers type, if the types of a register r specified in the two registers types are equal and the rest of them is in the subtyping relation. For example, we have $\alpha, \beta; \alpha = \beta \vdash \{r_1 : 42\}\{r_2 : \alpha\} \leq \{r_2 : \beta\}$.

$$\Delta; C \vdash \Gamma \leq \cdot \qquad \text{(SUBREGSNULL)}$$

$$\frac{\Delta; C \vdash \Gamma \leq \Gamma' \qquad \Delta; C \vdash \sigma = \sigma'}{\Delta; C \vdash \Gamma\,\{r : \sigma\} \leq \Gamma'\,\{r : \sigma'\}} \qquad \text{(SUBREGSREG)}$$

Figure 2.17: Subtyping rule (registers)

2.3 Example (an implementation of stacks)

This section describes an example implementation of one of the simplest memory management mechanism: memory stacks. Using a memory stack, memory regions can be allocated and deallocated in a LIFO manner. The memory stack implementation shown in this section grows from high memory addresses to low memory addresses. Another implementation that grow in the opposite direction is possible in a similar way.

In this section, we first show how empty memory stacks can be represented in TALK. Then, we describe how the operations for the memory stacks, that is, pop and push can be implemented in TALK.

Note that, in the following examples, the syntax of TALK is slightly extended for ease of understanding them. For example, the delimiter (;) between instructions are omitted. In addition, the label is explicitly denoted just before the instructions. Further, the arguments for the coerce instructions are sometimes omitted for brevity.

2.3.1 Empty stack

In TALK, the memory stack can be implemented with a variable-length array. The code in Figure 2.18 (note that the syntax is slightly modified for clarity) represents a program which takes an empty stack of size β in register `r4` as an argument. The array is fulfilled with junk values of the type $\exists\beta.\,\langle\beta\rangle$ (the integer constraints and the memory type is omitted) that represents an integer whose value is unknown. In addition, the number of the junk values, that is, the size of the stack is α_2. The register `r4` points to the end of the array ($\alpha_1 + \alpha_2 = \alpha_4$).

```
1 ∀ α1, α2, α4, ϵ. |α1 + α2 = α4|
2 [{α1 ↦ ∃ β. ⟨β⟩ [α2]} ⊗ ϵ]
3    (r4 : α4)
4 stack_example:
5         ...
```

Figure 2.18: Example of a stack implementation in TALK

2.3.2 Push

The code of Figure 2.19 is a program which takes an empty stack as an argument (register `r4`) and pushes an integer to the stack. First, the last one element of the stack is split (in line 5). This `split` is type-checked because the integer constraints of the label type indicate that the size of the stack is greater than one. Next, the element is unpacked in order to update its content (in line 6). Then, the stack pointer (`r4`) is shifted by one and the integer (`r1`) is stored into the element. After line 7, the register `r4` points to the rest of the stack whose size is $\alpha_3 - 1$.

The code of Figure 2.20 is a program which takes an empty stack as an argument (register `r4`) and pushes a tuple of size 2 to the stack.

First, the memory region for the tuple is allocated by splitting the stack (in line 5). This `split` is type-checked because the size of the stack is greater than 2, as specified in the integer constraints of the label type.

Next, we need to convert the allocated memory region to a tuple because it is an array. First, the array is split into the arrays of size 1 (in line 6). Then, the arrays are unpacked in order to update their contents (in line 7 and 8). This unpack is type-checked because the size of the arrays is 1,

```
1  ∀α1, α2, α3, α4, ϵ. |α3 > 1 ∧ α2 + α3 = α4|
2  [{α2 ↦ ∃β. ⟨β⟩ [α3]} ⊗ ϵ]
3     (r1 : α1, r4 : α4)
4  push_example_1:
5          split α2, (α3 - 1)
6          unpack α2 + α3 − 1
7          sub 1, r4, r4
8          st r1, [r4]
9
10         ...
```

Figure 2.19: Example of a stack implementation in TALK (push an integer)

```
1  ∀α1, α2, α3, α4, ϵ. |α3 > 2 ∧ α2 + α3 = α4|
2  [{α2 ↦ ∃β. ⟨β⟩ [α3]} ⊗ ϵ]
3     (r1 : α1, r4 : α4)
4  push_example_2:
5          split α2, (α3 - 2)
6          split (α2 + α3 − 2), 1
7          unpack (α2 + α3 − 2)
8          unpack (α2 + α3 − 2 + 1)
9          tuple_concat (α2 + α3 − 2) , (α2 + α3 − 2 + 1)
10         sub 2, r4, r4
11         st r1, [r4]
12         st r1, [r4 + 1]
13
14         ...
```

Figure 2.20: Example of a stack implementation in TALK (push a tuple)

that is, the arrays can be considered as tuples. Last, the unpacked tuples are concatenated into one tuple (in line 9). The arguments for the `unpack` instructions are omitted for brevity.

Then, the stack pointer (`r4`) is shifted by 2 (in line 10). Last, the allocated tuple is initialized by the value of the register `r1` (in line 11 and 12).

2.3.3 Pop

The code of Figure 2.21 is a program which takes a memory stack which contains one integer as an argument (the register `r4`) and pops the integer from the stack. First, the content of the element of the stack is stored in the register `r1` (in line 6). Next, the element is packed into an existential type $\exists\beta.\langle\beta\rangle$. The arguments for the `pack` instruction is omitted for the brevity. Then, the element is returned to the stack by concatenating the element to the stack (in line 7). This `concat` is type-checked because the array representing the stack and the element is adjacent from the integer constraints of the label type, and the type of the element is $\exists\beta.\langle\beta\rangle$ which is equal to the type of the elements of the array.

```
1  ∀ α2, α3, α4, β1, ϵ. |α2 + α3 = α4|
2  [{α2 ↦ ∃ β. ⟨β⟩ [α3]}⊗
3    {α4 ↦ ⟨β1⟩} ⊗ ϵ]
4     (r4 : α4)
5  pop_example_1:
6        ld [r4], r1
7        pack α4
8        concat α2, α4, 1
9        add 1, r4, r4
10
11       ...
```

Figure 2.21: Example of a stack implementation in TALK (pop an integer)

The code of Figure 2.22 is a program which takes a memory stack which contains one tuple of size 2 as an argument (the register `r4`) and pops the tuple from the stack. First, the contents of the tuple is stored in the register `r1` and `r2` (in line 6 and 7). Then, we need to deallocated the tuple and return it to the stack. To this end, we first split the tuple into the arrays of

size 1 (in line 8). Next, the arrays are packed into the existential type $\exists\beta.\,\langle\beta\rangle$ (in line 9 and 10). Then, the arrays are concatenated back to the stack (in line 10 and 11). These `concat` instructions are type-checked because all the arrays are adjacent from the integer constraints of the label type and the types are equal ($\exists\beta.\,\langle\beta\rangle$).

```
1  ∀α2, α3, α4, β1, β2, ϵ. |α2 + α3 = α4|
2  [{α2 ↦ ∃β. ⟨β⟩ [α3]}⊗
3    {α4 ↦ ⟨β1, β2⟩} ⊗ ϵ]
4     (r4 : α4)
5  pop_example_2:
6        ld [r4], r1
7        ld [r4 + 1], r2
8        tuple_split α5, 1
9        pack α5
10       pack α5 + 1
11       concat α5, α5 + 1, 1
12       concat α2, α5, 2
13       add 2, r4, r4
14
15       ...
```

Figure 2.22: Example of a stack implementation in TALK (pop a tuple)

2.3.4 Bound checking

In the programs of Section 2.3.2, the integer constraints of the label type assumes that the stacks have sufficient room for storing values. In practice, however, this is not necessarily the case. Thus, we need to check bound of stacks at runtime as the code of Figure 2.23.

The program in Figure 2.23 takes a memory stack as an argument (the register `r4`). In addition, it also takes the size of the stack as another argument (the register `r3`). Then, it checks whether the size of the stack is greater or equal to one (in line 5). Here we assume that the label `stack_overflow` is defined elsewhere. That is, if the stack has no room for storing an integer, it jumps to the error handler of the label `stack_overflow`.

```
1  ∀ α1, α2, α3, α4, ϵ. |α2 + α3 = α4|
2  [{α2 ↦ ∃ β. ⟨β⟩ [α3]} ⊗ ϵ]
3     (r1 : α1, r3 : α3, r4 : α4)
4  push_example_3:
5          ble r3, 1, stack_overflow
6          split α2, (α3 - 1)
7          unpack α2 + α3 − 1
8          sub 1, r4, r4
9          st r1, [r4]
10
11         ...
```

Figure 2.23: Example of bound checking for a memory stack

After the `ble` instruction, the type checker assumes that $\texttt{r3} \geq 1$, according to the typing rule BRANCH. Thus, the `split` instruction (in line 6) is type-checked because the premises of the typing rule SPLIT are satisfied.

2.4 Stack extension

The previous section describes how to implement memory stacks in TALK. Based on the idea, this section extends TALK with the memory stacks as a language primitive. The primary purpose of this extension is to make the examples shown later in this thesis simpler for ease of understanding. To this end, the extension shown in this section assumes that the stacks are unbounded. That is, the stacks can grow infinitely. Strictly speaking, this assumption is not sound because, in practice, if there are two stacks and they grow infinitely, they must conflict at some point. Therefore, the type system explained in this section is sound only if the stacks do not conflict. In this section, we give preference to understandability over soundness. (It is easy to further extend the extension with the bounded stacks, as expected from the previous section.)

2.4.1 Abstract machine

First, the abstract machine is extended as shown in Figure 2.24. The memory of the abstract machine now holds not only arrays but also stacks. The

stack (s) consists of a null stack ($\cdot$) and a pair of the value (v) and the stack (s). In addition, two instructions for manipulating the stacks (`push` and `pop`) are added. Their operational semantics are explained later.

$$
\begin{array}{lrll}
(memory) & M & ::= & \cdot \mid \{n \mapsto h\}M \\
(heap) & h & ::= & a \mid s \\
(stack) & s & ::= & \cdot \mid v :: s \\
(insts.) & I & ::= & \\
 & & \mid & \texttt{push}\ r_s, [r_d]; I \\
 & & \mid & \texttt{pop}\ [r_s], r_d; I
\end{array}
$$

Figure 2.24: Extension of abstract machine syntax

2.4.2 Types

The types are also extended according to the extension of the abstract machine, as shown in Figure 2.25. The type of the stacks is represented as the type of the null stack ($\cdot$), the pair type $\sigma :: st$ that represents the stack whose top elements has the type σ and the rest of the stack is st, and the stack type variable γ that represents stacks whose elements are unknown.

$$
\begin{array}{lrll}
(type\ var.) & \delta & ::= & \alpha, \gamma, \epsilon \\
(stack\ type) & st & ::= & \cdot \mid \sigma :: st \mid \gamma \\
(heap\ type) & ht & ::= & at \mid st \\
(memory\ type) & \Sigma & ::= & \cdot \mid \Sigma \otimes \{i \mapsto ht\} \mid \Sigma \otimes \epsilon \\
(constructor) & c & ::= & i \mid \Sigma \mid st
\end{array}
$$

Figure 2.25: Extension of types syntax

2.4.3 Instructions and operational semantics

To manipulate the stacks, two instructions (`push` and `pop`) are added to the abstract machine. $\texttt{push}\ r_s, [r_d]$ pushes the value stored in the register r_s to the stack that resides in the address specified by the register r_d. Then,

it decrements the register r_d by one. $\texttt{pop}\ [r_s], r_d$ pops the top of the stack that resides in the address specified by the register r_s and stores the value to the register r_d. Then, it increments the register r_s by one.

$$
\begin{array}{l}
(P, M\{R(r_d) \mapsto s\}, R, \texttt{push}\ r_s, [r_d]; I) \quad \mapsto_S \\
\qquad (P, M\{R(r_d) - 1 \mapsto R(r_s) :: s\}, R\{r_d \mapsto R(r_d) - 1\}, I) \\
(P, M\{R(r_s) \mapsto v :: s\}, R, \texttt{pop}\ [r_s], r_d; I) \quad \mapsto_S \\
\qquad (P, M\{R(r_s) + 1 \mapsto s\}, R\{r_d \mapsto v\}\{r_s \mapsto R(r_s) + 1\}, I)
\end{array}
$$

Figure 2.26: Extension of operational semantics (instructions)

2.4.4 Typing rules

According to the extension of the abstract machine and the operational semantics, the type system is also extended. As mentioned above, the type system is sound only if the stacks do not overlap.

Well-formedness of memory

$\Delta; C \vdash s : st$ states that, with the type variables Δ and under the assumption that the integer constraints C are satisfied, the stack s has the stack type st.

$$
\frac{\Delta; C \vdash v_j : \sigma_j}{\Delta; C \vdash v_1 :: \ldots :: v_n : \sigma_1 :: \ldots :: \sigma_n} \qquad \text{(STACK)}
$$

Figure 2.27: Extension of typing rules (machine state)

Well-formedness of instructions

The typing rule PUSH type-checks the `push` instruction. First, it checks whether if the value of the register r_d is a valid memory address in the memory type Σ and there is a stack at the address. Next, it extends the type of the stack by pushing the type of the register r_s. It also modifies the address of the stack and the type of the register r_d. Then, it type-checks the following instructions I.

The typing rule POP type-checks the `pop` instruction. First, it checks whether if the value of the register r_s is a valid memory address in the memory type Σ and there is a stack at the address. Next, it pops out the top of the stack and overwrites the type of the register r_d with it. It also modifies the address of the stack and the type of the register r_s. Then, it type-checks the rest of the instructions I.

$$\frac{\begin{array}{c}\Delta; C \vdash \Sigma = \Sigma' \otimes \{\Gamma\,(r_d) \mapsto st\} \\ \Sigma'' \equiv \Sigma' \otimes \{\Gamma\,(r_d) - 1 \mapsto \Gamma\,(r_s) :: st\} \\ \Delta; \Gamma\{r_d \mapsto \Gamma\,(r_d) - 1\}; C; \Sigma'' \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{push}\ r_s, [r_d]; I} \quad \text{(PUSH)}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Sigma = \Sigma' \otimes \{\Gamma\,(r_s) \mapsto \sigma :: st\} \\ \Sigma'' \equiv \Sigma' \otimes \{\Gamma\,(r_s) + 1 \mapsto st\} \\ \Delta; \Gamma\{r_s \mapsto \Gamma\,(r_s) + 1\}\{r_d \mapsto \sigma\}; C; \Sigma'' \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{pop}\ [r_s], r_d; I} \quad \text{(POP)}$$

Figure 2.28: Extension of typing rules (instructions)

Equality of stack types

The equality rules are naturally extended for the stack types. The rule EQSTACKEMPTY states that the empty stack type is equal to itself. The rule EQSTACKVAR states that the stack variables are equal if and only if they are syntactically equal. The rule EQSTACKCONS states that two stack types are equal if the types of their top elements are equal and the types of the rest of the stacks are equal.

$$\Delta; C \vdash \cdot = \cdot \quad \text{(EQSTACKEMPTY)}$$

$$\Delta; C \vdash \gamma = \gamma \quad \text{(EQSTACKVAR)}$$

$$\frac{\Delta; C \vdash \sigma = \sigma' \qquad \Delta; C \vdash st = st'}{\Delta; C \vdash \sigma :: st = \sigma' :: st'} \quad \text{(EQSTACKCONS)}$$

Figure 2.29: Equality rules (stack types)

2.5 Examples of stack types

This section shows an example of the usage of the stacks defined in the previous section. More concretely, we first explain how to implement function calls in TALK. Then, we show very simple memory management and multi-thread management code written in TALK. More practical and complex versions are introduced in Chapter 3.

2.5.1 Function calls

When a function is called, the return address is passed to the function by the caller. Therefore, if several function calls are nested, we need to store their return addresses in memory, because the number of registers is fixed. Typically, the return addresses are stored in a memory stack.

```
1  ∀α4, ϵ, γ. | · |[{α4 ↦ γ} ⊗ ϵ]
2    (r4 : α4, r3 : ∀α.| · |[{α ↦ γ} ⊗ ϵ](r4 : α))
3  rec_fun:
4        push r3, [r4]
5        movi rec_fun_ret, r3
6        apply r3 [ϵ, γ/ϵ, γ]
7        movi rec_fun_ret, r1
8        apply r1 [α4 − 1, ϵ, ∀α.| · |[{α ↦ γ} ⊗ ϵ](r4 : α) :: γ/α4, ϵ, γ]
9        jmp r1
10 ∀α4, ϵ, γ. | · |
11   [{α4 ↦ ∀α.| · |[{α ↦ γ} ⊗ ϵ](r4 : α) :: γ} ⊗ ϵ]
12   (r4 : α4)
13 rec_fun_ret:
14       pop [r4], r3
15       apply r3 [α4 + 1/α]
16       jmp r3
```

Figure 2.30: Example of a recursive function

The code of Figure 2.30 is a function which recursively calls itself infinitely. The function consists of two block of instructions. The first block calls itself recursively. The second block is executed after the return from

the recursive function call. Careful readers might notice that the second block is never executed in this case, but it does not affect correctness of the implementation.

The first block takes a memory stack as an argument with the register `r4` and the return address with the register `r3` (in line 2). The contents of the stack is unknown because the type is γ (in line 1). The code first saves the return address to the stack (in line 4). The type of the register `r4` becomes $\alpha_4 - 1$ and the type of the memory becomes

$$\{(\alpha_4 - 1) \mapsto \forall\alpha.|\cdot|[\{\alpha \mapsto \gamma\} \otimes \epsilon](\texttt{r4} : \alpha) :: \gamma\} \otimes \epsilon.$$

Next, the code sets new return address (`rec_fun_ret`) in the register `r3` (in line 5 and 6). The type of the register `r3` becomes

$$\forall\alpha_4.|\cdot|[\{\alpha_4 \mapsto \forall\alpha.|\cdot|[\{\alpha \mapsto \gamma\} \otimes \epsilon](\texttt{r4} : \alpha) :: \gamma\} \otimes \epsilon](\texttt{r4} : \alpha_4).$$

Then, the code prepares for the recursive call by setting itself to the register `r1` (in line 7 and 8). The type of the register `r1` becomes

$$\begin{aligned}&\forall.|\cdot|[\{\alpha_4 - 1 \mapsto \forall\alpha.|\cdot|[\{\alpha \mapsto \gamma\} \otimes \epsilon](\texttt{r4} : \alpha) :: \gamma\} \otimes \epsilon]\\ &(\texttt{r4} : \alpha_4 - 1,\\ &\ \texttt{r3} : \forall\alpha.|\cdot|[\{\alpha \mapsto \forall\alpha.|\cdot|[\{\alpha \mapsto \gamma\} \otimes \epsilon](\texttt{r4} : \alpha) :: \gamma\} \otimes \epsilon](\texttt{r4} : \alpha)).\end{aligned}$$

Last, the code jumps to itself (in line 9). The `jmp` instruction passes the type check of TALK because the current memory type and the types of the registers (`r3` and `r4`) satisfy the required precondition specified in the label type of the register `r1`.

After the return from the recursive function call (in line 13), the second block is executed. From the label type of the second block (from line 10 to 12), the code assumes that the register `r4` points to a memory stack whose top element is a return address of the function. The code first pops out the return address and prepares for the return (in line 14 and 15). The type of the register `r4` becomes $\alpha_4 + 1$, the type of the register `r3` becomes

$$\forall.|\cdot|[\{\alpha_4 + 1 \mapsto \gamma\} \otimes \epsilon](\texttt{r4} : \alpha_4 + 1),$$

and the type of the memory becomes

$$\{\alpha_4 + 1 \mapsto \gamma\} \otimes \epsilon.$$

Then, the code return to the return address (in line 16). The `jmp` instruction passes the type check of TALK because the current memory type and the type of the register `r4` satisfy the required precondition specified in the label type of the register `r3`.

2.5.2 Simple memory management

In Section 2.3, we described how to implement memory stacks in TALK. In this section, we show how to implement a more generic memory management mechanism (i.e., malloc/free) in TALK. More specifically, we describe very simple malloc/free code written in TALK.

From the viewpoint of type theory, memory is only a set of memory regions and the memory regions are no more than arrays. In addition, managing the memory is to change the types of the memory regions. Thus, the memory management code is easily implemented in TALK, because the type system of TALK supports the variable-length arrays and the strong updates of the type of the memory region, as described in Section 2.2.

The memory allocation and deallocation algorithms shown below is not so efficient, but they are sufficient to show the flexibility and expressiveness of TALK.

Type of the free memory

Figure 2.31 represents the type of the free memory. It is a list of variable-length arrays. Each element of the list is a tuple which has three elements. First element holds a pointer to an array. The size of the array is stored in second element of the tuple. Third element is a pointer to the next element. The memory type inside the existential type (in line 2 and 3) indicates that there exists a memory region which satisfies the memory type. The type system of TALK ensures that all the memory regions encapsulated in existential types are distinct each other. For example, memory type $\{\alpha_1 \mapsto \exists\, \beta_1.[\{\beta_1 \mapsto \gamma\}]\, \langle\beta_1\rangle\} \otimes \{\alpha_2 \mapsto \exists\, \beta_2.[\{\beta_2 \mapsto \gamma\}]\, \langle\beta_2\rangle\}$ indicates that α_1, α_2, β_1 and β_2 are different addresses. Strictly speaking, the definition of the list in Figure 2.31 represents an infinite list because the definition does not includes any list terminator. Therefore, it might be unrealistic because the free memory is finite. Section 2.6 explains how to extend the type system of TALK in order to support variant types (or union types).

$$
\begin{array}{ll}
1 & \mathit{FreeMem1} \equiv \\
2 & \quad \mu\eta.\exists\, \alpha_{mem}, \alpha_{size}, \alpha_{next}. \mid \cdot \mid [\{\alpha_{mem} \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha_{size}]\} \otimes \{\alpha_{next} \mapsto \eta\}] \\
3 & \quad\quad \langle \alpha_{mem}, \alpha_{size}, \alpha_{next} \rangle
\end{array}
$$

Figure 2.31: Type of the free memory (list of variable-length arrays)

Simple implementation of malloc

Figures 2.32 and 2.33 show a simple implementation of a memory allocator, `malloc1`. For clarity, the syntax of instructions are slightly extended. In addition, the arguments for the instructions are sometimes omitted. Further, the `apply` instruction is omitted because it is straightforward.

```
1  ∀ α_size, α_free, α_stk, γ, ϵ. | · | [{α_free ↦ FreeMem1} ⊗ {α_stk ↦ γ} ⊗ ϵ]
2    (r1 : α_size, r2 : α_free, r3 : ret_t, r4 : α_stk)
3  malloc1:
4        unroll α_free
5        unpack α_free
6        ld [r2 + 1], r5
7        ble r1, r5, malloc1_success
8        push r2, [r4]
9        push r3, [r4]
10       ld [r2 + 2], r2
11       movi malloc1_cont, r3
12       jmp malloc1

13 ∀ α_size, α_tag, α_stk, α_mem, α'_size, α_junk, α, α_free, γ, ϵ. | · |
14   [{α_tag ↦ ⟨α_mem, α'_size, α_junk⟩} ⊗ {α_mem ↦ ∃ β. ⟨β⟩ [α'_size]} ⊗
15     {α ↦ ∃ β. ⟨β⟩ [α_size]} ⊗ {α_free ↦ FreeMem1}
16     {α_stk − 2 ↦ ret_t :: α_tag :: γ} ⊗ ϵ] (r1 : α, r2 : α_free, r4 : α_stk − 2)
17 malloc_cont:
18       pop [r4], r3
19       pop [r4], r5
20       st r2, [r5 + 2]
21       mov r5, r2
22       pack α_tag
23       roll α_tag
24       jmp r3
```

Figure 2.32: Simple malloc implementation in TALK (1/2)

The code recursively traverses the free memory list until it finds the array (memory region) whose size is greater than or equal to the requested memory size. Then, it splits the array to the array of the requested memory

size and the rest, and returns the allocated array and links the rest to the free memory list.

The label type of `malloc1` indicates that the function takes a free memory (*FreeMem1* in line 1 of Figure 2.32) as an argument and returns an array of the specified size (α_{size} in line 2). The type of the allocated array is specified in line 17 of Figure 2.33. Note that the return type of the function is abbreviated as ret_t.

The function first unrolls and unpacks the packed free memory list in order to access its contents. Then, it checks whether the array of first element of the given free memory list satisfies the requested size (in line 6 and 7 of Figure 2.32). If so, the function jumps to `malloc1_success`. Otherwise, it tries the next element in the free memory list. First, it stores the current element of the list and the return address on the stack (in line 8 and 9). Then, it calls itself recursively (in line 11 and 12). After the return from the recursive call (label `malloc1_cont`), the function concatenates the saved element with the returned free memory list (in line 19 and 20) and returns it as the new free memory list (from line 21 to 24). Of course, the array allocated by the recursive call is also returned. Here the stack type $ret\_t :: \alpha_{tag} :: \gamma$ in the memory type (in line 16) represents a stack whose top element has the type ret_t and the next element has the type α_{tag} and the rest is unknown (γ).

The code of `malloc1_success` first splits the array of the first element of the given free memory list into the array of the requested size and the rest (in line 6 of Figure 2.33). The `split` instruction passes the type check of TALK because the type checker knows that the length of the array is greater (or equal) than the requested size from the label type of `malloc1_success` (in line 1). Then, it stores the information about the unused array and its size in the first element (from line 7 to 12) and returns the allocated array (from line 13 to 16).

Simple implementation of free

Figure 2.34 is a simple implementation of free. It is a bit peculiar because the function takes not only an array to be freed, but also a tuple of three elements which contains the information about the array (in line 2). This is because we made the algorithm as simple as possible for ease of understanding. The code simply concatenates the given tuple to the given free memory list along with the given array. The label type of `free1` indicates that the free array cannot be used any more because the array is deleted from the memory type after the function return (in line 5).

```
1  ∀ α_size, α_tag, α_stk, α_mem, α'_size, α_free, γ, ϵ. |α_size ≤ α'_size|
2  [{α_tag ↦ ⟨α_mem, α'_size, α_free⟩} ⊗ {α_mem ↦ ∃ β. ⟨β⟩ [α'_size]}⊗
3    {α_free ↦ FreeMem1} ⊗ {α_stk ↦ γ} ⊗ ϵ]
4     (r1 : α_size, r2 : α_tag, r3 : ret_t, r4 : α_stk)
5  malloc1_success:
6         split α_mem, α_size
7         ld [r2 + 1], r5
8         sub r1, r5, r6
9         st r6, [r2 + 1]
10        ld [r2], r5
11        add r1, r5, r6
12        st r6, [r2]
13        mov r5, r1
14        pack α_tag
15        roll α_tag
16        jmp r3
17 ret_t ≡ ∀ α, β. | · |[{α ↦ ∃β. ⟨β⟩ [α_size]}⊗
18        {β ↦ FreeMem1} ⊗ {α_stk ↦ γ} ⊗ ϵ] (r1 : α, r2 : β, r4 : α_stk)
```

Figure 2.33: Simple malloc implementation in TALK (2/2)

```
1  ∀ α_tag, α_free, α_mem, α_size, α_junk, ϵ. | · |
2  [{α_tag ↦ ⟨α_mem, α_size, α_junk⟩ ⊗ {α_mem ↦ ∃ β. ⟨β⟩ [α_size]}⊗
3    {α_free ↦ FreeMem1} ⊗ ϵ]
4     (r1 : α_tag, r2 : α_free,
5      r3 : ∀ α.| · |[{α ↦ FreeMem1} ⊗ ϵ](r1 : α))
6  free1:
7         st r2, [r1 + 2]
8         pack α_tag
9         roll α_tag
10        jmp r3
```

Figure 2.34: Simple free implementation in TALK

2.5.3 Simple multi-thread management

In this section, we describe how to implement a multi-thread management mechanism in TALK on single CPU machines (SMP is out of the scope of this thesis). At first glance, this seems to be impossible because TALK does not take into account multi-thread execution environments as described in Section 2.2, unlike other strictly typed languages that support multi-thread mechanisms as language primitives, such as Java [60] and C# [59]. However, it is possible because, on single CPU machines, threads are emulated by one single-thread of execution. For example, typical OS kernels and threading libraries implement threads as the pairs of the program counter and the memory stack. Then, switching threads can be implemented as follows. First, they store the program counter and the stack pointer of the running thread. Then, the address of the next thread stack is set to the stack pointer and they jump to the program counter of the next thread.

The scheme of multi-threading mechanism shown in this section follows the above approach. When a running thread wants to stop and context switch to another thread, the running thread calls the context switching function. The function takes the return address (label) of the thread, that is, the program counter of the thread. Next, it stores the return label on the memory stack of the running thread, and switches from the stack of the running thread to that of the thread to be run. Then, it loads the new return address, that is, the program counter of the new thread from the top of the stack. Last, it jumps to the loaded new return address. Thus, the context switch is completed. The contents of the registers of the threads can be saved before calling the context switch function, and restored after returning from the function.

Figure 2.35 shows a very simple context-switching function written in TALK. The type *thd_t* represents the type of thread contexts (in line 13). It is a tuple which contains only one element which holds a pointer to a stack. As described in the memory type of *thd_t*, the top element of the stack must be a program counter whose type is *pc_t* (in line 14). The rest of the stack is unknown (γ), but it satisfies the precondition of the program counter. (Note that integer constraints are omitted in this section, because they do not matter.)

As specified in the label type of the function (from line 1 to 3), the function takes the thread context of the thread to be run as an argument. The function switches execution context from the running thread to the thread specified by the argument. In addition, the running thread needs to give the return address (label) to the function. This is accomplished by pushing

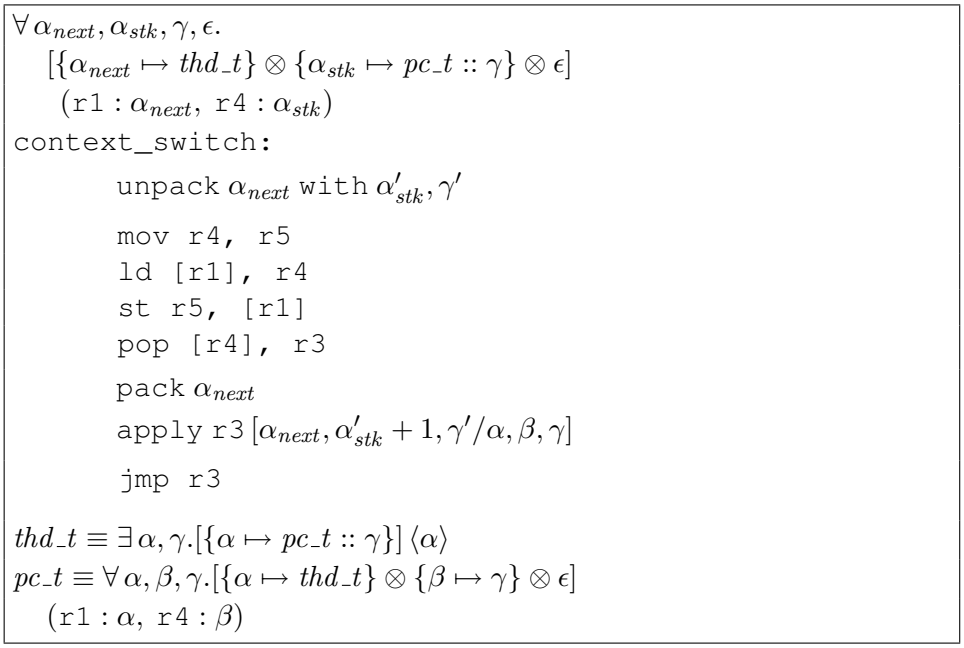

Figure 2.35: Example code of switching contexts without thread-local storage

the return address onto the stack of the thread (in line 2).

First, the function unpacks the thread context passed as the argument (in line 5) in order to access its contents. Next, the function loads the stack pointer of the new thread to register `r4` (in line 7) and stores the stack pointer of the current thread (in line 6 and 8). Then, it pops the program counter of the new thread to register `r3` (in line 9) and runs the new thread by jumping to the program counter (in line 12) after re-packing the thread context (in line 10) and instantiating the label type of the return address (in line 11).

The code is type checked as follows. First, after unpacking the thread context (in line 5), the memory type becomes

$$\{\alpha_{next} \mapsto \langle \alpha'_{stk} \rangle\} \otimes \{\alpha'_{stk} \mapsto pc\_t :: \gamma'\} \otimes \{\alpha_{stk} \mapsto pc\_t :: \gamma\} \otimes \epsilon.$$

Next, after switching the thread contexts (in line 8), the memory type becomes

$$\{\alpha_{next} \mapsto \langle \alpha_{stk} \rangle\} \otimes \{\alpha'_{stk} \mapsto pc\_t :: \gamma'\} \otimes \{\alpha_{stk} \mapsto pc\_t :: \gamma\} \otimes \epsilon,$$

and the type of the register `r4` becomes α'_{stk}, that is, the stack pointer points to the new stack. Then, after popping the new program counter from the new stack (in line 9), the type of the register `r3` becomes pc_t, the type of the register `r4` becomes $\alpha'_{stk} + 1$ and the memory type becomes

$$\{\alpha_{next} \mapsto \langle \alpha_{stk} \rangle\} \otimes \{\alpha'_{stk} + 1 \mapsto \gamma'\} \otimes \{\alpha_{stk} \mapsto pc\_t :: \gamma\} \otimes \epsilon.$$

Next, after re-packing the thread context (in line 10), the memory type becomes

$$\{\alpha_{next} \mapsto thd\_t\} \otimes \{\alpha'_{stk} + 1 \mapsto \gamma'\} \otimes \epsilon.$$

Here the argument for the `pack` instruction is $[\alpha_{stk}, \gamma | \{\alpha_{stk} \mapsto pc\_t :: \gamma\}]\ as\ thd\_t$. Next, the label type of the return address is instantiated in line 11. Then, the label type becomes

$$\forall. \left[\{\alpha_{next} \mapsto thd\_t\} \otimes \{\alpha'_{stk} + 1 \mapsto \gamma'\} \otimes \epsilon\right]\ (\mathtt{r1} : \alpha_{next}, \mathtt{r4} : \alpha'_{stk} + 1)\,.$$

The precondition indicated by the above label type is satisfied by the current memory and registers type. Thus, the last `jmp` instruction is type checked successfully.

2.6 Variant type extension

In this section, we explain how to extend TALK for supporting variant types. The easiest way to support variant types is directly introduce them to the type system. However, the approach has a big problem: the values that have the variant types cannot be created and manipulated directly by the language. For example, let us suppose that the tuple type of TALK is extended with the ML-type variant type. Then, the tuple that may contain an integer or a pointer to an integer can be represented as follows:

$$\mathit{Int\ of}\ \exists\beta.\,\langle\beta\rangle\ \mid \mathit{IntPtr\ of}\ \exists\beta.\,[\{\beta \mapsto \langle 0\rangle\}]\,\langle\beta\rangle\,.$$

The problem is that the above type does not tell how to create and use the values of *Int* and *IntPtr*. For example, let us suppose that we allow casts from the type $\exists\beta.\,\langle\beta\rangle$ to the above variant type. Then, when accessing the value of the variant type, we need to destruct the variant type and bring out the original type as the pattern match of the ML languages [17]. However, this is impossible in the operational semantics of TALK because we do not know how the variant values are represented in the abstract machine and, in fact, the abstract machine does not specify how to do that.

The naive solution to this problem is to fix the representation of the variant values in the abstract machine [96]. For example, let us suppose that we fix the representation of the values of the above variant type as follows:

$$\exists\beta.\,\langle 0, \beta\rangle\ \mid \exists\beta.\,[\{\beta \mapsto \langle 0\rangle\}]\,\langle 1, \beta\rangle\,.$$

Then, we can destruct the variant and bring out the original type by checking whether the first element of the tuple is 0 or 1.

In this approach, however, the representation of the variant is limited to the one specified by the type system. That is, to support another representation, we need to extend the type system. This contradicts the goal, designing a strictly typed language which is flexible and expressive enough to implement memory management facilities.

To solve the problem, we generalize the above approach by exploiting the integer constraints.

2.6.1 Types

Figure 2.36 shows the syntax of the extended types. The point is the introduction of the precondition Ψ. The precondition (Ψ) is a set of pairs of the integer constraints (C) and the memory type (Σ). If the precondition

always contains only one pair, the extended type system is equal to the original one described in Section 2.2.

$$
\begin{array}{lllll}
(\textit{label type}) & lt & ::= & \forall\Delta.\Psi\,(\Gamma) \\
(\textit{type}) & \tau & ::= & \ldots \mid \exists\Delta.\Psi\tau \mid \ldots \\
(\textit{precondition}) & \Psi & ::= & \cdot \mid (|C|\,[\Sigma])\,\Psi
\end{array}
$$

Figure 2.36: Extension of types

The label type is modified so that it specifies the precondition, instead of directly specifying the integer constraints and the memory type. The precondition of the label type means that the instructions of the label of the label type are valid, that is, their execution never gets stuck for all the pairs of the integer constraints and the memory type. In other words, if one pair of the precondition of the label type is satisfied, we can safely jump to the label that has the label type. For example, the following label type

$$
\begin{array}{l}
\forall\alpha_1, \alpha_2.\ (|\alpha_1 = \alpha_2|\,[\{\alpha_1 \mapsto \exists\beta.\,\langle\beta\rangle\}]) \\
\qquad\qquad (|\alpha_1 \neq \alpha_2|\,[\{\alpha_1 \mapsto \exists\beta.\,\langle\beta\rangle\} \otimes \{\alpha_2 \mapsto \exists\beta.\,\langle\beta\rangle\}]) \\
\qquad (\texttt{r1} : \alpha_1, \texttt{r2} : \alpha_2)
\end{array}
$$

indicates the instructions that take two pointers to integers with the register `r1` and `r2`, but the pointers may alias, that is, refer to the same integer. The first pair of the precondition indicates that the value of the register `r1` is equal to that of the register `r2` and there exists one integer at the address $\alpha_1 (= \alpha_2)$. The second pair indicates that the values of the register `r1` and `r2` are distinct and there are two integers in the memory.

The existential type is also modified so that it specifies the precondition. The precondition of the existential type means that there exists memory regions that satisfy the memory type and the integer constraints specified in one pair of the precondition. For example, the tuple that contains a pointer to an integer or the null pointer (0) can be represented as follows:

$$
\exists\alpha.\ (|\alpha = 0|\,[\cdot])(|\alpha \neq 0|\,[\{\alpha \mapsto \exists\beta.\,\langle\beta\rangle\}])\ \langle\alpha\rangle\,.
$$

Here the first pair of the precondition indicates that if $\alpha = 0$, then there exists no memory encapsulated in the existential type. The second pair indicates that if $\alpha \neq 0$, then there exists an integer at the address α.

The strength of the extended type system is that we can specify the representation of the variant types with the integer constraints. Thus, several

representations of the variant types can coexist in one type system. For example, the tuple of the previous paragraph can be also represented as follows:

$$\exists\alpha_{tag}, \alpha.\ (|\alpha_{tag} = 0|\,[\cdot])(|\alpha_{tag} = 1|\,[\{\alpha \mapsto \exists\beta.\,\langle\beta\rangle\}])\ \langle\alpha_{tag}, \alpha\rangle\,.$$

In this case, the tuple has the extra tag explicitly for representing the variant type. Here the first pair of the precondition indicates that if the first element of the tuple (the variant tag) is equal to 0, the tuple represents a null pointer. The second pair indicates that if the variant tag is equal to 1, the tuple represents a pointer to an integer. Further, the tuple can be represented as follows:

$$\exists\alpha.\ (|\alpha < 4096|\,[\cdot])(|\alpha \geq 4096|\,[\{\alpha \mapsto \exists\beta.\,\langle\beta\rangle\}])\ \langle\alpha\rangle\,.$$

In this case, the pointer is considered as a null pointer if the value is less that 4096.

2.6.2 Abstract machine

The abstract machine is almost unchanged despite the extension of the type. The only modification is that the coerce instructions for manipulating the existential packages are extended according to the extension of the existential types. The extended syntax of the abstract machine is shown in Figure 2.37.

$$(\mathit{insts}.) \quad I \quad ::= \quad \ldots \mid \mathtt{pack}_{[c_1,\ldots,c_n|\Sigma]as\exists\Delta.\Psi\tau}\ i; I \mid \ldots$$

Figure 2.37: Extension of abstract machine

2.6.3 Instructions and operational semantics

Although the types and the abstract machine are modified, the operational semantics of the abstract machine are unchanged. That is, the original instructions of TALK suffice for creating and manipulating the values of the variant types. Strictly speaking, the `pack` and `unpack` instructions now handle the extended existential types, but it does not affect the representation of their operational semantics.

2.6.4 Typing rules

According to the extension of the types, the typing rules need to be extended in order to define the meanings of the extension formally. Figure 2.38 shows the extended typing rules for the states of the abstract machine.

$$\frac{Dom(P) = Dom(\Phi) \quad \forall l \in Dom(P).\Phi(l) \equiv \forall \Delta.\Psi\,(\Gamma) \qquad \forall\,|C|\,[\Sigma] \in \Psi.\Delta; \Gamma; C; \Sigma \vdash P(l)}{\vdash P : \Phi} \text{ (PROGRAM)}$$

$$\frac{\Delta; C \vdash t : \tau'\,[c_1, \ldots, c_n/\Delta'] \qquad \theta \equiv [c_1, \ldots, c_n/\Delta'] \qquad \tau \equiv \exists \Delta'.\Psi\tau' \qquad \exists\,|C'|\,[\Sigma'] \in \Psi.\mathit{if}\ \Delta; C \models C'\theta,\ \mathit{then}\ \vdash M : \Sigma'\theta}{\Delta; C \vdash \mathtt{pack}_{[c_1,\ldots,c_n|M]\mathit{as}\ \tau}\,(t) : \tau} \text{ (TUPLEPACK)}$$

$$\frac{\forall \Delta'.\Psi'\,(\Gamma') \equiv \Phi\,(l) \qquad \theta \equiv [c_1, \ldots, c_n/\Delta''] \qquad \Psi'' \equiv \Psi'\theta \qquad \Gamma'' \equiv \Gamma'\theta \qquad \sigma \equiv \forall \Delta' \backslash \Delta''.\Psi''\,(\Gamma'')}{\Delta; C \vdash l\,[c_1, \ldots, c_n/\Delta''] : \sigma} \text{ (VALUELABEL)}$$

Figure 2.38: Extension of typing rules (machine state)

First, the typing rule PROGRAM is modified to consider the precondition (Ψ). The typing rules states that the program P is well-formed if all the instructions in P are well-formed according to the program type Φ. More concretely, for all the labels $l \in Dom(P)$, the instructions $P(l)$ are well-formed under the label type $\Phi(l)$.

As discussed informally in Section 2.6.1, the difference from the original typing rule is that the label type now contains the precondition Ψ. Therefore, we need to check whether the instructions are well-formed under all the pairs of the integer constraints and the memory types defined in the precondition.

Then, the typing rule TUPLEPACK is also modified in order to handle the precondition Ψ. The difference from the original typing rule is that the existential type specifies the candidates of the memory types (and the integer constraints) that can be encapsulated. Thus, the typing rule is modified so that it checks whether there exists one candidate that is satisfied by the current assumption of the type checker.

Last, the typing rule VALUELABEL is slightly modified because of the same reason as the typing rule PROGRAM.

Well-formedness of instructions

Now we show how to extend the typing rules for instructions. First, we describe the extension of the `branch` and `jmp` instructions (Figure 2.39). Then, we explain the extension of the `apply`, `pack` and `unpack` instructions (Figure 2.40).

$$\frac{\begin{array}{c}\Delta; \Gamma; C \vdash \Gamma\,(r_d) = \forall.\Psi'\,(\Gamma') \\ C_1 \equiv \Gamma\,(r_{s1})\,(=, \leq)\Gamma\,(r_{s2}) \\ C_2 \equiv \Gamma\,(r_{s1})\,(\neq, >)\Gamma\,(r_{s2}) \\ C'' \equiv C \wedge C_1 \\ \textit{If } \Delta; C \not\models \neg C_1, \textit{ then } \exists\,|C'|\,[\Sigma'] \in \Psi'. \\ \Delta; C'' \models C' \qquad \Delta; C'' \vdash \Sigma = \Sigma' \qquad \Delta; C'' \vdash \Gamma \leq \Gamma' \\ \textit{If } \Delta; C \not\models \neg C_2, \textit{ then } \Delta; \Gamma; C \wedge C_2; \Sigma \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash (\texttt{beq}, \texttt{ble})\; r_{s1}, r_{s2}, r_d; I} \quad \text{(BRANCH)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma; C \vdash \Gamma\,(r_d) = \forall.\Psi'\,(\Gamma') \\ \exists\,|C'|\,[\Sigma'] \in \Psi'. \\ \Delta; C \models C' \qquad \Delta; C \vdash \Sigma = \Sigma' \qquad \Delta; C \vdash \Gamma \leq \Gamma'\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{jmp}\; r_d} \quad \text{(JUMP)}$$

Figure 2.39: Extension of typing rules (instructions)

The most important difference of the typing rules BRANCH and JUMP from their original counterparts is that the label type now specifies several pairs of the memory types and the integer constraints and if one of the pairs is satisfied by the current assumption of the type checker, the rule considers that it is safe to jump to the label of the label type. This is because the typing rule PROGRAM states that the instructions are well-formed under the all of the pairs. Thus, both the BRANCH and JUMP typing rules chose one pair of the memory type and the integer constraints from the precondition of the target label type. Then, they check whether the current assumption (typing context) satisfies the chosen memory type and the integer constraints.

Another important difference is that the typing rule BRANCH now explicitly omits the typing of the taken branch and the non-taken branch if the conditions for the branches are apparently unsatisfiable according to the current typing context. For example, let us assume that the type of the register `r1` is 1 and that of the register `r2` is 2. Then, let us suppose that we are type-checking the following code fragment:

```
ble r1, r2, branch
```

...

The extended typing rule BRANCH omits the typing checking of the rest of the instructions, because $1 > 2$ is always unsatisfiable.

Strictly speaking, this omission is unnecessary, that is, the type system is sound without it. The reason why the omission is introduced is for practical reasons. The details are explained later.

$$\frac{\begin{array}{c}\Gamma(r) \equiv \forall\Delta'.\Psi'\,(\Gamma') \\ \theta \equiv [c_1, \ldots, c_n/\Delta''] \quad \Psi'' \equiv \Psi'\theta \quad \Gamma'' \equiv \Gamma'\theta \\ \sigma'_f \equiv \forall\Delta'\backslash\Delta''.\Psi''\,(\Gamma'') \quad \Delta; \Gamma\{r \mapsto \sigma'_f\}; C; \Sigma \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{apply}\ r\,[c_1, \ldots, c_n/\Delta'']\,; I} \quad \text{(APPLY)}$$

$$\frac{\begin{array}{c}\theta \equiv [c_1, \ldots, c_n/\Delta'] \quad \exists\,|C'|\,[\Sigma'] \in \Psi'.\mathit{if}\ \Delta; C \models C'\theta, \\ \mathit{then}\ \Delta; C \vdash \Sigma = \Sigma'' \otimes \{i \mapsto \tau\theta\} \otimes \Sigma'\theta \\ \Delta; \Gamma; C; \Sigma'' \otimes \{i \mapsto \exists\Delta'.\Psi'\tau\} \vdash I\end{array}}{\Delta; \Gamma; C; \Sigma \vdash \texttt{pack}_{[c_1,\ldots,c_n|\Sigma'[c_1,\ldots,c_n/\Delta']]as\exists\Delta'.\Psi'\tau}\ i; I} \quad \text{(PACK)}$$

$$\frac{\begin{array}{c}\Delta; C \vdash \Sigma = \Sigma'' \otimes \{i \mapsto \exists\Delta'.\Psi'\tau\} \quad \theta \equiv [\Delta''/\Delta'] \\ \forall\,|C'|\,[\Sigma'] \in \Psi'.\Delta\Delta''; \Gamma; C \wedge C'\theta; \Sigma'' \otimes \{i \mapsto \tau\theta\} \otimes \Sigma'\theta \vdash I\end{array}}{\Delta; C; \Sigma \vdash \texttt{unpack}\ i\ \texttt{with}\ \Delta''; I} \quad \text{(UNPACK)}$$

Figure 2.40: Extension of typing rules (coerce)

The typing rule APPLY is slightly modified so that it handles the precondition Ψ contained in the label type. The rule instantiates the label type (including the precondition) by substituting type variables, in the same way as the original one.

The typing rule PACK is also modified in order to handle the extended existential type. The rule checks whether there exists a pair of the memory type and the integer constraints that are satisfied by the memory to be encapsulated in the precondition.

The typing rule UNPACK is largely modified because it needs to handle all the pairs in the precondition. The rule first unpacks the existential type. Then, it checks whether the rest of the instructions are well-formed for all the pairs of the memory types and the integer constraints specified in the precondition of the existential type.

For example, let us assume that the type of the memory is

$$\{\alpha \mapsto \exists\beta.(|\beta = 0|\,[\cdot])(|\beta \neq 0|\,[\{\beta \mapsto \langle 0\rangle\}])\,\langle\beta\rangle\}.$$

The memory type indicates that there exists a pointer to an integer (0), or a null pointer, at the address α. Then, let us suppose that the type checker checks the code of Figure 2.41, where the type of the register `r1` is α.

```
1       unpack α
2       ld [r1], r2
3       beq r2, 0, null_pointer_exception
4       ld [r2], r3
```

Figure 2.41: Example of the usage of the variant type

First, the `unpack` instruction unpacks the existential package. Then, the memory type becomes

$$\{\alpha \mapsto \langle\beta\rangle\}$$

where $\beta = 0$, or

$$\{\alpha \mapsto \langle\beta\rangle\} \otimes \{\beta \mapsto \langle 0\rangle\}$$

where $\beta \neq 0$. Therefore, the typing rule UNPACK checks the rest of the instructions for each case.

If $\beta = 0$, the type checker ignores the second `ld` instruction according to the typing rule BRANCH, because $\beta \neq 0$ is unsatisfiable from the current assumption $\beta = 0$ (here we assume that the type of the label `null_pointer_exception` specifies no preconditions, that is, we are always able to jump to the label).

If $\beta \neq 0$, the type checker checks the second `ld` instruction and it passes the type check because the type of the register `r2` is β and the memory type indicates that there is an integer (0) at the address β.

Note that if the typing rule BRANCH does not omit the type checking explicitly, the code of Figure 2.41 never passes the type check, because the second `ld` instruction is ill-formed when $\beta = 0$ (there is nothing at the address β). This is why the BRANCH typing rule explicitly omits the typing of the taken branch and the non-taken branch if the conditions for the branches are unsatisfiable.

2.6.5 Equality rules

The equality rules are also extended according to the extension of the types (Figure 2.42). As expected, the equality rules for the existential types and the label types (EQEX and EXLABEL) are slightly modified because they now contain the precondition Ψ. First, they check whether the preconditions are equal or not with the rule EXPRE and/or EXPREEMPTY (explained

later). Then, they check the equality of the rest of the types for all the pairs of the precondition.

$$\frac{\Delta\Delta'; C \vdash \Psi_1 = \Psi_2 \quad \forall\, |C'|\,[\Sigma'] \in \Psi_1.\Delta\Delta'; C \wedge C' \vdash \tau_1 = \tau_2}{\Delta; C \vdash \exists\Delta'.\Psi_1\tau_1 = \exists\Delta'.\Psi_2\tau_2} \quad \text{(EQEX)}$$

$$\frac{\Delta\Delta'; C \vdash \Psi_1 = \Psi_2 \quad \forall\, |C'|\,[\Sigma'] \in \Psi_1.\Delta\Delta'; C \wedge C' \vdash \Gamma_1 = \Gamma_2}{\Delta; C \vdash \forall\Delta'.\Psi_1\,(\Gamma_1) = \forall\Delta'.\Psi_2\,(\Gamma_2)} \quad \text{(EQLABEL)}$$

$$\Delta; C \vdash \cdot = \cdot \quad \text{(EQPREEMPTY)}$$

$$\frac{\Delta; C \vdash C_1 = C_2 \qquad \Delta; C \wedge C_1 \vdash \Sigma_1 = \Sigma_2 \quad \Delta; C \vdash \Psi_1 = \Psi_2}{\Delta; C \vdash (|C_1|\,[\Sigma_1])\,\Psi_1 = (|C_2|\,[\Sigma_2])\,\Psi_2} \quad \text{(EQPRE)}$$

Figure 2.42: Extension of equality rules (types)

The equality of the preconditions is checked by the rule EQPREEMPTY and EQPRE. The rule EQPREEMPTY states that the empty precondition is equal to itself. The rule EQPRE first chooses one pair of the memory type and the integer constraints from each precondition. Then, it checks whether the chosen integer constraints are equal or not. In addition, it also checks whether the chosen memory types are equal under the extended assumption with the chosen integer constraints. Last, it checks the equality of the rest of the preconditions. Note that the precondition is a set, that is, the syntactic order is irrelevant, in the same way as the memory type.

2.7 Implementation

Based on the idea shown in Section 2.2, we implemented a TALK assembler and a TALK type checker for the IA-32 [21] architecture. The TALK assembler takes TALK code and emits binary executables annotated with the TALK type information. The format of the binary executables are usual ELF format. Therefore, they can be executed without any special runtime support. The TALK type checker takes the binary executables and type-checks them. Because the type system of TALK includes integer constraints, the type checker must be able to solve the constraints. To this end, we utilized the algorithm of the Omega test [78]. The TALK assembler and the TALK

type checker are available from the web site [91].

In this section, we describe how to map the real IA-32 architecture to the abstract CPU architecture shown in Section 2.2. The TALK language described in Section 2.2 is based on an abstract, virtual RISC CPU architecture, but the IA-32 architecture is a complex and peculiar CISC architecture. Thus, we need to make the connection between the real architecture and the virtual architecture in order to type check IA-32 executables. Although the correctness of the mappings explained in this section has not been proved yet, we believe that it is straightforward.

2.7.1 Execution mode

Basically, the IA-32 architecture has three execution modes: real mode, protected mode and virtual 8086 mode.

In real mode and virtual 8086 mode, the features of the CPU are severely limited, because, in these two modes, the CPU behaves as if it is an old 8086 CPU. For example, the largest memory size that can be handled by the CPU is only 1M bytes. In addition, virtual memory facilities of IA-32 CPUs are not available. Therefore, these two modes are rarely used by today's ordinary programs.

The only commonly-used program that is executed in real mode is a boot loaders (or a system initialization program). When an IA-32 system boots, its CPU runs in real mode. Then, the boot loader goes through the complicated procedures to switch the execution mode from real mode to protected mode.

The current implementation of TALK does not handle real mode and virtual 8086 mode. Therefore, we cannot write boot loaders in TALK. Theoretically, we can extend the type system to support these two modes. However, we believe that it is not a good idea to complicated the type system for supporting boot loaders, because they are hardly modified and executed only once per system boot.

Protected mode further consists of two sub-execution modes: 16 bit mode and 32 bit mode. In 16 bit mode, the largest memory size that can be handled by the CPU is 16M bytes. Thus, 16 bit mode is also rarely-used by today's ordinary programs. Therefore, the current implementation of TALK does not support 16 bit mode for the same reason as real mode and virtual 8086 mode.

In the following, we describe how to map the IA-32 32 bit protected mode to the abstract machine of Section 2.2.

2.7.2 Registers

The IA-32 CPU has only 8 general purpose registers (`eax`, `ecx`, `edx`, `ebx`, `esi`, `edi`, `ebp`, `esp`). Therefore, we need to fix the number of registers of the abstract machine to 8. Apparently, this does not break the soundness of the type system.

Other registers of IA-32 CPU, for example, floating-point registers and control registers are not handled by the current implementation of TALK. Although the floating-point arithmetic operations of IA-32 are fairly complicated, we believe that supporting floating-point registers is not so hard, because the operations are not used for pointer arithmetics.

One peculiar feature of the IA-32 architecture is that the general purpose registers can be accessed partly. For example, the size of the general purpose register `eax` is 32 bit, but we can access its lower 16 bits only by the register `ax`. In addition, the higher 8 bits of the register `ax` can be accessed by the register `ah` and the lower 8 bits can be accessed by the register `al`

The current implementation of TALK keeps track of the size of the general purpose registers by its type system and it is prohibited to read the register with a size which is different from the tracked one. For example, the code of Figure 2.43 does not pass the type check. First, the code initializes the register `eax` with a 32 bit integer in line 1 (here we assume that the size of the register `ecx` is also 32 bit). Thus, the type checker assumes that the size of the register `eax` is 32 bit. Then, the code tries to read the register `eax` by the register `ax` in line 2. Now, the type checker rejects the read because the size of the access register `ax`, 16, is different from the size of the register `eax`, 32.

```
1       mov %ecx, %eax
2       mov %ax, %dx // Type Error!
```

Figure 2.43: Example of accessing the part of the general purpose register `eax` (ill-typed)

2.7.3 Instructions

As for instructions, the biggest difference between the IA-32 architecture and the abstract machine of Section 2.2 is that the IA-32 instructions take not more than 2 operands. For example, the arithmetic add instruction is as follows:

```
add %eax, %ecx
```

The above instruction adds two integer values stored in the register `eax` and `ecx`, and stores the result to the register `ecx`.

This difference does not affect the soundness of TALK, because two-operand instructions are always representable with three-operand instructions. For example, the above `add` instruction is equal to the following instruction:

```
add %eax, %ecx, %ecx
```

Another big difference is that the IA-32 instructions can take "memory operands", that is, directly access memory regions without using memory load/store instructions explicitly. In other words, the IA-32 architecture has no specific load/store instructions. For example, the following instruction loads an integer value from the memory region specified by the register `eax`, adds the value and another integer value stored in the register `ecx`, and stores the result to the register `ecx`.

```
add (%eax), %ecx
```

This difference also does not affect the soundness of TALK, because the instructions that access memory regions with the memory operands are always representable with two instructions. For example, the above `add` instruction is equal to the following instructions:

```
ld [%eax], %tmp
add %tmp, %ecx, %ecx
```

Here the register `tmp` is introduced to the abstract machine for storing the loaded value temporarily. The introduction of the temporary register does not break the soundness of TALK, because the number of the registers are fixed ($n = 9$).

2.7.4 Memory addressing

As described above, the type system of TALK properly handles the IA-32 instructions that directly access the memory through the memory operands. However, there is a subtle problem that should be addressed in the memory operands of IA-32: memory addressing.

The memory addressing of IA-32 is somewhat complicated. The most complex addressing is represented as the following expression:

$$r_{base} + n_{scale} * r_{index} + n_{disp}.$$

Here r_{base} and r_{index} are general purpose registers. In addition, n_{scale} and n_{disp} are immediate values. Further, the possible value of n_{scale} is 1, 2, 4, or 8. On the other hand, the memory addressing of the abstract machine of Section 2.2 is as follows:

$$r_{base} + n_{disp}.$$

Therefore, we need to bridge the gap between the two memory addressing methods.

The difficulty is that r_{base} of the IA-32 addressing is not necessarily equal to r_{base} of the addressing of the abstract machine. For example, to access the 3rd element of the tuple that resides at the address α, the addressing in the abstract machine must be $r + 2$, where the type of the register r is α. However, in the IA-32 architecture, the addressing may be $r + 2$, where the type of r is α, $r + r + 2$, where the type of r is $\alpha/2$, or $r_b + 2 * r_i + 2$, where the type of r_b is 0 and the type of r_i is $\alpha/2$, and so on.

Theoretically, it is possible to figure out the values of the register r_{base} and the displacement value n_{disp} from the memory addressing of the IA-32 architecture. This is because the type system of TALK keeps separating memory regions from each other. To be more precise, the memory regions contained in the memory that is judged as "valid" by the typing rule of TALK do not overlap each other. Therefore, each address of the memory space resides in at most one memory region. Because the type checker knows the start address and the size of each memory region, it is able to find the memory region that contains the address specified in the memory addressing. For example, let us suppose that the type of the memory is

$$\{\alpha \mapsto \langle 1, 2, 3 \rangle\},$$

and the type of the registers is

$$\mathtt{r1} : \alpha + 1, \mathtt{r2} : 1.$$

Then, the memory addressing

$$\mathtt{r2} + 1 * \mathtt{r1} + 0$$

exactly specifies the third element of the tuple at the address α.

From the practical viewpoint, however, finding one memory region from a specified address is expensive because, at the worst case, the type checker must examine all the memory regions, that is, perform bound checks for all the memory regions with a constraint solver.

To solve the problem, the current implementation of TALK takes an extra argument for the memory addressing. The extra argument is the memory addressing of the abstract machine. Thus, the type checker is able to find the memory region from the memory addressing of IA-32 (and the extra argument), because all the type checker has to do is check whether the value of the memory address is equal to the value of the extra argument. For example, the above memory addressing `r2+1*r1+0` can be annotated with the extra argument $\alpha + 2$. Then, the type checker only checks whether

$$\alpha + 2 = \texttt{r2} + 1 * \texttt{r1} + 0.$$

2.7.5 Branch

In the abstract machine of Section 2.2, the branch instructions directly take the arguments to be compared. However, the branch instructions of IA-32 do not take them. Instead, the IA-32 CPU remembers the result of the previous arithmetic operation, and it figures out if a branch should be taken or not according to the stored result. For example, the following code jumps to the address specified by the label `branch` if `ecx` $\geq$ `eax` (`jae` stands for "jump if above or equal").

```
sub %eax, %ecx
jae branch
```

To support the IA-32 branch instructions, the type system of TALK is extended with one extra register (named `flag`) that holds the result of the previous arithmetic instruction. For example, when type checking the above code fragment, the type of the register `ecx` becomes $(\alpha_c - \alpha_a)$ after the `sub` instruction (here we assume that the initial types of the register `eax` and `ecx` are α_a and α_c. In addition, the type system keeps track the value as the type of the extra register. Then, the `jae` instruction is considered as the following instruction.

```
ble 0, %flag, branch
```

2.8 Limitations

2.8.1 Generic graph data structures

The type system of TALK is flexible and expressive enough to implement practical memory management code (e.g., malloc/free). However, it is not

enough to efficiently implement all the data structures that can be represented in ordinary typed languages, because the type system of TALK restricts pointer manipulation in some cases in order to keep track of aliasing relations between pointers. For example, generic graph data structures including directed acyclic graphs and cyclic graphs cannot be represented naturally in the type system of TALK. This is because once a memory region is encapsulated inside an existential package, the region cannot be accessed until the package is unpacked. For example, suppose that we represent the type of nodes in the generic graph data structures as follows:

$$\begin{array}{l} \mathit{Node} \equiv \\ \exists\, \alpha_{\mathit{next}}.\, |\cdot|[\{\alpha_{\mathit{next}} \mapsto \mathit{Node}\}]\, \langle \alpha_{\mathit{next}} \rangle\, . \end{array}$$

In the representation, however, different two nodes cannot point to the same node, because the two encapsulated memory regions must be distinct. That is, directed acyclic graphs cannot be implemented in the representation. A naive workaround to the problem is to give up capturing the structure of the graph in its type. For example, the following type can be used to implement the generic graph with 10 nodes:

$$\begin{array}{l} \mathit{Graph} \equiv \\ \exists\, \alpha_{\mathit{next}}.\, |0 \leq \alpha_{\mathit{next}} < 10|[\,]\, \langle \alpha_{\mathit{next}} \rangle\, [10]\, . \end{array}$$

It is only an array and each element of it represents a node in the graph. Each node holds an integer (α_{next}) which points to the next node in the graph.

A proper solution to the problem is to extend the type system of TALK so that the encapsulated memory regions inside existential packages can be aliased. The apparent problem of the solution is that, if we allow arbitrary aliases in existential packages, the memory safety is easily violated. We expect that the results of the previous works [62, 23, 31] in relaxing linearity temporarily in the linearly-typed languages can be applicable to the problem.

In addition, although we are able to implement malloc/free in TALK, generic conservative garbage collectors cannot be implemented in TALK. This is mainly because the current type system of TALK prohibits accessing the memory regions abstracted as memory type variables (ϵ). That is, the garbage in the abstracted memory regions cannot be collected safely. However, we still can implement functions that act like a garbage collector by following the approach of [98].

2.8.2 Race freedom on multi CPU machines

The type system of TALK is able to prevent the race conditions on single CPU machines (see Section 3.2). On multi-CPU machines, however, the memory safety, the control-flow safety and the race freedom cannot be ensured by the current type system of TALK. There are two problems in ensuring the safety properties on the multi-CPU machines.

First problem is that threads really run concurrently in the multi-CPU environment. In the single-CPU environment, threads do not really run concurrently, that is, they are emulated by a single thread. Therefore, it is easy to represent their operational semantics because just a single-thread semantics suffices. In the multi-thread environment, however, we need to consider concurrency, non-determinism and so on in their semantics. Therefore, it seems to be inevitable to adopt the approaches of process calculus [69, 68, 41].

Second problem is that the memory consistency of real CPU architectures is extremely complex. For example, suppose that three programs P, Q and R are executed on three CPUs C, D and E, respectively. Now, let us also suppose that the program P first wrote data to the address X, then the program Q wrote data to the address Y. According to the IA-32 architecture specification [21], the program R may see the write operations in reverse order, that is, data is first written to the address Y, then written to the address X. What is worse is that, even in the IA-32 architecture, the semantics of the memory consistency varies according to the CPU types. Thus, it is a challenging task to incorporate the complex memory consistency into the type system.

2.8.3 Deadlock and livelock freedom

The current type system of TALK cannot ensure the deadlock freedom and livelock freedom. For example, let us consider a simple function shown in Figure 2.44 (written in C for clarity). The function first acquires a synchronization lock in line 2, then it releases the lock before returning to its caller in line 4. The function passes the type check of TALK, because it is memory-safe and control-flow safe. However, it is apparent that the function may cause a deadlock if other threads try to acquire the lock because the function enters an infinite loop without releasing the held lock (in line 3).

The function of Figure 2.44 is too simple, so it seems to be easy to detect possible deadlocks by slightly extending the type system, but it is an

```
1 void Deadlock(void) {
2    Lock();
3    while (1);
4    Unlock();
5 }
```

Figure 2.44: A function that may cause a deadlock

illusion. To see why preventing deadlocks is difficult, consider the function of Figure 2.45, provided that the function `SomeFunc` is defined elsewhere and well-typed. The only difference from the function of Figure 2.44 is that the function of Figure 2.45 calls the function `SomeFunc` instead of directly entering an infinite loop. It may cause a deadlock if `SomeFunc` runs forever without releasing the held lock. That is, in order to prevent deadlocks, the type system must check whether `SomeFunc` will terminate or not. Unfortunately, however, this is an undecidable problem for Turing-complete languages, like TALK. One naive workaround is to prohibit backward jumps and function calls after acquiring synchronization locks. This approach, however, severely limits expressiveness and usefulness of the language, so it is impractical.

```
1 void Deadlock2(void) {
2    Lock();
3    SomeFunc();
4    Unlock();
5 }
```

Figure 2.45: Another function that may cause a deadlock

Preventing livelocks by the type system is harder than deadlocks, because it is almost the same as trying to verify fairness of thread schedulers.

2.8.4 Resource usage safety

Resource usage safety is the property that a program accesses a resource in a specified manner. For example, on the UNIX system, files must be first opened, then closed at the end. Between the open and the close, the files can be read and/or written any number of times. For another example, TCP

client sockets on the UNIX system must be first created, then connected to the server, and closed at the end of communication. Between the connect and the close, we can transmit and/or receive data with the sockets any number of times. The resource usage safety is also useful for ensuring the correctness of device drivers, because each device has its own semantics or procedure for correct operation.

The current type system of TALK cannot ensure the resource usage safety because it has no concern with how resources are accessed. Thus, the correctness of device drivers cannot be ensured in TALK, though their memory safety and control-flow safety are ensured. However, the type system can be extended to support the resource usage safety based on the previous works for ensuring the resource usage safety by type systems [54, 24, 95]. In fact, the current type system of TALK is able to ensure a simple form of the resource usage safety. For example, the proper usage of spin locks (that is, they must be acquired first, then released) can be enforced by the type system of TALK (see Section 3.2.4 for details).

Chapter 3

A prototype OS kernel written in TALK

Using the TALK assembler, we implemented a prototype OS kernel for the IA-32 architecture in TALK. The kernel provides a memory management facility, a multi-thread management facility and a very basic device control facility.

More specifically, the memory management facility takes control of almost all the available memory on the system and provides functions that allocate memory regions from the memory and deallocate them. The functions are used by other part of the kernel, as well as user programs (see Section 3.1 for details).

The multi-thread management facility provides functions that create a new thread, exit the running thread and yield the CPU to other threads. It also provides a mechanism for synchronizing threads in order to avoid race conditions (see Section 3.2 for details).

The current device control facility only provides device drivers that manage the video display and the keyboard of the system. In theory, other hardware devices (e.g., hard disk drives and network interfaces) can be supported without any problem.

As for user programs, the current kernel implementation does not provide a dynamic loading of the user programs. Moreover, it does not provide separate virtual address space for them because it never updates the page table after its initialization in order to ensure the memory safety and the control-flow safety (see Section 3.3 for details). Therefore, the user programs must be linked statically to the kernel. In addition, they must be written in TALK, otherwise the safety of the kernel cannot be ensured.

For booting the kernel, we utilize the GNU GRUB boot loader [37]. In addition, some peculiar boot procedures (e.g., segment preparation) are written in the untyped IA-32 assembly language. Except for them, the kernel is completely written in TALK. The size of the kernel is about 1700 lines of TALK code. It takes about 0.9 seconds to type-check the whole kernel on the Pentium 4 (3 GHz) processor with 1 GB RAM. The prototype OS kernel is available from the web site [91].

The rest of this chapter describes the implementation details of the kernel. Note that the examples shown in this chapter are written in TALK of Section 2.2 for ease of understanding. The real implementation is written in TALK for IA-32 of Section 2.7.

3.1 Memory management

The implementation described in the previous chapter (Section 2.5.2) meets the minimum requirements for malloc/free, but it is too simple for practical use. For example, after the several allocation and deallocation, the free memory will be fragmented. This section shows a more practical and complex implementation.

3.1.1 Type of the free memory

Figure 3.1 represents the type of the free memory. It is a list of variable-length arrays. Each element of the list is a variable-length array and a tuple which has two elements. The size of the array is stored in second element of the tuple. First element of the tuple is a pointer to the next element of the list.

```
1 FreeMem2 ≡
2   μη [α_self] .∃ α_next, α_size, α_mem. |α_mem = α_self + 2|
3     [{α_mem ↦ ∃β. ⟨β⟩ [α_size]} ⊗ {α_next ↦ η (α_next)}]
4       ⟨α_next, α_size⟩
```

Figure 3.1: Type of the free memory (list of variable-length arrays)

The difference from the type of the previous chapter (Figure 2.31) is that it makes the free memory itself (the variable-length arrays) and its header information (the tuples) adjacent, in order to avoid memory fragmentation. More concretely, the recursive type that represents the free

memory list takes one argument. This one argument is used to represent the address of the free memory (tuple) in its the type. For example, the type of the free memory which resides in the address α is described as $\{\alpha \mapsto \mathit{FreeMem2}(\alpha)\}$. Then, let us suppose that the free memory type is unrolled once. Then, the memory type becomes

$$\begin{aligned}\{\alpha \mapsto \exists \alpha_{next}, \alpha_{size}, \alpha_{mem}.|\alpha_{mem} = \alpha + 2| \\ [\{\alpha_{mem} \mapsto \exists \beta.\, \langle\beta\rangle\, [\alpha_{size}]\} \otimes \{\alpha_{next} \mapsto \mathit{FreeMem2}(\alpha_{next})\}] \\ \langle \alpha_{next}, \alpha_{size} \rangle\}.\end{aligned}$$

From the memory type above, we know that the tuple (at α) and the variable-length array (at α_{mem} inside the existential package) are adjacent by the integer constraint specified in the existential type ($\alpha_{mem} = \alpha + 2$).

3.1.2 Implementation of malloc

Figures 3.2 and 3.3 show an implementation of malloc. As in the previous chapter, the syntax of instructions are slightly extended. In addition, the `apply` instruction and the arguments for the `pack`, `unpack` and `roll` instructions are omitted for clarity.

As the simple implementation of malloc in the previous chapter, the code traverses the free memory list until it finds the array (memory region) whose size satisfies the requested size. If the array is found, the code splits the array into the array of the requested size and the rest. While the simple implementation in the previous chapter takes the top-half of the array, this implementation takes the bottom-half of the array in order to keep the adjacency of the header and the free memory region.

The label type of `malloc2` indicates that the function takes a free memory (*FreeMem2* in line 2 of Figure 3.2) as an argument and returns an array of the specified size (α_{size} in line 3). The type of the allocated array is specified in line 16 of Figure 3.3. Note that the return type of the function is abbreviated as ret_t.

The function first unrolls and unpacks the packed free memory list in order to access its contents. Then, it checks whether the array of first element of the given free memory list satisfies the requested size (in line 7 and 8 of Figure 3.2). If so, the function jumps to `malloc2_success`. Otherwise, it tries the next element in the free memory list. First, it stores the current element of the list and the return address on the stack (in line 9 and 10). Then, it calls itself recursively (from line 11 to 13). After the return from the recursive call (the instructions of the label `malloc2_cont`), it concatenates the saved element with the returned free memory list (in line 22 and 23)

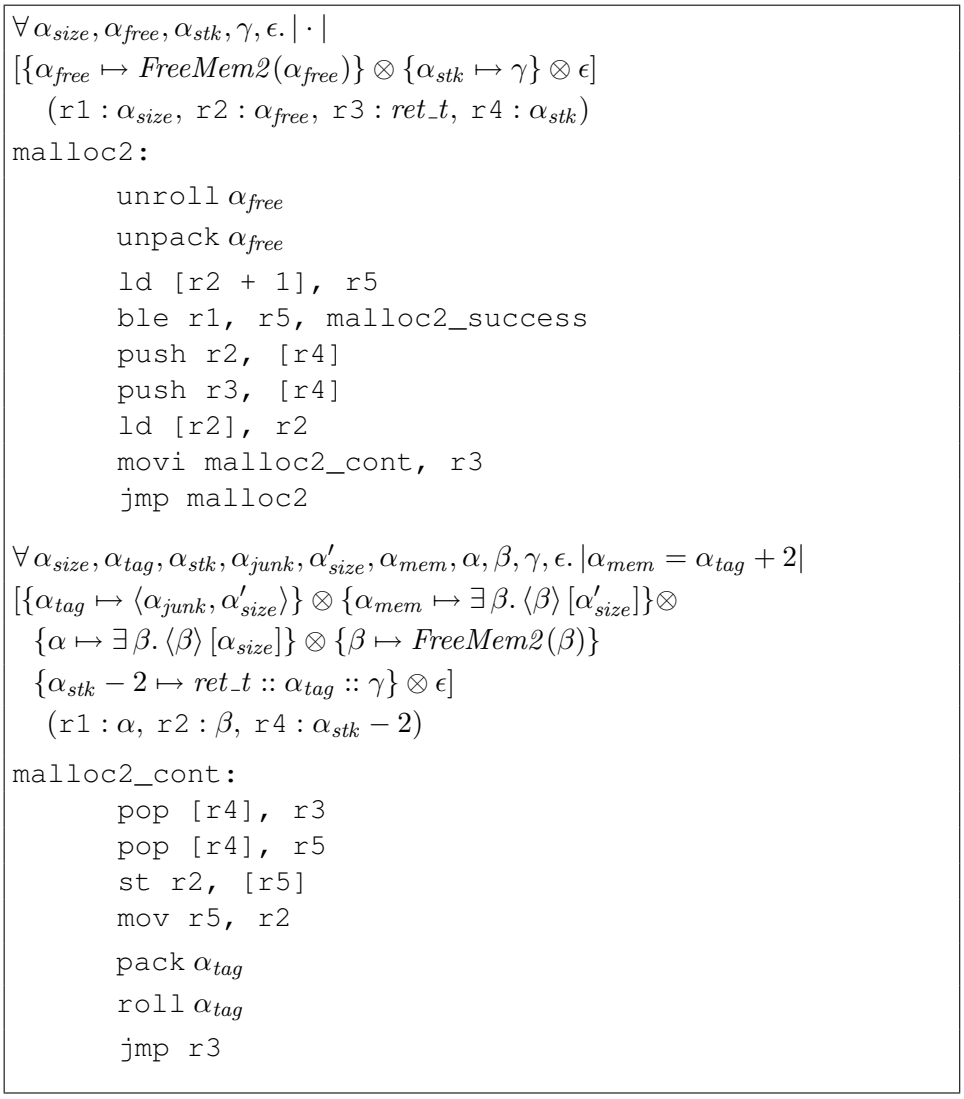

Figure 3.2: Implementation of malloc in TALK (1/2)

and returns it as a new free memory list (from line 24 to 26). Of course, the array allocated by the recursive call is also returned. Here the stack type $ret\_t :: \alpha_{tag} :: \gamma$ in the memory type (in line 17) represents a stack whose top element has type ret_t and next element has type α_{tag} and the rest is unknown (γ).

```
1  ∀ α_size, α_tag, α_stk, α_free, α'_size, α_mem, γ, ϵ. |α_size ≤ α'_size ∧ α_mem = α_tag + 2|
2  [{α_tag ↦ ⟨α_free, α'_size⟩} ⊗ {α_mem ↦ ∃ β. ⟨β⟩ [α'_size]}⊗
3    {α_free ↦ FreeMem2(α_free)} ⊗ {α_stk ↦ γ} ⊗ ϵ]
4     (r1 : α_size, r2 : α_tag, r3 : ret_t, r4 : α_stk)
5  malloc2_success:
6         split α_mem, (α'_size − α_size)
7         ld [r2 + 1], r5
8         sub r1, r5, r6
9         st r6, [r2 + 1]
10        mov r2, r5
11        add 2, r5, r5
12        add r6, r5, r1
13        pack α_tag
14        roll α_tag
15        jmp r3
16 ret_t ≡ ∀ α, β. | · |[{α ↦ ∃β. ⟨β⟩ [α_size]}⊗
17        {β ↦ FreeMem2(β)} ⊗ {α_stk ↦ γ} ⊗ ϵ]
18         (r1 : α, r2 : β, r4 : α_stk)
```

Figure 3.3: Implementation of malloc in TALK (2/2)

The code of `malloc2_success` first splits the array of the first element of the given free memory list into the array of the requested size and the rest (in line 6 of Figure 3.3). As described above, it allocates the memory from the bottom half of the array of the free memory region. The `split` instruction passes the type check of TALK because the type checker knows that the length of the array is greater (or equal) than the requested size from the label type of `malloc_success` (in line 1). Then, it rewrites the information about the unused array and its size in the second element (from line 7 to 9) and returns the allocated array (from line 10 to 15).

3.1.3 Implementation of free

Figure 3.4 is an implementation of free. First, the code converts the first two elements of the array to be freed into a tuple of size 2 (from line 7 to 11). Next, it initializes the tuple by the information of the address of the array to be freed and its size (from line 12 to 14). Then, it concatenates the tuple to the given free memory list along with the rest of the array. Last, it packs and rolls the tuple in order to make the tuple have the type $\mathit{FreeMem2}$. Specifically, in line 15, the type of the memory is

$$\begin{array}{l}\{\alpha_{mem} \mapsto \langle \alpha_{free}, \alpha_{size} - 2\rangle\} \\ \otimes\{\alpha_{mem} + 2 \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha_{size} - 2]\} \\ \otimes\{\alpha_{free} \mapsto \mathit{FreeMem2}(\alpha_{free})\} \otimes \epsilon.\end{array}$$

Next, by the `pack` instruction, the type becomes

$$\begin{array}{l}\{\alpha_{mem} \mapsto \exists\beta_{next}, \beta_{size}, \beta_{mem}.|\beta_{mem} = \alpha_{mem} + 2| \\ \quad [\{\beta_{mem} \mapsto \exists\beta.\, \langle\beta\rangle\, [\beta_{size}]\}\otimes \\ \quad\ \{\beta_{next} \mapsto \mathit{FreeMem2}(\beta_{next})\}] \\ \langle\beta_{next}, \beta_{size}\rangle\} \otimes \epsilon.\end{array}$$

Here the argument for the `pack` instruction is

$$\begin{array}{l}[\alpha_{free}, \alpha_{size} - 2, \alpha_{mem} + 2| \\ \{\alpha_{mem} + 2 \mapsto \exists\beta.\, \langle\beta\rangle\, [\alpha_{size} - 2]\} \otimes \{\alpha_{free} \mapsto \mathit{FreeMem2}(\alpha_{free})\}]\end{array}$$

(and the existential type itself). Then, by the `roll` instruction, the memory type becomes

$$\{\alpha_{mem} \mapsto \mathit{FreeMem2}(\alpha_{mem})\} \otimes \epsilon.$$

Here the argument for the `roll` instruction is $\mathit{FreeMem2}(\alpha_1)$. Thus, the function return in line 17 passes the type check of TALK, because the memory type satisfies the precondition specified in the label type of the register `r3`.

The freed memory cannot be accesses any more after the function return, because the array is removed from the memory type specified in the label type (in line 5).

3.1.4 Defragmentation

In addition to the implementation of malloc and free, this section describes the implementation of *d*efrag, which defragments the memory regions fragmented by the series of malloc and free.

```
1  ∀ α_mem, α_free, α_size, ϵ. |α_size > 2|
2  [{α_mem ↦ ∃ β. ⟨β⟩ [α_size]}⊗
3    {α_free ↦ FreeMem2(α_free)} ⊗ ϵ]
4     (r1 : α_mem, r2 : α_free, r4 : α_size,
5      r3 : ∀ α.| · |[{α ↦ FreeMem2(α)} ⊗ ϵ](r1 : α))
6  free2:
7         split α_mem, 2
8         split α_mem, 1
9         unpack α_mem
10        unpack α_mem + 1
11        tuple_concat α_mem, α_mem + 1
12        st r2, [r1]
13        sub 2, r4, r4
14        st r4, [r1 + 1]
15        pack α_mem
16        roll α_mem
17        jmp r3
```

Figure 3.4: Implementation of free in TALK

The implementation of defrag is shown in Figures 3.5, 3.6 and 3.7. The code takes a free memory list as an argument (in line 2 of Figure 3.5), defragments it, and returns it to the caller (in line 24 of Figure 3.7). The code assumes that the free memory list is sorted in ascending order with respect to the addresses of the memory regions, but the code never goes wrong if an unsorted memory list is passed. The sorting of free memory lists is straightforward, so the implementation is not shown in this thesis.

```
1  ∀ α_free, α_stk, γ, ϵ. | · |
2  [{α_free ↦ FreeMem2(α_free)} ⊗ {α_stk ↦ γ} ⊗ ϵ]
3    (r1 : α_free, r3 : ret_t, r4 : α_stk)
4  defrag:
5        unroll α_free
6        unpack α_free
7        ld [r1], r2
8        push r3, [r4]
9        movi defrag_cont, r3
10       jmp defrag_aux

11 ∀ α_free, α_stk, γ, ϵ. | · |
12 [{α_free ↦ FreeMem2(α_free)} ⊗ {α_stk − 1 ↦ ret_t :: γ} ⊗ ϵ]
13   (r1 : α_free, r4 : α_stk − 1)
14 defrag_cont:
15       pop [r4], r3
16       jmp r3
```

Figure 3.5: Implementation of defrag in TALK (1/3)

The code first unrolls and unpacks the free memory list in order to access its contents (in line 5 and 6 of Figure 3.5). Then, it calls the auxiliary function (`defrag_aux` of Figure 3.6) that takes the first memory region of the free memory list and the rest as arguments, and checks whether the first memory region is adjacent to the second memory region. If so, `defrag_aux` concatenates the two memory regions.

Specifically, the code of `defrag_aux` first calls the `defrag` function recursively (in line 9 and 10 of Figure 3.6) in order to defragment the rest of the free memory list. Careful readers might notice that the recursive call never returns because the free memory list is infinite in this presentation.

```
1  ∀ α_tag, α_free, α_size, α_stk, α_mem, γ, ϵ. |α_mem = α_tag + 2|
2  [{α_tag ↦ ⟨α_free, α_size⟩} ⊗ {α_mem ↦ ∃β. ⟨β⟩ [α_size]}
3  ⊗{α_free ↦ FreeMem2(α_free)} ⊗ {α_stk ↦ γ} ⊗ ϵ]
4     (r1 : α_tag, r2 : α_free, r3 : ret_t, r4 : α_stk)
5  defrag_aux:
6        push r1, [r4]
7        push r3, [r4]
8        mov r2, r1
9        movi defrag_aux_cont, r3
10       jmp defrag

11 ∀ α_tag, α_free, α_size, α_stk, α_mem, γ, ϵ. |α_mem = α_tag + 2|
12 [{α_tag ↦ ⟨α_free, α_size⟩} ⊗ {α_mem ↦ ∃β. ⟨β⟩ [α_size]}
13 ⊗{α_free ↦ FreeMem2(α_free)} ⊗ {α_stk − 2 ↦ ret_t :: α_tag :: γ} ⊗ ϵ]
14    (r1 : α_free, r4 : α_stk − 2)
15 defrag_aux_cont:
16       mov r1, r2
17       pop [r4], r3
18       pop [r4], r1
19       ld [r1 + 1], r5
20       add 2, r5, r5
21       add r1, r5, r5
22       beq r5, r2, defrag_concat
23       pack α_tag
24       roll α_tag
25       jmp r3
```

Figure 3.6: Implementation of defrag in TALK (2/3)

Here we do not care the infinity for ease of understanding the example.

After the recursive function call (in line 15), the code checks whether the first memory region and the second memory region are adjacent or not (from line 19 to 22). If they are not, the code simply returns the free memory list after packing and rolling it (from line 23 to 25). If they are adjacent, the code jumps to `defrag_concat` of Figure 3.7.

The code of `defrag_concat` first unrolls and unpacks the second memory region. Next, it concatenates the first memory region, the second memory region and the header of the second memory region (from line 8 to line 18). Along with the concatenation, the code collects the information about the concatenated memory region (from line 8 to 10, and from line 14 to 16). Then, the header of the first memory region is updated with the collected information so that the header reflects the newly created memory region (in line 19 and 20). Last, the code returns the free memory list to the caller after packing and rolling it.

3.1.5 Handling finite free memory

The implementation described so far assumes that the free memory is infinite. However, in a realistic situation, it is finite because the size of the physical memory is finite. This section describes how to handle the finite free memory in TALK.

Free memory list

The type of the free memory list now has the terminator by utilizing the extension for supporting variant types (explained in Section 2.6) in order to realize finite lists. Figure 3.8 shows the type definition. The precondition clause of line 3 indicates the list terminator. More specifically, if $\alpha_{next} = 0$ (that is, the first element of the tuple of line 6 is 0), then there exists no following element any more as indicated by the type of the encapsulated memory inside the existential type of line 2.

A subtle problem of the type definition of Figure 3.8 is that it does not match the free memory list passed by the BIOS program to the initial kernel routine. Therefore we need to convert the passed free memory list. Although the conversion itself is easy, we cannot write it in TALK because the free memory list passed by the BIOS adopts the 64-bit addressing, while TALK for IA-32 supports only the 32-bit addressing. Thus, the initial kernel routine written in an ordinary untyped IA-32 assembly language converts it after initializing the segment and paging mechanisms.

```
1  ∀ α_tag, α_free, α_size, α_stk, α_mem, γ, ϵ. |α_mem = α_tag + 2 ∧ α_free = α_mem + α_size|
2  [{α_tag ↦ ⟨α_free, α_size⟩} ⊗ {α_mem ↦ ∃β. ⟨β⟩ [α_size]}
3  ⊗{α_free ↦ FreeMem2(α_free)} ⊗ {α_stk ↦ γ} ⊗ ϵ]
4     (r1 : α_tag, r2 : α_free, r3 : ret_t, r4 : α_stk)
5  defrag_concat:
6          unroll α_free
7          unpack α_free with α'_next, α'_size, α'_mem
8          ld [r2 + 1], r5
9          add 2, r5, r5
10         ld [r2], r2
11         tuple_split α_free, 1
12         pack α_free
13         pack α_free + 1
14         ld [r1 + 1], r6
15         add r6, r5, r5
16         concat α_free, α_free + 1, 1
17         concat α_free, α_free + 2, α'_size
18         concat α_mem, α_free, (α_size + 2 + α'_size)
19         st r2, [r1]
20         st r5, [r1 + 2]
21         pack α_tag
22         roll α_tag
23         jmp r3
24 ret_t ≡ ∀ α. | · |{α ↦ FreeMem2(α)} ⊗ {α_stk ↦ γ} ⊗ ϵ]
25         (r1 : α, r4 : α_stk)
```

Figure 3.7: Implementation of defrag in TALK (3/3)

```
1 FreeMem ≡
2   μη [α_self] .∃ α_next, α_size, α_mem.
3    (|α_next = 0 ∧ α_mem = α_self + 2| [{α_mem ↦ ∃β. ⟨β⟩ [α_size]}])
4    (|α_next ≠ 0 ∧ α_mem = α_self + 2|
5     [{α_mem ↦ ∃β. ⟨β⟩ [α_size]} ⊗ {α_next ↦ η (α_next)}])
6      ⟨α_next, α_size⟩
```

Figure 3.8: Type of the free memory (finite list of variable-length arrays)

Memory allocation

The core algorithm of `malloc` does not change from Section 3.1.2. The only non-trivial difference is error-handling. If the memory region of the requested size cannot be allocated from the free memory list, `malloc` must indicate it to the caller. Therefore, the label type of `malloc` is changed as Figure 3.9 (its implementation is also changed slightly according to the type). The precondition clause of the label type of the return address (in line 4) indicates the error case. More specifically, if $\alpha = 0$, that is, the register `r1` is 0, then the memory type after `malloc` (in line 4) is almost the same as that of before `malloc` (in line 2), that is, no memory regions are allocated from the free memory list.

```
1 ∀ α_size, α_free, α_stk, γ, ϵ. | · |
2 [{α_free ↦ FreeMem(α_free)} ⊗ {α_stk ↦ γ} ⊗ ϵ]
3    (r1 : α_size, r2 : α_free, r3 : ret_t, r4 : α_stk)
4 ret_t ≡ ∀ α, β.(|α = 0| [{β ↦ FreeMem(β)} ⊗ {α_stk ↦ γ} ⊗ ϵ])
5                  (|α ≠ 0| [{α ↦ ∃β. ⟨β⟩ [α_size]}⊗
6                      {β ↦ FreeMem(β)} ⊗ {α_stk ↦ γ} ⊗ ϵ])
7                       (r1 : α, r2 : β, r4 : α_stk)
```

Figure 3.9: The label type of `malloc`

Concretely speaking, if sufficient memory regions are not found in the free memory list, `malloc` calls `defrag` in order to make the free memory regions (in the free memory list) as large as possible. Then, it retries the allocation. If it fails again, `malloc` gives up and returns 0 to the caller for indicating the request is unsatisfiable.

3.2 Multi-thread management

In this section, we show how to implement practical multi-thread management mechanisms in TALK. First, we describe how to implement thread-local storage by showing a context-switching function. Then, we explain how to create, schedule, and synchronize threads.

In the current implementation of the kernel, threads are represented as pairs of a memory stack and the pointer to it. Therefore, all the code of the multi-thread management mechanisms is completely written in TALK (except for the initialization of the very first thread).

3.2.1 Context switching function with thread-local storage

Although the context-switching function shown in the previous chapter (Section 2.5.3) works, it has one problem; it does not support thread-local storage. In the implementation, the running thread and the thread to be run must have the same memory type (except for their stacks). That is, all the memory is shared by all the threads on the system.

The code in Figure 3.10 is a more practical context-switch function that solves the problem by explicitly handling the thread-local storage. The ordinary instructions are unchanged from the implementation of Figure 2.35. Only the types and the arguments for the coerce instructions are changed.

In the memory type specified in the label type of the instructions (from line 1 to 3), the memory type variable ϵ_g represents the memory shared by all threads and the memory type variable ϵ_l represents the thread-local storage. *thd_t* ensures the thread-locality of ϵ_l because it encapsulates the memory represented by ϵ_l into the existential type (in line 13). Thus, the thread-local storage of a thread cannot be accessed directly by other threads.

The code is type checked as follows. First, after unpacking the thread context (in line 5), the memory type becomes

$$\{\alpha_{next} \mapsto \langle \alpha'_{stk} \rangle\} \otimes \{\alpha'_{stk} \mapsto pc\_t :: \gamma'\} \otimes \{\alpha_4 \mapsto pc\_t :: \gamma\} \otimes \epsilon_g \otimes \epsilon_l \otimes \epsilon'_l.$$

Then, after switching the thread contexts (in line 8), the memory type becomes

$$\{\alpha_{next} \mapsto \langle \alpha_4 \rangle\} \otimes \{\alpha'_{stk} \mapsto pc\_t :: \gamma'\} \otimes \{\alpha_4 \mapsto pc\_t :: \gamma\} \otimes \epsilon_g \otimes \epsilon_l \otimes \epsilon'_l.$$

and the type of the register `r4` becomes α'_{stk}. Then, after popping the new program counter from the new stack (in line 9), the type of the register `r3`

```
1  ∀ α_next, α_stk, γ, ε_g, ε_l.
2     [{α_next ↦ thd_t} ⊗ {α_stk ↦ pc_t :: γ} ⊗ ε_g ⊗ ε_l]
3      (r1 : α_next, r4 : α_stk)
4  context_switch:
5          unpack α_next with α'_stk, γ', ε'_l
6          mov r4, r5
7          ld [r1], r4
8          st r5, [r1]
9          pop [r4], r3
10         pack α_next
11         apply r3 [α_next, α'_stk + 1, γ', ε'_l/α, β, γ, ε_l]
12         jmp r3
13 thd_t ≡ ∃ α, γ, ε_l.[{α ↦ pc_t :: γ} ⊗ ε_l] ⟨α⟩
14 pc_t ≡ ∀ α, β, γ, ε_l.[{α ↦ thd_t} ⊗ {β ↦ γ} ⊗ ε_g ⊗ ε_l]
15    (r1 : α, r4 : β)
```

Figure 3.10: Example code of switching contexts with thread-local storage

becomes pc_t, the type of the register `r4` becomes $\alpha'_{stk} + 1$ and the memory type becomes

$$\{\alpha_{next} \mapsto \langle\alpha_4\rangle\} \otimes \{\alpha'_{stk} + 1 \mapsto \gamma'\} \otimes \{\alpha_4 \mapsto pc\_t :: \gamma\} \otimes \epsilon_g \otimes \epsilon_l \otimes \epsilon'_l.$$

Next, after packing the thread context (in line 10), the memory type becomes

$$\{\alpha_{next} \mapsto thd\_t\} \otimes \{\alpha'_{stk} + 1 \mapsto \gamma'\} \otimes \epsilon_g \otimes \epsilon'_l.$$

Here the argument for the `pack` instruction is $[\alpha_4, \gamma, \epsilon_l | \{\alpha_4 \mapsto pc\_t :: \gamma\} \otimes \epsilon_l]$ as thd_t. Next, the label type of the return address is instantiated in line 11. Then, the label type becomes

$$\forall. \left[\{\alpha_{next} \mapsto thd\_t\} \otimes \{\alpha'_{stk} + 1 \mapsto \gamma'\} \otimes \epsilon_g \otimes \epsilon'_l\right] \left(\mathtt{r1} : \alpha_{next}, \mathtt{r4} : \alpha'_{stk} + 1\right).$$

Now the precondition indicated by the above label type is satisfied by the current memory and registers type. Thus, the last `jmp` instruction is type checked successfully.

3.2.2 Creating threads

Creation of threads is almost the same as allocation of memory stacks. The thread creation routine of the kernel takes a code label which represents an entry point of the thread and an integer which represents a size of the thread stack as arguments. Then, it allocates a stack of the specified size by `malloc` and initializes it using the specified entry label. Then, the created thread is appended to a run queue which is just a linked list of threads. Figure 3.11 shows the type of the thread creation routine and Figure 3.12 shows the type of the run queue (a linked list of threads).

$$
\begin{array}{l}
\forall\, \alpha_{size}, \alpha_{free}, \alpha_{stk}, \gamma, \epsilon_g \epsilon_l.\, |\cdot| \\
[\{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon_g \otimes \epsilon_l] \\
\quad (\mathtt{r1} : \alpha_{size},\ \mathtt{r2} : \alpha_{free},\ \mathtt{r3} : \mathit{ret\_t},\ \mathtt{r4} : \alpha_{stk},\ \mathtt{r5} : \mathit{ent\_t}) \\
\mathit{ret\_t} \equiv \forall\, \alpha, \beta. (|\alpha = 0|\ [\{\beta \mapsto \mathit{FreeMem}(\beta)\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon_g \otimes \epsilon_l]) \\
\qquad\qquad (|\alpha \neq 0|\ [\{\alpha \mapsto \mathit{thd\_t}\} \otimes \\
\qquad\qquad\quad \{\beta \mapsto \mathit{FreeMem}(\beta)\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon_g]) \\
\qquad\qquad\quad (\mathtt{r1} : \alpha,\ \mathtt{r2} : \beta,\ \mathtt{r4} : \alpha_{stk}) \\
\mathit{thd\_t} \equiv \exists\, \alpha, \gamma, \epsilon_l. [\{\alpha \mapsto \mathit{pc\_t} :: \gamma\} \otimes \epsilon_l]\, \langle \alpha \rangle \\
\mathit{pc\_t} \equiv \forall\, \alpha, \beta, \gamma, \epsilon_l. [\{\alpha \mapsto \mathit{thd\_t}\} \otimes \{\beta \mapsto \gamma\} \otimes \epsilon_g \otimes \epsilon_l] \\
\quad (\mathtt{r1} : \alpha,\ \mathtt{r4} : \beta)
\end{array}
$$

Figure 3.11: The label type of the thread creation routine

$$
\begin{array}{l}
\mathit{ThdList} \equiv \\
\quad \mu\eta.\ \exists\, \alpha_{next}, \alpha_{thd}. \\
\quad (|\alpha_{next} = 0|\ [\{\alpha_{thd} \mapsto \mathit{thd\_t}\}]) \\
\quad (|\alpha_{next} \neq 0|\ [\{\alpha_{thd} \mapsto \mathit{thd\_t}\} \otimes \{\alpha_{next} \mapsto \eta\}]) \\
\qquad \langle \alpha_{next}, \alpha_{thd} \rangle
\end{array}
$$

Figure 3.12: Type of the run queue (finite list of threads)

Initializing the first thread As explained above, the thread creation routine is written in TALK, except for the creation of the very first thread. At first glance, the routine seems to be able to create it because there is no difference between it and other threads. However, it is impossible because

`malloc` used by the routine for creating a thread stack requires a memory stack. That is, `malloc` cannot be used without a thread context. Therefore, the routine for the first thread creation is written in the untyped IA-32 assembly language.

3.2.3 Scheduling threads

To schedule threads, all the kernel has to do is to take one thread from the run queue and let the taken thread run by using the context-switching function of Section 3.2.1. The current thread scheduler adopts a simple FIFO scheduling policy, that is, it takes a top thread from the run queue, runs it by context-switching it and the current thread, and appends the old-current thread to the run queue. The type of the scheduler is shown in Figure 3.13.

$$
\begin{array}{l}
1 \quad \forall\, \alpha_{cur}, \alpha_{thds}, \alpha_{stk}, \gamma, \epsilon_g \epsilon_l .\, |\cdot| \\
2 \quad [\{\alpha_{cur} \mapsto \exists \alpha.\, \langle \alpha \rangle\} \otimes \{\alpha_{thds} \mapsto \mathit{ThdList}\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon_g \otimes \epsilon_l] \\
3 \quad\quad (\mathtt{r1} : \alpha_{cur},\ \mathtt{r2} : \alpha_{thds},\ \mathtt{r3} : \mathit{ret\_t},\ \mathtt{r4} : \alpha_{stk}) \\
4 \quad \mathit{ret\_t} \equiv \forall\, \alpha, \beta .| \cdot | \\
5 \quad\quad [\{\alpha \mapsto \exists \alpha.\, \langle \alpha \rangle\} \otimes \{\beta \mapsto \mathit{ThdList}\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon_g \otimes \epsilon_l] \\
6 \quad\quad\quad (\mathtt{r1} : \alpha,\ \mathtt{r2} : \beta,\ \mathtt{r4} : \alpha_{stk})
\end{array}
$$

Figure 3.13: The label type of the thread scheduler

Note that the type of the scheduler indicates that the memory type never changes while calling the routine, that is, the routine seems to do nothing from the viewpoint of its caller.

The type itself, however, does not ensure that there will be no starvation condition, that is, the threads in the run queue will be executed eventually. For example, the type can be satisfied by a null function which does nothing. From the viewpoint of multi-thread management, the null function is regarded as a valid scheduler which always selects the running thread. Therefore, to prove starvation-freedom, we must refine the type system of TALK, or use other formal methods (e.g., model checkers and proof assistants), but it is out of the scope of this thesis.

3.2.4 Synchronizing threads

Using the thread scheduler described above and the variant type extension of TALK, a simple synchronized data structure for synchronizing threads can be implemented as shown in Figure 3.14. The data structure is a simple tuple of size 2. The first element (α_{locked}) represents a synchronization lock for another tuple pointed by the second element (α). If $\alpha_{locked} = 0$, the data structure is not locked. Otherwise, it is locked. As shown in the type of the memory encapsulated in the existential type, the tuple at the address α is hidden from programs, that is, they cannot access it.

$$
\begin{array}{l}
1 \quad \mathit{SynchData} \equiv \exists\, \alpha_{locked}, \alpha. \\
2 \quad\quad (|\alpha_{locked} = 0|\ [\{\alpha \mapsto \langle \alpha \rangle\}]) \\
3 \quad\quad (|\alpha_{locked} \neq 0|\ [\,]) \\
4 \quad\quad\quad \langle \alpha_{locked}, \alpha \rangle
\end{array}
$$

Figure 3.14: Type of a simple synchronized data structure

$$
\begin{array}{l}
1 \quad \forall\, \alpha_{synch}, \alpha_{stk}, \gamma, \epsilon.\, |\cdot| \\
2 \quad\quad [\{\alpha_{synch} \mapsto \mathit{SynchData}\} \otimes \{\alpha_{stk} \mapsto \mathit{ret\_t} :: \gamma\} \otimes \epsilon] \\
3 \quad\quad\ (\texttt{r1} : \alpha_{synch},\ \texttt{r4} : \alpha_{stk}) \\
4 \quad \texttt{lock:} \\
5 \quad\quad\quad\ \texttt{unpack}\ \alpha_{synch} \\
6 \quad\quad\quad\ \texttt{ld [r1], r3} \\
7 \quad\quad\quad\ \texttt{bne 0, r3, lock\_failed} \\
8 \quad\quad\quad\ \texttt{st 1, [r1]} \\
9 \quad\quad\quad\ \texttt{pop [r4], r3} \\
10 \quad\quad\quad\ \texttt{jmp r3} \\
11 \quad \mathit{ret\_t} \equiv \forall\, \alpha, \alpha_{stk}. \\
12 \quad\quad [\{\alpha_{synch} \mapsto \langle 1, \alpha \rangle\} \otimes \{\alpha \mapsto \langle \alpha \rangle\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon] \\
13 \quad\quad\ (\texttt{r4} : \alpha_{stk})
\end{array}
$$

Figure 3.15: An example routine for locking synchronized data (1/2)

Figures 3.15 and 3.16 show an example routine for locking the synchronized data. The type of the return address ($\mathit{ret\_t}$) indicates that the tuple encapsulated in the synchronized data is available for the caller of the routine

```
1  ∀ α_synch, α_locked, α, α_stk, γ, ϵ. |α_locked ≠ 0|
2    [{α_synch ↦ ⟨α_locked, α⟩} ⊗ {α_stk ↦ ret_t :: γ} ⊗ ϵ]
3     (r1 : α_synch, r4 : α_stk)
4  lock_failed:
5        pack α_synch
6        push r1, [r4]
7        push lock_failed_cont, [r4]
8        jmp scheduler
9  ∀ α_synch, α_stk, γ, ϵ. |·|
10   [{α_synch ↦ SynchData} ⊗ {α_stk ↦ α_synch :: ret_t :: γ} ⊗ ϵ]
11    (r4 : α_stk)
12 lock_failed_cont:
13       pop [r4], r1
14       jmp lock
```

Figure 3.16: An example routine for locking synchronized data (2/2)

(in line 12), that is, the routine returns to the caller only after the synchronized data is locked. The routine of Figure 3.15 first unpacks the synchronized data (in line 5) and checks whether it is locked or not (in line 7). If the synchronized data is not locked, the routine rewrites the lock variable (in line 8) and returns to the caller (in line 9 and 10). If the synchronized data is locked, it jumps to the routine of Figure 3.16. The routine first packs the synchronized data (in line 5) and calls the thread scheduler (in line 7 and 8) which satisfies the label type of Figure 3.13, that is, it makes the running thread yield to other threads. After the thread is rescheduled, the routine tries again by jumping to the routine of Figure 3.15.

Figure 3.17 shows an example routine for unlocking the locked data. First, the routine clears the lock (in line 5) and packs the data (in line 6). Then, it just returns to the caller (in line 7 and 8). The label type of the routine indicates that the tuple $\langle\alpha\rangle$ at the address α is encapsulated to the type *SynchData* after the routine, that is, the synchronized data is unlocked.

```
1  ∀ α_synch, α_locked, α, α_stk, γ, ϵ. |α_locked = 1|
2     [{α_synch ↦ ⟨α_lock, α⟩} ⊗ {α ↦ ⟨α⟩} ⊗ {α_stk ↦ ret_t :: γ} ⊗ ϵ]
3      (r1 : α_synch, r4 : α_stk)
4  unlock:
5          st 0, [r1]
6          pack α_synch
7          pop [r4], r3
8          jmp r3
9  ret_t ≡ ∀ α_stk.
10    [{α_synch ↦ SynchData} ⊗ {α_stk ↦ γ} ⊗ ϵ]
11    (r4 : α_stk)
```

Figure 3.17: An example routine for unlocking synchronized data

3.3 Boot procedures

Generally speaking, boot procedures of systems require dynamic program loading and self-rewriting code. For example, typical IA-32 systems first execute a special program called BIOS (The BIOS program is typically hard-wired to the systems as ROM). Then, the BIOS program searches a disk drive (including hard disk drives and optical drives) which contains a valid boot sector, which is part of an operating system. The problem is that the size of the boot sector is limited to 512 bytes. Therefore, if the size of the OS kernel is larger than 512 bytes, the program in the boot sector must load other part of the OS kernel to the memory of the system (i.e., dynamic program loading). In addition, the memory image of the initial boot sector is typically useless, after the kernel is fully loaded to the memory. Thus, the typical OS kernel reuses the memory occupied by the boot sector as free memory (i.e., self-rewriting code).

Because the TALK type system does not support the dynamic program loading or self-rewriting code currently, we give up writing the boot procedure in TALK and use the GRUB boot loader [37], which is one of the most popular boot loaders. One benefit of using GRUB is that we do not need to care about the complex boot procedures, because GRUB can load whole kernel image from disk drives and place it in the memory of the system. Another benefit is that it provides useful debug information to the kernel developers. For example, GRUB warns users if the kernel image to

be loaded is invalid with respect to the specification [38] defined by GRUB.

Theoretically, it is not impossible to extend the type system of TALK in order to support the dynamic program loading and the self-rewriting code. The extension becomes important when considering kernel modules. Loading the kernel modules into the kernel is just the dynamic program loading, and unloading them is easier than the self-rewriting code.

3.3.1 IA-32 specific boot procedures

As described above, we use the GRUB boot loader for loading whole kernel image to the memory. Thus, large part of burden of the boot procedures is shifted from the kernel to GRUB. However, there still remain confusing procedures that are peculiar to the IA-32 architecture. This section describes the remaining procedures. As expected, they cannot be written in TALK.

Segments The worst thing of the IA-32 architecture is the segmented memory model. The memory model of IA-32 consists of two memory models (In fact, three, if we consider the paging mechanism described later). At the lowest level, the memory is flat in the sense that memory addressing is represented as just one integer. For example, the memory addressing with the integer 10000 will access the 10000th byte data in the memory (if it exists). This "flat memory model" is very simple and natural. Therefore, almost all the current CPU architectures adopt the flat model. The type system of TALK also assumes the flat memory model as described in Section 2.2.

However, on the IA-32 CPUs, programs cannot directly see the flat memory because there exists an extra memory model over the flat memory model. In the extra memory model, the memory consists of "segments". Each segment is a contiguous memory region. In the memory model, memory addressing is a pair of a segment selector, which specifies a segment in the memory, and an offset into the segment specified by the segment selector.

All the segments in the memory are mapped to the underlying flat memory by the tables called GDT (Global Descriptor Table) and LDT (Local Descriptor Table). The kernel must prepare GDT (LDT is optional) and pass it to the CPU. For example, suppose that the kernel sets GDT so that a segment whose size is 10000 bytes is mapped to the address 20000 of the flat memory model. Then, the memory addressing that specifies the offset 1000 into the segment is automatically converted to the flat memory addressing 21000 by the CPU.

Thus, what the kernel has to do first is prepare GDT (and LDT). Otherwise, the kernel cannot make any valid memory access, because the memory model that programs can use on IA-32 CPUs is only the segmented memory model. In fact, the segmented memory model can be disabled virtually by setting GDT so that a segment whose size is 4GB (= the largest memory address of the flat memory model) is mapped at the address 0 of the flat memory model. Then, the offset into the address becomes equal to the address of the flat memory model. Even so, the kernel still needs to prepare GDT for virtually disabling the segmented memory model.

What is worse is that the existence of the paging (virtual memory) mechanism. The paging mechanism of the IA-32 architecture can be enabled only after GDT is configured properly. Therefore, after enabling the paging, the kernel must reconfigure GDT. This is because the paging mechanism introduces another flat memory model between the segmented memory model and the flat memory model.

We believe that it is useless (though not impossible) to extend the type system of TALK for supporting these historical relics. Thus, the part of the kernel that initializes GDT (twice) is not written in TALK. Because Intel and other industry companies, such as IBM, Microsoft and AMD, are working for establishing a standard for more cleaner boot procedures [20, 34], the bizarre characteristic will be resolved in the near future.

Surprisingly enough, the procedures described above for the segment memory model is only half of the story. The GRUB boot loader deals with the other half (including transition from 16-bit execution mode to 32-bit execution mode).

Paging Unlike the segments, the paging mechanism (or the virtual memory mechanism) of IA-32 is almost a standard one. When the paging is enabled, the virtual flat memory space is stacked on the physical flat memory space. The mapping between the virtual memory space and the physical memory space is specified with page tables. Thus, the kernel needs to prepare the page tables before enabling the paging mechanism.

The peculiarity of the IA-32 paging mechanism is that it supports different kinds of page table formats. For example, the simplest format is just a standard two-level page table. In the format, the page size is fixed to 4KB. Another format is also almost a two-level page table, but it supports the 4MB page, as well as the 4KB page. In addition to these two-level page table formats, IA-32 supports three-level page tables in order to handle big physical memory (64GB or above).

Currently, the kernel utilizes the simplest page table format (two-level and 4KB pages) for the ease of implementation. At the boot time, the kernel initializes a page table so that it maps the physical memory address 0 to the virtual memory address 0xc0000000. (The value 0xc0000000 itself has no special meaning. We just followed the design decision of the Linux kernel.) Therefore, the kernel cannot handle big memory larger than 1GB, but it is sufficiently large for the prototype implementation. The initialization of the page table is currently not written in TALK.

Once the page table initialized, the memory region in which the page table resides is completely hidden from the TALK programs by the type system, because the current kernel does not support user processes, so it is unnecessary to manipulate the initialized page table. More specifically, the memory type of the initial TALK routine does not contain the memory region. Thus, the memory safety is ensured because the flat memory model assumption of TALK is never violated by erroneous page table manipulations.

3.4 Device drivers

The current kernel includes simple video and keyboard drivers that are written in TALK. This section describes their implementation.

At first glance, writing device drivers in typed programming languages seems to be difficult because there exist many kinds of devices and each of them has different characteristics. On reflection, however, what the device drivers do basically is to transfer data from the memory of the system to the devices and/or vice versa, no matter how the devices behave intricately. Therefore, it is not so difficult to write them in TALK and ensure their memory and control-flow safeties with the type check of TALK.

3.4.1 Video driver

The video system of the IA-32 system (more precisely, the PC/AT system) is based on memory-mapped I/O. That is, for drawing a dot, a line, and/or a character on the screen of the system, all the video driver has to do is to write data to a special memory region which is mapped to VRAM (video ram) of the video device. The current video system of PC/AT supports various display resolutions (from 320 x 200 pixels to 7680 x 4800 pixels) and color depths (from 1 bpp to 32 bpp), but the current video driver of the kernel supports only a simple graphic mode which is the default of BIOS. In

theory, other graphic modes can be easily supported without any problem.

More specifically, the graphic mode that the current video driver supports can display 80 x 25 characters in up to 16 colors on the screen of size 640 x 200 pixels. Its video memory is mapped to the address 0xb8000 and its size is 4000 bytes (= 80 x 25 x 2). To print a character, all the video driver has to do is just to write two bytes in the video memory. The first byte specifies the character and the second byte specifies the attributes (foreground and background colors, blinking and intensity). For example, to print 'A' with white (foreground color) and black (background color) at the location (12, 8) on the screen, the driver has to write the value 41 (which represents 'A' in ASCII code) to the address 0xb8518 and the value 7 to the address 0xb8519.

Figure 3.18 shows a routine which prints a character on the screen. The memory type (in line 2) indicates that the video memory is located at the address `0xb8000` and the type of the video memory is represented as the array of size 4000. (Note that we assume the size of each array element is 1 byte in this example.) From the viewpoint of programs, the video memory is almost the same as ordinary physical memory. Therefore, it is natural to represent the video memory as a fixed-length array. First, the routine takes a character to be printed and its location on the screen as arguments (in line 3). The integer constraints of its label type ensures that the location is surely inside the video memory (in line 1). Then, the routine calculates the corresponding address in the video memory from the specified location (from line 5 to 7). Next, it splits the video memory in order to access the calculated memory address (from line 8 to 10). Then, it unpacks the two bytes chopped off with `split` (in line 11 and 12). Next, it writes the specified character data to the calculated address and the attribute value 7 to the next address (from line 13 to 15). It makes the video device print the character with white (foreground color) and black (background color) on the screen. Then, it restores the video memory to its original array type with `pack` and `concat` (from line 16 to 20). Last, it returns to the caller (in line 21 and 22).

3.4.2 Keyboard driver

In the PC/AT system, keyboards are controlled by KBC (keyboard controller) named 8042. KBC is connected to CPU with I/O ports. Therefore, we first explain them.

The I/O ports can be regarded as a particular kind of memory. For example, if a CPU writes data to an I/O port, the data is transparently trans-

```
1  ∀ α_c, α_x, α_y, α_stk, γ, ϵ. |0 ≤ α_x < 80 ∧ 0 ≤ α_y < 25|
2     [{0xb8000 ↦ ∃α. ⟨α⟩ (4000)} ⊗ {α_stk ↦ ret_t :: γ} ⊗ ϵ]
3      (r1 : α_c, r2 : α_x, r3 : α_y, r4 : α_stk)
4  print_char:
5        mul r3, 160, r3
6        add r2, r3, r2
7        add r2, 0xb8000, r2
8        split 0xb8000, α_off
9        split 0xb8000 + α_off, 2
10       split 0xb8000 + α_off, 1
11       unpack 0xb8000 + α_off
12       unpack 0xb8000 + α_off + 1
13       st r1, [r2]
14       add r2, 1, r2
15       st 7, [r2]
16       pack 0xb8000 + α_off
17       pack 0xb8000 + α_off + 1
18       concat 0xb8000 + α_off, 0xb8000 + α_off + 1, 1
19       concat 0xb8000 + α_off, 0xb8000 + α_off + 2, 3998 − α_off
20       concat 0xb8000, 0xb8000 + α_off, 4000 − α_off
21       pop [r4], r3
22       jmp r3

23 α_off ≡ α_x + α_y ∗ 160
24 ret_t ≡ ∀ α_stk.[{0xb8000 ↦ ∃α. ⟨α⟩ (4000)} ⊗ {α_stk ↦ γ} ⊗ ϵ]
25    (r4 : α_stk)
```

Note that we assume that the size of the existential package ($\exists\alpha.\ \langle\alpha\rangle$) is 1 byte in this example.

Figure 3.18: A routine which prints a character on the screen

fered to the device connected to the port. In addition, if a device writes data to an I/O port, a CPU can read the data from it. Therefore, the semantics of the I/O ports are different from one device to another. To cope with the diversity, the type system of TALK for IA-32 handles the I/O ports conservatively. More specifically, it assumes that data written to I/O ports are forgotten, that is, successive reads for the I/O ports may not see the written data. In addition, the type system makes some I/O ports are read-only or write-only, according to the specification of the devices.

KBC of the IA-32 system interacts with keyboards as follows. If a key is pressed or released, a keyboard notifies KBC. Then, KBC writes the information of the key and whether it is pressed or released, into the I/O port of the address 0x60. Next, KBC writes its status information which indicates that there is new available data, into the I/O port of the address 0x64. Last, KBC interrupts CPU to announce that the status of KBC has been changed.

As expected from the above description of KBC, all the keyboard driver has to do is to read the I/O port 0x64 to get the status of KBC, then read the I/O port 0x60 to get the information about the key pressed or released, if the KBC status indicates that there is new data. Thus, the keyboard driver is written completely in TALK. One problem of the current implementation of the keyboard driver is that it performs polling, that is, it reads the I/O port 0x64 periodically. This is because the type system of TALK does not directly support interrupts.

Chapter 4

Related work

4.1 Hardware protection

4.1.1 Microkernel

Traditional approaches of trying to ensure safety of OS kernels utilize protection mechanisms of CPU hardware. For example, microkernels [1, 40, 85, 7, 44, 16, 30, 63] try to minimize trusted computing base by executing OS components in non-privileged protection domains enforced by CPU hardware, as much as possible. For example, in the Mach [1] operating system, device drivers, file systems and part of virtual memory managers can be executed in the user-mode of CPU. For another example, in the approach of Exokernel [30], device drivers, file systems, part of virtual memory managers, interprocess communication mechanisms and thread schedulers can be executed in the user-mode.

The strength of the approach of microkernels is that we need not to ensure and/or verify safety of OS components executed in the non-privileged protection domains because it is ensured by the protection mechanism of CPU.

On the other hand, there are four problems in the microkernel approach. First, even in microkernel systems, basic memory management and multithread management code are executed in a privileged protection domain. Therefore, their memory and control-flow safety cannot be ensured at all. Second, the approach can detect errors of the OS components executed in the non-privileged protection domains at runtime, but cannot prevent them from failing. For example, suppose that a file system driver executed in the user-mode makes an invalid memory access. Then, CPU detects the error and lets the microkernel kill the driver. Thus, the kernel is surely protected

from the error, but users cannot access files managed by the driver anymore. Third, the cost of communication between protection domains, that is, the microkernel and the OS components executed in the non-privileged protection domains, is not negligible. For example, in the early version of Mach 3 operating system, overall performance was degenerated up to 66% compared to a traditional kernel. L4 microkernel [63] dramatically reduced the performance degeneration of the microkernel approach [45], but it sacrifices safety of the microkernel and still adds a little overhead. Forth, the unit of the protection domains of CPU is sometimes too coarse to protect data. For example, in the IA-32 architecture [21], the unit is 4 KB basically. That is, a 4 KB protection domain may be required to protect 1 byte data, in the worst case. Mondriaan Memory Protection [100] can manage protection domains at the granularity of machine words, but there exists no CPUs that implement Mondriaan Memory Protection so far.

The approach of this thesis does not suffer these problems. First, the type system of TALK is expressive and powerful enough to write memory management and multi-thread management code. Therefore, their memory safety and control-flow safety is ensured by the type check of TALK. Second, programs that pass the type check of TALK never cause runtime errors. Third, the approach does not incur any runtime overhead because the type check of TALK is performed statically, not at runtime. Forth, the unit of protection is sufficiently fine-grained because the type system of TALK can handles byte data. The disadvantage of the approach, compared to the microkernel approach, is that programs must be built with TALK. Thus, it is difficult to directly apply the approach to existing programs.

Note that the approach of this thesis does not depend on any hardware protection mechanism. Therefore, it can be applied to limited computing-environments that lack the hardware protection mechanisms, such as embedded systems.

4.1.2 Virtualization

Virtualization [15, 28, 99, 5, 13, 79] is a technique that creates virtual machines (VM) on real hardware by software [13, 79] or combination of software and hardware assists [15, 28, 99, 5, 22]. The VMs are carefully controlled by a privileged program called virtual machine monitor (VMM) so that programs cannot notice whether they are running in the real hardware or not. For example, Bochs [13] creates VMs by emulating IA-32 CPU in a pure software approach. The VMM of Bochs is just an IA-32 interpreter which runs in the user-mode of CPU. For another example, VMware [28]

and Xen [5] creates VMs by utilizing the protection mechanism of the IA-32 architecture. Basically, the VMM of VMware and Xen is a controller of the protection mechanism which must be run in the kernel-mode.

The good point of the virtualization is that, even if a program in a VM behaves badly and crashes the VM, other VMs and the real hardware are not affected. Therefore, safety of the real hardware are ensured by running OSes in VMs without verifying their safety.

The problem of the virtualization is that it cannot prevent runtime errors from occurring, as in the case of the microkernel approach. In fact, the argument about the microkernel is applicable to the approach of virtualization by substituting VMM for microkernel. In addition, the cost of running many VMs on the real hardware is considerable.

4.2 Model checking

Model checking is a method of formally verifying finite models of programs according to specified formal specifications. To put it bluntly, what model checking does is to execute the target program, trace all the possible states of the execution and checks whether the specified property is satisfied. Thus, in theory, model checking can verify any safety property, including the memory safety and the control-flow safety, though it is semi-algorithm because target programs may have infinite states in their execution. In practice, however, it does not make much sense because it is nothing more than finding bugs in a program by executing it. That is, the naive model checking cannot be applied to large programs because of the state explosion problem. Therefore, existing model checkers (e.g., [51, 4, 48]) abstracts the program states and the transition between them (that is, execution) in order to avoid the state explosion. The disadvantage of the abstraction is that it limits the range of safety properties that are verifiable. Note that, despite the effort of the abstraction, existing model checkers still face with the state explosion problem. Therefore, the cost of verification (CPU time and memory consumption) tends to be considerably larger than that of the type checking of TALK. The rest of this section describes three existing model checkers.

SPIN SPIN [51] is a model checker for distributed programs (e.g., processes and threads in OS) which can verify linear temporal logic (LTL) properties of them. For example, it can verify deadlock and livelock freedom of programs. Thus, as for the comparison of verifiable safety proper-

ties, the SPIN model checker seems to be superior to the approach of this thesis, at first glance. However, there are two problems in its approach.

First problem is that it can only handle programs written in the PROMELA programming language [51]. Therefore, to verify programs written in other languages, they must be translated to PROMELA first. This means that the soundness of the verification depends on the correctness of translation. That is, if there is a bug in the translation, we cannot believe the result of the verification at all. From another standpoint, it can be said that the SPIN model checker gives up the task of the abstraction and delegates its burden to its users. FeaVer [52] includes an assist tool for abstracting C programs to PROMELA, but it does not guarantee the correctness of the abstraction.

Second problem is that, though LTL properties can be verified, it cannot verify the memory safety of programs. This is because PROMELA itself is a memory-safe language. PROMELA is designed for expressing the inherent parallel nature of distributed programs. Therefore, it does not even have a dynamic memory allocation mechanism in order to simplify problems. In sum, to verify LTL properties of OS with the SPIN model checker, its memory safety must be verified by other means first, for example, with the approach of this thesis.

SLAM SLAM [4] is a model checker which can verify safety properties of C programs. We first explain the expressiveness of the SLAM model checker by usage examples. To verify some property, the property must be represented in the SLIC specification language.

For example, Figure 4.1 shows an example SLIC specification for proper usage of spin locks. First, the state of a spin lock is specified from line 1 to 4. It consists of two constant values (`Locked` and `Unlocked`) and its initial state is `Unlocked`. Then, the transition rule of the state is specified from line 6 to 11 and from line 13 to 18. If the lock acquire function `Lock` is called and the state is `Unlocked`, the state changes to `Locked` (in line 10). If the state is `Locked`, the execution is aborted (in line 8). In addition, if the lock release function `Unlock` is called and the state is `Locked`, the state changes to `Unlocked` (in line 17). If the state is `Unlocked`, the execution is aborted (in line 15).

Next, suppose that we are trying to verify the program of Figure 4.2. The SLAM model checker translates the specification of Figure 4.1 and the program of Figure 4.2 into one integrated C program as shown in Figure 4.3. Then, it checks whether the label `SLIC_ERROR` is reachable or not in the C program.

```
1  state {
2     enum { Unlocked = 0, Locked = 1 }
3       state = Unlocked;
4  }
5
6  Lock.return {
7     if (state == Locked)
8        abort;
9     else
10       state = Locked;
11 }
12
13 Unlock.return {
14    if (state == Unlocked)
15       abort;
16    else
17       state = Unlocked;
18 }
```

Figure 4.1: An example SLIC specification for proper usage of spin locks

```
1  void example (void) {
2     Lock();
3     Unlock();
4  }
```

Figure 4.2: A simple C code which accesses spin locks

```
1  enum { Unlocked = 0, Locked = 1 }
2  state = Unlocked;
3
4  void slic_abort() {
5     SLIC_ERROR: ;
6  }
7
8  void Lock_return (void) {
9     if (state == Locked)
10       slic_abort();
11    else
12       state = Locked;
13 }
14
15 void Unlock_return (void) {
16    if (state == Unlocked)
17       slic_abort();
18    else
19       state = Unlocked;
20 }
21
22 void example (void) {
23    Lock();
24    Lock_return();
25    Unlock();
26    Unlock_return();
27 }
```

Figure 4.3: A C code which is generated by SLAM

As shown in the above examples, what the SLAM model checker does is just to solve the reachability problem of single thread programs. Therefore, the safety properties that can be verified by SLAM are not so complex. For example, deadlock and livelock freedom cannot be checked by SLAM. In fact, the type system of TALK is able to ensure the proper usage of spin locks shown in the above examples (see Section 3.2.4). Of course, the expressiveness of SLIC specification is strictly greater than that of the type system of TALK because the general reachability problem is undecidable while the type check of TALK is decidable. However, the gap is not so large unlike that of the SPIN model checker. We expect that the gap can be largely eliminated by integrating the type system of resource usage analysis [54, 55] into TALK.

In addition to the limited expressiveness, a big problem of the SLAM model checker is that it cannot verify the memory safety and the control-flow safety of C programs. It assumes that the memory safety and the control-flow safety are verified by other means, for example, with the approach of this thesis. Moreover, pointer arithmetic is largely restricted in the SLAM model checker. Therefore, it cannot verify properties of memory management code, because the code exploits complex pointer arithmetic.

Further, note that the C programming language is surely a low-level language, but there still exists a large gap between it and assembly/machine languages. That is, the result of the SLAM model checker is only applicable to the binary executables only if the C compiler used to generate them is correct. However, it is really hard to verify the correctness of compilers. On the other hand, in the approach of this thesis, the TALK type checker is able to type-check programs at the level of binary executables.

The rest of the verification algorithm is as follows. First, it extracts an abstract model (called boolean programs), by using the theorem provers Simplify [26] and Vampyre [12], from the target program, according to a set of predicates. The set is just an empty set at first, but refined as the algorithm proceeds. Next, it searches a path which reaches an error state on the abstract model. If the path is not found, it means that the target program satisfies the specification. Otherwise, it checks whether the found path is real or not, according to the target program. If the path is real, it means that the target program does not satisfies the specification. Otherwise, it refines the predicates and retries from the extraction of an abstract model.

Despite the above abstraction, the SLAM model checker is still considerably slow. For example, [3] shows that it takes 98 seconds to verify a floppy device driver which is 6500 lines of C code (the experimental environment is unknown because [3] does not mention it). This is because it

sometimes requires an exponential number of calls to the theorem provers. Thus, its running time is dominated by the cost of theorem proving.

BLAST BLAST [48] is another model checker which can verify safety properties of C programs. In fact, the safety properties that can be verified by the BLAST model checker are almost the same as that of the SLAM model checker, because its specification language is almost the same as that of SLAM (aside from its surface syntax), the specification written in the language is woven into the target C program, what the model checker really does is just to solve the reachability problem, and its verification algorithm (an abstract-check-refine loop) is basically the same as SLAM. Thus, the argument about the SLAM model checker is also applicable to the BLAST model checker.

One advantage of BLAST is its smart abstraction technique, called lazy abstraction. The point is that, when refining an abstract model with the refined abstract predicates, it refines only the part of the abstract model that the refinement of the predicates affects, not the whole abstract model. Thanks to the lazy abstraction, the number of calls to the theorem provers is largely reduced. However, it is still significantly slow in some cases. For example, [47] shows that it takes about 0.5 to 429 seconds to verify the proper usage of spin locks for programs of about 20000 lines of C code, on a 700 MHz Pentium III processor with 256 MB RAM.

4.3 Verification with proof assistants

The most straightforward approach to ensure safety of an OS is to directly verify that its implementation satisfies the safety by hand with proof assistants (e.g., [86, 75, 76, 77]). The advantage of the approach is that almost arbitrary properties can be verified, while what the type system of TALK directly verifies is only the memory safety and the control-flow safety. For example, the deadlock freedom and the livelock freedom can be verified by extracting a model of process-calculus from the implementation and proving that they are satisfied on the model and the extraction is correct, with the proof assistants. Thus, as for the range of verifiable properties, the approach is far more flexible than TALK and the approach of model checking.

However, there are two problems in the approach. First problem is that ordinary OS developers are not familiar with the proof assistants. In fact, they may have no idea how to formally prove the memory safety and the control-flow safety of their programs. On the other hand, in the approach

of this thesis, all the programmers have to do is just annotating their assembly programs with the type information of TALK (by hand or some other means). Then, the type checker of TALK automatically verifies the memory safety and the control-flow safety of the programs. In addition, we believe that, compared to the proof assistants, the type system of TALK is easier to understand for them because they know a little about type systems, thanks to the weakly-typed programming language, C.

Second problem is that the proof is not synchronized with the implementation. That is, if developers modify the implementation, they have to prove the safety of the modified implementation again. This severely limits development efficiency because proving the safety with the proof assistants is much harder than modifying the implementation with text editors. On the other hand, in the approach of TALK, all they have to do is just modifying their programs. Then, the TALK type checker automatically verifies its memory safety and control-flow safety.

The rest of this section describes three works that verify safety of OSes with the proof-assistants.

Kit Kit [11] is an OS whose safety is verified at the machine code level. More specifically, the verified safety property is a process isolation property, that is, processes do not interfere with each other, except process communication. The property is represented in the Boyer-Moore logic, and its proof is mechanically checked by the Boyer-Moore theorem prover [14].

The proof consists of a formal definition of a communicating process, a formal specification of an OS kernel which manages a fixed number of communicating processes, the proof of a theorem which states that the specification correctly implements each process, a formal definition of a machine on which to implement the OS kernel, the machine code implementation of the kernel, and the proof of a theorem which states that the machine code running on the machine correctly implements the formal specification. In sum, the proof says that the machine code implementation of the kernel correctly implements the abstract model of processes and the processes are correctly isolated in the model.

One problem of the approach of Kit is that, though Kit is a small and simple OS, it is still really hard to generate the proof. In fact, it took eighteen months to complete the proof of the process isolation property [11]. This is because the proof of Kit was made from scratch. What is worse is that Kit does not support dynamic allocation of resources. The generation of the proof will become even harder if the dynamic allocation is considered. On

the other hand, it took a few weeks to write an prototype OS kernel which provides a dynamic memory allocation facility in TALK.

L4.verified The L4.verified [93] project is an ongoing project which tries to verify safety of the L4 microkernel [63]. As of writing this thesis, the correctness of the virtual memory subsystem of L4 has been verified [92]. On the other hand, this thesis does not handle the virtual memory so far (see Chapter 3).

The approach of the verification is as follows. First, the abstract model of the virtual memory subsystem is extracted from the informal description of L4, and the model is formally defined in the theorem prover Isabelle [75]. The model consists of the virtual address spaces and the operations on them. Then, the correctness of the model is proved. For example, it proves the property that all the valid pages are mapped to physical pages, and so on. Next, the abstract model is refined to a more concrete model formally, that is, without violating the correctness of the abstract model. More specifically, the concrete model consists of page tables and the operations on them. For example, an address lookup operation in the abstract model is formally implemented as a program which operates on the page tables. Last, the concrete model is further refined to an implementation in the C programming language. Thus, it is verified that the implementation correctly implements the virtual memory subsystem of L4.

One problem of the approach of L4.verified is that the cost of the verification is too high. In fact, [93] shows that it took 1.5 person years to obtain the verified implementation of the virtual memory subsystem, and all the specifications and proofs for the verification run to about 14000 lines of proof scripts of Isabelle, while the whole L4 microkernel consists of about 10000 lines of code.

VFiasco The VFiasco [50] project is an ongoing project which tries to verify safety of the Fiasco microkernel [49]. The safety properties that the project attempts to prove are the termination of the page-fault handlers and the correctness of the memory allocator, though nothing has been proved yet, as of writing this thesis. On the other hand, we have implemented a memory-safe memory allocator in TALK (see Section 3.1 for details), though the termination property cannot be ensured by the type system of TALK.

The unique point of the project is that it tries to verify properties at the level of C++ [58] source code, because Fiasco is written in C++. To this end, it tries to define a semantics of C++ and represent the semantics in the

PVS [76] proof assistant, though the definition has not been completed yet and only the semantics of the C++ data types is available so far.

One problem of the VFiasco approach is that C++ is too complex to define a formal semantics. For example, C++ supports class inheritance, operator overloading, runtime type casts, exception mechanisms, and so on. Even though the semantics is defined, there is another problem. The problem is that it is difficult to build a C++ compiler which follows the defined semantics. On the other hand, the approach of this thesis does not suffer the problems. In fact, the semantics of TALK is simpler than expected because it is almost the same as that of the underlying CPU architecture.

4.4 Strictly typed programming languages

There are several operating systems that are partly written in strictly typed languages. However, none of them directly implemented memory management and multi-thread management code, while we implemented them in TALK.

House [43] is an operating system written in Haskell programming language [88]. In House, a simple graphic driver, a PS2 mouse driver and drivers for network cards (NE2000 and Intel PRO/100), a GUI window system and a network protocol stack are implemented in Haskell. In addition, House supports execution of user programs in a separate address space. Unlike the approach of this thesis, however, memory management facility cannot be implemented in Haskell and the kernel of House includes a garbage collector which is written in C.

Desert Spring-Time [25] is an operating system written in OCaml programming language [17]. In Desert Spring-Time, an IDE driver, drivers for network cards (NE2000 and Realtek 8139), a simple graphic driver, drivers for PS/2 mouse and keyboard and a network protocol stack are implemented in OCaml. As the House operating system, however, memory management facility is not implemented in OCaml. Therefore, Desert Spring-Time includes a garbage collector written in C for memory management.

Singularity [53] is an operating system written in Sing# programing language [53], which is an extension of Spec# programming language [6], which is an extension of C# programming language [59]. The good point of Sing# is that programmers can specify the precondition and postcondition that must be satisfied by their programs. The static checker of Sing# verifies them with the Simplify theorem prover [26] and inserts check code to

the program in order to check the behavior of the programs at runtime. In Singularity, several I/O device drivers (including supports for DMA) are written in Sing#. However, memory management and multi-thread management code is not written in Sing#. In Singularity, multi-thread management and inter-process communication mechanisms are implemented as language extensions of Sing#.

Lisp OSes ([70, 32] and others) are written in Lisp [66]. There are two problems in the Lisp OSes. One problem is that Lisp is a dynamically typed language, that is, they cannot prevent runtime errors from occurring, as in the case of the hardware approach. Another problem is that they sometimes require special hardware (Lisp machines) in order to achieve high performance.

The SPIN operating system [9] is an extensible operating system that can be extended safely by inserting extensions that are written in Modula-3 programming language [74]. The goal of SPIN is to extend the functionality of the kernel safely, not to ensure safety of the kernel itself. Therefore, memory management and multi-thread management mechanisms are not implemented in Modula-3.

Besides the problem of inability to write memory management and multi-thread management code, all the above OSes have the problem that trusted computing base (TCB) becomes huge. TCB is part of OS on which safety of the whole system depends. Therefore, minimizing TCB as small as possible is important to make OS reliable. In the above OSes, their safety is depends on the correctness of external compilers for Haskell, OCaml, Lisp and Modula-3, respectively. That is, their TCB includes the external compilers. Generally speaking, complexity of language compilers is comparable to or exceeds that of OS.

On the other hand, in the approach of this thesis, TCB is considerably small compared to that of the above approaches because safety is depends only on the correctness of the type checker of TALK. We need not to trust external compilers or assemblers because the type check of TALK can be performed on binary executables. Thus, as for TCB, the approach of this thesis is more reliable than the above approaches.

In addition, their approaches have the problem that programmers cannot use their favorite languages to modify and/or extend the OSes. For example, if a programmer wants to modify the House operating system, she must use Haskell. This severely limits the development of the OSes.

At first glance, the problem seems to be also applicable to the approach of this thesis, but there is an essential difference. The difference is that programmers need not to write TALK code directly because TALK can be used

as a target language of high-level languages. That is, programmers can use their favorite languages if there exist compilers from them to TALK. For example, if there are compilers from strictly-typed C dialects [61, 83] to TALK, it helps C programmers to write safe OS kernels. From the theoretical viewpoint, it is not so difficult to make a compiler from strictly-typed programming languages to TALK, because the type-preserving compilation from a typed high-level language to a typed assembly language has been already studied by [71]. Moreover, even the standard C can be used by programmers if we can build a compiler which compiles C programs to TALK. Although C is not a strictly typed language, it may be possible with the approach of CCured [73] that is a C compiler that inserts dynamic checking code to C programs in order to ensure the memory safety. (Note that CCured itself does not allow programmers to write memory management code and depends on the external memory management mechanism, as other strictly-typed programming languages.)

4.4.1 Types and memory management

Linear type systems [94] ensure that a memory region is accessed only once. That is, they can prevent pointers from aliasing. Therefore, the memory region can be reused safely. There exist TALs based on the linear types [18, 2]. One big problem of the linear types is that the expressiveness of linearly-typed languages is largely limited because no aliases are allowed.

Alias type systems [82, 96] do not prevent pointers from aliasing, but track the information about aliases for reusing memory regions safely. Thus, the alias type systems are more expressive than the linear type systems. However, it is impossible to implement practical memory management in the original alias type system because it does not support variable-length arrays. As described in Section 2.2, TALK is based on the alias types and extended to support variable-length arrays and integer constraints. Thus, practical memory management can be implemented in TALK.

Hawblitzel et al. [46] extends the alias type system for implementing flexible memory management. The similarity between the approach of TALK and theirs is that both introduce integer constraints to the alias types. The important difference is that, in their type system, variable-length arrays are realized as a combination of fixed-length tuples and recursive types. However, there are two problems in their approach. One problem is that elements of an array cannot be accessed in $O(1)$ order because the array type must be unrolled ($O(n)$ time at worst) in advance. The other problem is that it requires runtime type checks for managing arrays. To solve these

problems, they extended their type system intricately for detecting useless runtime type-checks as precisely as possible. For example, they defined 'split' of arrays as a function, and showed that the function is not needed at runtime, with their complex typing rules. On the other hand, there are no such problems in the type system of TALK because it directly supports the variable-length arrays as language primitives. Thus, the type system of TALK is much simpler than theirs and yet powerful enough to implement memory management code.

DTAL [101] is a typed assembly language extended with the dependent type. As the type system of TALK, DTAL also introduces integer constraints to its type system. However, DTAL is not flexible enough to implement memory management because memory reusing is impossible. The goal of DTAL is to type-check array bound-checking.

In region-based memory management [89, 90, 42], heap values are allocated in one of memory regions. When a memory region is deallocated, all the heap values in the region are deallocated. The region-based memory management does not allow programmers to directly manage memory. Calculus of Capability [23] extends the region-based memory management and allows programmers to explicitly allocate and deallocate memory regions, but memory regions cannot be reused explicitly and the heap values allocated in memory regions cannot be managed directly.

Shape analysis [27, 39, 81] is an analysis which estimates the shape (e.g., tree, DAG or cyclic graph) of the data structure that is accessible from pointers. Although the shape analysis is developed in the research area of compiler optimization, it can be used for detecting pointer aliases because it determines whether two pointers point to the same data structure. However, the approach of the shape analysis cannot be applied directly to memory management because it is a conservative analysis. In addition, the analysis can tell whether if a data structure can be deallocated safely, but programmers cannot reuse the data structure explicitly.

Chapter 5

Conclusion

This thesis has proposed a new strictly typed programming language which is flexible and powerful enough to implement operating systems (OS). By writing in the strictly typed language, the memory safety and the control-flow safety of OS is automatically and efficiently verified thorough the type check of the language.

Ensuring safety of OS kernels has been considered to be extremely hard, because the OS kernels have been written in weakly typed or untyped languages. For example, the approach of using model checkers to verify safety properties is only applicable for the very small part of the OS kernels because of the problem of the state explosion. In addition, the approach of using theorem provers and/or proof assistants to prove safety properties of OS kernels has one problem that usual programmers are not familiar with the theorem provers and/or proof assistants. In addition, it has another problem that the safety property must be proved again, if the OS kernels are rewritten.

Compared to the above approaches, the proposed approach is more practical because there is no state explosion problem and the programmers do not need to learn how to use the theorem provers and/or the proof assistants. Although the safety ensured by the proposed approach is just the memory safety and the control-flow safety, we believe that the proposed approach has made a big step, because the more sophisticated safety properties (e.g., the deadlock freedom, the multi-thread safety on SMP and the resource usage safety) can be ensured on the basis of the basic type safety.

Concretely speaking, this thesis has proposed the new strictly and statically typed assembly language (TALK) that is flexible and powerful enough to implement OS kernels. Traditional strictly typed programming languages

are not enough because the important components of the OS kernels (e.g., memory management mechanisms and multi-thread management mechanisms) cannot be implemented in them. To cope with the problem, the type system of TALK supports variable-length arrays (the arrays whose size is not known until runtime), integer constraints between variables, explicit alias tracking, and split/concatenation of the variable-length arrays.

More specifically, the contribution of this thesis is that we have first realized that integrating the integer constraints and the explicit alias tracking with the variable-length arrays and their split/concatenation enables us to write practical memory management code in strictly-typed languages. The integer constraints and the explicit alias tracking have been separately studied by different research areas.

Then, this thesis has shown how to implement memory management and multi-thread mechanisms in TALK. From the viewpoint of type theory, the memory management is managing free memory regions (the variable-length arrays) and changing their types when the regions are allocated and/or deallocated. Thus, the memory management can be implemented in TALK, because it directly supports the variable-length arrays and enables programmers to destructively update the contents of the memory regions, thanks to the explicit alias tracking. Further, at the lowest level, the multi-thread management is just managing the memory stacks that represent the execution context of threads. Thus, TALK is sufficient to implement the multi-thread management, because the memory stacks are just the variable-length arrays and TALK supports them.

In addition, this thesis has described an implementation of TALK on the IA-32 architecture. It has also presented a prototype implementation of an OS kernel which is written in TALK. The OS kernel provides memory management mechanisms and multi-thread mechanisms. Although the very early boot procedure is not written in TALK, the other parts are completely written in TALK.

5.1 Future Direction

One big problem of the current implementation of TALK is that it does not support interrupts. There are two possible approaches to support interrupts.

One approach is to make threads handle interrupts. First, the kernel creates threads for each interrupt vector. The kernel also associates an interrupt handler to each interrupt vector. Then, if an interrupt occurs, the asso-

ciated interrupt handler just wakes up the associated thread. The interrupt is processed by the thread. The advantage of this approach is that we need not to modify the type system of TALK because multi-thread management can be implemented in TALK, as shown in this thesis. The disadvantage is that the trusted computing base becomes a bit larger because the memory and control-flow safety of the interrupt handler cannot be ensured. More specifically, the type system of TALK cannot prevent race conditions between the interrupt handlers and normal program execution.

The other approach is to modify the type system of TALK. From the viewpoint of the type system, the interrupt handlers are special functions that may be called anywhere in programs whenever interrupts are not disabled. That is, the memory and control-flow safety of the interrupt handlers can be verified by checking whether if the preconditions specified in their label types are always satisfied whenever interrupts are not disabled. To realize this approach, we need to modify the typing rules of TALK and the type system so that it can keep track of disabled interrupts. The advantage of this approach is that the whole kernel including interrupt handlers can be type-checked by the type checker. The disadvantage is that we need to modify the implementation of the TALK assembler and type checker according to the extension of the type system.

Appendix A

Type Soundness

A.1 Inversion lemmas

Lemma A.1 (Value inversion: integer)
If $\Delta; C \vdash n : \sigma$, then $\sigma \equiv i$.

Proof From the typing rules, the last typing rule of the derivation $\Delta; C \vdash n : \sigma$ is VALUEINTEGER. Thus, from the typing rule, σ is some integer i.

Lemma A.2 (Value inversion: label)
If $\Delta; C \vdash l[c_1, \ldots, c_n/\Delta''] : \sigma$, then $\sigma \equiv \forall\Delta'.|C|[\Sigma](\Gamma)$.

Proof From the typing rules, the last typing rule of the derivation $\Delta; C \vdash l[c_1, \ldots, c_n/\Delta''] : \sigma$ is VALUELABEL. Thus, from the typing rule, σ is some label type $\forall\Delta'.|C|[\Sigma](\Gamma)$.

Lemma A.3 (Memory inversion)
If $\vdash M\{n \mapsto a\} : \Sigma$, then $\Sigma \equiv \Sigma' \otimes \{n \mapsto at\}$ and $\cdot; \cdot \vdash a : at$.

Proof First, let $M \equiv \{n_1 \mapsto a_1\} \ldots \{n_m \mapsto a_m\}$. Then, from $\texttt{GU}(M\{n \mapsto a\})$, $\forall i.n_i \neq n$. Thus, by the typing rule MEMORY, $\Sigma \equiv \{n_1 \mapsto at_1\} \otimes \ldots \otimes \{n_m \mapsto at_m\} \otimes \{n \mapsto at \otimes \Sigma''$, where $\forall i.\cdot; \cdot \vdash a_i : at_i$ and $\cdot; \cdot \vdash a : at$. Here let $\Sigma' \equiv \{n_1 \mapsto at_1\} \otimes \ldots \otimes \{n_m \mapsto at_m\} \otimes \Sigma''$. Then, $\Sigma \equiv \Sigma' \otimes \{n \mapsto at\}$ and $\cdot; \cdot \vdash a : at$.

Lemma A.4
If $\Delta; C \vdash \Sigma_1 \otimes \{i_1 \mapsto at_1\} = \Sigma_2 \otimes \{i_2 \mapsto at_2\}$, $\Delta; C \models i_1 = i_2$ and $\Delta; C \vdash at_1 = at_2$, then $\Delta; C \vdash \Sigma_1 = \Sigma_2$.

Proof By induction on the derivation of $\Delta; C \vdash \Sigma_1 \otimes \{i_1 \mapsto at_1\} = \Sigma_2 \otimes \{i_2 \mapsto at_2\}$. The proof is by case analysis on the last rule of the derivation.

- Case EQMEMEMPTY
  This case never occurs because $\Sigma_1 \otimes \{i_1 \mapsto at_1\} \not\equiv \cdot$.

- Case EQMEMLOC
  The proof is by case analysis of the premises of the typing rule EQMEMLOC.

  - Case $\Delta; C \vdash \Sigma_1 = \Sigma_2$, $\Delta; C \models i_1 = i_2$ and $\Delta; C \vdash at_1 = at_2$
    From the premise, we have $\Delta; C \vdash \Sigma_1 = \Sigma_2$.
  - Case $\Delta; C \vdash \Sigma_1' \otimes \{i_1 \mapsto at_1\} = \Sigma_2' \otimes \{i_2 \mapsto at_2\}$, $\Delta; C \models i_1' = i_2'$ and $\Delta; C \vdash at_1' = at_2'$, where $\Sigma_1 \equiv \Sigma_1' \otimes \{i_1' = at_1'\}$ and $\Sigma_2 \equiv \Sigma_2' \otimes \{i_2' = at_2'\}$
    By the induction hypothesis, we have $\Delta; C \vdash \Sigma_1' = \Sigma_2'$. Now, by the typing rule EQMEMLOC, we have $\Delta; C \vdash \Sigma_1' \otimes \{i_1 \mapsto at_1\} = \Sigma_2' \otimes \{i_2 \mapsto at_2\}$. That is, $\Delta; C \vdash \Sigma_1 = \Sigma_2$.
  - Case $\Delta; C \vdash \Sigma_1' \otimes \{i_1 \mapsto at_1\} = \Sigma_2$, $\Delta; C \models i_1' = i_2$ and $\Delta; C \vdash at_1' = at_2$, where $\Sigma_1 \equiv \Sigma_1' \otimes \{i_1' = at_1'\}$
    By the typing rule EQMEMLOC, we have $\Delta; C \vdash \Sigma_1' \otimes \{i_1 \mapsto at_2\} = \Sigma_1' \otimes \{i_1' \mapsto at_1'\} (\equiv \Sigma_1)$, because $\Delta; C \models i_1 = i_2 = i_1'$, $\Delta; C \vdash at_1 = at_2 = at_1'$ and $\Delta; C \vdash \Sigma_1' = \Sigma_1'$. Thus, $\Delta; C \vdash \Sigma_1 = \Sigma_2$.
  - Case $\Delta; C \vdash \Sigma_2' \otimes \{i_2 \mapsto at_2\} = \Sigma_2$, $\Delta; C \models i_1 = i_2'$ and $\Delta; C \vdash at_1 = at_2'$, where $\Sigma_2 \equiv \Sigma_2' \otimes \{i_2' = at_2'\}$
    Same as the previous case.

- Case EQMEMVAR
  By the typing rule, we have $\Sigma_1 \equiv \Sigma_1' \otimes \epsilon$, $\Sigma_2 \equiv \Sigma_2' \otimes \epsilon$, $\Delta; C \vdash \Sigma_1' \otimes \{i_1 \mapsto at_1\} = \Sigma_2' \otimes \{i_2 \mapsto at_2\}$. Here, by the induction hypothesis, $\Delta; C \vdash \Sigma_1' = \Sigma_2'$. Now, by the typing rule EQMEMVAR, we have $\Delta; C \vdash \Sigma_1 = \Sigma_2$.

- Case EQMEMZEROARRAYL
  The proof is by case analysis of the premises of the typing rule.

  - Case $\Delta; C \vdash \Sigma_1 = \Sigma_2 \otimes \{i_2 \mapsto at_2\}$, $\Delta; C \vdash j_1 = 0$, where $at_1 \equiv \tau_1[j_1]$
    Let $at_2 \equiv \tau_2[j_2]$. Then, from the typing rule EQARRAY, $\Delta; C \vdash j_1 = j_2 = 0$. Here, by the typing rule EQMEMZEROARRAYR, we have $\Delta; C \vdash \Sigma_1 = \Sigma_2$.

- Case $\Delta; C \vdash \Sigma_1' \otimes \{i_1 \mapsto at_1 = \Sigma_2 \otimes \{i_2 \mapsto at_2\}$ and $\Delta; C \vdash j_1' = 0$, where $\Sigma_1 \equiv \Sigma_1' \otimes \{i_1' \mapsto \tau'[j_1']\}$
  By the induction hypothesis, we have $\Delta; C \vdash \Sigma_1' = \Sigma_2$. Here, by the typing rule EQMEMZEROARRAYL, we have $\Delta; C \vdash \Sigma_1' \otimes \{i_1' \mapsto \tau'[j_1']\} = \Sigma_2$.

- Case EQMEMZEROARRAYR
  Same as Case EQMEMZEROARRAYL.

Lemma A.5
If $\Delta; C \vdash \Sigma_1 \otimes \epsilon = \Sigma_2 \otimes \epsilon$, then $\Delta; C \vdash \Sigma_1 = \Sigma_2$.

Proof By induction on the derivation of $\Delta; C \vdash \Sigma_1 \otimes \epsilon = \Sigma_2 \otimes \epsilon$. The proof is by case analysis on the last rule of the derivation.

- Case EQMEMEMPTY
  This case never occurs because $\Sigma_1 \otimes \epsilon \not\equiv \cdot$.

- Case EQMEMLOC
  From the typing rule, we have $\Delta; C \vdash \Sigma_1' \otimes \epsilon = \Sigma_2' \otimes \epsilon$, where $\Sigma_1 \equiv \Sigma_1' \otimes \{i_1 \mapsto at_1\}$ and $\Sigma_2 \equiv \Sigma_2' \otimes \{i_2 \mapsto at_2\}$. Here, by the induction hypothesis, we have $\Delta; C \vdash \Sigma_1' = \Sigma_2'$. Now, by the typing rule EQMEMLOC, $\Delta; C \vdash \Sigma_1 = \Sigma_2$.

- Case EQMEMVAR
  The proof is by case analysis of the premises of the typing rule.

  - Case $\Delta; C \vdash \Sigma_1 = \Sigma_2$
    From the premise, we have $\Delta; C \vdash \Sigma_1 = \Sigma_2$.
  - Case $\Delta; C \vdash \Sigma_1' \otimes \epsilon = \Sigma_2' \otimes \epsilon$, where $\Sigma_1 \equiv \Sigma_1' \otimes \epsilon'$ and $\Sigma_2 \equiv \Sigma_2' \otimes \epsilon'$
    By the induction hypothesis, we have $\Delta; C \vdash \Sigma_1' = \Sigma_2'$. Here, by the typing rule EQMEMVAR, $\Delta; C \vdash \Sigma_1' \otimes \epsilon' = \Sigma_2' \otimes \epsilon'$. That is, $\Delta; C \vdash \Sigma_1 = \Sigma_2$.

- Case EQMEMZEROARRAYL
  By the typing rule, we have $\Delta; C \vdash \Sigma_1' \otimes \epsilon = \Sigma_2 \otimes \epsilon$ and $\Delta; C \models j = 0$, where $\Sigma_1 \equiv \Sigma_1' \otimes \{i \mapsto \tau[j]\}$. By the induction hypothesis, we have $\Delta; C \vdash \Sigma_1' = \Sigma_2$. Now, by the typing rule EQMEMZEROARRAYL, $\Delta; C \vdash \Sigma_1' \otimes \{i \mapsto \tau[j]\} = \Sigma_2$. That is, $\Delta; C \vdash \Sigma_1 = \Sigma_2$.

- Case EQMEMZEROARRAYR
  Same as Case EQMEMZEROARRAYL.

Lemma A.6
If $\Delta; C \vdash \Sigma_1 \otimes \{i \mapsto \tau[j]\} = \Sigma_2$ and $\Delta; C \models j = 0$, then $\Delta; C \vdash \Sigma_1 = \Sigma_2$.

Proof By induction on the derivation of $\Delta; C \vdash \Sigma_1 \otimes \{i_1 \mapsto at_1\}$. The proof is by case analysis on the last rule of the derivation.

- Case EQMEMEMPTY
  This case never occurs because $\Sigma_1 \otimes \{i \mapsto \tau[j]\} \not\equiv \cdot$.

- Case EQMEMLOC
  The proof is by case analysis of the premises of the typing rule.

  - Case $\Delta; C \vdash \Sigma_1' \otimes \{i \mapsto \tau[j]\} = \Sigma_2'$, where $\Sigma_1 \equiv \Sigma_1' \otimes \{i_1' \mapsto at_1'\}$, $\Sigma_2 \equiv \Sigma_2' \otimes \{i_2' \mapsto at_2'\}$, $\Delta; C \models i_1' = i_2'$ and $\Delta; C \vdash at_1' = at_2'$
    By the induction hypothesis, we have $\Delta; C \vdash \Sigma_1' = \Sigma_2'$. Now, by the typing rule EQMEMLOC, we have $\Delta; C \vdash \Sigma_1' \otimes \{i_1' \mapsto at_1'\} = \Sigma_2' \otimes \{i_2' \mapsto at_2'\}$. That is, $\Delta; C \vdash \Sigma_1 = \Sigma_2$.
  - Case $\Delta; C \vdash \Sigma_1 = \Sigma_2'$, where $\Sigma_2 \equiv \Sigma_2' \otimes \{i_2' \mapsto at_2'\}$, $\Delta; C \models i = i_2'$ and $\Delta; C \vdash \tau[j] = at_2'$
    Let $at_2' \equiv \tau'[j']$. Then, from the typing rule EQARRAY, we have $\Delta; C \vdash \tau = \tau'$ and $\Delta; C \models j' = j = 0$. Therefore, by the typing rule EQMEMZEROARRAYR, we have $\Delta; C \vdash \Sigma_1 = \Sigma_2' \otimes \{i_2' \mapsto at_2'\}$. That is, $\Delta; C \vdash \Sigma_1 = \Sigma_2$.

- Case EQMEMVAR
  By the typing rule, we have $\Delta; C \vdash \Sigma_1' \otimes \{i \mapsto \tau[j]\} = \Sigma_2'$, where $\Sigma_1 \equiv \Sigma_1' \otimes \epsilon$ and $\Sigma_2 \equiv \Sigma_2' \otimes \epsilon$. Here, by the induction hypothesis, $\Delta; C \vdash \Sigma_1' = \Sigma_2'$. Now, by the typing rule EQMEMVAR, we have $\Delta; C \vdash \Sigma_1' \otimes \epsilon = \Sigma_2' \otimes \epsilon$. That is, $\Delta; C \vdash \Sigma_1 = \Sigma_2$.

- Case EQMEMZEROARRAYL
  The proof is by case analysis of the premises of the typing rule.

  - Case $\Delta; C \vdash \Sigma_1 = \Sigma_2$
    From the premise, we have $\Delta; C \vdash \Sigma_1 = \Sigma_2$.
  - Case $\Delta; C \vdash \Sigma_1' \otimes \{i \mapsto \tau[j]\} = \Sigma_2$ and $\Delta; C \models j' = 0$, where $\Sigma_1 \equiv \Sigma_1' \otimes \{i' \mapsto \tau'[j']\}$
    By the induction hypothesis, we have $\Delta; \Sigma_1' = \Sigma_2$. Then, by the typing rule EQMEMZEROARRAYL, $\Delta; C \vdash \Sigma_1' \otimes \{i' \mapsto \tau'[j']\} = \Sigma_2$. That is, $\Delta; C \vdash \Sigma_1 = \Sigma_2$.

- Case EQMEMZEROARRAYR
  From the typing rule, we have $\Delta; C \vdash \Sigma_1 \otimes \{i \mapsto \tau[j]\} = \Sigma_2'$ and $\Delta; C \models j' = 0$, where $\Sigma_2 \equiv \Sigma_2' \otimes \{i' \mapsto \tau'[j']\}$. Here, by the induction hypothesis, we have $\Delta; C \vdash \Sigma_1 = \Sigma_2'$. Now, by the typing rule EQMEMZEROARRAYR, $\Delta; C \vdash \Sigma_1 = \Sigma_2' \otimes \{i' \mapsto \tau'[j']\}$. That is, $\Delta; C \vdash \Sigma_1 = \Sigma_2$.

Lemma A.7
If $\Delta; C \vdash \Sigma_1 = \Sigma_2 \otimes \{i \mapsto \tau[j]\}$, then $\Delta; C \vdash \Sigma_1 = \Sigma_2$.

Proof Same as the proof of the lemma A.7.

Lemma A.8
If $\Delta; C \vdash \Sigma \otimes \Sigma' = \Sigma_1 \otimes \Sigma_2$ and $\Delta; C \vdash \Sigma = \Sigma_1$, then $\Delta; C \vdash \Sigma' = \Sigma_2$.

Proof By induction on the derivation of $\Delta; C \vdash \Sigma = \Sigma_1$. The proof is by case analysis on the last rule of the derivation.

- Case EQMEMEMPTY
  By the typing rule, we have $\Sigma \equiv \Sigma_1 \equiv \cdot$. That is, $\Sigma \otimes \Sigma' \equiv \Sigma'$ and $\Sigma_1 \otimes \Sigma_2 \equiv \Sigma_2$. Therefore, $\Delta; C \vdash \Sigma' = \Sigma_2$.

- Case EQMEMLOC
  By the typing rule, we have $\Sigma \equiv \Sigma'' \otimes \{i_1 \mapsto at_1\}$, $\Sigma_1 \equiv \Sigma_1'' \otimes \{i_2 \mapsto at_2\}$, $\Delta; C \vdash \Sigma'' = \Sigma_1''$, $\Delta; C \models i_1 = i_2$ and $\Delta; C \vdash at_1 = at_2$.

  Here, by the lemma A.4, we have $\Delta; C \vdash \Sigma'' \otimes \Sigma' = \Sigma_1'' \otimes \Sigma_2$. Then, by the induction hypothesis, $\Delta; C \vdash \Sigma' = \Sigma_2$.

- Case EQMEMVAR
  By the typing rule, we have $\Sigma \equiv \Sigma''\epsilon$, $\Sigma_1 \equiv \Sigma_1''\epsilon$ and $\Delta; C \vdash \Sigma'' = \Sigma_1''$.

  Here, by the lemma A.5, we have $\Delta; C \vdash \Sigma'' \otimes \Sigma' = \Sigma_1'' \otimes \Sigma_2$. Then, by the induction hypothesis, $\Delta; C \vdash \Sigma' = \Sigma_2$.

- Case EQMEMZEROARRAYL
  By the typing rule, we have $\Sigma \equiv \Sigma'' \otimes \{i \mapsto \tau[j]\}$, $\Delta; C \models j = 0$ and $\Delta; C \vdash \Sigma'' = \Sigma_1$.

  Here, by the lemma A.6, we have $\Delta; C \vdash \Sigma'' \otimes \Sigma' = \Sigma_1'' \otimes \Sigma_2$. Then, by the induction hypothesis, $\Delta; C \vdash \Sigma' = \Sigma_2$.

- Case EQMEMZEROARRAYR
  Same as Case EQMEMZEROARRAYL.

Lemma A.9
If $\Delta; C \vdash \Sigma = \Sigma'$ and $\Delta; C \vdash \Sigma_1 = \Sigma_2$, then $\Delta; C \vdash \Sigma \otimes \Sigma_1 = \Sigma' \otimes \Sigma_2$.

Proof By induction on the derivation of $\Delta; C \vdash \Sigma = \Sigma'$. The proof is by case analysis on the last rule of the derivation.

- Case EQMEMEMPTY
  By the typing rule, $\Sigma \equiv \Sigma' \equiv \cdot$. Therefore, $\Sigma \otimes \Sigma_1 \equiv \Sigma_1$ and $\Sigma' \otimes \Sigma_2 \equiv \Sigma_2$. Thus, $\Delta; C \vdash \Sigma \otimes \Sigma_1 = \Sigma' \otimes \Sigma_2$.

- Case EQMEMLOC
  By the typing rule, we have $\Sigma \equiv \Sigma_1' \otimes \{i_1 \mapsto at_1\}$, $\Sigma' \equiv \Sigma_2' \otimes \{i_2 \mapsto at_2\}$, $\Delta; C \vdash \Sigma_1' = \Sigma_2'$, $\Delta; C \models i_1 = i_2$ and $\Delta; C \vdash at_1 = at_2$.

  Here, by the induction hypothesis, we have $\Delta; C \vdash \Sigma_1' \otimes \Sigma_1 = \Sigma_2' \otimes \Sigma_2$. Now, by the typing rule EQMEMLOC, we have $\Delta; C \vdash \Sigma_1' \otimes \Sigma_1 \otimes \{i_1 \mapsto at_1\} = \Sigma_2' \otimes \Sigma_2 \otimes \{i_2 \mapsto at_2\}$. That is, $\Delta; C \vdash \Sigma \otimes \Sigma_1 = \Sigma' \otimes \Sigma_2$.

- Case EQMEMVAR
  By the typing rule, we have $\Sigma \equiv \Sigma_1' \otimes \epsilon$, $\Sigma' \equiv \Sigma_2' \otimes \epsilon$ and $\Delta; C \vdash \Sigma_1' = \Sigma_2'$.

  Here, by the induction hypothesis, we have $\Delta; C \vdash \Sigma_1' \otimes \Sigma_1 = \Sigma_2' \otimes \Sigma_2$. Now, by the typing rule EQMEMVAR, $\Delta; C \vdash \Sigma_1' \otimes \Sigma_1 \otimes \epsilon = \Sigma_2' \otimes \Sigma_2 \otimes \epsilon$. That is, $\Delta; C \vdash \Sigma \otimes \Sigma_1 = \Sigma' \otimes \Sigma_2$.

- Case EQMEMZEROARRAYL
  By the typing rule, we have $\Sigma \equiv \Sigma_1' \otimes \{i \mapsto \tau[j]\}$, $\Delta; C \models j = 0$ and $\Delta; C \vdash \Sigma_1' = \Sigma'$.

  Here, by the induction hypothesis, we have $\Delta; C \vdash \Sigma_1' \otimes \Sigma_1 = \Sigma' \otimes \Sigma_2$. Now, by the typing rule EQMEMZEROARRAYL, $\Delta; C \vdash \Sigma_1' \otimes \Sigma_1 \otimes \{i \mapsto \tau[j]\} = \Sigma' \otimes \Sigma_2$. That is, $\Delta; C \vdash \Sigma \otimes \Sigma_1 = \Sigma' \otimes \Sigma_2$.

- Case EQMEMZEROARRAYR
  Same as Case EQMEMZEROARRAYL.

Lemma A.10
If $\vdash MM' : \Sigma \otimes \Sigma'$ and $\vdash M' : \Sigma'$, then $\vdash M : \Sigma$.

Proof Let $M \equiv \{n_1 \mapsto a_1\} \ldots \{n_k \mapsto a_k\}$, $M' \equiv \{n_1' \mapsto a_1'\} \ldots \{n_m' \mapsto a_m'\}$, $\Sigma_1 \equiv \{n_1 \mapsto at_1\} \otimes \ldots \otimes \{n_k \mapsto at_k\}$, $\Sigma_2 \equiv \{n_1' \mapsto at_1'\} \otimes \ldots \otimes \{n_m' \mapsto at_m'\}$. Then, by the typing rule MEMORY, $\vdash M : \Sigma_1$ and $\vdash M' : \Sigma_2$. In addition, we have $\cdot; \cdot \vdash \Sigma \otimes \Sigma' = \Sigma_1 \otimes \Sigma_2$ and $\cdot; \cdot \vdash \Sigma' = \Sigma_2$, because $\vdash MM' : \Sigma \otimes \Sigma'$.

Here, by the lemma A.8, we have $\cdot; \cdot \vdash \Sigma = \Sigma_1$. Now, by the lemma A.40, $\vdash M : \Sigma$.

Lemma A.11
If $\vdash M : \Sigma, \vdash M' : \Sigma'$ and $\texttt{GU}(MM')$, then $\vdash MM' : \Sigma \otimes \Sigma'$.

Proof Let $M \equiv \{n_1 \mapsto a_1\} \ldots \{n_k \mapsto a_k\}$ and $M' \equiv \{n'_1 \mapsto a'_1\} \ldots \{n'_m \mapsto a'_m\}$. Then, by the typing rule MEMORY, we have $\cdot;\cdot \vdash \{n_1 \mapsto at_1\} \otimes \ldots \otimes \{n_k \mapsto at_k\} = \Sigma$ and $\cdot;\cdot \vdash \{n'_1 \mapsto at'_1\} \otimes \ldots \otimes \{n'_m \mapsto at'_m\} = \Sigma'$, where $\forall i.\cdot;\cdot \vdash a_i : at_i$ and $\forall i.\cdot;\cdot \vdash a'_i : at'_i$.

Here, by the typing rule MEMORY, we have $\vdash MM' : \{n_1 \mapsto at_1\} \otimes \ldots \otimes \{n_k \mapsto at_k\} \otimes \{n'_1 \mapsto at'_1\} \otimes \ldots \otimes \{n'_m \mapsto at'_m\}$ because $\texttt{GU}(MM')$. Now, by the lemma A.9, $\cdot;\cdot \vdash \{n_1 \mapsto at_1\} \otimes \ldots \otimes \{n_k \mapsto at_k\} \otimes \{n'_1 \mapsto at'_1\} \otimes \ldots \otimes \{n'_m \mapsto at'_m\} = \Sigma \otimes \Sigma'$. Thus, by the lemma A.40, we have $\vdash MM' : \Sigma \otimes \Sigma'$.

A.2 Type substitution lemmas

Lemma A.12 (Type substitution: integer constraints)
If $\Delta\Delta'\Delta'';C \models C'$, then $\Delta\Delta'';C\,[c_1,\ldots,c_n/\Delta'] \vdash C'\,[c_1,\ldots,c_n/\Delta']$.

Proof From the definition of the relation $\models$, C' is deduced from C, no matter how the integer variables in $\Delta\Delta'\Delta''$ are instantiated. Therefore, $C\,[c_1,\ldots,c_n/\Delta']$ is deduced from $C\,[c_1,\ldots,c_n/\Delta']$, no matter how the integer variables in $\Delta\Delta''$ are instantiated. Thus, $\Delta\Delta'';C\,[c_1,\ldots,c_n/\Delta'] \vdash C'\,[c_1,\ldots,c_n/\Delta']$.

Lemma A.13 (Type substitution: integer constraints equality)
If $\Delta\Delta'\Delta'';C \vdash C_1 = C_2$, then $\Delta\Delta'';C\,[c_1,\ldots,c_n/\Delta'] \vdash C_1\,[c_1,\ldots,c_n/\Delta'] = C_2\,[c_1,\ldots,c_n/\Delta']$.

Proof From the typing rule EQCSTRT, we have $\Delta\Delta'\Delta'';C \wedge C_1 \models C_2$ and $\Delta\Delta'\Delta'';C \wedge C_2 \models C_1$. Here, by the lemma A.12, $\Delta\Delta'';C\theta \wedge C_1\theta \models C_2\theta$ and $\Delta\Delta'';C\theta \wedge C_2\theta \models C_1\theta$. Thus, by the typing rule EQCSTRT, $\Delta\Delta'';C\theta \vdash C_1\theta = C_2\theta$.

Lemma A.14 (Type substitution: value type equality)
If $\Delta\Delta'\Delta'';C \vdash \sigma = \sigma'$, then $\Delta\Delta'';C\,[c_1,\ldots,c_n/\Delta'] \vdash \sigma\,[c_1,\ldots,c_n/\Delta'] = \sigma'\,[c_1,\ldots,c_n/\Delta']$.

Lemma A.15 (Type substitution: tuple type equality)
If $\Delta\Delta'\Delta'';C \vdash \tau = \tau'$, then $\Delta\Delta'';C\,[c_1,\ldots,c_n/\Delta'] \vdash \tau\,[c_1,\ldots,c_n/\Delta'] = \tau'\,[c_1,\ldots,c_n/\Delta']$.

Lemma A.16 (Type substitution: array type equality)
If $\Delta\Delta'\Delta'';C \vdash at = at'$, then $\Delta\Delta'';C\,[c_1,\ldots,c_n/\Delta'] \vdash at\,[c_1,\ldots,c_n/\Delta'] = at'\,[c_1,\ldots,c_n/\Delta']$.

Lemma A.17 (Type substitution: memory type equality)
If $\Delta\Delta'\Delta'';C \vdash \Sigma = \Sigma'$, then $\Delta\Delta'';C\,[c_1,\ldots,c_n/\Delta'] \vdash \Sigma\,[c_1,\ldots,c_n/\Delta'] = \Sigma'\,[c_1,\ldots,c_n/\Delta']$.

Lemma A.18 (Type substitution: registers type equality)
If $\Delta\Delta'\Delta'';C \vdash \Gamma = \Gamma'$, then $\Delta\Delta'';C\,[c_1,\ldots,c_n/\Delta'] \vdash \Gamma\,[c_1,\ldots,c_n/\Delta'] = \Gamma'\,[c_1,\ldots,c_n/\Delta']$.

Proof By induction on the typing derivation. the type substitution lemmas A.14,A.15,A.16,A.17 and A.18 are proved simultaneously. Let $\theta = [c_1,\ldots,c_n/\Delta']$.

Proof of the lemma A.14
The proof is by case analysis on the last rule of the derivation.

- Case EQINT
  From the typing rule, $\Delta\Delta'\Delta'';C \models i = i'$, where $\sigma \equiv i$ and $\sigma' \equiv i'$. This means that $i = i'$ is deduced from C for all the instantiation of the integer variables in $\Delta\Delta'\Delta''$. Therefore, $\Delta\Delta'';C\theta \models i\theta = i'\theta$. Thus, by the typing rule EQINT, $\Delta\Delta'';C\theta \vdash \sigma\theta = \sigma'\theta$.

- Case EQLABEL
  From the typing rule, $\Delta\Delta'\Delta''\Delta_1;C \vdash C_1 = C_2$, $\Delta\Delta'\Delta''\Delta_1;C \wedge C_1 \vdash \Sigma_1 = \Sigma_2$ and $\Delta\Delta'\Delta''\Delta_1;C \wedge C_1 \vdash \Gamma_1 = \Gamma_2$, where $\sigma_1 \equiv \forall\Delta_1.|C_1|[\Sigma_1](\Gamma_1)$, $\sigma_2 \equiv \forall\Delta_1.|C_2|[\Sigma_2](\Gamma_2)$. By the induction hypothesis and the lemmas A.13, A.17 and A.18, we have $\Delta\Delta''\Delta_1;C\theta \vdash C_1\theta = C_2\theta$, $\Delta\Delta''\Delta_1;C\theta \wedge C_1\theta \vdash \Sigma_1\theta = \Sigma_2\theta$ and $\Delta\Delta''\Delta_1;C\theta \wedge C_1\theta \vdash \Gamma_1\theta = \Gamma_2\theta$. Thus, by the typing rule EQLABEL, $\Delta\Delta'';C\theta \vdash \sigma\theta = \sigma'\theta$.

Proof of the lemma A.15
The proof is by case analysis on the last rule of the derivation.

- Case EQTUPLE
  From the typing rule, $\forall i.\Delta\Delta'\Delta'';C \vdash \sigma_i = \sigma'_i$, where $\tau = \langle\sigma_1,\ldots,\sigma_n\rangle$ and $\tau' = \langle\sigma'_1,\ldots,\sigma'_n\rangle$. Here, by the induction hypothesis and the lemma A.14, $\forall i.\Delta\Delta'';C\theta \vdash \sigma_i\theta = \sigma'_i\theta$. Thus, by the typing rule EQTUPLE, $\Delta\Delta'';C\theta \vdash \tau\theta = \tau'\theta$.

- Case EQEX
  From the typing rule, $\Delta\Delta'\Delta''\Delta_1; C \vdash C_1 = C_2$, $\Delta\Delta'\Delta''\Delta_1; C \wedge C_1 \vdash \Sigma_1 = \Sigma_2$ and $\Delta\Delta'\Delta''\Delta_1; C \wedge C_1 \vdash \tau_1 = \tau_2$, where $\tau = \exists\Delta_1.|C_1|[\Sigma_1]\tau_1$, $\tau' = \exists\Delta_1.|C_2|[\Sigma_2]\tau_2$. Here, by the induction hypothesis and the lemmas A.13 and A.17, we have $\Delta\Delta''\Delta_1 \vdash C_1\theta = C_2\theta$, $\Delta\Delta''\Delta_1; C\theta \wedge C_1\theta \vdash \Sigma_1\theta = \Sigma_2\theta$ and $\Delta\Delta''\Delta_1; C\theta \wedge C_1\theta \vdash \tau_1\theta = \tau_2\theta$. Thus, by the typing rule EQEX, $\Delta\Delta''; C\theta \vdash \tau\theta = \tau'\theta$.

- Case EQREC
  From the typing rule, we have $\tau \equiv \tau'$. where $\tau = \rho\,(c_1, \ldots, c_n)$. Thus, by the typing rule EQREC, $\Delta\Delta''; C\theta \vdash \tau\theta = \tau'\theta$.

Proof of the lemma A.16

From the typing rule EQARRAY, we have $\Delta\Delta'\Delta''; C \vdash \tau = \tau'$ and $\Delta\Delta'\Delta''; C \vdash i = i'$, where $at \equiv \tau[i]$ and $at' \equiv \tau'[i']$. Here, by the induction hypothesis and the lemmas A.15 and A.14 $\Delta\Delta''; C\theta \vdash \tau\theta = \tau'\theta$ and $\Delta\Delta''; C\theta \vdash i\theta = i'\theta$. Thus, by the typing rule EQARRAY, $\Delta\Delta''; C\theta \vdash at\theta = at'\theta$.

Proof of the lemma A.17

The proof is by case analysis on the last rule of the derivation.

- Case EQMEMEMPTY
  From the typing rule, we have $\Delta_1; C_1 \vdash \cdot = \cdot$ for any Δ_1 and C_1. Thus, $\Delta\Delta''; C\theta \vdash \cdot = \cdot$.

- Case EQMEMLOC
  From the typing rule, we have $\Delta\Delta'\Delta''; C \vdash \Sigma_1 = \Sigma_2$, $\Delta\Delta'\Delta''; C \vdash i_1 = i_2$ and $\Delta\Delta'\Delta''; C \vdash at_1 = at_2$, where $\Sigma \equiv \Sigma_1 \otimes \{i_1 \mapsto at_1\}$ and $\Sigma' \equiv \Sigma_2 \otimes \{i_2 \mapsto at_2\}$. Here, by the induction hypothesis, $\Delta\Delta''; C\theta \vdash \Sigma_1\theta = \Sigma_2\theta$. In addition, by the lemmas A.14 and A.16, $\Delta\Delta''; C\theta \vdash i_1\theta = i_2\theta$ and $\Delta\Delta''; C\theta \vdash at_1\theta = at_2\theta$. Thus, by the typing rule EQMEMLOC, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma'\theta$.

- Case EQMEMVAR
  From the typing rule, we have $\Delta\Delta'\Delta''; C \vdash \Sigma_1 = \Sigma_2$, where $\Sigma \equiv \Sigma_1 \otimes \epsilon$ and $\Sigma' \equiv \Sigma_2 \otimes \epsilon$. Now, by the induction hypothesis, $\Delta\Delta''; C\theta \vdash \Sigma_1\theta = \Sigma_2\theta$. Here if $\epsilon \notin \Delta'$, then by the typing rule EQMEMVAR, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma'\theta$. Otherwise (if $\epsilon \in \Delta'$), $\Sigma\theta \equiv \Sigma_1\theta \otimes \Sigma''$ and $\Sigma'\theta \equiv \Sigma_2\theta \otimes \Sigma''$, where $[\Sigma''/\epsilon] \in \theta$. Here $\Delta\Delta''; C\theta \vdash \Sigma'' = \Sigma''$. Thus, by the lemma A.9, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma'\theta$.

- Case EQMEMZEROARRAYL
  From the typing rule, we have $\Delta\Delta'\Delta''; C \vdash \Sigma'' = \Sigma'$, $\Delta\Delta'\Delta''; C \vdash i_2 = 0$, where $\Sigma \equiv \Sigma'' \otimes \{i_1 \mapsto \tau[i_2]\}$. Here, by the induction hypothesis, $\Delta\Delta''; C\theta \vdash \Sigma''\theta = \Sigma'\theta$. In addition, by the lemma A.14, $\Delta\Delta''; C\theta \vdash i_2\theta = 0$. Thus, by the typing rule EQMEMZEROARRAYL, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma'\theta$.

- Case EQMEMZEROARRAYR
  Same as the case EQMEMZEROARRAYL.

Proof of the lemma A.18
The proof is by case analysis on the last rule of the derivation.

- Case EQREGSNULL
  From the typing rule, we have $\Delta_1; C_1 \vdash \cdot = \cdot$ for any Δ_1 and C_1. Thus, $\Delta\Delta''; C\theta \vdash \cdot = \cdot$.

- Case EQREGSREG
  From the typing rule, we have $\Delta\Delta'\Delta''; C \vdash \Gamma_1 = \Gamma_2$ and $\Delta\Delta'\Delta''; C \vdash \sigma = \sigma'$, where $\Gamma \equiv \Gamma_1\{r : \sigma\}$ and $\Gamma' \equiv \Gamma_2\{r : \sigma'\}$. Now, by the induction hypothesis and the lemma A.14, $\Delta\Delta''; C\theta \vdash \Gamma_1\theta = \Gamma_2\theta$ and $\Delta\Delta''; C\theta \vdash \sigma\theta = \sigma'\theta$. Thus, by the typing rule EQREGSREG, $\Delta\Delta''; C\theta \vdash \Gamma\theta = \Gamma'\theta$.

Lemma A.19 (Type substitution: registers type subtyping)
If $\Delta\Delta'\Delta''; C \vdash \Gamma \leq \Gamma'$, then $\Delta\Delta''; C\,[c_1, \ldots, c_n/\Delta'] \vdash \Gamma\,[c_1, \ldots, c_n/\Delta'] \leq \Gamma'\,[c_1, \ldots, c_n/\Delta']$.

Proof By case analysis on the last rule of the derivation. Let $\theta \equiv [c_1, \ldots, c_n/\Delta']$.

- Case SUBREGSNULL
  From the typing rule, $\Gamma' \equiv \cdot$. Thus, by the typing rule SUBREGSNULL, $\Delta\Delta''; C\theta \vdash \Gamma\theta \leq \cdot$.

- Case SUBREGSREG
  From the typing rule, we have $\Delta\Delta'\Delta''; C \vdash \Gamma_1 \leq \Gamma_2$ and $\Delta\Delta'\Delta''; C \vdash \sigma = \sigma'$, where $\Gamma \equiv \Gamma_1\{r : \sigma\}$ and $\Gamma' \equiv \Gamma_2\{r : \sigma'\}$. Now, by the induction hypothesis and the lemma A.14, $\Delta\Delta''; C\theta \vdash \Gamma_1\theta \leq \Gamma_2\theta$ and $\Delta\Delta''; C\theta \vdash \sigma\theta = \sigma'\theta$. Thus, by the typing rule SUBREGSREG, $\Delta\Delta''; C\theta \vdash \Gamma\theta \leq \Gamma'\theta$.

Lemma A.20 (Type substitution: value)
If $\Delta\Delta'\Delta''; C \vdash v : \sigma$, then $\Delta\Delta''; C\,[c_1, \ldots, c_n/\Delta'] \vdash v\,[c_1, \ldots, c_n/\Delta'] : \sigma\,[c_1, \ldots, c_n/\Delta']$.

Proof By case analysis on the last rule of the derivation. Let $\theta \equiv [c_1, \ldots, c_n/\Delta']$.

- Case VALUEINTEGER
  From the typing rule, $\Delta\Delta'\Delta''; C \models n = i$, where $v \equiv n$ and $\sigma \equiv i$. Here, by the lemma A.12, $\Delta\Delta''; C\theta \models n = i\theta$. Thus, by the typing rule VALUEINTEGER, we have $\Delta\Delta''; C\theta \vdash n = i\theta$. That is, $\Delta\Delta''; C\theta \vdash v\theta : \sigma\theta$.

- Case VALUELABEL
  From the typing rule, $v \equiv l\theta'$ and $\Delta\Delta'\Delta''; C \vdash \sigma = \forall\Delta_1\backslash\Delta_2.|C''|[\Sigma''](\Gamma'')$, where $\theta' \equiv [c'_1, \ldots, c'_m/\Delta_2]$, $C'' \equiv C'\theta'$, $\Sigma'' \equiv \Sigma'\theta'$, $\Gamma'' \equiv \Gamma'\theta'$ and $\Phi(l) \equiv \forall\Delta_1.|C'|[\Sigma'](\Gamma')$. Now, by the lemma A.14, $\Delta\Delta''; C\theta\sigma\theta = (\forall\Delta_1\backslash\Delta_2.|C''|[\Sigma''](\Gamma''))\theta$. Here $(\forall\Delta_1\backslash\Delta_2.|C''|[\Sigma''](\Gamma''))\theta \equiv \forall\Delta_1\backslash\Delta_2.|C''\theta|[\Sigma''\theta](\Gamma''\theta)$. In addition, let $\theta'' \equiv [c_1\theta, \ldots, c_n\theta/\Delta'']$. Then, $C'' \equiv C'\theta''$, $\Sigma'' \equiv C'\theta''$, $\Gamma'' \equiv \Gamma'\theta''$ and $l\theta'\theta \equiv l\theta''$. Thus, by the typing rule VALUELABEL, $\Delta\Delta''; C\theta \vdash v\theta : \sigma\theta$.

Lemma A.21 (Type substitution: instructions)
If $\Delta\Delta'\Delta''; \Gamma; C; \Sigma \vdash I$, then $\Delta\Delta''; \Gamma[c_1, \ldots, c_n/\Delta']; C[c_1, \ldots, c_n/\Delta']; \Sigma[c_1, \ldots, c_n/\Delta'] \vdash I[c_1, \ldots, c_n/\Delta']$.

Proof By case analysis on the last rule of the derivation. Let $\theta \equiv [c_1, \ldots, c_n/\Delta']$.

- Case LOAD
  From the typing rule, we have $I = \texttt{ld}\ [r_s + n], r_d; I'$, $\Delta\Delta'\Delta''; C \vdash \Sigma = \Sigma' \otimes \{\Gamma(r_s) \mapsto \langle\sigma_1, \ldots, \sigma_n, \ldots, \sigma'_n\rangle\}$ and $\Delta\Delta'\Delta''; \Gamma\{r_d \mapsto \sigma_n\}; C; \Sigma \vdash I'$. Here, by the lemma A.17, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma'\theta \otimes \{\Gamma(r_s)\theta \mapsto \langle\sigma_1\theta, \ldots, \sigma_n\theta, \ldots, \sigma'_n\theta\rangle\}$. Therefore, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma'\theta \otimes \{\Gamma\theta(r_s) \mapsto \langle\sigma_1\theta, \ldots, \sigma_n\theta, \ldots, \sigma'_n\theta\rangle\}$ because $\Gamma(r_s)\theta = \Gamma\theta(r_s)$. Now, by the induction hypothesis, $\Delta\Delta''; \Gamma\theta\{r_d \mapsto \sigma_n\theta\}; C\theta; \Sigma\theta \vdash I'\theta$. Thus, by the typing rule LOAD, $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I\theta$.

- Case STORE
  From the typing rule, we have $I = \texttt{st}\ r_s, [r_d + n]; I'$, $\Delta\Delta'\Delta''; C \vdash \Sigma = \Sigma' \otimes \{\Gamma(r_d) \mapsto \langle\sigma_1, \ldots, \sigma_n, \ldots, \sigma'_n\rangle\}$ and $\Delta\Delta'\Delta''; \Gamma; C; \Sigma' \otimes \{\Gamma(r_d) \mapsto \langle\sigma_1, \ldots, \Gamma(r_s), \ldots, \sigma'_n\rangle\} \vdash I'$. Here, by the lemma A.17, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma'\theta \otimes \{\Gamma(r_d)\theta \mapsto \langle\sigma_1\theta, \ldots, \sigma_n\theta, \ldots, \sigma_{n'}\theta\rangle\}$. Therefore, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma'\theta \otimes \{\Gamma\theta(r_d) \mapsto \langle\sigma_1\theta, \ldots, \sigma_n\theta, \ldots, \sigma_{n'}\theta\rangle\}$, because $\Gamma\theta(r_d) = \Gamma(r_d)\theta$. Now, by the induction hypothesis, $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma'\theta \otimes \{\Gamma\theta(r_d) \mapsto \langle\sigma_1\theta, \ldots, \Gamma\theta(r_s), \ldots, \sigma'_n\rangle\}$. Thus, by the typing rule STORE, $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I\theta$.

- Case MOVE
  From the typing rule, we have $I = \texttt{mov}\ r_s, r_d; I'$ and $\Delta\Delta'\Delta''; \Gamma\{r_d \mapsto \Gamma(r_s)\}; C; \Sigma \vdash I'$. Here, by the induction hypothesis, $\Delta\Delta''; \Gamma\theta\{r_d \mapsto \Gamma\theta(r_s)\}; C\theta; \Sigma\theta \vdash I'\theta$. Thus, by the typing rule MOVE, $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I\theta$.

- Case MOVEI
  From the typing rule, we have $I = \texttt{movi}\ v, r_d; I'$, $\Delta\Delta'\Delta'' \vdash v : \sigma$ and $\Delta\Delta'\Delta''; \Gamma\{r_d \mapsto \sigma\}; C; \Sigma \vdash I'$. Here, by the induction hypothesis and the lemma A.20, $\Delta\Delta'' \vdash v\theta : \sigma\theta$ and $\Delta\Delta'' \vdash; \Gamma\theta\{r_d \mapsto \sigma\theta\}; C\theta; \Sigma\theta \vdash I'\theta$. Thus, by the typing rule MOVEI $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I\theta$.

- Case ARITH
  From the typing rule, we have $I = (\texttt{add}, \texttt{sub}, \texttt{mul})\ r_{s_1}, r_{s_2}, r_d; I'$ and $\Delta\Delta'\Delta''; \Gamma\{r_d \mapsto \Gamma(r_{s_2})(+,-,*)\Gamma(r_{s_1})\}; C; \Sigma \vdash I$. Thus, by the induction hypothesis, $\Delta\Delta''; \Gamma\theta\{r_d \mapsto \Gamma\theta(r_{s_2})(+,-,*)\Gamma\theta(r_{s_1})\}; C\theta; \Sigma\theta \vdash I\theta$. Thus, by the typing rule ARITH, $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I\theta$.

- Case BRANCH
  From the typing rule, we have $I = (\texttt{beq}, \texttt{ble})\ r_{s_1}, r_{s_2}, r_d; I'$, $\Delta\Delta'\Delta''; C \vdash \Gamma(r_d) = \forall.|C'|[\Sigma'](\gamma')$, $\Delta\Delta'\Delta''; C'' \vdash C'$, $\Delta\Delta'\Delta''; C'' \vdash \Sigma = \Sigma'$, $\Delta\Delta'\Delta''; C'' \vdash \Gamma \leq \Gamma'$ and $\Delta\Delta'\Delta''; C \wedge \Gamma(r_{s_1})(\neq,>)\Gamma(r_{s_2}); \Sigma \vdash I'$, where $C'' \equiv C \wedge \Gamma(r_{s_1})(=,\leq)\Gamma(r_{s_2})$. Here, by the induction hypothesis, $\Delta\Delta''; C\theta \vdash \Gamma\theta(r_d) = (\forall.|C'|[\Sigma'](\Gamma'))\theta = \forall.|C'\theta|[\Sigma'\theta](\Gamma'\theta)$ (by the lemma A.14, $\Delta\Delta''; C''\theta \vdash C'\theta$ (by the lemma A.12, $\Delta\Delta''; C''\theta \vdash \Sigma\theta = \Sigma'\theta$ (by the lemma A.17), $\Delta\Delta''; C''\theta \vdash \Gamma\theta \leq \Gamma'\theta$ (by the lemma A.19) and $\Delta\Delta''; C\theta \wedge \Gamma\theta(r_{s_1})(\neq,>)\Gamma\theta(r_{s_2}); \Sigma\theta \vdash I'\theta$. Thus, by the typing rule BRANCH, we have $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I\theta$.

- Case JUMP
  From the typing rule, we have $I = \texttt{jmp}\ r_d$, $\Delta\Delta'\Delta''; C \vdash \Gamma(r_d) = \forall.|C'|[\Sigma'](\Gamma')$, $\Delta\Delta'\Delta''; C \vdash C'$, $\Delta\Delta'\Delta''; C \vdash \Sigma = \Sigma'$ and $\Delta\Delta'\Delta''; C \vdash \Gamma \leq \Gamma'$. Here, by the induction hypothesis, $\Delta\Delta''; C\theta \vdash \Gamma\theta(r_d) = (\forall.|C'|[\Sigma'](\Gamma'))\theta = \forall.|C'\theta|[\Sigma'\theta](\Gamma'\theta)$, $\Delta\Delta''; C\theta \vdash C'\theta$, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma'\theta$ and $\Delta\Delta''; C\theta \vdash \Gamma\theta \leq \Gamma'\theta$. Thus, by the typing rule JUMP, $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I\theta$.

- Case APPLY
  From the typing rule, we have $I = \texttt{apply}\ r\ [c'_1, \ldots, c'_m/\Delta_2]\ ; I'$, $\Gamma(r) \equiv \forall\Delta_1.|C'|[\Sigma'](\Gamma')$ and $\Delta\Delta'\Delta''; \Gamma\{r \mapsto \forall\Delta_1\backslash\Delta_2.|C''|[\Sigma''](\Gamma'')\}; C; \Sigma \vdash I'$, where $C'' \equiv C'\theta'$, $\Sigma'' \equiv \Sigma'\theta'$, $\Gamma'' \equiv \Gamma'\theta'$ and $\theta' \equiv [c'_1, \ldots, c'_m/\Delta_2]$. Here, by the induction hypothesis, $\Delta\Delta''; \Gamma\theta\{r \mapsto (\forall\Delta_1\backslash\Delta_2.|C''|[\Sigma''](\Gamma''))\theta\}; C\theta; \Sigma\theta \vdash$

$I'\theta$. Now, $(\forall\Delta_1\backslash\Delta_2.|C''|[\Sigma''](\Gamma''))\theta \equiv \forall\Delta_1\backslash\Delta_2.|C''\theta|[\Sigma''\theta](\Gamma''\theta)$. Here let $\theta'' \equiv [c'_1\theta, \ldots, c'_m\theta/\Delta_2]$. Then, $C''\theta \equiv (C'\theta')\theta = (C'\theta)\theta''$, $\Sigma'' \equiv (\Sigma'\theta')\theta = (\Sigma'\theta)\theta''$ and $\Gamma'' \equiv (\Gamma'\theta')\theta = (\Gamma'\theta)\theta''$. Thus, we have $\Delta\Delta''; \Gamma\theta\{r \mapsto \forall\Delta_1\backslash\Delta_2.|(C'\theta)\theta''|[(\Sigma'\theta)\theta'']((\Gamma'\theta)\theta'')\}; C\theta; \Sigma\theta \vdash I\theta$. Therefore, by the typing rule APPLY, $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I\theta$ because $I\theta \equiv \texttt{apply}\ r\theta''; I'\theta$.

- Case ROLL
  From the typing rule, we have $I = \texttt{roll}_\tau\ i; I'$, $\Delta\Delta'\Delta''; C \vdash \Sigma = \Sigma' \otimes \{i \mapsto \tau'[\mu\eta[\Delta_1].\tau'/\eta][c'_1, \ldots, c'_m/\Delta_1]\}$ and $\Delta\Delta'\Delta''; \Gamma; C; \Sigma' \otimes \{i \mapsto \tau\} \vdash I'$, where $\tau \equiv \mu\eta[\Delta_1].\tau'(c'_1, \ldots, c'_m)$. Here, by the induction hypothesis and the lemma A.17, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma'\theta \otimes \{i\theta \mapsto (\tau'[\mu\eta[\Delta_1].\tau'/\eta][c'_1, \ldots, c'_m/\Delta_1])\theta\}$ and $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma'\theta \otimes \{i\theta \mapsto \tau\theta\} \vdash I'\theta$. Now, $(\tau'[\mu\eta[\Delta_1].\tau'/\eta][c'_1, \ldots, c'_m/\Delta_1])\theta = \tau'\theta[\mu\eta[\Delta_1].\tau'\theta/eta][c'_1\theta, \ldots, c'_m\theta/\Delta_1]$. Thus, by the typing rule ROLL, $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I\theta(= \texttt{roll}_{\tau\theta}\ i\theta; I'\theta)$.

- Case UNROLL
  From the typing rule, we have $I = \texttt{unroll}\ i; I'$, $\Delta\Delta'\Delta''; C \vdash \Sigma = \Sigma' \otimes \{i \mapsto \mu\eta[\Delta_1].\tau'(c'_1, \ldots, c'_m)\}$ and $\Delta\Delta'\Delta''; \Gamma; C; \Sigma' \otimes \{i \mapsto \tau'[\mu\eta[\Delta_1].\tau'/\eta][c'_1, \ldots, c'_m/\Delta_1] \vdash I'$. Here, by the induction hypothesis and the lemma A.17, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma'\theta \otimes \{\imath\theta(\mu\eta[\Delta_1].\tau'(c'_1, \ldots, c'_m))\theta\}$ and $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma'\theta \otimes \{i\theta \mapsto (\tau'[\mu\eta[\Delta_1].\tau'/\eta][c'_1, \ldots, c'_m/\Delta_1])\theta \vdash I'\theta$. Now, because $(\mu\eta[\Delta_1].\tau'(c'_1, \ldots, c'_m))\theta = \mu\eta[\Delta_1].\tau'\theta(c'_1\theta, \ldots, c'_m\theta)$ and $(\tau'[\mu\eta[\Delta_1].\tau'/\eta][c'_1, \ldots, c'_m/\Delta_1])\theta = \tau'\theta[\mu\eta[\Delta_1].\tau'\theta/\eta][c'_1\theta, \ldots, c'_m\theta/\Delta_1]$, we have $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I\theta(= \texttt{unroll}\ i\theta; I'\theta)$ by the typing rule UNROLL.

- Case PACK
  From the typing rule, we have $I = \texttt{pack}_{[c'_1, \ldots, c'_m|\Sigma'\theta']\ as\ \tau}\ i; I'$, $\Delta\Delta'\Delta''; C \vdash \Sigma = \Sigma'' \otimes \{i \mapsto \tau'\theta'\} \otimes \Sigma'\theta'$, $\Delta\Delta'\Delta''; C \models C'\theta'$ and $\Delta\Delta'\Delta''; \Gamma; C; \Sigma'' \otimes \{i \mapsto \tau\} \vdash I'$, where $\theta' \equiv [c'_1, \ldots, c'_m/\Delta_1]$ and $\tau \equiv \exists\Delta'.|C'|[\Sigma']\tau'$. Here, by the induction hypothesis and the lemma A.17, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma''\theta \otimes \{i\theta \mapsto (\tau'\theta')\theta\} \otimes (\Sigma'\theta')\theta$, $\Delta\Delta''; C\theta \models (C'\theta')\theta$ and $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma''\theta \otimes \{i\theta \mapsto \tau\theta\} \vdash I'\theta$ Now, let $\theta'' \equiv [c'_1\theta, \ldots, c'_m\theta/\Delta_1]$. Then, $(\tau'\theta')\theta \equiv (\tau'\theta)\theta''$, $(\Sigma'\theta')\theta \equiv (\Sigma'\theta)\theta''$ and $(C'\theta')\theta \equiv (C'\theta)\theta''$. In addition, $\tau\theta \equiv \exists\Delta_1.|C'\theta|[\Sigma'\theta]\tau'\theta$. Thus, we have $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma''\theta \otimes \{i\theta \mapsto (\tau'\theta)\theta''\}$, $\Delta\Delta''; C\theta \vdash (C'\theta)\theta''$ and $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma''\theta \otimes \{i\theta \mapsto \exists\Delta_1.|C'\theta|[\Sigma'\theta]\tau'\theta$. Here $I\theta = \texttt{pack}_{[c'_1\theta, \ldots, c'_m|\Sigma'\theta'']\ as\ \exists\Delta_1.|C'\theta|[\Sigma'\theta]\tau'\theta}\ i\theta; I'\theta$. Therefore, by the typing rule PACK, $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I$.

- Case UNPACK
  From the typing rule, we have $I = \texttt{unpack}\ i\ \texttt{with}\ \Delta_2; I'$, $\Delta\Delta'\Delta''; C \vdash$

$\Sigma = \Sigma'' \otimes \{i \mapsto \exists \Delta_1.|C'|[\Sigma']\tau\}$ and $\Delta\Delta'\Delta''\Delta_2; \Gamma; C \wedge C'\,[\Delta_2/\Delta_1]\,; \Sigma'' \otimes \{i \mapsto \tau\,[\Delta_2/\Delta_1]\} \otimes \Sigma'\,[\Delta_2/\Delta_1] \vdash I'$. Here, by the induction hypothesis and the lemma A.17, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma''\theta \otimes \{i\theta \mapsto (\exists \Delta_1.|C'|[\Sigma']\tau)\theta\}$ and $\Delta\Delta''\Delta_2; \Gamma\theta; C\theta \wedge (C'\,[\Delta_2/\Delta_1])\theta; \Sigma''\theta \otimes \{i\theta \mapsto (\tau\,[\Delta_2/\Delta_1])\theta\} \otimes (\Sigma'\,[\Delta_2/\Delta_1])\theta \vdash I'\theta$. Now, $(\exists \Delta_1.|C'|[\Sigma']\tau)\theta \equiv \exists \Delta_1.|C'\theta|[\Sigma'\theta]\tau\theta$, $(C'\,[\Delta_2/\Delta_1])\theta \equiv (C'\theta)\,[\Delta_2/\Delta_1]$, $(\tau\,[\Delta_2/\Delta_1])\theta \equiv (\tau\theta)\,[\Delta_2/\Delta_1]$ and $(\Sigma'\,[\Delta_2/\Delta_1])\theta \equiv (\Sigma'\theta)\,[\Delta_2/\Delta_1]$. Thus, by the typing rule UNPACK, we have $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I\theta$, because $I\theta \equiv \texttt{unroll}\ i\theta; I'\theta$.

- Case SPLIT
  From the typing rule, we have $I = \texttt{split}\ i_1, i_2; I'$, $\Delta\Delta'\Delta''; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \tau[j_1]\}$, $\Delta\Delta'\Delta''; C \models 0 \leq i_2 \leq j_1$ and $\Delta\Delta'\Delta''; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \tau[i_2]\} \otimes \{k_1 \mapsto \tau[k_2]\} \vdash I'$, where $k_1 \equiv i_1 + \mathit{sizeof}(\tau) * i_2$ and $k_2 \equiv j_1 - i_2$. Here, by the induction hypothesis and the lemmas A.17 and A.12, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma'\theta \otimes \{i_1\theta \mapsto \tau\theta[j_1\theta]\}$, $\Delta\Delta''; C\theta \models 0 \leq i_2\theta \leq j_1\theta$ and $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma'\theta \otimes \{i_1\theta \mapsto \tau\theta[i_2\theta]\} \otimes \{k_1\theta \mapsto \tau\theta[k_2\theta]\} \vdash I'\theta$. Now, because $\mathit{sizeof}(\tau) = \mathit{sizeof}(\tau\theta)$, by the typing rule SPLIT, we have $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I\theta (= \texttt{split}\ i_1\theta, i_2\theta; I'\theta)$.

- Case CONCAT
  From the typing rule, we have $I = \texttt{conat}\ i_1, i_2, i_3; I'$, $\Delta\Delta'\Delta''; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \tau[i_2]\} \otimes \{j_1 \mapsto \tau[j_2]\}$, $\Delta\Delta'\Delta''; C \models j_1 = i_1 + \mathit{sizeof}(\tau) * i_2$ and $\Delta\Delta'\Delta''; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \tau[i_2 + j_2]\} \vdash I'$. Here, by the induction hypothesis and the lemmas A.17 and A.12, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma'\theta \otimes \{i_1\theta \mapsto \tau\theta[i_2\theta]\} \otimes \{j_1\theta \mapsto \tau\theta[j_2\theta]\}$ and $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma'\theta \otimes \{i_1\theta \mapsto \tau\theta[i_2\theta + j_2\theta]\} \vdash I'\theta$. Now, because $\mathit{sizeof}(\tau) = \mathit{sizeof}(\tau\theta)$, by the typing rule CONCAT, we have $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I\theta (= \texttt{concat}\ i_1\theta, i_2\theta, i_3\theta; I'\theta)$.

- Case TUPLESPLIT
  From the typing rule, we have $I = \texttt{tuple\_split}\ i_1, n_2; I'$, $\Delta\Delta'\Delta''; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n \rangle\}$, $\Delta\Delta'\Delta''; C \models 0 < n_2 < n$ and $\Delta\Delta'\Delta''; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_{n_2} \rangle\} \otimes \{i_1 + n_2 \mapsto \langle \sigma_{n_2+1}, \ldots, \sigma_n \rangle\} \vdash I'$. Here, by the induction hypothesis and the lemmas A.17 and A.12, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma'\theta \otimes \{i_1\theta \mapsto \langle \sigma_1\theta, \ldots, \sigma_n\theta \rangle\}$, $\Delta\Delta''; C\theta \models 0 < n_2 < n$ and $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma'\theta \otimes \{i_1\theta \mapsto \langle \sigma_1\theta, \ldots, \sigma_{n_2}\theta \rangle\} \otimes \{i_1\theta + n_2 \mapsto \langle \sigma_{n_2+1}\theta, \ldots, \sigma_n\theta \rangle\} \vdash I'\theta$. Now, by the typing rule TUPLESPLIT, $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I\theta (= \texttt{tuple\_split}\ i_1\theta, n_2; I'\theta)$.

- Case TUPLECONCAT
  From the typing rule, we have $I = \texttt{tuple\_concat}\ i_1, i_2; I'$, $\Delta\Delta'\Delta''; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n \rangle\} \otimes \{i_2 \mapsto \langle \sigma'_1, \ldots, \sigma'_m \rangle\}$, $\Delta\Delta'\Delta''; C \models i_2 =$

$i_1 + n$ and $\Delta\Delta'\Delta''; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_m \rangle\} \vdash I'$. Here, by the induction hypothesis and the lemmas A.17 and A.12, $\Delta\Delta''; C\theta \vdash \Sigma\theta = \Sigma'\theta \otimes \{i_1\theta \mapsto \langle \sigma_1\theta, \ldots, \sigma_n\theta \rangle\} \otimes \{i_2\theta \mapsto \langle \sigma'_1\theta, \ldots, \sigma'_m\theta \rangle\}$, $\Delta\Delta''; C\theta \models i_2\theta = i_1\theta + n$ and $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma'\theta \otimes \{i_1\theta \mapsto \langle \sigma_1\theta, \ldots, \sigma_n\theta, \sigma'_1\theta, \ldots, \sigma'_m\theta \rangle\} \vdash I'\theta$. Now, by the typing rule TUPLECONCAT, $\Delta\Delta''; \Gamma\theta; C\theta; \Sigma\theta \vdash I\theta (= \texttt{tuple\_concat}\ i_1\theta, i_2\theta; I'\theta)$.

A.3 Constraints weakening lemmas

Lemma A.22 (Constraints weakening: integer constraints)
If $\Delta; C' \models C$ and $\Delta; C \models C''$, then $\Delta; C' \models C''$.

Proof From the definition of the relation $\models$, C'' is deduced from C and C is deduced from C'. That is, C'' is deduced from C. Thus, $\Delta; C' \models C''$.

Lemma A.23 (Constraints weakening: integer constraints equality)
If $\Delta; C' \models C$ and $\Delta; C \vdash C_1 = C_2$, then $\Delta; C' \vdash C_1 = C_2$.

Proof From the typing rule, we have $\Delta; C \wedge C_1 \models C_2$ and $\Delta; C \wedge C_2 \models C_1$. Here, from the definition of the relation $\models$, $\Delta; C' \wedge C_1 \models C \wedge C_1$ and $\Delta; C' \wedge C_2 \models C \wedge C_2$. Thus, by the typing rule EQCSTRT, $\Delta; C' \vdash C_1 = C_2$.

Lemma A.24 (Constraints weakening: value type equality)
If $\Delta; C' \models C$ and $\Delta; C \vdash \sigma = \sigma'$, then $\Delta; C' \vdash \sigma = \sigma'$.

Lemma A.25 (Constraints weakening: tuple type equality)
If $\Delta; C' \models C$ and $\Delta; C \vdash \tau = \tau'$, then $\Delta; C' \vdash \tau = \tau'$.

Lemma A.26 (Constraints weakening: array type equality)
If $\Delta; C' \models C$ and $\Delta; C \vdash at = at'$, then $\Delta; C' \vdash at = at'$.

Lemma A.27 (Constraints weakening: memory type equality)
If $\Delta; C' \models C$ and $\Delta; C \vdash \Sigma = \Sigma'$, then $\Delta; C' \vdash \Sigma = \Sigma'$.

Lemma A.28 (Constraints weakening: registers type equality)
If $\Delta; C' \models C$ and $\Delta; C \vdash \Gamma = \Gamma'$, then $\Delta; C' \vdash \Gamma = \Gamma'$.

Proof By induction on the typing derivation. the lemmas A.24, A.25, A.26, A.27 and A.28 are proved simultaneously.

Proof of the lemma A.24
The proof is by case analysis on the last rule of the derivation.

- Case EQINT
  From the typing rule, we have $\sigma \equiv i$, $\sigma' \equiv i'$ and $\Delta; C \models i = i'$. Here, by the lemma A.22, $\Delta; C' \models i = i'$. Thus, by the typing rule EQINT, we have $\Delta; C' \models \sigma = \sigma'$.

- Case EQLABEL
  From the typing rule, we have $\Delta\Delta'; C \vdash C_1 = C_2$, $\Delta\Delta'; C \wedge C_1 \vdash \Sigma_1 = \Sigma_2$ and $\Delta\Delta'; C \wedge C_1 \vdash \Gamma_1 = \Gamma_2$, where $\sigma \equiv \forall\Delta'.|C_1|[\Sigma_1](\Gamma_1)$ and $\sigma' \equiv \forall\Delta'.|C_2|[\Sigma_2](\Gamma_2)$. Here, by the lemma A.23, $\Delta\Delta'; C' \vdash C_1 = C_2$. Next, by the definition of the relation $\models$, we have $\Delta\Delta'; C' \wedge C_1 \models C \wedge C_1$. Therefore, by the induction hypothesis and the lemma A.27, $\Delta\Delta'; C' \wedge C_1 \vdash \Sigma_1 = \Sigma_2$. In addition, by the induction hypothesis and the lemma A.28, $\Delta\Delta'; C' \wedge C_1 \vdash \Gamma_1 = \Gamma_2$. Thus, by the typing rule EQLABEL, we have $\Delta; C' \vdash \sigma = \sigma'$.

Proof of the lemma A.25
The proof is by case analysis on the last rule of the derivation.

- Case EQTUPLE
  From the typing rule, $\forall i.\Delta; C \vdash \sigma_i = \sigma'_i$, where $\tau \equiv \langle\sigma_1, \ldots, \sigma_n\rangle$ and $\tau' \equiv \langle\sigma'_1, \ldots, \sigma'_n\rangle$. Here, by the induction hypothesis and the lemma A.24, $\forall i.\Delta; C7 \vdash \sigma_i = \sigma'_i$. Thus, by the typing rule EQTUPLE, we have $\Delta; C \vdash \tau = \tau'$.

- Case EQEX
  From the typing rule, we have $\Delta\Delta'; C \vdash C_1 = C_2$, $\Delta\Delta'; C \wedge C_1 \vdash \Sigma_1 = \Sigma_2$ and $\Delta\Delta'; C \wedge C_1 \vdash \tau_1 = \tau_2$, where $\tau \equiv \exists\Delta'.|C_1|[\Sigma_1]\tau_1$ and $\tau' \equiv \exists\Delta'.|C_2|[\Sigma_2]\tau_2$. Here, by the lemma A.23, $\Delta\Delta'; C' \vdash C_1 = C_2$. Next, by the induction hypothesis and the lemma A.27, $\Delta\Delta'; C' \wedge C_1 \vdash \Sigma_1 = \Sigma_2$ because $\Delta\Delta'; C' \wedge C_1 \models C \wedge C_1$. Last, by the induction hypothesis, $\Delta\Delta'; C' \wedge C_1 \vdash \tau_1 = \tau_2$. Thus, by the typing rule EQEX, we have $\Delta; C' \vdash \tau = \tau'$.

- Case EQREC
  From the typing rule, $\tau \equiv\equiv \tau'\rho\,(c_1, \ldots, c_n)$. Thus, by the typing rule EQREC, we have $\Delta; C' \vdash \tau = \tau'$, because the rule does not depend on integer constraints.

Proof of the lemma A.26

From the typing rule EQARRAY, we have $\Delta; C \vdash \tau = \tau'$ and $\Delta; C \models i = i'$, where $at \equiv \tau[i]$ and $at' \equiv \tau'[i']$. Here, by the induction hypothesis and the lemma A.25, $\Delta; C' \vdash \tau = \tau'$. In addition, by the lemma A.22, $\Delta; C' \models i = i'$. Thus, by the typing rule EQARRAY, we have $\Delta; C \vdash at = at'$

Proof of the lemma A.27

The proof is by case analysis on the last rule of the derivation.

- Case EQMEMEMPTY
  From the typing rule, $\Sigma \equiv \Sigma' \equiv \cdot$. Thus, by the typing rule EQMEMEMPTY, $\Delta; C' \vdash \Sigma = \Sigma'$.

- Case EQMEMLOC
  From the typing rule, we have $\Sigma \equiv \Sigma_1 \otimes \{i_1 \mapsto at_1\} = \Sigma_2 \otimes \{i_2 \mapsto at_2\}$, $\Delta; C \vdash \Sigma_1 = \Sigma_2$, $\Delta; C \models i_1 = i_2$ and $\Delta; C \vdash at_1 = at_2$. Here, by the induction hypothesis, $\Delta; C' \vdash \Sigma_1 = \Sigma_2$. Next, by the lemma A.22, $\Delta; C' \vdash i_1 = i_2$. Then, by the induction hypothesis and the lemma A.26, $\Delta; C' \vdash at_1 = at_2$. Thus, by the typing rule EQMEMLOC, we have $\Delta; C' \vdash \Sigma = \Sigma'$.

- Case EQMEMVAR
  From the typing rule, we have $\Sigma \equiv \Sigma_1 \otimes \epsilon$ and $\Sigma' \equiv \Sigma_2 \otimes \epsilon$ and $\Delta; C \vdash \Sigma_1 = \Sigma_2$. Here, by the induction hypothesis, we have $\Delta; C' \vdash \Sigma_1 = \Sigma_2$. Thus, by the typing rule EQMEMVAR, $\Delta; C' \vdash \Sigma = \Sigma'$.

- Case EQMEMZEROARRAYL
  From the typing rule, we have $\Sigma \equiv \Sigma_1 \otimes \{i_1 \mapsto \tau[i_2]\}$, $\Delta; C \vdash \Sigma_1 = \Sigma'$ and $\Delta; C \models i_2 = 0$. Here, by the induction hypothesis, $\Delta; C' \vdash \Sigma_1 = \Sigma'$. Then, by the lemma A.22, $\Delta; C' \models i_2 = 0$. Thus, by the typing rule EQMEMZEROARRAYL, we have $\Delta; C' \vdash \Sigma = \Sigma'$.

- Case EQMEMZEROARRAYR
  Same as the case EQMEMZEROARRAYL.

Proof of the lemma A.28

The proof is by case analysis on the last rule of the derivation.

- Case EQREGSNULL
  From the typing rule, $\Gamma \equiv \Gamma' \equiv \cdot$. Now, by the typing rule EQREGSNULL, $\Delta; C' \vdash \Gamma = \Gamma'$, because the rule does not depend on integer constraints.

- Case EQREGSREG
  From the typing rule, $\Gamma \equiv \Gamma_1\{r : \sigma\}$, $\Gamma' \equiv \Gamma_2\{r : \sigma'\}$, $\Delta; C \vdash \Gamma_1 = \Gamma_2$ and $\Delta; C \vdash \sigma = \sigma'$. Here, by the induction hypothesis, $\Delta; C' \vdash \Gamma_1 = \Gamma_2$. In addition, by the induction hypothesis and the lemma A.24, $\Delta; C' \vdash \sigma = \sigma'$. Thus, by the typing rule EQREGSREG, $\Delta; C \vdash \Gamma = \Gamma'$.

Lemma A.29 (Constraints weakening: registers type subtyping)
If $\Delta; C' \models C$ and $\Delta; C \vdash \Gamma \leq \Gamma'$, then $\Delta; C' \vdash \Gamma \leq \Gamma'$.

Proof By induction on the derivation $\Delta; C \vdash \Gamma \leq \Gamma'$. The proof is by case analysis on the last rule of the derivation.

- Case SUBREGSNULL
  From the typing rule, $\Gamma' \equiv \cdot$. Thus, $\Delta; C \vdash \Gamma \leq \Gamma'$, because the rule does not depend on integer constraints.

- Case SUBREGSREG
  From the typing rule, $\Gamma \equiv \Gamma_1\{r : \sigma\}$, $\Gamma \equiv \Gamma_2\{r : \sigma'\}$, $\Delta; C \vdash \Gamma_1 \leq \Gamma_2$ and $\Delta; C \vdash \sigma = \sigma'$. Here, by the induction hypothesis, $\Delta; C' \vdash \Gamma_1 \leq \Gamma_2$. In addition, by the lemma A.24, $\Delta; C' \vdash \sigma = \sigma'$. Thus, by the typing rule SUBREGSREG, $\Delta; C' \vdash \Gamma \leq \Gamma'$.

Lemma A.30 (Constraints weakening: value type)
If $\Delta; C' \models C$ and $\Delta; C \vdash v : \sigma$, then $\Delta; C' \vdash v : \sigma$.

Proof By case analysis on the last rule of the derivation $\Delta; C \vdash v : \sigma$.

- Case VALUEINTEGER
  From the typing rule, $\Delta; C \models n = i$, where $v \equiv n$ and $\sigma \equiv i$. Here, by the lemma A.22, $\Delta; C' \models n = i$. Therefore, by the typing rule VALUEINTEGER, we have $\Delta; C' \vdash v : \sigma$.

- Case VALUELABEL
  From the typing rule, $v \equiv l\theta$ and $\Delta; C \vdash \sigma = \forall\Delta'\backslash\Delta''.|C''|[\Sigma'']$, where $\theta \equiv [c_1, \ldots, c_n/\Delta'']$, $\Phi(l) \equiv \forall\Delta'.|C_1|[\Sigma'](\Gamma')$, $C'' \equiv C_1\theta$, $\Sigma'' \equiv \Sigma'\theta$ and $\Gamma'' \equiv \Gamma'\theta$. Here, by the lemma A.24, $\Delta; C' \vdash \sigma = \forall\Delta'\backslash\Delta''.|C''|[\Sigma''](\Gamma'')$. Thus, by the typing rule VALUELABEL, we have $\Delta; C \vdash v : \sigma$.

Lemma A.31 (Constraints weakening: tuple type)
If $\Delta; C' \models C$ and $\Delta; C \vdash t : \tau$, then $\Delta; C' \vdash t : \tau$.

Proof By straightforward induction on the derivation of $\Delta; C \vdash t : \tau$.

Lemma A.32 (Constraints weakening: instructions)
If $\Delta; C' \models C$ and $\Delta; \Gamma; C; \Sigma \vdash I$, then $\Delta; \Gamma; C'; \Sigma \vdash I$.

Proof By induction on the derivation of $\Delta; \Gamma; C; \Sigma \vdash I$. The proof is by case analysis on the last rule of the derivation.

- Case LOAD
  From the typing rule, we have $I \equiv \texttt{ld}\ [r_s + n], r_d; I'$, $\Delta; C \vdash \Sigma = \Sigma' \otimes \{\Gamma(r_s) \mapsto \langle \ldots, \sigma_n, \ldots \rangle\}$ and $\Delta; \Gamma\{r_d \mapsto \sigma_n\}; C; \Sigma \vdash I'$. Here, by the lemma A.27, $\Delta; C' \vdash \Sigma = \Sigma' \otimes \{\Gamma(r_s) \mapsto \langle \ldots, \sigma_n, \ldots \rangle\}$. In addition, by the induction hypothesis, $\Delta; \Gamma\{r_d \mapsto \sigma_n\}; C'; \Sigma \vdash I'$. Thus, by the typing rule LOAD, we have $\Delta; \Gamma; C'; \Sigma \vdash I$.

- Case STORE
  From the typing rule, we have $I \equiv \texttt{st}\ r_s, [r_d + n]; I'$, $\Delta; C \vdash \Sigma = \Sigma' \otimes \{\Gamma(r_d) \mapsto \langle \ldots, \sigma_n, \ldots \rangle\}$ and $\Delta; \Gamma; C; \Sigma' \otimes \{\Gamma(r_d) \mapsto \langle \ldots, \Gamma(r_s), \ldots \rangle\} \vdash I'$. Here, by the lemma A.27, $\Delta; C' \vdash \Sigma = \Sigma' \otimes \{\Gamma(r_d) \mapsto \langle \ldots, \sigma_n, \ldots \rangle\}$. In addition, by the induction hypothesis, $\Delta; \Gamma; C'; \Sigma' \otimes \{\Gamma(r_d) \mapsto \langle \ldots, \Gamma(r_s), \ldots \rangle\} \vdash I'$. Thus, by the typing rule STORE, we have $\Delta; \Gamma; C'; \Sigma \vdash I$.

- Case MOVE
  From the typing rule, we have $I \equiv \texttt{mov}\, r_s, r_d; I'$ and $\Delta; \Gamma\{r_d \mapsto \Gamma(r_s)\}; C; \Sigma \vdash I'$. Here, by the induction hypothesis, $\Delta; \Gamma\{r_d \mapsto \Gamma(r_s)\}; C'; \Sigma \vdash I'$. Thus, by the typing rule MOVE, we have $\Delta; \Gamma; C'; \Sigma \vdash I$.

- Case MOVEI
  From the typing rule, we have $I \equiv \texttt{movi}\ v, r_d; I'$, $\Delta; C \vdash v : \sigma$ and $\Delta; \Gamma\{r_d \mapsto \sigma\}; C; \Sigma \vdash I'$. Here, by the lemma A.30, $\Delta; C' \vdash v : \sigma$. In addition, by the induction hypothesis, $\Delta; \Gamma\{r_d \mapsto \sigma\}; C'; \Sigma \vdash I'$. Thus, by the typing rule MOVEI, we have $\Delta; \Gamma; C'; \Sigma \vdash I$.

- Case ARITH
  From the typing rule, we have $I \equiv (\texttt{add}, \texttt{sub}, \texttt{mul})\ r_{s_1}, r_{s_2}, r_d; I'$ and $\Delta; \Gamma\{r_d \mapsto \Gamma(r_{s_2})(+, -, *)\Gamma(r_{s_1})\}; C; \Sigma \vdash I'$. Here, by the induction hypothesis, $\Delta; \Gamma\{r_d \mapsto \Gamma(r_{s_2})(+, -, *)\Gamma(r_{s_1})\}; C'; \Sigma \vdash I'$. Thus, by the typing rule ARITH, we have $\Delta; \Gamma; C'; \Sigma \vdash I$.

- Case BRANCH
  From the typing rule, we have $I \equiv (\texttt{beq}, \texttt{ble})\ r_{s_1}, r_{s_2}, r_d; I'$, $\Delta; C \vdash \Gamma(r_d) = \forall.|C_1|[\Sigma'](\Gamma')$, $\Delta; C'' \models C_1$, $\Delta; C'' \vdash \Sigma = \Sigma'$ $\Delta; C'' \vdash \Gamma \leq \Gamma'$, and $\Delta; \Gamma; C \wedge \Gamma(r_{s_1})(\neq, >)\Gamma(r_{s_2}); \Sigma \vdash I'$, where $C'' \equiv C \wedge \Gamma(r_{s_1})(=, \leq)\Gamma(r_{s_2})$. Now, let $C_2 \equiv C' \wedge \Gamma(r_{s_1})(=, \leq)\Gamma(r_{s_2})$. Here, by the

lemma A.24, $\Delta; C' \vdash \Gamma(r_d) = \forall.|C_1|[\Sigma'](\Gamma')$. In addition, by the lemma A.22, we have $\Delta; C_2 \models C_1$. Next, by the lemma A.27, $\Delta; C_2 \vdash \Sigma = \Sigma'$. Then, by the lemma A.29, $\Delta; C_2 \vdash \Gamma \leq \Gamma'$. Last, by the induction hypothesis, $\Delta; \Gamma; C' \wedge \Gamma(r_{s_1})(\neq, >)\Gamma(r_{s_2}); \Sigma \vdash I'$. Thus, by the typing rule BRANCH, we have $\Delta; \Gamma; C'; \Sigma \vdash I$.

- Case JUMP
  From the typing rule, we have $I \equiv \texttt{jmp}\, r_d$, $\Delta; C \vdash \Gamma(r_d) = \forall.|C_1|[\Sigma'](\Gamma')$, $\Delta; C \models C_1$, $\Delta; C \vdash \Sigma = \Sigma'$ and $\Delta; C \vdash \Gamma \leq \Gamma'$. Here, by the lemma A.24, $\Delta; C' \vdash \Gamma(r_d) = \forall.|C_1|[\Sigma'](\Gamma')$. Next, by the lemma A.22, $\Delta; C' \models C_1$. In addition, by the lemma A.27, $\Delta; C' \vdash \Sigma = \Sigma'$. Last, by the lemma A.29, $\Delta; C' \vdash \Gamma \leq \Gamma'$. Thus, by the typing rule JUMP, we have $\Delta; \Gamma; C'; \Sigma \vdash I$.

- Case APPLY
  From the typing rule, we have $I \equiv \texttt{apply}\, r\theta; I'$, $\Delta; \Gamma\{r \mapsto \sigma_f\}; C; \Sigma \vdash I$, where $\sigma_f \equiv \forall\Delta'\backslash\Delta''.|C''|[\Sigma''](\Gamma'')$, $C'' \equiv C_1\theta$, $\Sigma'' \equiv \Sigma'\theta$, $\Gamma'' \equiv \Gamma'\theta$, $\Gamma(r) \equiv \forall\Delta'.|C_1|[\Sigma'](\Gamma')$ and $\theta \equiv [c_1, \ldots, c_n/\Delta'']$. Here, by the induction hypothesis, $\Delta; \Gamma\{r \mapsto \sigma_f\}; C'; \Sigma \vdash I'$. Thus, by the typing rule APPLY, $\Delta; \Gamma; C'; \Sigma \vdash I$.

- Case ROLL
  From the typing rule, we have $I \equiv \texttt{roll}_\tau\ (c_1, \ldots, c_n)$, $\Delta; C \vdash \Sigma = \Sigma' \otimes \{i \mapsto \tau'[\mu\eta[\Delta'].\tau'/\eta][c_1, \ldots, c_n/\Delta']\}$ and $\Delta; \Gamma; C; \Sigma' \otimes \{i \mapsto \tau\} \vdash I'$, where $\tau \equiv \mu\eta[\Delta'].\tau'\,(c_1, \ldots, c_n)$. Here, by the lemma A.27, $\Delta; C' \vdash \Sigma = \Sigma' \otimes \{i \mapsto \tau'[\mu\eta[\Delta'].\tau'/\eta][c_1, \ldots, c_n/\Delta']\}$. In addition, by the induction hypothesis, $\Delta; \Gamma; C'; \Sigma' \otimes \{i \mapsto \tau\} \vdash I'$. Thus, by the typing rule ROLL, $\Delta; \Gamma; C'; \Sigma \vdash I$.

- Case UNROLL
  From the typing rule, we have $I \equiv \texttt{unroll}\ i; I'$, $\Delta; C \vdash \Sigma = \Sigma' \otimes \{i \mapsto \mu\eta[\Delta'].\tau'(c_1, \ldots, c_n)\}$ and $\Delta; \Gamma; C; \Sigma' \otimes \{i \mapsto \tau'[\mu\eta[\Delta'].\tau'/\eta][c_1, \ldots, c_n/\Delta']\} \vdash I'$. Here, by the lemma A.27, $\Delta; C \vdash \Sigma = \Sigma' \otimes \{i \mapsto \mu\eta[\Delta'].\tau'(c_1, \ldots, c_n)\}$.
  In addition, by the induction hypothesis, $\Delta; \Gamma; C'; \Sigma' \otimes \{i \mapsto \tau'[\mu\eta[\Delta'].\tau'/\eta][c_1, \ldots, c_n/\Delta'$
  I'. Thus, by the typing rule UNROLL, $\Delta; \Gamma; C'; \Sigma \vdash I$.

- Case PACK
  From the typing rule, we have $I \equiv \texttt{pack}_{[c_1,\ldots,c_n|\Sigma'\theta]\, as\ \tau}\, i; I'$, $\Delta; C \vdash \Sigma = \Sigma'' \otimes \{i \mapsto \tau'\theta\} \otimes \Sigma'\theta$, $\Delta; C \models C_1\theta$ and $\Delta; \Gamma; C; \Sigma'' \otimes \{i \mapsto \tau\} \vdash I'$, where $\tau \equiv \exists\Delta'.|C_1|[\Sigma']\tau'$ and $\theta \equiv [c_1, \ldots, c_n/\Delta']$. Here, by the lemma A.27, $\Delta; C' \vdash \Sigma = \Sigma'' \otimes \{i \mapsto \tau'\theta\} \otimes \Sigma'\theta$. Next, by the lemma A.22, $\Delta; C' \models$

$C_1\theta$. In addition, by the induction hypothesis, $\Delta; \Gamma; C'; \Sigma'' \otimes \{i \mapsto \tau\} \vdash I'$. Thus, by the typing rule PACK, $\Delta; \Gamma; C'; \Sigma \vdash I$.

- Case UNPACK
  From the typing rule, we have $I \equiv \texttt{unpack}\ i\ \texttt{with}\ \Delta''; I'$, $\Delta; C \vdash \Sigma = \Sigma'' \otimes \{i \mapsto \exists \Delta'.|C_1|[\Sigma']\tau\}$ and $\Delta\Delta''; \Gamma; C \wedge C_1\theta; \Sigma' \otimes \{i \mapsto \tau\theta\} \otimes \Sigma'\theta \vdash I$, where $\theta \equiv [\Delta''/\Delta']$. Here, by the lemma A.27, $\Delta; C' \vdash \Sigma = \Sigma'' \otimes \{i \mapsto \exists \Delta'.|C_1|[\Sigma']\tau\}$. Now, by the lemma A.22, $\Delta\Delta''; C' \wedge C_1\theta \models C \wedge C_1\theta$. Therefore, by the induction hypothesis, $\Delta\Delta''; \Gamma; C' \wedge C_1\theta; \Sigma'' \otimes \{i \mapsto \tau\theta\} \otimes \Sigma'\theta \vdash I'$. Thus, by the typing rule UNPACK, we have $\Delta; \Gamma; C'; \Sigma \vdash I$.

- Case SPLIT
  From the typing rule, we have $I \equiv \texttt{split}\ i_1, i_2; I'$, $\Delta; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \tau[j_1]\}$, $\Delta; C \models 0 \leq i_2 \leq j_1$ and $\Delta; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \tau[i_2]\} \otimes \{k_1 \mapsto \tau[k_2]\} \vdash I'$, where $k_1 \equiv i_1 + \mathit{sizeof}(\tau) * i_2$ and $k_2 \equiv j_1 - i_2$. Here, by the lemma A.27, $\Delta; C' \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \tau[j_1]\}$. Next, by the lemma A.22, $\Delta; C' \models 0 \leq i_2 \leq j_1$. In addition, by the induction hypothesis, $\Delta; \Gamma; C'; \Sigma' \otimes \{i_1 \mapsto \tau[i_2]\} \otimes \{k_1 \mapsto \tau[k_2]\} \vdash I'$. Thus, by the typing rule SPLIT, we have $\Delta; \Gamma; C'; \Sigma \vdash I$.

- Case CONCAT
  From the typing rule, we have $I \equiv \texttt{concat}\ i_1, j_1, j_2; I'$, $\Delta; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \tau[i_2]\} \otimes \{j_1 \mapsto \tau[j_2]\}$, $\Delta; C \models j_1 = i_1 + \mathit{sizeof}(\tau) * i_2$ and $\Delta; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \tau[i_2 + j_2]\} \vdash I'$. Here, by the lemma A.27, $\Delta; C' \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \tau[i_2]\} \otimes \{j_1 \mapsto \tau[j_2]\}$. Then, by the lemma A.22, $\Delta; C' \models j_1 = i_1 + \mathit{sizeof}(\tau) * i_2$. In addition, by the induction hypothesis, $\Delta; \Gamma; C'; \Sigma' \otimes \{i_1 \mapsto \tau[i_2 + j_2]\} \vdash I'$. Thus, by the typing rule CONCAT, we have $\Delta; \Gamma; C'; \Sigma \vdash I$.

- Case TUPLESPLIT
  From the typing rule, we have $I \equiv \texttt{tuple\_split}\ i_1, n_2; I'$, $\Delta; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n \rangle\}$, $\Delta; C \models 0 < n_2 < n$ and $\Delta; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_{n_2} \rangle\} \otimes \{i_1 + n_2 \mapsto \langle \sigma_{n_2+1}, \ldots, \sigma_n \rangle\} \vdash I'$. Here, by the lemma A.27, $\Delta; C' \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n \rangle\}$. Then, by the lemma A.22, $\Delta; C' \models 0 < n_2 < n$. In addition, by the induction hypothesis, $\Delta; \Gamma; C'; \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_{n_2} \rangle\} \otimes \{i_1 + n_2 \mapsto \langle \sigma_{n_2+1}, \ldots, \sigma_n \rangle\} \vdash I'$. Thus, by the typing rule TUPLESPLIT, we have $\Delta; \Gamma; C'; \Sigma \vdash I$.

- Case TUPLECONCAT
  From the typing rule, we have $I \equiv \texttt{tuple\_concat}\ i_1, i_2; i'$, $\Delta; C \vdash$

$\Sigma = \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n \rangle\} \otimes \{i_2 \mapsto \langle \sigma'_1, \ldots, \sigma'_m \rangle\}$, $\Delta; C \models i_2 = i_1 + n$ and $\Delta; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_m \rangle\} \vdash I'$. Here, by the lemma A.27, $\Delta; C' \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n \rangle\} \otimes \{i_2 \mapsto \langle \sigma'_1, \ldots, \sigma'_m \rangle\}$. Next, by the lemma A.22, $\Delta; C' \models i_2 = i_1 + n$. In addition, by the induction hypothesis, $\Delta; \Gamma; C'; \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_m \rangle\} \vdash I'$. Thus, by the typing rule TUPLECONCAT, we have $\Delta; \Gamma; C'; \Sigma \vdash I$.

Lemma A.33
If $\Delta; C \vdash a : \tau[i]$, then $\Delta; C \models i > 0$.

Proof From the typing rule ARRAY, we have $a = \langle t_1, \ldots, t_n \rangle$ and $\Delta; C \models n = i$. Thus, $\Delta; C \models i > 0$, because $n > 0$

Lemma A.34
If $\vdash M : \Sigma$, then $\forall m \in Dom(\Sigma).n_m \geq 0$ where $\Sigma(m) \equiv \tau_m[n_m]$.

Proof From the typing rule MEMORY, $M \equiv \{n'_1 \mapsto a_1\} \ldots \{n'_k \mapsto a_k\}$, $\Sigma \equiv \{n'_1 \mapsto \tau_1[n_1]\} \otimes \ldots \otimes \{n'_k \mapsto \tau_k[n_k]\} \otimes \Sigma'$, $\forall i. \cdot; \cdot \vdash a_i : \tau_i[n_i]$ and $\forall m \in Dom(\Sigma').\cdot; \cdot \vdash \Sigma'(m) = \tau_m[0]$. Therefore, $\forall i \in \{n'_1, \ldots, n'_k\}.n_i > 0$ by the lemma A.33. In addition, $\forall m \in Dom.n_m = 0$ from the typing rule EQARRAY. Thus, $\forall m \in Dom(\Sigma).n_m \geq 0$.

Lemma A.35 (Null Array Addition)
If $\vdash M : \Sigma$, then $\vdash M : \Sigma \otimes \{n \mapsto \tau[0]\}$.

Proof From the typing rule MEMORY, we have $GU(M)$, $\forall i. \cdot; \cdot \vdash a_i : at_i$ and $\forall m \in Dom(\Sigma').\cdot; \cdot \vdash \Sigma'(m) = \tau_m[0]$, where $M \equiv \{n_1 \mapsto a_1\} \ldots \{n_k \mapsto a_k\}$ and $\Sigma \equiv \{n_1 \mapsto at_1\} \otimes \ldots \otimes \{n_k \mapsto at_k\} \otimes \Sigma'$. Here $\Sigma \otimes \{n \mapsto \tau[0]\} \equiv \Sigma \equiv \{n_1 \mapsto at_1\} \otimes \ldots \otimes \{n_k \mapsto at_k\} \otimes \Sigma' \otimes \{n \mapsto \tau[0]\}$. In addition, $\forall m \in Dom(\Sigma' \otimes \{n \mapsto \tau[0]\}).\cdot; \cdot \vdash \Sigma'(m) = \tau_m[0]$. Thus, by the typing rule MEMORY, $\vdash M : \Sigma \otimes \{n \mapsto \tau[0]\}$.

Lemma A.36 (Null Array Deletion)
If $\vdash M : \Sigma \otimes \{n \mapsto \tau[0]\}$, then $\vdash M : \Sigma$.

Proof From the typing rule MEMORY, we have $GU(M)$, $\forall i. \cdot; \cdot \vdash a_i : at_i$ and $\forall m \in Dom(\Sigma').\cdot; \cdot \vdash \Sigma'(m) = \tau_m[0]$, where $M \equiv \{n_1 \mapsto a_1\} \ldots \{n_k \mapsto a_k\}$ and $\Sigma \otimes \{n \mapsto \tau[0]\} \equiv \{n_1 \mapsto at_1\} \otimes \ldots \otimes \{n_k \mapsto at_k\} \otimes \Sigma'$. Here $\Sigma' \equiv \Sigma'' \otimes \{n \mapsto \tau[0]\}$ because $\forall i.m_i > 0$ where $at_i \equiv \tau_i[m_i]$, by the lemma A.33. Therefore, $\Sigma \equiv \{n_1 \mapsto at_1\} \otimes \ldots \otimes \{n_k \mapsto at_k\} \otimes \Sigma''$. Thus, by the typing rule MEMORY, $\vdash M : \Sigma$ because $\forall m \in Dom(\Sigma'').\cdot; \cdot \vdash \Sigma''(m) = \tau_m[0]$.

A.4 Transitivity lemmas

Lemma A.37 (Value type transitivity)
If $\Delta; C \vdash v : \sigma$ and $\Delta; C \vdash \sigma = \sigma'$, then $\Delta; C \vdash v : \sigma'$.

Proof By induction on the derivation of $\Delta; C \vdash v : \sigma$. The proof is by case analysis on the last rule of the derivation.

- Case VALUEINTEGER
  From the typing rule, we have $\Delta; C \models v = \sigma$. Therefore, $\Delta; C \models v = \sigma'$. Thus, by the typing rule VALUEINTEGER, $\Delta; C \vdash v : \sigma'$.

- Case VALUELABEL
  From the typing rule, we have $\Delta; C \vdash \sigma = \forall \Delta' \backslash \Delta''.|C''|[\Sigma'']\left(\Gamma''\right)$, for some Σ'' and Γ''. Now, we have $\Delta; C \vdash \sigma' = \forall \Delta' \backslash \Delta''.|C''|[\Sigma'']\left(\Gamma''\right)$. Thus, from the typing rule VALUELABEL, $\Delta; C \vdash v : \sigma'$.

Lemma A.38 (Tuple type transitivity)
If $\Delta; C \vdash t : \tau$ and $\Delta; C \vdash \tau = \tau'$, then $\Delta; C \vdash t : \tau'$.

Proof By induction on the derivation of $\Delta; C \vdash t : \tau$. The proof is by case analysis on the last rule of the derivation.

- Case TUPLE
  From the typing rule, we have $t \equiv \langle v_1, \ldots, v_n \rangle$, $\tau \equiv \langle \sigma_1, \ldots, \sigma_n \rangle$ and $\forall j. \Delta; C \vdash v_j : \sigma_j$. Because of the form of τ, the last rule of the derivation $\Delta; C \vdash \tau = \tau'$ is EQTUPLE. Therefore, $\forall j. \Delta; C \vdash \sigma_j = \sigma'_j$, where $\tau' \equiv \langle \sigma'_1, \ldots, \sigma'_n \rangle$. Here, by the lemma A.37, $\forall j. \Delta; C \vdash v_j : \sigma'_j$. Thus, $\Delta; C \vdash t : \tau'$.

- Case TUPLEROLL
  From the typing rule, we have $t \equiv \texttt{roll}(t')$ and $\Delta; C \vdash t' \left[\rho/\eta\right] \left[c_1, \ldots, c_n/\Delta_1\right]$, where $\tau \equiv \rho\left(c_1, \ldots, c_n\right)$ and $\rho \equiv \mu\eta \left[\Delta_1\right] .\tau_1$. Because of the form of τ, the last rule of the derivation $\Delta; C \vdash \tau = \tau'$ is EQREC. Therefore, $\tau' \equiv \rho\left(c_1, \ldots, c_n\right)$. That is, $\tau \equiv \tau'$. Thus, by the typing rule TUPLEROLL, $\Delta; C \vdash t : \tau' (= \tau)$.

- Case TUPLEPACK
  From the typing rule, we have $t \equiv \texttt{pack}_{[c_1,\ldots,c_n|M]}(t')$, $\Delta; C \vdash t' : \tau_1 \left[c_1, \ldots, c_n/\Delta_1\right], \vdash M : \Sigma_1 \left[c_1, \ldots, c_n/\Delta_1\right]$ and $\Delta; C \models C_1 \left[c_1, \ldots, c_n/\Delta_1\right]$, where $\tau \equiv \exists \Delta_1.|C_1|[\Sigma_1]\tau_1$. Because of the form of τ, the last rule of the derivation $\Delta; C \vdash \tau = \tau'$ is EQEX. Therefore, $\Delta\Delta_1; C \vdash C_1 = C_2$,

$\Delta\Delta_1; C \wedge C_1 \vdash \Sigma_1 = \Sigma_2$ and $\Delta\Delta_1; C \wedge C_1 \vdash \tau_1 = \tau_2$, where $\tau' \equiv \exists\Delta_1.|C_2|[\Sigma_2]\tau_2$. Here, by the lemma A.15, $\Delta; C \wedge C_1\,[c_1, \ldots, c_n/\Delta_1] \vdash \tau_1\,[c_1, \ldots, c_n/\Delta_1] = \tau_2\,[c_1, \ldots, c_n/\Delta_1]$, because $C\,[c_1, \ldots, c_n/\Delta_1] \equiv C$. Now, by the induction hypothesis, $\Delta; C \wedge C_1\,[c_1, \ldots, c_n/\Delta_1] \vdash t' : \tau_2\,[c_1, \ldots, c_n/\Delta_1]$. Here, by the lemma A.31, $\Delta; C \vdash t' : \tau_2\,[c_1, \ldots, c_n/\Delta_1]$.

Next, by the lemma A.17, $\Delta; C \wedge C_1\,[c_1, \ldots, c_n/\Delta_1] \vdash \Sigma_1\,[c_1, \ldots, c_n/\Delta_1] = \Sigma_2\,[c_1, \ldots, c_n/\Delta_1]$. Here, from the $\vdash M : \Sigma_1\,[c_1, \ldots, c_n/\Delta_1]$, $\Sigma_1\,[c_1, \ldots, c_n/\Delta_1]$ does not contain any type variables. Therefore, $\cdot;\cdot \vdash \Sigma_1\,[c_1, \ldots, c_n/\Delta_1] = \Sigma_2\,[c_1, \ldots, c_n/\Delta_1]$. Thus, $\vdash M : \Sigma_2\,[c_1, \ldots, c_n/\Delta_1]$.

Then, by the lemma A.13, $\Delta; C \vdash \Sigma_1\,[c_1, \ldots, c_n/\Delta_1] = \Sigma_2\,[c_1, \ldots, c_n/\Delta_1]$. Here, by the typing rule EQCSTRT, $\Delta; C \wedge C_1\,[c_1, \ldots, c_n/\Delta_1] \models C_2\,[c_1, \ldots, c_n/\Delta_1]$. Now, by the lemma A.22, $\Delta; C \models C_2\,[c_1, \ldots, c_n/\Delta_1]$, because $\Delta; C \models C_1\,[c_1, \ldots, c_n/\Delta_1]$ and $\Delta; C \models C$.

Thus, $\Delta; C \vdash t : \tau'$.

Lemma A.39 (Array type transitivity)
If $\Delta; C \vdash a : at$ and $\Delta; C \vdash at = at'$, then $\Delta; C \vdash a : at'$.

Proof From the typing rule ARRAY, we have $a \equiv \langle t_1, \ldots, t_n\rangle$, $\forall j.\Delta; C \vdash t_j : \tau$ and $\Delta; C \models n = i$, where $at \equiv \tau[i]$. Now, from the typing rule EQARRAY, $\Delta; C \vdash \tau = \tau'$ and $\Delta; C \models i = i'$, where $at' \equiv \tau'[i']$. Here, by the induction hypothesis and the lemma A.38, $\forall j.\Delta; C \vdash t_j : \tau'$. In addition, $\Delta; C \models n = i'$. Thus, by the typing rule ARRAY, $\Delta; C \vdash a : at'$.

Lemma A.40 (Memory type transitivity)
If $\vdash M : \Sigma$ and $\cdot;\cdot \vdash \Sigma = \Sigma'$, then $\vdash M : \Sigma'$.

Proof By induction on the derivation of $\cdot;\cdot \vdash \Sigma = \Sigma'$. The proof is by case analysis on the last rule of the derivation.

- Case EQMEMEMPTY
  From the typing rule, $\Sigma \equiv \Sigma' \equiv \cdot$. Therefore, by the typing rule MEMORY, $M \equiv \cdot$ and $GU(M)$. Thus, $\vdash M : \Sigma'$ by the typing rule MEMORY.

- Case EQMEMLOC
  From the typing rule, $\Sigma \equiv \Sigma_1 \otimes \{n_1 \mapsto at_1\}$, $\Sigma' \equiv \Sigma_2 \otimes \{n_2 \mapsto at_2\}$, $\cdot;\cdot \vdash \Sigma_1 = \Sigma_2$, $\cdot;\cdot \models n_1 = n_2$ and $\cdot;\cdot \vdash at_1 = at_2$.

  Here let $at_1 \equiv \tau_1[m_1]$ and $at_2 \equiv \tau_2[m_2]$. Then, by the typing rule EQARRAY, $\cdot;\cdot \vdash m_1 = m_2$. Here, by the lemma A.34, $m_1 (= m_2) \geq 0$

If $m_1(= m_2) > 0$, then, by the lemma A.43, $M = M' \otimes \{n_1 \mapsto a\}$ where $\cdot;\cdot \vdash a : at_1$ and $\vdash M' : \Sigma_1$. Now, by the induction hypothesis, $\vdash M' : \Sigma_2$. In addition, by the lemma A.39, $\cdot;\cdot \vdash a : at_2$. Therefore, by the lemma A.11, $\vdash M : \Sigma'$, because $\Sigma' \equiv \Sigma_2 \otimes \{n_2 \mapsto at_2\} = \Sigma'$ and $M = M' \otimes \{n_2 \mapsto a\}$.

If $m_1(= m_2) = 0$, then, by the lemma A.36, $\vdash M : \Sigma_1$. Here, by the induction hypothesis, $\vdash M : \Sigma_2$. Now, by the lemma A.35, $\vdash M : \Sigma'$.

- Case EQMEMVAR
  This rule never be used because $\Delta \equiv \cdot$.

- Case EQMEMZEROARRAYL
  By the lemma B.9, we have $\cdot;\cdot \vdash \Sigma' = \Sigma$. Then, the rest is the same as the case EQMEMZEROARRAYR.

- Case EQMEMZEROARRAYR
  From the typing rule, $\Sigma' \equiv \Sigma_2 \otimes \{n_1 \mapsto \tau[n_2]\}$, $\cdot;\cdot \vdash \Sigma = \Sigma_2$ and $\cdot;\cdot \models n_2 = 0$. Now, by the induction hypothesis, $\vdash M : \Sigma_2$. Then, by the lemma A.35, $\vdash M : \Sigma'$.

Lemma A.41 (Registers type equality transitivity)
If $\vdash R : \Gamma$ and $\cdot;\cdot \vdash \Gamma = \Gamma'$, then $\vdash R : \Gamma'$.

Proof By straightforward induction on the derivation of $\cdot;\cdot \vdash \Gamma = \Gamma'$.

Lemma A.42 (Registers type subtyping)
If $\vdash R : \Gamma$ and $\cdot;\cdot \vdash \Gamma \leq \Gamma'$, then $\vdash R : \Gamma'$.

Proof By straightforward induction on the derivation of $\cdot;\cdot \vdash \Gamma \leq \Gamma'$.

A.5 Canonical forms lemmas

Lemma A.43 (Canonical form: memory)
If $\vdash M : \Sigma \otimes \{n \mapsto \tau[n']\}$ and $n' > 0$, then $M = M'\{n \mapsto a\}$, $\vdash M' : \Sigma$ and $\cdot;\cdot \vdash a : \tau[n']$.

Proof By induction on the size of the memory M.

- Case $|M| = 0$
  By the typing rule MEMORY, we have $\cdot;\cdot \vdash \cdot = \Sigma \otimes \{n \mapsto \tau[n']\}$. However, this contradicts the lemma A.48. Thus, this case never occurs.

- Case $|M| = 1$
  Let $M \equiv \{m \mapsto a\}$. Then, by the typing rule MEMORY, we have $\cdot;\cdot \vdash \{m \mapsto at\} = \Sigma \otimes \{n \mapsto \tau[n']\}$ and $\cdot;\cdot \vdash a : at$.

  Here if $m \neq n$, it contradicts the lemma A.48. Thus, $m = n$. In addition, by the lemma A.48, we have $\cdot;\cdot \vdash at = \tau[n']$ and $\cdot;\cdot \vdash \cdot = \Sigma$. Now, by the lemma A.39, $\cdot;\cdot \vdash a : \tau[n']$. Here, by the typing rule MEMORY, $\vdash \cdot : \Sigma$.

- Case $|M| = k$
  Let $M \equiv M'\{m \mapsto a\}$. Then, by the typing rule MEMORY, we have $\cdot;\cdot \vdash \Sigma'' \otimes \{m \mapsto at\} = \Sigma \otimes \{n \mapsto \tau[n']\}, \vdash M' : \Sigma''$ and $\cdot;\cdot \vdash a : at$.

  Here if $m \neq n$, then $\Sigma'' \equiv \Sigma_1 \otimes \{n \mapsto \tau'[n'']\}$, $\cdot;\cdot \vdash \Sigma_1 \otimes \{m \mapsto at\} = \Sigma$ and $\Delta; C \vdash \tau = \tau'$ and $\Delta; C \models n' = n''$, by the lemma A.48. Now, by the induction hypothesis, $M' \equiv M''\{n \mapsto a'\}, \vdash M'' : \Sigma_1, \cdot;\cdot \vdash a' : \tau'[n'']$ Here let $M_1 \equiv M''\{m \mapsto a\}$. Then, $M \equiv M_1\{n \mapsto a'\}$. Now, by the lemma A.11, $\vdash M_1 : \Sigma_1 \otimes \{m \mapsto at\}$, because $\vdash \{m \mapsto a\} : \{m \mapsto at\}$. Thus, $\vdash M_1 : \Sigma$, because of the lemma A.40. In addition, by the lemma A.39, $\cdot;\cdot \vdash a' : \tau[n']$.

Lemma A.44 (Canonical form: array)
If $\cdot;\cdot \vdash \langle t_1, \ldots, t_n \rangle : at$, then $\forall i.\cdot;\cdot \vdash t_i : \tau$ and $\cdot;\cdot \models n = m$, where $at \equiv \tau[m]$.

Proof From the typing rule ARRAY, we have $at \equiv \tau[m]$, $\forall i.\cdot;\cdot \vdash t_i : \tau$ and $\cdot;\cdot \models n = m$.

Lemma A.45 (Canonical form: tuple)
If $\cdot;\cdot \vdash t : \tau$, then

- if $\tau \equiv \langle \sigma_1, \ldots, \sigma_n \rangle$, then $t \equiv \langle v_1, \ldots, v_n \rangle$ and $\forall i.\cdot;\cdot \vdash t_i : \sigma_i$.
- if $\tau \equiv \mu\eta[\Delta].\tau'(c_1, \ldots, c_n)$, then $t \equiv \texttt{roll}(t')$ and $\cdot;\cdot \vdash t' : \tau'[\mu\eta[\Delta].\tau'/\eta][c_1, \ldots, c_n/\Delta]$.
- if $\tau \equiv \exists\Delta.|C|[\Sigma]\tau'$, then $t \equiv \texttt{pack}_{[c_1,\ldots,c_n|M]}(t')$, $\cdot;\cdot \vdash t' : \tau'[c_1, \ldots, c_n/\Delta]$ and $\vdash M : \Sigma[c_1, \ldots, c_n/\Delta]$.

Proof If $\tau \equiv \langle \sigma_1, \ldots, \sigma_n \rangle$, then the last rule of the derivation $\cdot;\cdot \vdash t : \tau$ is TUPLE. Therefore, from the typing rule TUPLE, we have $t \equiv \langle v_1, \ldots, v_n \rangle$ and $\forall i.\cdot;\cdot \vdash v_i : \sigma_i$.

If $\tau \equiv \mu\eta[\Delta].\tau'(c_1, \ldots, c_n)$, then the last rule of the derivation $\cdot;\cdot \vdash t : \tau$ is TUPLEROLL. Therefore, from the typing rule TUPLEROLL, we have $t \equiv \texttt{roll}(t')$ and $\cdot;\cdot \vdash t' : \tau'[\mu\eta[\Delta].\tau'/\eta][c_1, \ldots, c_n/\Delta]$.

If $\tau \equiv \exists\Delta.|C|[\Sigma]\tau'$, then the last rule of the derivation of $\cdot;\cdot \vdash t : \tau$ is TUPLEPACK. Therefore, from the typing rule TUPLEPACK, we have $t \equiv \texttt{pack}_{[c_1,\ldots,c_n|M]}(t')$, $\cdot;\cdot \vdash t' : \tau'[c_1,\ldots,c_n/\Delta]$ and $\vdash M : \Sigma[c_1,\ldots,c_n/\Delta]$.

Lemma A.46 (Canonical form: value)
If $\Delta; C \vdash v : \sigma$, then

- if $\sigma = i$, then $\Delta; C \models v = i$.
- if $\sigma = \forall\Delta'.|C'|[\Sigma'](\Gamma')$, then $v = l\theta$ and $\Delta; C \vdash \sigma = \forall\Delta''\backslash\Delta_1|C''\theta|[\Sigma''\theta](\Gamma''\theta)$ where $\theta \equiv [c_1,\ldots,c_n/\Delta_1]$ and $\Phi(l) \equiv \forall\Delta_2.|C''|[\Sigma''](\Gamma'')$

Proof If $\sigma = i$, then the last derivation of $\Delta; C \vdash v : \sigma$ is the typing rule VALUEINTEGER. Thus, by the typing rule, we have $\Delta; C \models v = i$.

If $\sigma = \forall\Delta'.|C'|[\Sigma'](\Gamma')$, then the last derivation of $\Delta; C \vdash v : \sigma$ is the typing rule VALUELABEL. Therefore, by the typing rule, we have $\Delta; C \vdash \sigma = \forall\Delta_1\backslash\Delta_2.|C_2|[\Sigma_2](\Gamma_2)$ and $v \equiv l\theta$, where $\theta \equiv [c_1,\ldots,c_n/\Delta_2]$, $Phi(l) \equiv \forall\Delta_1.|C_1|[\Sigma_1](\Gamma_1)$, $C_2 \equiv C_1\theta$, $\Sigma_2 \equiv \Sigma_1\theta$ and $\Gamma_2 \equiv \Gamma_1\theta$.

Lemma A.47 (Canonical form: memory type zero arrays)
If $\Delta; C \vdash \Sigma = \Sigma' \otimes \{i' \mapsto at'\}$ and $\Delta; C \models j' = 0$ (where $at' \equiv \tau'[j']$), then $\Delta; C \vdash \Sigma = \Sigma'$,

Proof By straightforward induction on the derivation of $\Delta; C \vdash \Sigma = \Sigma' \otimes \{i' \mapsto at'\}$.

Lemma A.48 (Canonical form: memory type location)
If $\Delta; C \vdash \Sigma = \Sigma' \otimes \{i' \mapsto at'\}$ and $\Delta; C \models j' > 0$ (where $at' \equiv \tau'[j']$), then $\Sigma \equiv \Sigma'' \otimes \{i \mapsto at\}$, $\Delta; C \vdash \Sigma'' = \Sigma'$, $\Delta; C \models i = i'$ and $\Delta; C \vdash at = at'$.

Proof By induction on the derivation of $\Delta; C \vdash \Sigma = \Sigma' \otimes \{i' \mapsto \tau'[j']\}$. The proof is by case analysis on the last rule of the derivation.

- Case EQMEMEMPTY
  This case never occurs because $\Sigma' \otimes \{i' \mapsto at'\} \not\equiv \cdot$.
- Case EQMEMLOC
  By the typing rule, we have $\Sigma \equiv \Sigma'' \otimes \{i \mapsto \tau[j]\}$, $\Delta; C \vdash \Sigma'' = \Sigma'$, $\Delta; C \models i = i'$ and $\Delta; C \vdash at = at'$.
- Case EQMEMVAR
  By the typing rule, we have $\Sigma' \equiv \Sigma_1 \otimes \epsilon$, $\Sigma \equiv \Sigma_2 \otimes \epsilon$ and $\Delta; C \vdash \Sigma_2 = \Sigma_1 \otimes \{i' \mapsto at'\}$. Now, by the induction hypothesis, $\Sigma_2 \equiv \Sigma_2' \otimes \{i \mapsto$

$at\}$, $\Delta; C \vdash \Sigma_2' = \Sigma_1$, $\Delta; C \models i = i'$ and $\Delta; C \vdash at = at'$. Here let $\Sigma'' \equiv \Sigma_2' \otimes \epsilon$. Then, $\Sigma \equiv \Sigma'' \otimes \{i \mapsto at\}$. In addition, by the typing rule EQMEMVAR, we have $\Delta; C \vdash \Sigma'' = \Sigma'$.

- Case EQMEMZEROARRAYL
  By the typing rule, we have $\Sigma' \equiv \Sigma_1 \otimes \{k \mapsto \tau''[m]\}$, $\Delta; C \models m = 0$ and $\Delta; C \vdash \Sigma = \Sigma_1 \otimes \{i' \mapsto at'\}$. Now, by the induction hypothesis, $\Sigma \equiv \Sigma'' \otimes \{i \mapsto at\}$, $\Delta; C \vdash \Sigma'' = \Sigma_1$, $\Delta; C \models i = i'$ and $\Delta; C \vdash at = at'$. Here, by the typing rule EQMEMVAR, we have $\Delta; C \vdash \Sigma'' = \Sigma' (\equiv \Sigma_1 \otimes \{k \mapsto \tau''[m]\})$.

- Case EQMEMZEROARRAYR
  Same as Case EQMEMZEROARRAYL.

Lemma A.49 (Canonical form: memory type variable)
If $\Delta; C \vdash \Sigma = \Sigma' \otimes \epsilon$, then $\Sigma \equiv \Sigma'' \otimes \epsilon$ and $\Delta; C \vdash \Sigma'' = \Sigma'$.

Proof By induction on the derivation of $\Delta; C \vdash \Sigma = \Sigma' \otimes \epsilon$. The proof is by case analysis of the last rule of the derivation.

- Case EQMEMEMPTY
  This case never occurs because $\Sigma' \otimes \epsilon \not\equiv \cdot$.

- Case EQMEMLOC
  By the typing rule, we have $\Sigma \equiv \Sigma_2 \otimes \{i \mapsto at\}$, $\Sigma' \equiv \Sigma_1 \otimes \{i' \mapsto at'\}$, $\Delta; C \vdash \Sigma_2 = \Sigma_1 \otimes \epsilon$, $\Delta; C \models i = i'$ and $\Delta; C \vdash at = at'$. Here, by the induction hypothesis, we have $\Sigma_2 \equiv \Sigma_2' \otimes \epsilon$ and $\Delta; C \vdash \Sigma_2' = \Sigma_1$. Now, let $\Sigma'' \equiv \Sigma_2' \otimes \{i \mapsto at\}$. Then, $\Sigma \equiv \Sigma'' \otimes \epsilon$. In addition, by the typing rule EQMEMLOC, $\Delta; C \vdash \Sigma'' = \Sigma'$.

- Case EQMEMVAR
  By the typing rule, we have $\Sigma \equiv \Sigma'' \otimes \epsilon$ and $\Delta; C \vdash \Sigma'' = \Sigma'$.

- Case EQMEMZEROARRAYL
  By the typing rule, we have $\Sigma' \equiv \Sigma_1 \otimes \{i' \mapsto \tau'[j']\}$, $\Delta; C \models j' = 0$ and $\Delta; C \vdash \Sigma = \Sigma_1 \otimes \epsilon$. Here, by the induction hypothesis, we have $\Sigma = \Sigma'' \otimes \epsilon$, $\Delta; C \vdash \Sigma'' = \Sigma_1$. Now, by the typing rule EQMEMZEROARRAYR, we have $\Delta; C \vdash \Sigma'' = \Sigma' (\equiv \Sigma_1 \otimes \{i' \mapsto \tau'[j']\})$.

- Case EQMEMZEROARRAYR
  Same as Case EQMEMZEROARRAYR.

A.6 Preservation lemma

Lemma A.50 (Preservation)
If $\vdash S$ and there exists S' such that $S \mapsto S'$, then $\vdash S'$.

Proof By cases on the operational rules.

- Case `ld`
  Let $S \equiv (P, M, R, \texttt{ld}\ [r_s + n], r_d; I)$. Then, $S' \equiv (P, M, R\{r_d \mapsto v_n\}, I)$, where $M \equiv M'\{n' \mapsto \langle\langle \ldots, v_n, \ldots\rangle\rangle\}$ and $n' \equiv R(r_s)$. From the assumption $\vdash S$, we have $P \vdash \Phi$, $\vdash M : \Sigma$, $\vdash R : \Gamma$ and $\cdot;\Gamma;\cdot;\Sigma \vdash \texttt{ld}\ [r_s + n], r_d; I$.

  Now, from the typing rule LOAD, $\cdot;\cdot \vdash \Sigma = \Sigma' \otimes \{n' \mapsto \langle \ldots, \sigma_n, \ldots\rangle\}$ and $\cdot;\Gamma\{r_d \mapsto \sigma_n\};\cdot;\Sigma \vdash I$, because $n' \equiv R(r_s) \equiv \Gamma(r_s)$.

  Here, by the lemma A.40, $\vdash M'\{n' \mapsto \langle\langle \ldots, v_n, \ldots\rangle\rangle\} : \Sigma' \otimes \{n' \mapsto \langle \ldots, \sigma_n, \ldots\rangle\}$. Thus, by the lemma A.43,$\cdot;\cdot \vdash \langle\langle \ldots, v_n, \ldots\rangle\rangle : \langle \ldots, \sigma_n, \ldots\rangle$.

  Therefore, from the typing rule REGISTERS, we have $\cdot;\cdot \vdash R\{r_d \mapsto v_n\} : \Gamma\{r_d \mapsto \sigma_n\}$. Thus, $\vdash S'$.

- Case `st`
  Let $S \equiv (P, M \equiv M'\{n' \mapsto \langle\langle \ldots, v_n, \ldots, \rangle\rangle\}, R, \texttt{st}\ r_s, [r_d + n]; I)$. Then, $S' \equiv (P, M'\{n' \mapsto \langle\langle \ldots, v, \ldots\rangle\rangle\}, R, I)$, where $n' \equiv R(r_d)$ and $v \equiv R(r_s)$. From the assumption $\vdash S$, we have $P \vdash \Phi$, $\vdash M : \Sigma$, $\vdash R : \Gamma$ and $\cdot;\Gamma;\cdot;\Sigma \vdash \texttt{st}\ r_s, [r_d + n]; I$.

  Now, from the typing rule STORE, $\cdot;\cdot \vdash \Sigma = \Sigma' \otimes \{n' \mapsto \langle \ldots, \sigma_n, \ldots\rangle\}$ and $\cdot;\Gamma;C;\Sigma' \otimes \{n' \mapsto \langle \ldots, \Gamma(r_s), \ldots\rangle\} \vdash I$, because $n' \equiv R(r_d) \equiv \Gamma(r_d)$.

  Here, by the lemma A.40, $\vdash M'\{n' \mapsto \langle\langle \ldots, v_n, \ldots\rangle\rangle\} : \Sigma' \otimes \{n' \mapsto \langle \ldots, \sigma_n, \ldots\rangle\}$. Now, by the lemma A.43,$\cdot;\cdot \vdash \langle\langle \ldots, v, \ldots\rangle\rangle : \langle \ldots, \sigma_n, \ldots\rangle$. Therefore, by the lemma A.10, $\cdot;\cdot \vdash M' : \Sigma'$. Then, by the lemma A.11, $\vdash M'\{n' \mapsto \langle\langle \ldots, v, \ldots\rangle\rangle\} : \Sigma' \otimes \{n' \mapsto \langle \ldots, \sigma, \ldots\rangle\}$ where $\sigma \equiv \Gamma(r_s)$, because $\cdot;\cdot \vdash \{n' \mapsto \langle\langle \ldots, v, \ldots\rangle\rangle\} : \{n' \mapsto \langle \ldots, \sigma, \ldots\rangle\}$. Thus, $\vdash S'$.

- Case `mov`
  Let $S \equiv (P, M, R, \texttt{mov}\ r_s, r_d; I)$. Then, $S' \equiv (P, M, R\{r_d \mapsto v\}, I)$, where $v \equiv R(r_s)$. From the assumption $\vdash S$, we have $P \vdash \Phi$, $\vdash M : \Sigma$, $\vdash R : \Gamma$ and $\cdot;\Gamma;\cdot;\Sigma \vdash \texttt{mov}\ r_s, r_d; I$.

  From $\vdash R : \Gamma$, we have $\cdot;\cdot \vdash v : \sigma$, where $\sigma \equiv \Gamma(r_s)$. Therefore, $\vdash R\{r_d \mapsto v\} : \Gamma\{r_d \mapsto \sigma\}$. Last, from the typing rule MOVE, $\cdot;\Gamma\{r_d \mapsto \sigma\};\cdot;\Sigma \vdash I$. Thus, $\vdash S'$.

- Case `movi`
  Let $S \equiv (P, M, R, \texttt{movi}\ v, r_d; I)$. Then, $S' \equiv (P, M, R\{r_d \mapsto v\}, I)$. From the assumption $\vdash S$, we have $P \vdash \Phi, \vdash M : \Sigma, \vdash R : \Gamma$ and $\cdot; \Gamma; \cdot; \Sigma \vdash \texttt{movi}\ v, r_d; I$.

  Now, from the typing rule MOVEI, $\cdot; \Gamma\{r_d \mapsto \sigma\}; \cdot; \Sigma \vdash I$ and $\cdot; \cdot \vdash v : \sigma$. Therefore, $\vdash R\{r_d \mapsto v\} : \Gamma\{r_d \mapsto \sigma\}$. Thus, $\vdash S'$.

- Case `add`,`sub` and `mul`
  Let $S \equiv (P, M, R, (\texttt{add}, \texttt{sub}, \texttt{mul})\ r_{s1}, r_{s2}, r_d; I)$. Then, $S' \equiv (P, M, R\{r_d \mapsto v\}, I)$, where $v \equiv R(r_{s2})(+, -, *)R(r_{s1})$. From the assumption $\vdash S$, we have $P \vdash \Phi, \vdash M : \Sigma, \vdash R : \Gamma$ and $\cdot; \Gamma; \cdot; \Sigma \vdash (\texttt{add}, \texttt{sub}, \texttt{mul})\ r_{s1}, r_{s2}, r_d; I$.

  Now, from the typing rule ARITH, $\cdot; \Gamma\{r_d \mapsto \Gamma(r_{s2})(+, -, *)\Gamma(r_{s1})\}; \cdot; \Sigma \vdash I$. Here $\vdash R\{r_d \mapsto v\} : \Gamma\{r_d \mapsto \Gamma(r_{s2})(+, -, *)\Gamma(r_{s1})\}$ because $R(r_{s2}) \equiv \Gamma(r_{s2})$ and $R(r_{s1}) \equiv \Gamma(r_{s1})$. Thus, $\vdash S'$.

- Case `beq` and `ble`
  Let $S \equiv (P, M, R, (\texttt{beq}, \texttt{ble})\ r_{s1}, r_{s2}, r_d; I)$. From the assumption $\vdash S$, we have $P \vdash \Phi, \vdash M : \Sigma, \vdash R : \Gamma$ and $\cdot; \Gamma; \cdot; \Sigma \vdash (\texttt{beq}, \texttt{ble})\ r_{s1}, r_{s2}, r_d; I$.

  First, if $R(r_{s1})(=, \leq)R(r_{s2})$, then $S' \equiv (P, M, R, I'\theta)$ where $P(l) \equiv I'$ and $R(r_d) \equiv l\theta$ and $\theta \equiv [c_1, \ldots, c_n/\Delta]$. Now, from $P \vdash \Phi$, we have $\Delta_1; \Gamma_1; C_1; \Sigma_1 \vdash I'$, where $\Phi(l) \equiv \forall\Delta_1.\ |C_1|\ [\Sigma_1]\ (\Gamma_1)$. Here, by the assumption, $\cdot; \cdot \vdash l\theta : \Gamma(r_d)$. In addition, by the lemma A.2, $\Gamma(r_d) \equiv \forall\Delta_2.\ |C_2|\ [\Sigma_2]\ (\Gamma_2)$ and $\cdot; \cdot \vdash \Gamma(r_d) = \Delta_1 \backslash \Delta.|C_1\theta|[\Sigma_1\theta](\Gamma_1\theta)$.

  Now, from the typing rule BRANCH, we have $\cdot; \cdot \vdash \Delta_1 \backslash \Delta.|C_1\theta|[\Sigma_1\theta](\Gamma_1\theta) = \Gamma(r_d) == \forall.|C'|[\Sigma'](\Gamma')$. Here, by the typing rule EQLABEL, we have $\Delta_1 \equiv \Delta, \cdot; \cdot \vdash C_1\theta = C', \cdot; \cdot \vdash \Sigma_1\theta = \Sigma', \cdot; \cdot \vdash \Gamma_1\theta = \Gamma'$.

  In addition, from the typing rule BRANCH, we have $\cdot; C \models C', \cdot; C \vdash \Sigma = \Sigma'$ and $\cdot; C \vdash \Gamma \leq \Gamma'$, where $C \equiv \Gamma(r_{s1})(=, \leq)\Gamma(r_{s2})$. Here $\cdot; \cdot \models \Gamma(r_{s1})(=, \leq)\Gamma(r_{s2})$ because $\Gamma(r_{s1}) = R(r_{s1})$ and $\Gamma(r_{s2}) = R(r_{s2})$. Therefore, by the lemmas A.22, A.27 and A.29, we have $\cdot; \cdot \models C'$, $\cdot; \cdot \vdash \Sigma = \Sigma'$ and $\cdot; \cdot \vdash \Gamma \leq \Gamma'$. Further, from $\cdot; \cdot \vdash C_1\theta = C'$, we have $\cdot; C' \models C_1\theta$, Thus, $\cdot; \cdot \models C_1\theta$.

  Here, by the type substitution lemma A.21, we have $\cdot; \Gamma_1\theta; C_1\theta; \Sigma_1\theta \vdash I'\theta$. Now, by the lemma A.32, $\cdot; \Gamma_1\theta; \cdot; \Sigma_1\theta \vdash I'\theta$, because $\cdot; \cdot \models C_1\theta$. Therefore, to prove $\vdash S'$, we need to show that $\vdash M : \Sigma_1\theta$ and $\vdash R : \Gamma_1\theta$. Here, by the lemma A.40, $\vdash M : \Sigma_1\theta$, because $\cdot; \cdot \vdash \Sigma_1\theta = \Sigma' = \Sigma$. In addition, by the lemmas A.41 and A.42, $\vdash R : \Gamma_1\theta$, because $\cdot; \cdot \vdash \Gamma_1\theta = \gamma' \geq \Gamma$. Thus, $\vdash S'$.

Second, if $R(r_{s1})(\neq,>)R(r_{s2})$, then $S' =\equiv (P, M, R, I)$. From the typing rule BRANCH, we have $\cdot;\Gamma;\Gamma(r_{s1})(\neq,>)\Gamma(r_{s2});\Sigma \vdash I$. Here $\cdot;\cdot \models \Gamma(r_{s1})(\neq,>)\Gamma(r_{s2})$ because $\Gamma(r_{s1}) \equiv R(r_{s1})$ and $\Gamma(r_{s2}) \equiv R(r_{s2})$. Therefore, by the lemma A.32, we have $\cdot;\Gamma;\cdot;\Sigma \vdash I$. Thus, $\vdash S'$.

- Case `jmp`
  Let $S \equiv (P, M, R, \texttt{jmp}\ r_d)$. Then, $S' \equiv (P, M, R, I\theta)$ where $P(l) \equiv I$, $R(r_d) \equiv l\theta$ and $\theta \equiv [c_1, \ldots, c_n/\Delta]$. From the assumption $\vdash S$, we have $P \vdash \Phi, \vdash M : \Sigma, \vdash R : \Gamma$ and $\cdot;\Gamma;\cdot;\Sigma \vdash \texttt{jmp}\ r_d$.

  Here, from $P \vdash \Phi$, we have $\Delta_1;\Gamma_1;C_1;\Sigma_1 \vdash I$ where $\Phi(l) \equiv \forall\Delta_1.\ |C_1|\ [\Sigma_1]\ (\Gamma_1)$ Now, by the assumption, $\cdot;\cdot \vdash l\theta : \Gamma(r_d)$. In addition, by the lemma A.2, $\Gamma(r_d) \equiv \forall\Delta_2.\,|C_2|\,[\Sigma_2]\,(\Gamma_2)$ and $\cdot;\cdot \vdash \Gamma(r_d) = \Delta_1\backslash\Delta.|C_1\theta|[\Sigma_1\theta](\Gamma_1\theta)$.

  Now, from the typing rule JUMP, we have $\cdot;\cdot \vdash \forall\Delta_1\backslash\Delta.|C_1\theta|[\Sigma_1\theta](\Gamma_1\theta) = \forall.|C'|[\Sigma'](\Gamma')$. Here, by the typing rule EQLABEL, we have $\Delta_1 \equiv \Delta$, $\cdot;\cdot \vdash C_1\theta = C'$, $\cdot;\cdot \vdash \Sigma_1\theta = \Sigma'$, $\cdot;\cdot \vdash \Gamma_1\theta = \Gamma'$.

  In addition, from the typing rule JUMP, we have $\cdot;\cdot \models C'$, $\cdot;\cdot \vdash \Sigma = \Sigma'$ and $\cdot;\cdot \vdash \Gamma \leq \Gamma'$. Further, by $\cdot;\cdot \vdash C_1\theta = C'$, we have $\cdot;\cdot \models C_1\theta$, because $\cdot;C' \models C_1\theta$ from the typing rule EQLABEL and the lemma A.22.

  Here, by the type substitution lemma A.21, we have $\cdot;\Gamma_1\theta;C_1\theta;\Sigma_1\theta \vdash I'\theta$. Now, by the lemma A.32, $\cdot;\Gamma_1\theta;\cdot;\Sigma_1\theta \vdash I'\theta$, because $\cdot;\cdot \models C_1\theta$. Therefore, to prove $\vdash S'$, we need to show that $\vdash M : \Sigma_1\theta$ and $\vdash R : \Gamma_1\theta$. Here, by the lemma A.40, $\vdash M : \Sigma_1\theta$, because $\cdot;\cdot \vdash \Sigma_1\theta = \Sigma' = \Sigma$. In addition, by the lemmas A.41 and A.42, $\vdash R : \Gamma_1\theta$, because $\cdot;\cdot \vdash \Gamma_1\theta = \Gamma' \geq \Gamma$. Thus, $\vdash S'$.

- Case `apply`
  Let $S \equiv (P, M, R, \texttt{apply}\ r\theta; I)$ and $\theta \equiv [c_1, \ldots, c_n/\Delta]$. Then, $S' \equiv (P, M, R\{r \mapsto v\theta\}, I)$, where $v \equiv R(r) \equiv l\theta'$ and $\theta' \equiv [c'_1, \ldots, c'_m/\Delta']$.

  From the assumption $\vdash S$, we have $\vdash P \vdash \Phi, \vdash M : \Sigma$ and $\vdash R : \Gamma$. Here let $\Phi(l) = \forall\Delta_1.|C_1|[\Sigma_1](\Gamma_1)$. Then, from the lemma A.2 and $\cdot;\cdot \vdash v : \Gamma(r), \Gamma(r) \equiv \forall\Delta_2.|C_2|[\Sigma_2](\Gamma_2)$ and $\cdot;\cdot \vdash \Gamma(r) = \forall\Delta_1\backslash\Delta'.|C_1\theta'|[\Sigma_1\theta'](\Gamma_1\theta')$. Thus, from the typing rule EQLABEL, $\Delta_2 \equiv \Delta_1\backslash\Delta'$, $\Delta_2;\cdot \vdash C_2 = C_1\theta'$, $\Delta_2;C_2 \vdash \Sigma_2 = \Sigma_1\theta'$, $\Delta_2;C_2 \vdash \Gamma_2 = \Sigma_1\theta'$.

  Here, by the lemmas A.13, A.17 and A.18, $\Delta_2\backslash\Delta;\cdot C_2\theta = C_1\theta'$, $\Delta_2\backslash\Delta;C_2\theta \vdash \Sigma_2\theta = \Sigma_1\theta'$, $\Delta_2\backslash\Delta;C_2\theta \vdash \Gamma_2\theta = \Sigma_1\theta'$. Now, let $\sigma_f \equiv \forall\Delta_2\backslash\Delta.|C_2\theta|[\Sigma_2\theta](\Gamma_2\theta)$. Then, by the typing rule EQLABEL, $\sigma_f = \forall\Delta_1\backslash\Delta'\Delta.|C_1\theta'\theta|[\Sigma_1\theta'\theta](\Gamma_1\theta'\theta)$.

  Now, by the typing rule APPLY, we have $\cdot;\Gamma\{r \mapsto \sigma_f\};\cdot;\Sigma \vdash I$. Thus, to prove $\vdash S'$, we need to show $\cdot;\cdot \vdash R\{r \mapsto v\theta\} : \Gamma\{r \mapsto$

$\sigma_f\}$. That is, we need to show $\cdot;\cdot \vdash v\theta : \sigma_f$. Here, by the typing rule VALUELABEL, $\cdot;\cdot \vdash l\theta'\theta(= v\theta) : \forall\Delta_1\backslash\Delta'\Delta.|C_1\theta'\theta|[\Sigma_1\theta'\theta](\Gamma_1\theta'\theta)$. Thus, by the lemma A.37, we have $\cdot;\cdot \vdash v\theta : \sigma_f$. Thus, $\vdash S'$.

- Case `roll`
  Let $S \equiv (P, M\{n \mapsto \langle t\rangle\}, R, \texttt{roll}_\tau\ n; I)$. Then, $S' \equiv (P, M\{n \mapsto \langle \texttt{roll}\ (t)\rangle\}, R, I)$. From the assumption $\vdash S$, we have $\vdash P : \Phi$, $\vdash M\{n \mapsto \langle t\rangle\} : \Sigma$, $\vdash R : \Gamma$ and $\cdot;\Gamma;\cdot;\Sigma \vdash \texttt{roll}_\tau\ n; I$.

  Here, from the typing rule ROLL, we have $\cdot;\cdot \vdash \Sigma = \Sigma' \otimes \{i \mapsto \tau'[\mu\eta[\Delta'].\tau'/\eta][c_1, \ldots, c_n/\Delta']\}$ and $\cdot;\Gamma;\cdot;\Sigma' \otimes \{i \mapsto \tau\} \vdash I$, where $\tau = \mu\eta[\Delta'].\tau'(c_1, \ldots, c_n)$.

  Now, by the lemma A.40, $\vdash M\{n \mapsto \langle t\rangle\} : \Sigma' \otimes \{n \mapsto \tau'[\mu\eta[\Delta'].\tau'/\eta][c_1, \ldots, c_n/\Delta']\}$. Thus, by the lemma A.43, $\cdot;\cdot \vdash \{n \mapsto \langle t\rangle\} : \{n \mapsto \tau'[\mu\eta[\Delta'].\tau'/\eta][c_1, \ldots, c_n/\Delta']\}$. Therefore, by the lemma A.10, $\vdash M : \Sigma'$.

  Here $\vdash \{n \mapsto \langle \texttt{roll}(t)\rangle\} : \{n \mapsto \tau\}$ because $\cdot;\cdot \vdash \texttt{roll}(t) : \tau$. Thus, by the lemma A.11, $\vdash M\{n \mapsto \langle \texttt{roll}(t)\rangle\} : \Sigma' \otimes \{n \mapsto \tau\}$. Thus, $\vdash S'$.

- Case `unroll`
  Let $S \equiv (P, M\{n \mapsto \langle \texttt{roll}\ (t)\rangle\}, R, \texttt{unroll}\ n; I)$. Then, $S' = (P, M\{n \mapsto \langle t\rangle\}, R, I)$. From the assumption $\vdash S$, we have $\vdash P : \Phi$, $\vdash M\{n \mapsto \langle \texttt{roll}(t)\rangle\} : \Sigma$, $\vdash R : \Gamma$ and $\cdot;\Gamma;\cdot;\Sigma \vdash \texttt{unroll}\ n; I$.

  Here, from the typing rule UNROLL, we have $\cdot;\cdot \vdash \Sigma = \Sigma' \otimes \{n \mapsto \tau\}$ and $\cdot;\Gamma;\cdot;\Sigma' \otimes \{n \mapsto \tau'[\mu\eta[\Delta'].\tau'/\eta][c_1, \ldots, c_n/\Delta']\} \vdash I$, where $\tau \equiv \mu\eta[\Delta'].\tau'(c_1, \ldots, c_n)$.

  Now, by the lemma A.40, $\vdash M\{n \mapsto \langle \texttt{roll}(t)\rangle\} : \Sigma' \otimes \{n \mapsto \tau\}$. Here, by the lemma A.43, $\vdash \{n \mapsto \langle \texttt{roll}(t)\rangle\} : \{n \mapsto \tau\}$. Therefore, by the lemma A.10, $\vdash M : \Sigma'$.

  Here, by the typing rule TUPLEROLL, $\cdot;\cdot \vdash t : \tau'[\mu\eta[\Delta'].\tau'/\eta][c_1, \ldots, c_n/\Delta']$. Thus, $\vdash \{n \mapsto \langle t\rangle\} : \{n \mapsto \tau'[\mu\eta[\Delta'].\tau'/\eta][c_1, \ldots, c_n/\Delta']$. Therefore, by the lemma A.11, $\vdash M\{n \mapsto \langle t\rangle\} : \Sigma' \otimes \{n \mapsto \tau'[\mu\eta[\Delta'].\tau'/\eta][c_1, \ldots, c_n/\Delta']$. Thus, $\vdash S'$.

- Case `pack`
  Let $S \equiv (P, M\{n \mapsto \langle t\rangle\}M', R, \texttt{pack}_{[c_1,\ldots,c_n|\Sigma_1]as\ \tau}\ n; I)$. Then, $S' \equiv (P, M\{n \mapsto \left\langle \texttt{pack}_{[c_1,\ldots,c_n|M']}\ (t)\right\rangle\}, R, I)$, where $Dom(M') = \{n | n \in Dom(\Sigma_1) s.t. \Sigma_1(n) \equiv \tau_n[m(> 0)]\}$. From the assumption $\vdash S$, we have $\vdash P : \Phi$, $\vdash R : \Gamma$, $\vdash M\{n \mapsto \langle t\rangle\}M' : \Sigma$ and $\cdot;\Gamma;\cdot;\Sigma \vdash \texttt{pack}_{[c_1,\ldots,c_n|\Sigma]as\ \tau}\ n; I$.

Now, from the typing rule PACK, we have $\tau \equiv \exists\Delta'.|C'|[\Sigma']\tau'$, $\cdot;\cdot \vdash \Sigma = \Sigma'' \otimes \{n \mapsto \tau'\theta\} \otimes \Sigma'\theta$, $\cdot;\cdot \models C'\theta$ and $\cdot;\Gamma;\cdot;\Sigma'' \otimes \{n \mapsto \tau\} \vdash I$, where $\theta \equiv [c_1, \ldots, c_n/\Delta']$ and $\Sigma_1 \equiv \Sigma'\theta$.

Here, by the lemma A.40, $\vdash M\{n \mapsto \langle t\rangle\}M' : \Sigma'' \otimes \{n \mapsto \tau'\theta\} \otimes \Sigma_1$. Then, by the lemma A.10, $\vdash M : \Sigma''$ and $\vdash M' : \Sigma'\theta(= \Sigma_1)$. In addition, by the lemma A.43,$\cdot;\cdot \vdash \{n \mapsto \langle t\rangle\} : \{n \mapsto \tau'\theta\}$. That is, $\cdot;\cdot \vdash t : \tau'\theta$.

Now, from the typing rule TUPLEPACK, $\cdot;\cdot \vdash \mathtt{pack}_{[c_1,\ldots,c_n|M']}(t) : \tau$. Thus, $\cdot;\cdot \vdash \{n \mapsto \langle t\rangle\} : \{n \mapsto \tau\}$. Here, by the lemma A.11, $\vdash M\{n \mapsto \left\langle \mathtt{pack}_{[c_1,\ldots,c_n|M']}(t)\right\rangle\} : \Sigma'' \otimes \{n \mapsto \tau\}$. Thus, $\vdash S'$.

- Case `unpack`
  Let $S \equiv (P, M, R, \mathtt{unpack}\ n\ \mathtt{with}\ \Delta; I)$. Then, $S' \equiv (P, M''\{n \mapsto \langle t\rangle\}M', R, I\,[c_1, \ldots, c_n/\Delta])$, where $M = M''\{n \mapsto \left\langle \mathtt{pack}_{[c_1,\ldots,c_n|M']}\,(t)\right\rangle\}$. From the assumption $\vdash S$, we have $\vdash P : \Phi$, $\vdash M : \Sigma$, $\vdash R : \Gamma$ and $\cdot;\Gamma;\cdot;\Sigma \vdash \mathtt{unpack}\ n\ \mathtt{with}\ \Delta; I$.

  Now, from the typing rule UNPACK, we have $\cdot;\cdot \vdash \Sigma = \Sigma'' \otimes \{n \mapsto \tau\}$, $\Delta;\Gamma;C'\theta;\Sigma'' \otimes \{n \mapsto \tau'\theta\} \otimes \Sigma'\theta \vdash I$, where $\theta \equiv [\Delta/\Delta']$ and $\tau \equiv \exists\Delta'.|C'|[\Sigma']\tau'$.

  Here, by the lemma A.40, $\vdash M''\{n \mapsto \left\langle \mathtt{pack}_{[c_1,\ldots,c_n|M']}(t)\right\rangle : \Sigma'' \otimes \{n \mapsto \tau\}$. Then, by the lemma A.10, $\vdash M'' : \Sigma''$. In addition, by the lemma A.43,$\cdot;\cdot \vdash \{n \mapsto \left\langle \mathtt{pack}_{[c_1,\ldots,c_n|M']}(t)\right\rangle : \{n \mapsto \tau\}$. Then, $\cdot;\cdot \vdash \mathtt{pack}_{[c_1,\ldots,c_n|M']}(t) : \tau$.

  Now, from the type TUPLEPACK, we have $\cdot;\cdot \vdash t : \tau'[c_1, \ldots, c_n/\Delta']$, $\vdash M' : \Sigma'[c_1, \ldots, c_n/\Delta']$ and $\cdot;\cdot \models C'[c_1, \ldots, c_n/\Delta']$.

  Here, by the lemma A.21, $\cdot;\Gamma\theta';C'\theta\theta';\Sigma''\theta' \otimes \{n \mapsto \tau'\theta\theta'\} \otimes \Sigma'\theta\theta' \vdash I\theta'$, where $\theta' \equiv [c_1, \ldots, c_n/\Delta]$. Here $\Gamma\theta' \equiv \Gamma$ and $\Sigma''\theta' \equiv \Sigma''$, because Γ and Σ'' do not contain the type variables of Δ. Thus, $\cdot;\Gamma;C'\theta\theta';\Sigma'' \otimes \{n \mapsto \tau'\theta\theta'\} \otimes \Sigma'\theta\theta' \vdash I\theta'$. Here $\theta\theta' \equiv [c_1, \ldots, c_n/\Delta']$. Thus, $\cdot;\Gamma;C'\,[c_1, \ldots, c_n/\Delta'];\Sigma'' \otimes \{n \mapsto \tau'\,[c_1, \ldots, c_n/\Delta']\} \otimes \Sigma'\,[c_1, \ldots, c_n/\Delta'] \vdash I\theta'$. Here, by the lemma A.32 and $\cdot;\cdot \models C'[c_1, \ldots, c_n/\Delta']$, we have $\cdot;\Gamma;\cdot;\Sigma'' \otimes \{n \mapsto \tau'\,[c_1, \ldots, c_n/\Delta']\} \otimes \Sigma'\,[c_1, \ldots, c_n/\Delta'] \vdash I\theta'$.

  Now, by the lemma A.11, $\vdash M''\{n \mapsto \langle t\rangle\}M' : \Sigma''\{n \mapsto \tau'\,[c_1, \ldots, c_n/\Delta']\} \otimes \Sigma'\,[c_1, \ldots, c_n/\Delta']$. Thus, $\vdash S'$.

- Case `split`
  Let $S \equiv (P, M, R, \mathtt{split}\ n_1, n_2; I)$. From the assumption $\vdash S$, we have $\vdash P : \Phi$, $\vdash M : \Sigma$, $\vdash R : \Gamma$ and $\cdot;\Gamma;\cdot;\Sigma \vdash \mathtt{split}\ n_1, n_2; I$.

First, if $n_2 = 0$, then $S' \equiv (P, M, R, I)$. By the typing rule SPLIT, we have $\vdash M : \Sigma' \otimes \{n_1 \mapsto \tau[n]\}$ and $\cdot;\Gamma;\cdot;\Sigma' \otimes \{n_1 \mapsto \tau[0]\} \otimes \{n_1 \mapsto \tau[n]\} \vdash I$. Therefore, if $\vdash M : \Sigma' \otimes \{n_1 \mapsto \tau[0]\} \otimes \{n_1 \mapsto \tau[n]\}$, then $\vdash S'$. Here, by the typing rule EQMEMZEROARRAYL, we have $\cdot;\cdot \vdash \Sigma' \otimes \{n_1 \mapsto \tau[0]\} \otimes \{n_1 \mapsto \tau[n]\} = \Sigma' \otimes \{n_1 \mapsto \tau[n]\}$. Now, by the lemma A.40, we have $\vdash M : \Sigma' \otimes \{n_1 \mapsto \tau[0]\} \otimes \{n_1 \mapsto \tau[n]\}$. Thus, $\vdash S'$.

Second, if $n_2 \neq 0$ and $M \equiv M'\{n_1 \mapsto \langle t_1, \ldots, t_{n_2}\rangle\}$, then $S' \equiv (P, M, R, I)$. Here, by the typing rule SPLIT, we have $\cdot;\cdot \vdash \Sigma = \Sigma' \otimes \{n_1 \mapsto \tau\,[n_2]\}$ and $\cdot;\Gamma;\cdot;\Sigma' \otimes \{n_1 \mapsto \tau\,[n_2]\} \otimes \{(n_1 + \sum_{i=1}^{n_2} \mathit{sizeof}\,(t_i)) \mapsto \tau\,[0]\}$.

Now, from the typing rule EQMEMZEROARRAYR, $\cdot;\cdot \vdash \Sigma' \otimes \{n_1 \mapsto \tau\,[n_2]\} = \Sigma' \otimes \{n_1 \mapsto \tau\,[n_2]\} \otimes \{(n_1 + \mathit{sizeof}\,(\tau) * n_2) \mapsto \tau\,[0]\}$. Therefore, by the lemma A.40, $\vdash M : \Sigma' \otimes \{n_1 \mapsto \tau\,[n]\} \otimes \{(n_1 + \mathit{sizeof}\,(\tau) * n_2) \mapsto \tau\,[0]\}$. Thus, $\vdash S' (= S)$.

Last, if $0 < n_2 < n$, then $S' \equiv (P, M'\{n_1 \mapsto \langle t_1, \ldots, t_{n_2}\rangle\}\{n' \mapsto \langle t_{n_2+1}, \ldots, t_n\rangle, R, I)$, where $M \equiv M'\{n_1 \mapsto \langle t_1, \ldots, t_{n_2}, \ldots, t_n\rangle\}$ and $n' \equiv n_1 + \sum_{i=1}^{n} \mathit{sizeof}\,(t_i)$. Here, by the typing rule SPLIT, we have $\cdot;\cdot \vdash \Sigma = \Sigma' \otimes \{n_1 \mapsto \tau\,[n]\}$ and $\cdot;\Gamma;\cdot;\Sigma' \otimes \{n_1 \mapsto \tau\,[n_2]\} \otimes \{(n_1 + \mathit{sizeof}\,(\tau) * n_2) \mapsto \tau\,[n - n_2]\}$.

Now, by the lemma A.40, $\vdash M'\{n_1 \mapsto \langle t_1, \ldots, t_{n_2}, \ldots, t_n\rangle\} : \Sigma' \otimes \{n_1 \mapsto \tau\,[n]\}$. Here, by the lemma A.10, $\vdash M' : \Sigma'$ and $\vdash \{n_1 \mapsto \langle t_1, \ldots, t_{n_2}, \ldots, t_n\rangle\} : \{n_1 \mapsto \tau\,[n]\}$. That is, $forall.\cdot;\cdot \vdash \tau_i : \tau$. Therefore, $\cdot;\cdot \vdash \langle t_1, \ldots, t_{n_2}\rangle : \tau[n_2]$ and $\cdot;\cdot \vdash \langle t_{n_2+1}, \ldots, t_n\rangle : \tau[n - n_2]$. Thus, by the lemma A.11, $\vdash M'\{n_1 \mapsto \langle t_1, \ldots, t_{n_2}\rangle\}\{n' \mapsto \langle t_{n_2+1}, \ldots, t_n\rangle\} : \Sigma' \otimes \{n_1 \mapsto \tau[n_2]\} \otimes \{n' \mapsto \tau[n - n2]\}$. Here, by the definition of *sizeof* and $\forall i.\mathit{sizeof}\,(t_i) = \mathit{sizeof}\,(\tau)$, $n' = n_1 + \mathit{sizeof}\,(\tau) * n_2$. Therefore, $\vdash M'\{n_1 \mapsto \langle t_1, \ldots, t_{n_2}\rangle\}\{n' \mapsto \langle t_{n_2+1}, \ldots, t_n\rangle\} : \Sigma' \otimes \{n_1 \mapsto \tau[n_2]\} \otimes \{(n_1 + \mathit{sizeof}\,(\tau) * n_2) \mapsto \tau[n - n2]\}$. Thus, $\vdash S'$.

- Case `concat`
  Let $S \equiv (P, M, R, \texttt{concat}\ n_1, n_2, n_3; I)$. From the assumption $\vdash S$, we have $\vdash P : \Phi$, $\vdash M : \Sigma$, $\vdash R : \Gamma$ and $\cdot;\Gamma;\cdot;\Sigma \vdash \texttt{concat}\ n_1, n_2, n_3; I$.

  First, if $n_3 = 0$ and $n_1 = n_2$, then $S' \equiv (P, M, R, I)$. By the typing rule CONCAT, we have $\cdot;\cdot \vdash \Sigma = \Sigma' \otimes \{n_1 \mapsto \tau[0]\} \otimes \{n_2 \mapsto \tau[0]\}$ and $\cdot;\Gamma;\cdot;\Sigma' \otimes \{n_1 \mapsto \tau[0]\} \vdash I$. Now, by the lemma A.40, $\vdash M : \Sigma' \otimes \{n_1 \mapsto \tau[0]\} \otimes \{n_2 \mapsto \tau[0]\}$.

  Here, by the typing rule EQMEMZEROARRAYL, $\cdot;\cdot \vdash \Sigma' \otimes \{n_1 \mapsto$

$\tau[0]\} \otimes \{n_2 \mapsto \tau[0]\} = \Sigma' \otimes \{n_1 \mapsto \tau[0]\}$. Now, by the lemma A.40, $\vdash M : \Sigma' \otimes \{n_1 \mapsto \tau[0]\}$. Thus, $\vdash S'$.

Second, if $n_3 > 0$ and $n_1 = n_2$, then $S' \equiv (P, M, R, I)$. By the typing rule CONCAT, we have $\cdot;\cdot \vdash \Sigma = \Sigma' \otimes \{n_1 \mapsto \tau[0]\} \otimes \{n_2 \mapsto \tau[n_3]\}$ and $\cdot;\Gamma;\cdot;\Sigma' \otimes \{n_1 \mapsto \tau[n_3]\} \vdash I$. Now, by the lemma A.40, we have $\vdash M : \Sigma' \otimes \{n_1 \mapsto \tau[0]\} \otimes \{n_2 \mapsto \tau[n_3]\}$

Here, by the typing rule EQMEMZEROARRAYL, $\cdot;\cdot \vdash \Sigma' \otimes \{n_1 \mapsto \tau[0]\} \otimes \{n_2 \mapsto \tau[n_3]\} = \Sigma' \otimes \{n_2 \mapsto \tau[n_3]\} = \Sigma' \otimes \{n_1 \mapsto \tau[n_3]\}$. Now, by the lemma A.40, $\vdash M : \Sigma' \otimes \{n_1 \mapsto \tau[n_3]\}$. Thus, $\vdash S'$.

Third, if $n_3 = 0$ and $n_1 \neq n_2$, then $S' \equiv (P, M, R, I)$, $M \equiv M'\{n_1 \mapsto \langle t_1, \ldots, t_n\rangle\}$ and $n_2 = n_1 + \sum_{i=1}^{n} \mathit{sizeof}(t_i)$. By the typing rule CONCAT, we have $\cdot;\cdot \vdash \Sigma = \Sigma' \otimes \{n_1 \mapsto \tau[n]\} \otimes \{n_2 \mapsto \tau[0]\}$ and $\cdot;\Gamma;\cdot;\Sigma' \otimes \{n_1 \mapsto \tau[n]\} \vdash I$. Now, by the lemma A.40, $\vdash M : \Sigma' \otimes \{n_1 \mapsto \tau[n]\} \otimes \{n_2 \mapsto \tau[0]\}$

Here, by the typing rule EQMEMZEROARRAYL, $\cdot;\cdot \vdash \Sigma' \otimes \{n_1 \mapsto \tau[n]\} \otimes \{n_2 \mapsto \tau[0]\} = \Sigma' \otimes \{n_1 \mapsto \tau[n]\}$. Now, by the lemma A.40, $\vdash M : \Sigma' \otimes \{n_1 \mapsto \tau[n]\}$. Thus, $\vdash S'$.

Last, if $n_3 > 0$ and $n_1 \neq n_2$, then $S' \equiv (P, M'\{n_1 \mapsto \langle t_1, \ldots, t_n, t'_1, \ldots, t'_{n_3}\rangle\}, R, I)$, $M \equiv M'\{n_1 \mapsto \langle t_1, \ldots, t_n\rangle\}\{n_2 \mapsto \langle t'_1, \ldots, t'_{n_3}\rangle\}$ and $n_2 = n_1 + \sum_{i=1}^{n} \mathit{sizeof}(t_i)$.

By the typing rule CONCAT, we have $\cdot;\cdot \vdash \Sigma = \Sigma' \otimes \{n_1 \mapsto \tau\,[n]\} \otimes \{n_2 \mapsto \tau\,[n_3]\}$, $\cdot;\cdot \vdash n_2 = n_1 + \mathit{sizeof}(\tau) * n$ and $\cdot;\Gamma;\cdot;\Sigma' \otimes \{n_1 \mapsto \tau[n + n_3]\} \vdash I$.

Here, by the lemma A.40, $\vdash M'\{n_1 \mapsto \langle t_1, \ldots, t_n\rangle\}\{n_2 \mapsto \langle t'_1, \ldots, t'_{n_3}\rangle\} : \Sigma' \otimes \{n_1 \mapsto \tau\,[n]\} \otimes \{n_2 \mapsto \tau\,[n_3]\}$ Now, by the lemma A.10, we have $\vdash M' : \Sigma'$, $\vdash \{n_1 \mapsto \langle t_1, \ldots, t_n\rangle\} : \{n_1 \mapsto \tau\,[n]\}$ and $\vdash \{n_2 \mapsto \langle t'_1, \ldots, t'_{n_3}\rangle\} : \{n_2 \mapsto \tau\,[n_3]\}$. Thus, $\forall i.\tau_i : \tau$ and $\forall i.\tau'_i : \tau$. Therefore, $\cdot;\cdot \vdash \langle t_1, \ldots, t_n, t'_1, \ldots, t'_{n_3}\rangle : \tau[n + n_3]$. Here, by the lemma A.11, $\vdash M'\{n_1 \mapsto \langle t_1, \ldots, t_n, t'_1, \ldots, t'_{n_3}\rangle : \Sigma' \otimes \{n_1 \mapsto \tau[n + n_3]\}$. Thus, $\vdash S'$.

- Case `tuple_split`
  Let $S \equiv (P, M\{n_1 \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\}, R, \texttt{tuple\_split}\ n_1, n_2; I)$. Then, $S' \equiv (P, M\{n_1 \mapsto \langle\langle v_1, \ldots, v_{n_2}\rangle\rangle\}\{n'_1 \mapsto \langle\langle v_{n_2+1}, \ldots, v_n\rangle\rangle\}, R, I)$, where $n'_1 = n_1 + n_2$. From the assumption $\vdash S$, we have $\vdash P : \Phi$, $\vdash M\{n_1 \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\} : \Sigma$, $\vdash R : \Gamma$ and $\cdot;\Gamma;\cdot;\Sigma \vdash \texttt{tuple\_split}\ n_1, n_2; I$

  Here, by the typing rule TUPLESPLIT, we have $\cdot;\cdot \vdash \Sigma = \Sigma' \otimes \{n_1 \mapsto \langle\sigma_1, \ldots, \sigma_n\rangle\}$, $\cdot;\cdot \models 0 < n_2 < n$, $\cdot;\Gamma;\cdot;\Sigma' \otimes \{n_1 \mapsto \langle\sigma_1, \ldots, \sigma_{n_2}\rangle\} \otimes \{n_1 + n_2 \mapsto \langle\sigma_{n_2+1}, \ldots, \sigma_n\rangle\} \vdash I$.

Now, by the lemma A.40, $\vdash M\{n_1 \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\} : \Sigma' \otimes \{n_1 \mapsto \langle\sigma_1, \ldots, \sigma_n\rangle\}$. Here, by the lemma A.10, $\vdash M : \Sigma'$ and $\vdash \{n_1 \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\} : \{n_1 \mapsto \langle\sigma_1, \ldots, \sigma_n\rangle\}$. That is, $\forall i.\cdot;\cdot \vdash v_i : \sigma_i$. Therefore, $\cdot;\cdot \vdash \langle v_1, \ldots, v_{n_2}\rangle : \langle\sigma_1, \ldots, \sigma_{n_2}\rangle$ and $\cdot;\cdot \vdash \langle v_{n_2+1}, \ldots, v_n\rangle : \langle\sigma_{n_2+1}, \ldots, \sigma_n\rangle$. Thus, by the lemma A.11, $\vdash M\{n_1 \mapsto \langle v_1, \ldots, v_{n_2}\rangle\}\{n'_1 \mapsto \langle v_{n_2+1}, \ldots, v_n\rangle\} : \Sigma' \otimes \{n_1 \mapsto \langle\sigma_1, \ldots, \sigma_{n_2}\rangle\} \otimes \{n'_1 \mapsto \langle\sigma_{n_2+1}, \ldots, \sigma_n\rangle\}$. Thus, $\vdash S'$.

- Case `tuple_concat`
  Let $S = (P, M\{n_1 \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\}\{n_2 \mapsto \langle\langle v'_1, \ldots, v'_m\rangle\rangle\}, R, \texttt{tuple\_split}\ n_1, n_2; I)$. Then, $S' = (P, M\{n_1 \mapsto \langle\langle v_1, \ldots, v_n, v'_1, \ldots, v'_m\rangle\rangle\}\{n'_1 \mapsto \langle\langle v_{n_2+1}, \ldots, v_n\rangle\rangle\}, R, I)$, where $n_2 = n_1 + n$. From the assumption $\vdash S$, we have $\vdash P : \Phi$, $\vdash M' : \Sigma$, $\vdash R : \Gamma$ and $\cdot;\Gamma;\cdot;\Sigma \vdash \texttt{tuple\_concat}\ n_1, n_2; I$.

  Now, by the typing rule TUPLECONCAT, we have $\cdot;\cdot \vdash \Sigma = \Sigma' \otimes \{n_1 \mapsto \langle\sigma_1, \ldots, \sigma_n\rangle\} \otimes \{n_2 \mapsto \langle\sigma'_1, \ldots, \sigma'_m\rangle\}$, $\cdot;\cdot \models n_2 = n_1 + n$ and $\cdot;\Gamma;\cdot;\Sigma' \otimes \{n_1 \mapsto \langle\sigma_1, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_m\rangle\} \vdash I$.

  Here, by the lemma A.40, $\vdash M\{n_1 \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\}\{n_2 \mapsto \langle\langle v'_1, \ldots, v'_m\rangle\rangle\} : \Sigma' \otimes \{n_1 \mapsto \langle\sigma_1, \ldots, \sigma_n\rangle\} \otimes \{n_2 \mapsto \langle\sigma'_1, \ldots, \sigma'_m\rangle\}$. In addition, by the lemma A.10, $\vdash M : \Sigma'$, $\vdash \{n_1 \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\} : \{n_1 \mapsto \langle\sigma_1, \ldots, \sigma_n\rangle\}$ and $\vdash \{n_2 \mapsto \langle\langle v'_1, \ldots, v'_m\rangle\rangle\} : \{n_2 \mapsto \langle\sigma'_1, \ldots, \sigma'_m\rangle\}$. That is, $\forall i.\cdot;\cdot \vdash v_i : \sigma_i$ and $\forall i.\cdot;\cdot \vdash v'_i : \sigma'_i$. Therefore, by the lemma A.11, $\vdash M\{n_1 \mapsto \langle\langle v_1, \ldots, v_n, v'_1, \ldots, v'_m\rangle\rangle\} : \Sigma' \otimes \{n_1 \mapsto \langle\sigma_1, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_m\rangle\}$. Thus, $\vdash S'$.

A.7 Progress lemma

Lemma A.51 (Progress)
If $\vdash S$, then there exists S' such that $S \mapsto S'$.

Proof Let us suppose that $S = (P, M, R, I)$. From the typing rule $STATE$ and $\vdash S$, we have $\vdash P : \Phi$, $\vdash M : \Sigma$, $\vdash R : \Gamma$ and $\cdot;\Gamma;\cdot;\Sigma \vdash I$. We prove by case analysis on the last typing rule of the typing derivation $\cdot;\Gamma;\cdot;\Sigma \vdash I$.

- Case LOAD
  From the typing rule, $I = \texttt{ld}\ [r_s + n], r_d; I'$. In addition, $\vdash M : \Sigma' \otimes \{n' \mapsto \langle\ldots, \sigma_n, \ldots\rangle\}$ by the lemma A.40, where $n' = \Gamma(r_s) = R(r_s)$. Therefore, by the lemma A.43, $M = M'\{n' \mapsto \langle\langle\ldots, v, \ldots\rangle\rangle\}$. Thus, by the operational semantics of `ld`, there exists S' such that $S \mapsto S'$.

- Case STORE
  From the typing rule, $I = \texttt{st}\ r_s, [r_d + n]; I'$. In addition, $\vdash M : \Sigma' \otimes$

$\{n' \mapsto \langle \ldots, \sigma_n, \ldots \rangle\}$ by the lemma A.40, where $n' = \Gamma(r_d) = R(r_d)$. Therefore, by the lemma A.43, $M = M'\{n' \mapsto \langle\langle \ldots, v_n, \ldots \rangle\rangle\}$. Thus, by the operational semantics of `st`, there exists S' such that $S \mapsto S'$.

- Case MOVE
  From the typing rule, $I = \texttt{mov}\ r_s, r_d; I'$. Thus, by the operational semantics of `mov`, there exists S' such that $S \mapsto S'$.

- Case MOVEI
  From the typing rule, $I = \texttt{movi}\ v, r_d; I'$. Thus, by the operational semantics of `movi`, there exists S' such that $S \mapsto S'$.

- Case ARITH
  From the typing rule, $I = (\texttt{add}, \texttt{sub}, \texttt{mul})\ r_{s_1}, r_{s_2}, r_d; I'$. Thus, by the operational semantics of (`add`, `sub`, `mul`), there exists S' such that $S \mapsto S'$.

- Case BRANCH
  From the typing rule, $I = (\texttt{beq}, \texttt{ble})\ r_{s_1}, r_{s_2}, r_d; I'$. Here, by the assumption $\vdash R : \Gamma$, $\cdot;\cdot \vdash v : \Gamma(r_d)$, where $v = R(r_d)$. Now, by the lemma A.37, $\cdot;\cdot \vdash v : \forall.|C'|[\Sigma'](\Gamma')$. Therefore, by the lemma A.46, $v = l[c_1, \ldots, c_n/\Delta]$. Thus, by the operational semantics of (`beq`, `ble`), there exists S' such that $S \mapsto S'$.

- Case JUMP
  From the typing rule, $I = \texttt{jmp}\ r_d$. Here, by the assumption $\vdash R : \Gamma$ and $\cdot;\cdot \vdash v : \Gamma(r_d)$, where $v = R(r_d)$. Now, by the lemma A.37, $\cdot;\cdot \vdash v : \forall.|C'|[\Sigma'](\Gamma')$. Therefore, by the lemma A.46, $v = l[c_1, \ldots, c_n/\Delta]$. Thus, by the operational semantics of `jmp`, there exists S' such that $S \mapsto S'$.

- Case APPLY
  From the typing rule, $I = \texttt{apply}\ r\,[c_1, \ldots, c_n/\Delta]\,; I'$. Here, by the assumption $\vdash$: Γ, $\cdot;\cdot \vdash v : \forall\Delta'.|C'|[\Sigma'](\Gamma')(= \Gamma(r))$. Therefore, by the lemma A.46, $v = l[c_1, \ldots, c_n/\Delta]$. Thus, by the operational semantics of `apply`, there exists S' such that $S \mapsto S'$.

- Case ROLL
  From the typing rule, $I = \texttt{roll}_\tau\ n; I'$. In addition, $\vdash M : \Sigma' \otimes \{n \mapsto \tau'[\mu\eta[\Delta'].\tau'/\eta][c_1, \ldots, c_n/\Delta']\}$ by the lemma A.40, where $\tau = \mu\eta[\Delta'].\tau'(c_1, \ldots, c_n)$. Now, by the lemma A.43 and the lemma A.44, $M = M'\{n \mapsto \langle t \rangle\}$. Thus, by the operational semantics of `roll`, there exists S' such that $S \mapsto S'$.

- Case UNROLL
  From the typing rule, $I = \texttt{unroll}\, n; I'$. In addition, by the lemma A.40, $\vdash M : \Sigma' \otimes \{n \mapsto \tau\}$, where $\tau = \mu\eta[\Delta'].\tau'(c_1, \ldots, c_n)$. Now, by the lemma A.43 and the lemma A.44, $M = M'\{n \mapsto \langle t \rangle\}$ and $\cdot;\cdot \vdash t : \tau$. Here, by the lemma A.45, $t = \texttt{roll}(t')$. Thus, by the operational semantics of `unroll`, there exists S' such that $S \mapsto S'$.

- Case PACK
  From the typing rule, $I = \texttt{pack}_{[c_1,\ldots,c_n|\Sigma'[c_1,\ldots,c_n/\Delta']]as\ \tau}\, n; I'$, where $\tau = \exists\Delta.|C|[\Sigma']\tau'$. In addition, by the lemma A.40, $\vdash M : \Sigma'' \otimes \{n \mapsto \tau'[c_1, \ldots, c_n/\Delta]\} \otimes \Sigma'[c_1, \ldots, c_n/\Delta']$. Now, by the lemma A.43, $M = M'\{n \mapsto \langle t \rangle\}M''$ and $Dom(M'') \subseteq Dom(\Sigma'[c_1, \ldots, c_n/\Delta'])$. Thus, by the operational semantics of `pack`, there exists S' such that $S \mapsto S'$.

- Case UNPACK
  From the typing rule, $I = \texttt{unpack}\ n\ \texttt{with}\ \Delta; I'$. By the lemma A.40, $\vdash M : \Sigma'' \otimes \{n \mapsto \tau\}$, where $\tau \equiv \exists\Delta'.|C'|[\Sigma']\tau'\}$. Here, by the lemma A.43, the lemma A.44 and the lemma A.45, we have $M = M'\{n \mapsto \langle t \rangle\}$, where $t = \texttt{pack}_{[c_1,\ldots,c_n|M'']}(t')$, $\cdot;\cdot \vdash t' : \tau'[c_1, \ldots, c_n/\Delta']$ and $\vdash M'' : \Sigma'[c_1, \ldots, c_n/\Delta']$. Thus, by the operational semantics of `unpack`, there exists S' such that $S \mapsto S'$.

- Case SPLIT
  From the typing rule, $I = \texttt{split}\ n_1, n_2; I'$.

  If $n_2 = 0$, then there exists S' such that $S \mapsto S'$ by the operational semantics of `split` (the second rule).

  Otherwise (if $n_2 > 0$), by the lemma A.40, $\vdash M : \Sigma' \otimes \{n_1 \mapsto \tau[m]\}$ and $n_2 \leq m$. Now, by the lemma A.43, $M = M'\{n \mapsto \langle t_1, \ldots, t_m \rangle\}$. Here if $n_2 = m$, then there exists S' such that $S \mapsto S'$ by the operational semantics of `split` (the third rule). Otherwise (if $0 < n_2 < m$), there exists S' such that $S \mapsto S'$ by the operational semantics of `split` (the first rule).

- Case CONCAT
  From the typing rule, $I = \texttt{concat}\ n_1, n_2, n_3; I'$.

  If $n_1 = n_2$ and $n_3 = 0$, then, by the operational semantics of `concat` (the forth rule), there exists S' such that $S \mapsto S'$.

  If $n_1 = n_2$ and $n_3 > 0$, then, by the lemma A.40, $\vdash M : \Sigma' \otimes \{n_1 \mapsto \tau[0]\} \otimes \{n_2 \mapsto \tau[n_3]\} = \Sigma' \otimes \{n_1 \mapsto \tau[n_3]\}$. Now, by the lemma A.43,

$M = M'\{n_1 \mapsto \langle t_1, \ldots, t_{n_3}\rangle$. Thus, by the operational semantics of `concat` (the third rule), there exists S' such that $S \mapsto S'$.

If $n_1 < n_2$ and $n_3 = 0$, then, by the lemma A.40, $\vdash M : \Sigma' \otimes \{n_1 \mapsto \tau[m]\} \otimes \{n_2 \mapsto \tau[0]\} = \Sigma' \otimes \{n_1 \mapsto \tau[m]\}$, where $n_2 = n_1 + \mathit{sizeof}(\tau) * m$. Now, by the lemma A.43, $M = M'\{n_1 \mapsto \langle t_1, \ldots, t_m\rangle$ and $\forall i. \cdot; \cdot \vdash t_i : \tau$. Therefore, $\mathit{sizeof}(\tau) * m = \sum_{i=1}^{m} \mathit{sizeof}(t_i)$. Thus, by the operational semantics of `concat` (the second rule), there exists S' such that $S \mapsto S'$.

If $n_1 < n_2$ and $n_3 > 0$, then, by the lemma A.40, $\vdash M : \Sigma' \otimes \{n_1 \mapsto \tau[m]\} \otimes \{n_2 \mapsto \tau[n_3]\}$, where $n_2 = n_1 + \mathit{sizeof}(\tau) * m$. Now, by the lemma A.43, $M = M'\{n_1 \mapsto \langle t_1, \ldots, t_m\rangle\}\{n_2 \mapsto \langle t'_1, \ldots, t'_{n_3}\rangle\}$ and $\forall i. \cdot; \cdot \vdash t_i : \tau$. Therefore, $\mathit{sizeof}(\tau) * m = \sum_{i=1}^{m} \mathit{sizeof}(t_i)$. Thus, by the operational semantics of `concat` (the first rule), there exists S' such that $S \mapsto S'$.

- Case TUPLESPLIT
  From the typing rule, $I =$ `tuple_split` n_1, n_2. By the lemma A.40, $\vdash M : \Sigma' \otimes \{n_1 \mapsto \langle \sigma_1, \ldots, \sigma_n\rangle\}$. Now, by the lemma A.43, the lemma A.44, the lemma A.45 and the lemma A.46, $M = M'\{n_1 \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\}$. Here, by the typing rule TUPLESPLIT, $n_2 < n$. Thus, by the operational semantics of `tuple_split`, there exists S' such that $S \mapsto S'$.

- Case TUPLECONCAT
  From the typing rule, $I =$ `tuple_concat` n_1, n_2. By the lemma A.40, $\vdash M : \Sigma' \otimes \{n_1 \mapsto \langle \sigma_1, \ldots, \sigma_n\rangle\}\{n_2 \mapsto \langle \sigma'_1, \ldots, \sigma'_m\rangle\}$. Now, by the lemma A.43, the lemma A.44, the lemma A.45 and the lemma A.46, $M = M'\{n_1 \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\}\{n_2 \mapsto \langle\langle v'_1, \ldots, v'_m\rangle\rangle\}$. Here, by the typing rule TUPLECONCAT, $n_2 = n_1 + n$. Thus, by the operational semantics of `tuple_concat`, there exists S' such that $S \mapsto S'$.

Theorem A.52 (Type Soundness)
If $\vdash S$ and $S \mapsto^* S'$, then S' is not stuck.

Proof By induction on the length n of the evaluation steps.

- Case $n = 0$
  S' $(= S)$ is not stuck because of the lemma A.51.

- Case $n = k$
  Let S'' be the state after one evaluation step, that is, $S \mapsto S''$. Then,

from the lemma A.50, $\vdash S''$. By the assumption of the induction, S' is not stuck, because $S'' \mapsto^{k-1} S'$.

Appendix B

Basic Lemmas for Proving Type Soundness

B.1 Reflection

Lemma B.1
$\Delta; C \vdash C' = C'$.

Proof By the definition of the relation $\models$, we have $\Delta; C \wedge C' \models C'$. Thus, by the typing rule EQCSTRT, $\Delta; C \vdash C' = C'$.

Lemma B.2
$\Delta; C \vdash \Gamma = \Gamma$.

Proof By induction on the structure of Γ.

- Case $\Gamma \equiv \cdot$
  From the typing rule EQREGSNULL, we have $\Delta; C \vdash \Gamma = \Gamma$.

- Case $\Gamma \equiv \Gamma'\{r : \sigma\}$
  By the induction hypothesis, $\Delta; C \vdash \Gamma' = \Gamma'$. In addition, by the lemma B.3, $\Delta; C \vdash \sigma = \sigma$. Thus, by the typing rule EQREGSREG, we have $\Delta; C \vdash \Gamma = \Gamma$.

Lemma B.3
$\Delta; C \vdash \sigma = \sigma$.

Proof By case analysis of the form of σ.

- Case $\sigma \equiv i$
  By the definition of the relation $\models$, we have $\Delta; C \models i = i$. Thus, $\Delta; C \vdash \sigma = \sigma$.

- Case $\sigma \equiv \forall \Delta'.|C'|[\Sigma'](\Gamma')$
  By the lemma B.1, $\Delta\Delta'; C \vdash C' = C'$. Next, by the lemma B.6, $\Delta\Delta'; C \wedge C' \vdash \Sigma' = \Sigma'$. In addition, by the lemma B.2, $\Delta\Delta'; C \wedge C' \vdash \Gamma' = \Gamma'$. Thus, by the typing rule EQLABEL, we have $\Delta; C \vdash \sigma = \sigma$.

Lemma B.4
$\Delta; C \vdash \tau = \tau$.

Proof By induction on the structure of τ.

- Case $\tau \equiv \langle \sigma_1, \ldots, \sigma_n \rangle$
  By the lemma B.3, $\forall i.\Delta; C \vdash \sigma_1 = \sigma_1$. Thus, by the typing rule EQTUPLE, we have $\Delta; C \vdash \tau = \tau$.

- Case $\tau \equiv \exists \Delta'.|C'|[\Sigma']\tau'$
  By the lemma B.1, $\Delta\Delta'; C \vdash C' = C'$. Next, by the lemma B.6, $\Delta\Delta'; C \wedge C' \vdash \Sigma' = \Sigma'$. In addition, by the induction hypothesis, $\Delta\Delta'; C \wedge C' \vdash \tau' = \tau'$. Thus, by the typing rule EQEX, we have $\Delta; C \vdash \tau = \tau$.

- Case $\tau \equiv \rho\,(c_1, \ldots, c_n)$
  By the typing rule EQREC, we have $\Delta; C \vdash \tau = \tau$.

Lemma B.5
$\Delta; C \vdash at = at$.

Proof Let $at \equiv \tau[i]$. Then, by the lemma B.4 and the definition of the relation $\models$, we have $\Delta; C \vdash \tau = \tau$ and $\Delta; C \models i = i$. Thus, we have $\Delta; C \vdash at = at$.

Lemma B.6
$\Delta; C \vdash \Sigma = \Sigma$.

Proof By induction on the structure of Σ.

- Case $\Sigma \equiv \cdot$
  By the typing rule EQMEMEMPTY, $\Delta; C \vdash \Sigma = \Sigma$.

- Case $\Sigma \equiv \Sigma' \otimes \{i \mapsto at\}$
  By the induction hypothesis, $\Delta; C \vdash \Sigma' = \Sigma'$. Now, by the typing rule EQMEMLOC and the lemma B.5, $\Delta; C \vdash \Sigma' \otimes \{i \mapsto at\} = \Sigma' \otimes \{i \mapsto at\}$. That is, $\Delta; C \vdash \Sigma = \Sigma$.

- Case $\Sigma \equiv \Sigma' \otimes \epsilon$
  By the induction hypothesis, $\Delta; C \vdash \Sigma' = \Sigma'$. Now, by the typing rule EQMEMVAR, $\Delta; C \vdash \Sigma' \otimes \epsilon = \Sigma' \otimes \epsilon$. That is, $\Delta; C \vdash \Sigma = \Sigma$.

B.2 Symmetry

Lemma B.7
If $\Delta; C \vdash \tau = \tau'$, then $\Delta; C \vdash \tau' = \tau$.

Proof By straightforward case analysis on the last rules of the derivation of $\Delta; C \vdash \tau = \tau'$.

Lemma B.8
If $\Delta; C \vdash at = at'$, then $\Delta; C \vdash at' = at$.

Proof Let $at \equiv \tau[i]$ and $at' \equiv \tau'[i']$. Then, by the typing rule, we have $\Delta; C \vdash \tau = \tau'$ and $\Delta; C \models i = i'$. Here, by the lemma B.7 and the definition of the relation $\models$, we have $\Delta; C \vdash \tau' = \tau$ and $\Delta; C \models i' = i$. Thus, $\Delta; C \vdash at' = at$.

Lemma B.9
If $\Delta; C \vdash \Sigma = \Sigma'$, then $\Delta; C \vdash \Sigma' = \Sigma$.

Proof By induction on the derivation of $\Delta; C \vdash \Sigma = \Sigma'$. The proof is by case analysis on the last rule of the derivation.

- Case EQMEMEMPTY
  By the typing rule, we have $\Sigma \equiv \Sigma' \equiv \cdot$. Thus, by the typing rule EQMEMEMPTY, $\Delta; C \vdash \Sigma' = \Sigma$.

- Case EQMEMLOC
  By the typing rule, we have $\Sigma \equiv \Sigma_1 \otimes \{i_1 \mapsto at_1\}$, $\Sigma' \equiv \Sigma_2 \otimes \{i_2 \mapsto at_2\}$, $\Delta; C \vdash \Sigma_1 = \Sigma_2$, $\Delta; C \models i_1 = i_2$ and $\Delta; C \vdash at_1 = at_2$. Here, by the induction hypothesis, we have $\Delta; C \vdash \Sigma_2 = \Sigma_1$. Then, by the definition of the relation $\models$, $\Delta; C \vdash i_2 = i_1$. In addition, by the lemma B.8, $\Delta; C \vdash at_2 = at_1$. Thus, we have $\Delta; C \vdash \Sigma' = \Sigma$.

- Case EQMEMVAR
  By the typing rule, we have $\Sigma \equiv \Sigma_1 \otimes \epsilon$, $\Sigma' \equiv \Sigma_2 \otimes \epsilon$ and $\Delta; C \vdash \Sigma_1 = \Sigma_2$. Here, by the induction hypothesis, $\Delta; C \vdash \Sigma_2 = \Sigma_1$. Now, by the typing rule EQMEMVAR, $\Delta; C \vdash \Sigma_2 \otimes \epsilon = \Sigma_1 \otimes \epsilon$. That is, $\Delta; C \vdash \Sigma' = \Sigma$.

- Case EQMEMZEROARRAYL
  By the typing rule, we have $\Sigma \equiv \Sigma_1 \otimes \{i_1 \mapsto \tau[i_2]\}$, $\Delta; C \vdash \Sigma_1 = \Sigma'$ and $\Delta; C \models i_2 = 0$. Here, by the induction hypothesis, we have $\Delta; C \vdash \Sigma' = \Sigma_1$. Now, by the typing rule EQMEMZEROARRAYR, we have $\Delta; C \vdash \Sigma' = \Sigma$.

- Case EQMEMZEROARRAYR
  Same as Case EQMEMZEROARRAYL.

B.3 Transitivity

Lemma B.10
If $\Delta; C \vdash \tau = \tau'$ and $\Delta; C \vdash \tau' = \tau''$, then $\Delta; C \vdash \tau = \tau''$.

Proof By straightforward case analysis on the last rule of the derivation of $\Delta; C \vdash \tau = \tau'$.

Lemma B.11
If $\Delta; C \vdash at = at'$ and $\Delta; C \vdash at' = at''$, then $\Delta; C \vdash at = at''$.

Proof Let $at \equiv \tau[i]$, $at' \equiv \tau'[i']$ and $at'' \equiv \tau''[i'']$. Then, by the typing rule, we have $\Delta; C \vdash \tau = \tau'$, $\Delta; C \vdash \tau' = \tau''$, $\Delta; C \models i = i'$ and $\Delta; C \models i' = i''$. Here, by the lemma B.10 and the definition of the relation $\models$, we have $\Delta; C \vdash \tau = \tau''$ and $\Delta; C \models i = i''$. Thus, $\Delta; C \vdash at = at''$.

Lemma B.12
If $\Delta; C \vdash \Sigma = \Sigma'$ and $\Delta; C \vdash \Sigma' = \Sigma''$, then $\Delta; C \vdash \Sigma = \Sigma''$.

Proof By induction on the derivation of $\Delta; C \vdash \Sigma = \Sigma'$. The proof is by case analysis on the last rule of the derivation.

- Case EQMEMEMPTY
  By the typing rule, we have $\Sigma \equiv \Sigma' \equiv \cdot$. Thus, we have $\Delta; C \vdash \cdot = \Sigma''$. That is, $\Delta; C \vdash \Sigma = \Sigma''$.

- Case EQMEMLOC
  By the typing rule, we have $\Sigma \equiv \Sigma_1 \otimes \{i_1 \mapsto at_1\}$, $\Sigma' \equiv \Sigma_2 \otimes \{i_2 \mapsto at_2\}$, $\Delta; C \vdash \Sigma_1 = \Sigma_2$, $\Delta; C \models i_1 = i_2$ and $\Delta; C \vdash at_1 = at_2$.

  Here let $at_2 \equiv \tau[j]$. If $\Delta; C \models j = 0$, by the lemma A.47, we have $\Delta; C \vdash \Sigma_2 = \Sigma''$. Now, by the induction hypothesis, $\Delta; C \vdash \Sigma_1 = \Sigma''$. Here, by the typing rule EQMEMZEROARRAYL, we have $\Delta; C \vdash \Sigma = \Sigma''$, because $\Delta; C \models j' = 0$ where $at_1 \equiv \tau'[j']$.

  Otherwise (if $\Delta; C \models j \neq 0$), by the lemma A.48, we have $\Sigma'' \equiv \Sigma_3 \otimes \{i_3 \mapsto at_3\}$, $\Delta; C \vdash \Sigma_2 = \Sigma_3$, $\Delta; C \models i_2 = i_3$ and $\Delta; C \vdash at_2 = at_3$. Here, by the induction hypothesis, $\Delta; C \vdash \Sigma_1 = \Sigma_3$. Next, from the definition of the relation $\models$, we have $\Delta; C \models i_1 = i_3$. Then, by the lemma B.11, $\Delta; C \vdash at_1 = at_3$. Now, by the typing rule EQMEMLOC, we have $\Delta; C \vdash \Sigma_1 \otimes \{i_1 \mapsto at_1\} = \Sigma_3 \otimes \{i_3 \mapsto at_3\}$. That is, $\Delta; C \vdash \Sigma = \Sigma''$.

- Case EQMEMVAR
  By the typing rule, we have $\Sigma \equiv \Sigma_1 \otimes \epsilon$, $\Sigma' \equiv \Sigma_2 \otimes \epsilon$ and $\Delta; C \vdash \Sigma_1 = \Sigma_2$. Here, by the lemma A.49, we have $\Sigma'' \equiv \Sigma_3 \otimes \epsilon$ and $\Delta; C \vdash \Sigma_2 = \Sigma_3$. Now, by the induction hypothesis, $\Delta; C \vdash \Sigma_1 = \Sigma_3$. Then, by the typing rule EQMEMVAR, we have $\Delta; C \vdash \Sigma_1 \otimes \epsilon = \Sigma_3 \otimes \epsilon$. That is, $\Delta; C \vdash \Sigma = \Sigma''$.

- Case EQMEMZEROARRAYL
  By the typing rule, we have $\Sigma \equiv \Sigma_1 \otimes \{i_1 \mapsto \tau[i_2]\}$, $\Delta; C \vdash \Sigma_1 = \Sigma'$ and $\Delta; C \models i_2 = 0$. Here, by the induction hypothesis, we have $\Delta; C \vdash \Sigma_1 = \Sigma''$. Now, by the typing rule EQMEMZEROARRAYL, we have $\Delta; C \vdash \Sigma = \Sigma''$.

- Case EQMEMZEROARRAYR
  By the typing rule, we have $\Sigma' \equiv \Sigma_1 \otimes \{i_1 \mapsto \tau[i_2]\}$, $\Delta; C \vdash \Sigma = \Sigma_1$ and $\Delta; C \models i_2 = 0$. Here, by the lemma A.47, we have $\Delta; C \vdash \Sigma_1 = \Sigma''$. Now, by the induction hypothesis, $\Delta; C \vdash \Sigma = \Sigma''$.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for UNIX development. In *Summer 1986 USENIX Conference*, pages 93–112, 1986.

[2] D. Aspinall and A. Compagnoni. Heap bounded assembly language. *Automated Reasoning*, 31:261–302, 2003.

[3] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming Language Design and Implementation (PLDI '01)*, pages 203–213, 2001.

[4] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177, 2003.

[6] M. Barnett, K. Rustan, M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of the Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS '04)*, 2004.

[7] N. Batlivala, B. Gleeson, J. R. Hamrick, S. Lurndal, D. Price, J. Soddy, and V. Abrossimov. Experience with SVR4 over Chorus. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 223–242. USENIX Association, 1992.

[8] M. Beck, H. Bohme, U. Kunitz, R. Magnus, M. Dziadzka, and D. Verworner. *Linux Kernel Internals*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 267–284, 1995.

[10] G. Betarte, C. Cornes, N. Szasz, and A. Tasistro. Specification of a smart card operating system. In *Types for Proofs and Programs: International Workshop, TYPES '99*, pages 77–93, 1999.

[11] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[12] D. Blei, C. Harrelson, R. Jhala, R. Majumdar, G. C. Necula, S. P. Rahul, W. Weimer, and D. Weitz. The Vampyre theorem prover. `http://www.cs.ucla.edu/~rupak/Vampyre`.

[13] Bochs: the cross platform ia-32 emulator. `http://bochs.sourceforge.net`.

[14] R. Boyer and J. S. Moore. *A Computational Logic Handbook*. Number 23 in Perspectives in Computing. Academic Press, 1988.

[15] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[16] M. I. Bushnell. Towards a new strategy of OS design. Technical report, GNU's Bulletin, Jan. 1994.

[17] E. Chailloux, P. Manoury, and B. Pagano. *Developing Applications with Objective Caml*. O'Reilly France, 2000. `http://caml.inria.fr`.

[18] J. Cheney and G. Morrisett. A linearly typed assembly language. Technical report, Department of Computer Science, Cornell University, 2003.

[19] Sendmail Consortium. sendmail. `http://www.sendmail.org`.

[20] Intel Corporation. EFI: Extensible firmware interface specification. `http://developer.intel.com`.

[21] Intel Corporation. Intel IA-32 architecture software developer's manual. `http://developer.intel.com`.

[22] Intel Corporation. Intel virtualization technology specification for the IA-32 Intel architecture. `http://developer.intel.com`.

[23] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*, pages 262–275, 1999.

[24] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*, pages 59–69, 2001.

[25] Desert Spring-Time. `http://dst.purevoid.org`.

[26] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[27] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation (PLDI '94)*, pages 230–241, 1994.

[28] S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture, Oct. 1998. US Patent, 6397242.

[29] R. K. Dybvig. *The Scheme Programming Language: ANSI Scheme*. Prentice Hall PTR, 1996.

[30] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 251–226, 1995.

[31] M. Fahndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, pages 13–24, 2002.

[32] F. V. Fjeld. Movitz: a common lisp x86 development platform. `http://common-lisp.net/project/movitz/`.

[33] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of the 8th European Symposium on Programming Languages and Systems (ESOP '99)*, pages 91–108, 1999.

[34] The UEFI forum. UEFI: Unified extensible firmware interface specification. `http://www.uefi.org`.

[35] The Apache Software Foundation. Apache HTTP Server. `http://www.apache.org`.

[36] Free Software Foundation. GNU C Library. `http://www.gnu.org/software/libc/`.

[37] Free Software Foundation. GNU GRUB. `http://www.gnu.org/software/grub/`.

[38] Free Software Foundation. Multiboot specification. `http://www.gnu.org/software/grub/manual/multiboot`.

[39] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '96)*, pages 1–15, 1996.

[40] D. B. Golub, R. W. Dean, A. Forin, and R. F. Rashid. UNIX as an application program. In *USENIX Summer*, pages 87–95, 1990.

[41] D. Grossman. Type-safe multithreading in cyclone. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '03)*, pages 13–25, 2003.

[42] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, pages 282–293, 2002.

[43] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in haskell. In *Proceedings of the 10th International Conference on Functional Programming (ICFP '05)*, pages 116–128, 2005.

[44] G. Hamilton and P. Kougiouris. The spring nucleus: A microkernel for objects. Technical Report TR-93-14, Sun Microsystems, April 1993.

[45] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 66–77, 1997.

[46] C. Hawblitzel, E. Wei, H. Huang, E. Krupski, and L. Wittie. Low-level linear memory management. In *Proceedings of the 2nd workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE '04)*, 2004.

[47] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 526–538. Springer, Jul. 2002.

[48] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proceedings of the 10th International SPIN Workshop (SPIN '03)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, May 2003.

[49] M. Hohmuth and H. Hartig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 217–230, 2001.

[50] M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 2nd ECOOP Workshop on Programming Languages and Operating Systems*, 2005.

[51] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[52] G. J. Holzmann and M. H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.

[53] G. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. D. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[54] A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, pages 332–342, 2002.

[55] A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, 2005.

[56] Internet Systems Consortium. Inc. BIND: Berkeley internet name domain. `http://www.isc.org`.

[57] International Organization for Standardization. *ISO/IEC 9899:1990: Programming languages — C*. 1990.

[58] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. 1998.

[59] International Organization for Standardization. *ISO/IEC 23270:2003: C# Language Specification*. 2003.

[60] Java Community Process. JSR 901: Java language specification, March 2004. `http://jcp.org`.

[61] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, 2002.

[62] N. Kobayashi. Quasi-linear types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*, pages 29–42, 1999.

[63] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 237–250, 1995.

[64] M. Lutz. *Programming python*. O'Reilly & Associates, Inc., 1996.

[65] J. Mauro and R. McDougall. *Solaris internals: core kernel architecture*. Sun Microsystems, Inc., 2001.

[66] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960.

[67] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The design and implementation of the 4.4BSD operating system*. Addison Wesley Longman Publishing Co., Inc., 1996.

[68] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1982.

[69] R. Milner. The polyadic pi-calculus: a tutorial. In *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

[70] D. Moon. Genera retrospective. In *Proceedings of the IEEE 1991 International Workshop on Object Orientation in Operating Systems (IWOOOS '91)*, pages 2–8, 1991.

[71] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, 1999.

[72] K. Shirane N. Kobayashi. Type-based information flow analysis for low-level languages. In *Proceedings of the 3rd Asian Workshop on Programming Languages and Systems (APLAS '02)*, 2002.

[73] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, pages 128–139, 2002.

[74] G. Nelson, editor. *System Programming in Modula-3*. Prentice Hall, 1991.

[75] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[76] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (CADE '92)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, Jun 1992.

[77] F. Pfenning and C. Schürmann. System description: Twelf — A metalogical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE '99)*, pages 202–206. Springer-Verlag LNAI 1632, 1999.

[78] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[79] QEMU. `http://fabrice.bellard.free.fr/qemu/`.

[80] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, 2004.

[81] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.

[82] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP '00)*, pages 366–381. Springer-Verlag, 2000.

[83] G. Smith and D. Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(1–3):49–72, 1998.

[84] SourceForge.net. `http://sourceforge.net`.

[85] A. S. Tanenbaum, M. F. Kaashoek, R. Renesse, and H. E. Bal. The Amoeba distributed operating system: a status report. *Computer Communications*, 14:324–335, 1991.

[86] The Coq Development Team. The Coq proof assistant reference manual version 8.0. Technical report, INRIA, 2004. `http://coq.inria.fr/`.

[87] D. Thomas and A. Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley, 2000.

[88] S. Thompson. *Haskell: the Craft of Functional Programming*. Addison-Wesley, March 1999.

[89] M. Tofte and J. P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '94)*, pages 188–201, 1994.

[90] M. Tofte and J. P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[91] TOS project. `http://web.yl.is.s.u-tokyo.ac.jp/~tosh/tos/`.

[92] H. Tuch and G. Klein. Verifying the L4 virtual memory subsystem. In *Proceedings of NICTA Formal Methods Workshop on Operating Systems Verification*, pages 73–97. NICTA Technical Report 0401005T-1, National ICT Australia, 2004.

[93] H. Tuch, G. Klein, and G. Heiser. OS verification — now! In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005.

[94] D. Turner, P. Wadler, and C. Mossion. Once upon a type. In *Proceedings of the ACM International Conference on Functional Programming and Computer Architecture*, 1995.

[95] D. Walker. A type system for expressive security policies. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '00)*, pages 254–267, 2000.

[96] D. Walker and G. Morrisett. Alias types for recursive data structures. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC '00)*, 2000.

[97] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl (3rd Edition)*. O'Reilly, July 2000.

[98] D. C. Wang and A. W. Appel. Type-preserving garbage collectors. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*, pages 166–178, 2001.

[99] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *USENIX Annual Technical Conference*, Jun. 2002.

[100] E. Witchel, J. Cates, and K. Asanovic. Mondriaan memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2002.

[101] H. Xi and R. Harper. A dependently typed assembly language. In *Proceedings of the 6th International Conference on Functional Programming (ICFP '01)*, pages 169–180, 2001.

[102] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '99)*, pages 214–227, January 1999.