Writing practical memory management code with a strictly typed assembly language (Extended Version)

Toshiyuki Maeda
Graduate School of Information Science and Technology, The University of Tokyo
7-3-1, Hongo, Bunkyo-ku,
Tokyo, Japan
tosh@is.s.u-tokyo.ac.jp

Akinori Yonezawa
Graduate School of Information Science and Technology, The University of Tokyo
7-3-1, Hongo, Bunkyo-ku,
Tokyo, Japan
yonezawa@is.s.u-tokyo.ac.jp

ABSTRACT

Memory management (e.g., malloc/free) cannot be implemented in traditional strictly typed programming languages because they do not allow programmers to reuse memory regions, in order to preserve memory safety. Therefore, they depend on external memory management facilities, such as garbage collection. Thus, many important programs that require explicit memory management (e.g., operating systems) have been written in weakly typed programming languages (e.g., C). To address the problem, we designed a new strictly and statically typed assembly language which is flexible and expressive enough to implement memory management. The key idea of our typed assembly language is that it supports variable-length arrays (the arrays whose size is not known until runtime) as language primitives, maintains integer constraints between variables, and keeps track of pointer aliases. Based on the idea, we implemented a prototype implementation of the language. We also implemented a small operating system kernel which provides a memory management facility with the language.

1. INTRODUCTION

Today, computers (PCs, cell-phones, etc.) 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 and many programs come to be written in strictly typed languages (e.g., Java [11], C# [4], Objective Caml [15]).

However, there still exist programs that have not been written in strictly typed languages. For example, existing OSes (e.g., Linux, FreeBSD, Windows XP and Solaris) are written in C and assembly languages. In addition, many Internet servers, for example, Apache httpd server, sendmail mail server and BIND DNS server are written in C. Therefore, many vulnerabilities that may result in a serious security breach (e.g., system crash and/or intrusion) are found in the programs, because it is very hard to ensure and/or verify their safety.

One of the reasons why the programs are written in weakly-typed languages despite the security problem is that existing strictly typed programming languages do not allow programmers to directly manage memory. In simple words, malloc/free cannot be implemented in the languages. For example, Java, C# and Objective Caml depend on an external memory management facility, garbage collection. In the languages, programmers cannot free or reuse memory regions explicitly.

To address the problem, we designed and implemented a strictly and statically typed language which is flexible and expressive enough to implement memory management (malloc/free), and actually implemented memory management code in the language. The language is a variant of Typed Assembly Language (TAL) [14, 13] extended with the support for variable-length arrays (the arrays whose size is not known until runtime) and integer constraints, and the idea of tracking aliases of pointers explicitly [20, 25].

There are two reasons why we chose TAL for writing memory management code. The first reason is that TAL can express low-level memory operations that are necessary for implementing memory management, because TAL is an ordinary assembly language (except for being typed). The second reason is that the TAL programs can be type-checkd at the level of binary executables by annotating the executables with the type information of TAL. This means that we can verify the safety of memory management code with the type-checker without its source code.

The rest of the paper is organized as follows. First, we argue what is required to implement memory management in Sec. 2. Next, we explain our language, called TALK, in Sec. 3. We also describe its type system and type-checking in the section. Next, we explain how memory management can be implemented using TALK in Sec. 4. Then, we briefly describe the prototype implementation of our language and the small toy OS kernel built with the implementation in Sec. 5. Last, we mention related work in Sec. 6 and conclude this paper in Sec. 7.

2. REQUISITES FOR IMPLEMENTING MEMORY MANAGEMENT

2.1 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. 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. 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.

2.2 Integer constraints

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. This contradicts our goal, designing the statically typed programming language that is able to implement memory management without external trusted memory management facilities.

To avoid the problem, we need to introduce the idea of dependent type [27, 26] to the type system. For example, the type of a variable-length array of integers can be represented as `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 (more specifically, singleton types), 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 that 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` with the singleton types. For example, the type of `x` must be `int`(β), instead of `int`. Here, 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. Such 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.

2.3 Alias tracking

The previous section 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. Fig. 1 is an example C code which performs this *memory reuse*. The function reuses the memory region pointed by pointer `x` (line 3).

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

Figure 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, because memory safety cannot be ensured. For example, using the function in Fig. 1, we can write a function as in Fig. 2. 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 (line 4). Thus, 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.

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

Figure 2: Example of C code that breaches memory safety

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

To solve the problem, we need to introduce the idea of alias types [25] to the type system in order to keep track of aliases explicitly. The basic idea of the alias types 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 alias types, the code in Fig. 1 can be rewritten as in Fig. 3. First, the type of the argument `x` is changed from `int**` to `ptr(p)` (line 2). `ptr(p)` indicates that `x` is a pointer which points to the address `p`. Next, the part 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 (line 1). The other memory type added after the function indicates the state of the memory after the function is executed (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 body of the function stores an integer to the memory region at the address p (line 4). Therefore, the memory type after the function indicates that the memory region at the address p is an integer. (Please note that the alias type system ensures that p and q are different integers, because the memory type is a map.)

```
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 3: Example of pseudo code based on the idea of alias type

In addition, the code of Fig. 2 can be rewritten as Fig. 4. The type check of the alias type system rejects the rewritten code at line 5, because after the function call (`pointer_to_int`, line 4), the type of the memory region is changed from a pointer type (`ptr(q)`) to the integer 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 4: Example of pseudo code that causes a type error

2.4 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 types. 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 `int` $[a]$ (here we assume that $a > 0$). Now, let us think of the code in Fig. 5 that allocates one element from the top of the free memory and reuses it as a pointer to integer (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 need a notion of split (and concatenation) of arrays in 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

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

of size `a2` at address `p2`, where `a` $=$ `a1` $+$ `a2` and `p2` $=$ `p` $+$ `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 Fig. 5 can be rewritten as in Fig. 6. The `split` operation (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 6: Example of pseudo code that allocates a pointer to integer from free memory (complete)

3. OUR LANGUAGE: TALK

This section introduces our strictly typed assembly language for writing memory management code. Although the language explained in this section is based on a virtual CPU architecture, the actual implementation is based on the IA-32 [10] assembly language. In this section, we first explain its abstract machine and types. Then, its operational semantics and typing rules are described. The syntax of our language is shown in Fig. 7 and Fig.8.

3.1 Abstract Machine

A state of the abstract machine consists of program P, memory M, registers R and instructions I. The instructions I is explained in Sec. 3.3. The program P is a map from label l to the instructions I. The registers R is a map from register r to value v.

The memory M is a map from an integer n to heap value h. The heap value h is array a or stack s. The array a consists of tuples t, and the tuples consist of values v. (`roll` (t) and $\texttt{pack}_{[c_1,\ldots,c_n|M]}\,(t)$ are introduced only for formal arguments of recursive types and existential types.) The value v is integer n or the label l. (The suffix of the label l $([c_1,\ldots,c_n/\Delta])$is required only by type checkers. It has no meaning at runtime.) The stack s consists of the values v. Strictly speaking, the stacks are not necessary because they can be represented by the arrays. However, in this paper, we introduce the stacks for ease of understating examples. In addition, we assume that the stacks are unbounded in

this paper. The actual implementation handles the bounded stacks properly.

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

Figure 7: Syntax of Abstract Machine

3.2 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 consists of maps from the integer type i to heap type ht, and type variables ϵ. For example, $\{\mathtt{0x}c0345810 \mapsto ht\}$ represents memory in which there is only one heap value of the type ht at the address $\mathtt{0x}c0345810$. On the other hand, $\{\mathtt{0x}c0345810 \mapsto ht\} \otimes \epsilon$ represents memory in which there is a heap value of the type ht at the address $\mathtt{0x}c0345810$ and there may be some other data (or not). The type variable ϵ indicates that there may be some other data in the memory.

The heap type ht is array type at or stack type st. 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 the variable-length arrays.

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 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 stack type st consists of null stack type $(\cdot)$, a pair of the small value type σ which represents the top of the stack and the stack type st which represents the rest of the stack, and type variable γ. For example, $42 :: \gamma$ represents the type of a stack whose top element is an integer value (42) and the rest is unknown (γ).

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 is 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 our 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 our 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, $\forall\alpha, \beta, \epsilon.\,|\beta \leq 128|\,[\epsilon \otimes \{\alpha \mapsto \langle 0 \rangle\,[\beta]\}]\,(\mathtt{r1} : \alpha)$ represents instructions that take a pointer (register $\mathtt{r1}$) 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, $\exists\alpha, \beta.\,|\beta \leq 128|\,[\{\alpha \mapsto \langle 0 \rangle\,[\beta]\}]\,\langle\alpha\rangle$ 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.

In addition, the actual implementation of our language supports a variant type, but we do not explain it in this paper for brevity.

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

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

Figure 8: Syntax of Types

3.3 Instructions and Operational Semantics

There are two kinds of instructions in our language. 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. Fig. 9 and Fig. 10 represent their operational semantics. (In this paper, $e\,[b/a]$ 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 $e\,[b_1/a_1, b_2/a_2]$.)

3.3.1 Ordinary Instructions

There are 12 ordinary instructions in our language. `ld` $[r_s + n], r_d$ is a 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 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 our language, there is no reference types or pointer types. Memory addresses are only integers. Therefore, the address calculation are performed with the arithmetic instructions.

`push` $r_s, [r_d]$ is an instruction for manipulating memory stacks. It 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. `pop` $[r_s], r_d$ is the other instruction for manipulating memory stacks. It 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.

`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.

3.3.2 Coerce Instructions

There are 9 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 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), a stack type variable (γ) or a memory type variable (ϵ).

`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`).

`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 [25], 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 array types. `split` splits an array type into two adjacent array types and `concat` concatenates two adjacent array types into one array type. These instructions are used to access an element of an array (see Sec. 3.4.4 for details). In addition, they are useful for implementing memory management facilities (see Sec. 4 for details).

`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 our language, 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 example of Sec. 4 for details)

3.4 Typing Rules

Typing rules are shown in Fig. 11, Fig. 12 and Fig. 13. $\vdash S$ states that the abstract machine state S is well-formed. If (P, M, R, I) is well-formed, there exists a certain state (P, M', R', I') which is also well-formed and $(P, M, R, I) \mapsto_S (P, M', R', I')$, that is, no runtime errors occur. Strictly speaking, the above statement holds true if all the stacks in memory never conflict with other heap values. This is because the language presented in this paper assumes that the stacks can grow infinitely, but this is impossible in practice. Thus, we can prove the above statement except for the rules for the stacks. This is only a minor matter, because the stacks are not necessary as mentioned in Sec. 3.1

Our type system is based on the one proposed in [25]. Although Walker and Morrisett proved type soundness in [25], their language falls short of a practical language. This is why, for the purpose of writing practical memory management code, we essentially added a variable-length array type to their type system. We claim that those changes are unlikely to break type soundness, because, in TALK, the strong updates that may change types of memory regions are only for tuples (the arrays of size 1) as described in Sec. 3.4.4. This is almost the same as in the type system of [25]. The only concern is the effect of split/concatenation of variable-length arrays, but the typing rules for them are quite straightforward.

The judgement of abstract machine states consists of the judgement of program, memory, registers and instructions.

3.4.1 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 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 Sec. 3.4.4.

3.4.2 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.

3.4.3 Well-formedness of memory

$\vdash M : \Sigma$ states that the memory M has the memory type Σ (MEMORY). The judgement rule checks whether

$$
\begin{array}{lcl}
(P, M\{R(r_s) \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\}, R, \texttt{ld}\ [r_s + n'], r_d; I) & \mapsto_S & (P, M\{R(r_s) \mapsto \langle\langle v_1, \ldots, v_n\rangle\rangle\}, R\{r_d \mapsto v_{n'}\}, I) \\
(P, M\{R(r_d) \mapsto \langle\langle \ldots, v_{n'}, \ldots\rangle\rangle\}, R, \texttt{st}\ r_s, [r_d + n']; I) & \mapsto_S & (P, M\{R(r_d) \mapsto \langle\langle \ldots, R(r_s), \ldots\rangle\rangle\}, R, I) \\
(P, M, R, \texttt{mov}\ r_s, r_d; I) & \mapsto_S & (P, M, R\{r_d \mapsto R(r_s)\}, I) \\
(P, M, R, \texttt{movi}\ v, r_d; I) & \mapsto_S & (P, M, R\{r_d \mapsto v\}, I) \\
(P, M, R, \texttt{add}\ r_{s1}, r_{s2}, r_d; I) & \mapsto_S & (P, M, R\{r_d \mapsto R(r_{s2}) + R(r_{s1})\}, I) \\
(P, M, R, \texttt{sub}\ r_{s1}, r_{s2}, r_d; I) & \mapsto_S & (P, M, R\{r_d \mapsto R(r_{s2}) - R(r_{s1})\}, I) \\
(P, M, R, \texttt{mul}\ r_{s1}, r_{s2}, r_d; I) & \mapsto_S & (P, M, R\{r_d \mapsto R(r_{s2}) * R(r_{s1})\}, I) \\
(P, M\{R(r_d) \mapsto s\}, R, \texttt{push}\ r_s, [r_d]; I) & \mapsto_S & (P, M\{R(r_d) - 1 \mapsto R(r_s) :: s\}, R\{r_d \mapsto R(r_d) - 1\}, I) \\
(P, M\{R(r_s) \mapsto v :: s\}, R, \texttt{pop}\ [r_s], r_d; I) & \mapsto_S & (P, M\{R(r_s) + 1 \mapsto s\}, R\{r_d \mapsto v\}\{r_s \mapsto R(r_s) + 1\}, I) \\
(P, M, R, \texttt{beq}\ r_{s1}, r_{s2}, r_d; I) & \mapsto_S & \textit{if}\ R(r_{s1}) = R(r_{s2})\ \textit{then}\ (P, M, R, P(l)\,[c_1, \ldots, c_n/\Delta]) \\
 & & \textit{else}\ (P, M, R, I) \\
 & & \textit{where}\ R(r_d) = l\,[c_1, \ldots, c_n/\Delta] \\
(P, M, R, \texttt{ble}\ r_{s1}, r_{s2}, r_d; I) & \mapsto_S & \textit{if}\ R(r_{s1}) \leq R(r_{s2})\ \textit{then}\ (P, M, R, P(l)\,[c_1, \ldots, c_n/\Delta]) \\
 & & \textit{else}\ (P, M, R, I) \\
 & & \textit{where}\ R(r_d) = l\,[c_1, \ldots, c_n/\Delta] \\
(P, M, R, \texttt{jmp}\ r_d) & \mapsto_S & (P, M, R, P(l)\,[c_1, \ldots, c_n/\Delta]) \\
 & & \textit{where}\ R(r_d) = l\,[c_1, \ldots, c_n/\Delta]
\end{array}
$$

Figure 9: Operational Semantics (instructions)

$$
\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\}, R, \texttt{unroll}\ n; I) & \mapsto_S & (P, M\{n \mapsto \langle t\rangle\}, R, I) \\
\Big(P, M\{n \mapsto \langle t\rangle\}M', R, \texttt{pack}_{[c_1,\ldots,c_n|\Sigma]\,as\ \tau}\ n; I\Big) & \mapsto_S & \Big(P, M\{n \mapsto \Big\langle \texttt{pack}_{[c_1,\ldots,c_n|M']}\,(t)\Big\rangle\}, R, I\Big) \\
 & & \textit{where}\ Dom(M') \subseteq Dom(\Sigma) \\
(P, M\{n \mapsto \Big\langle \texttt{pack}_{[c_1,\ldots,c_n|M']}\,(t)\Big\rangle\}, R, & & \\
\qquad \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\}, R, \texttt{split}\ n_1, n_2; I) & \mapsto_S & (P, M\{n_1 \mapsto \langle t_1, \ldots, t_{n_2}\rangle\}\{n'_1 \mapsto \langle t_{n_2+1}, \ldots, t_n\rangle\}, R, I) \\
 & & \textit{where}\ 0 < n_2 < n\ \textit{and}\ n'_1 = n_1 + \textstyle\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\}, 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\}\{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 + \textstyle\sum_{i=1}^{n} \textit{sizeof}\ (t_i) \\
(P, M\{n_1 \mapsto \langle t_1, \ldots, t_n\rangle\}, 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 + \textstyle\sum_{i=1}^{n} \textit{sizeof}\ (t_i) \\
(P, M\{n_1 \mapsto \langle t_1, \ldots, t_m\rangle\}, 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\}, R, \texttt{tuple\_split}\ n_1, n_2; I) & \mapsto_S & (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) \\
 & & \textit{where}\ n'_1 = n_1 + n_2 \\
(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, & & \\
\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 10: Operational Semantics (coerce)

the heap value $M(n)$ has the heap type $\Sigma(n)$ for each address $n \in Dom(M)$. In addition, the rule states that all the heap values in the memory (including encapsulated memory regions inside existential packages) do not overlap each other (which is denoted as $\texttt{GU}(M)$) in order to keep track of pointer aliases properly.

$\Delta; C \vdash a : at$ 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 $\Delta; C \models i = 2$. We do not show the rules for $\Delta; C \models C'$ (read as C' can be deduced from C) in this paper. It is well-known that the problem of integer constraints is decidable if the constraints are linear. The only instruction that may introduce a non-linear constraint is the `mul` instruction.

$\Delta; C \vdash s : st$ 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.

$\Delta; C \vdash t : \tau$ states that the element t of an array has the type τ. 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 TUPLE_PACK is for existential types and the typing rule TUPLE_ROLL is for recursive types.

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

3.4.4 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.

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\forall \Delta'.\, |C'|\,[\Sigma']\,(\Gamma') \equiv \Phi\,(l) \qquad \theta \equiv [c_1, \ldots, c_n/\Delta''] \\ C'' \equiv C'\theta \qquad \Sigma'' \equiv \Sigma'\theta \qquad \Gamma'' \equiv \Gamma'\theta \\ \Delta; C \vdash \sigma = \forall \Delta' \backslash \Delta''.\, |C''|\,[\Sigma'']\,(\Gamma'')\end{array}}{\Delta; C \vdash l\,[c_1, \ldots, c_n/\Delta''] : \forall \Delta.|C_1|[\Sigma_1](\Gamma_1)} \text{ (VALUE\_LABEL)}$$

Figure 11: Typing rules (machine state)

Because LOAD and STORE only permit load/store operations for arrays whose size is 1, 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 Sec. 2.4).

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 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 PUSH type-checks the `push` instruction. First, it checks whether 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 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.

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, $\alpha; \cdot \vdash \{\texttt{r1} \mapsto \alpha\} \leq \{\texttt{r1} \mapsto \alpha\}$ and $\alpha; \cdot \vdash \{\texttt{r1} \mapsto \alpha, \texttt{r2} \mapsto 42\} \leq \{\texttt{r1} \mapsto \alpha\}$, but $\alpha; \cdot \nvdash \{\texttt{r1} \mapsto \alpha\} \leq \{\texttt{r1} \mapsto \alpha, \texttt{r2} \mapsto 42\}$. 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 can 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 our language. 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 our language 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 our language 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.

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.

The typing rule 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.

The typing rule 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.

The typing rule 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

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

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

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

$$\frac{\Delta; C \vdash v : \sigma \qquad \Delta; \Gamma\{r_d \mapsto \sigma\}; C; \Sigma \vdash I}{\Delta; \Gamma; C; \Sigma \vdash \texttt{movi}\ v, r_d; I}\ (\textsc{movei})$$

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

$$\frac{\Delta; C \vdash \Sigma = \Sigma' \otimes \{\Gamma\,(r_d) \mapsto st\} \quad \Delta; \Gamma\{r_d \mapsto \Gamma\,(r_d) - 1\}; C; \Sigma' \otimes \{\Gamma\,(r_d) - 1 \mapsto \Gamma\,(r_s) :: st\} \vdash I}{\Delta; \Gamma; C; \Sigma \vdash \texttt{push}\ r_s, [r_d]; I}\ (\textsc{push})$$

$$\frac{\Delta; C \vdash \Sigma = \Sigma' \otimes \{\Gamma\,(r_s) \mapsto \sigma :: st\} \quad \Delta; \Gamma\{r_s \mapsto \Gamma\,(r_s) + 1\}\{r_d \mapsto \sigma\}; C; \Sigma' \otimes \{\Gamma\,(r_s) + 1 \mapsto st\} \vdash I}{\Delta; \Gamma; C; \Sigma \vdash \texttt{pop}\ [r_s], r_d; I}\ (\textsc{pop})$$

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

$$\frac{\Delta; C \vdash \Gamma\,(r_d) = \forall.\, |C'|\,[\Sigma']\,(\Gamma') \qquad \Delta; C \models C' \qquad \Delta; C \vdash \Sigma = \Sigma' \qquad \Delta; C \vdash \Gamma \leq \Gamma'}{\Delta; \Gamma; C; \Sigma \vdash \texttt{jmp}\ r_d}\ (\textsc{jump})$$

Figure 12: Typing rules (instructions)

type with it. Here $sizeof(\tau)$ is the size of the tuple represented by the type τ. If τ is a recursive type $(\mu \ldots \tau')$ or an existential type $(\exists \ldots \tau')$, $sizeof(\tau)$ is recursively applied to the inner tuple type τ'. Because $sizeof(\tau)$ is always a constant integer, the rule SPLIT and CONCAT never generate non-linear constraints. Please 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.

The typing rule TUPLE_SPLIT and TUPLE_CONCAT are almost the same as SPLIT and CONCAT, but they typecheck the split and concatenation of tuples.

4. MEMORY MANAGEMENT WITH TALK

In this section, we show simple memory management code which is written in TALK. Although its algorithm is simple and naive, we believe that it is sufficient to show the flexibility and expressiveness of TALK.

4.1 Representation of the free memory

Fig. 14 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 the second element of the tuple. The first element of the tuple is a pointer to the next element of the list. The one argument for the recursive type represents the address of the tuple itself. Therefore, the integer constraint inside the existential type (line 2) indicates that the tuple and the variable-length array are adjacent. In addition, the memory type inside the existential type (line 3) indicates that there surely exists a memory region which satisfies the memory type. Strictly speaking, the definition of the list in Fig. 14 represents an infinite list because the definition does not include any list terminator. Therefore, it might be unrealistic because the free memory is finite. To define finite lists, the type system of TALK supports variant types (or union types), but we do not explain them in this paper for clarity.

4.2 Implementation of malloc

Fig. 15 is a simple implementation of malloc. For clarity, the syntax of instructions are slightly extended. In addition, the `apply` instruction and the argument for the `pack`, `unpack` and `roll` instructions are omitted for clarity.

The label type of `malloc` indicates that the function takes a free memory ($FreeMem(\alpha_{free})$ at line 2) as an argument and returns an array of the specified size (α_{size}) at line 3). The type of the allocated array is specified at line 61. Please note that the return type of the function is abbreviated as ret_t.

The function first checks whether the array of first element of the given free memory list satisfies the requested size (line 11). If so, the function jumps to `malloc_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 (line 13 and 14). Then, it calls itself recursively (from line 16 to 21). After the return from the recursive call (the instructions of the label `malloc_cont`), it concatenates the saved element with the returned free memory

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

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

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

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

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

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

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

$$\frac{\Delta; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n \rangle\} \quad \Delta; C \models 0 < n_2 < n \quad \Delta; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_{n_2} \rangle\} \otimes \{i_1 + n_2 \mapsto \langle \sigma_{n_2+1}, \ldots, \sigma_n \rangle\} \vdash I}{\Delta; \Gamma; C; \Sigma \vdash \texttt{tuple\_split}\ i_1, n_2; I} \text{(TUPLE\_SPLIT)}$$

$$\frac{\Delta; C \vdash \Sigma = \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n \rangle\} \otimes \{i_2 \mapsto \langle \sigma'_1, \ldots, \sigma'_m \rangle\} \quad \Delta; C \models i_2 = i_1 + n \quad \Delta; \Gamma; C; \Sigma' \otimes \{i_1 \mapsto \langle \sigma_1, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_m \rangle\} \vdash I}{\Delta; \Gamma; C; \Sigma \vdash \texttt{tuple\_concat}\ i_1, i_2; I} \text{(TUPLE\_CONCAT)}$$

Figure 13: Typing rules (coerce instructions)

$$\begin{array}{l} \mathit{FreeMem} \equiv \\ \quad \mu\eta\,[\alpha_{self}].\exists\,\alpha_{next}, \alpha_{size}, \alpha_{mem}.\ |\alpha_{mem} = \alpha_{self} + 2| \\ \quad\quad [\{\alpha_{mem} \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size}]\} \otimes \{\alpha_{next} \mapsto \eta\,(\alpha_{next})\}] \\ \quad\quad \langle \alpha_{next}, \alpha_{size} \rangle \end{array}$$

Figure 14: Type of the free memory (list of variable-length arrays)

```
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  malloc:
5          unroll α_free
6          unpack α_free
7          ld [r2 + 1], r5
8          # try the first element
9          movi malloc_success, r10
10         apply r10
11         ble r1, r5, r10
12         # not enough, try next
13         push r2, [r4] # save local vars.
14         push r3, [r4]
15         # recursive call
16         ld [r2], r2
17         movi malloc_cont, r3
18         apply r3
19         movi malloc, r10
20         apply r10
21         jmp r10

22 ∀ α_size, α_tag, α_stk, α_junk, α_mem, α'_size, α, α_free, γ, ϵ.
23 |α_mem = α_tag + 2|
24 [{α_tag ↦ ⟨α_junk, α'_size⟩} ⊗ {α_mem ↦ ∃ β. ⟨β⟩ [α'_size]}⊗
25   {α ↦ ∃ β. ⟨β⟩ [α_size]} ⊗ {α_free ↦ FreeMem(α_free)}
26   {α_stk − 2 ↦ ret_t :: α_tag :: γ} ⊗ ϵ]
27     (r1 : α, r2 : α_free, r4 : α_stk − 2)
28 malloc_cont:
29         pop [r4], r3 # restore local vars.
30         pop [r4], r5
31         # link the previous element
32         # to the returned free memory list
33         st r2, [r5]
34         mov r5, r2
35         pack α_tag
36         roll α_tag
37         apply r3 [α, α_tag/α, α_free]
38         jmp r3

39 ∀ α_size, α_tag, α_stk, α_free, α_mem, α'_size, γ, ϵ.
40 |α_size ≤ α'_size ∧ α_mem = α_tag + 2|
41 [{α_tag ↦ ⟨α_free, α'_size⟩} ⊗ {α_mem ↦ ∃ β. ⟨β⟩ [α'_size]}⊗
42   {α_free ↦ FreeMem(α_free)} ⊗ {α_stk ↦ γ} ⊗ ϵ]
43     (r1 : α_size, r2 : α_tag, r3 : ret_t, r4 : α_stk)
44 malloc_success:
45         # allocate memory by split
46         # from the end of the array
47         split α_mem, (α'_size − α_size)
48         # rewrite the tag
49         ld [r2 + 1], r5
50         sub r1, r5, r6
51         st r6, [r2 + 1]
52         # set the address of
53         # the allocated memory to r1
54         mov r2, r5
55         add 2, r5, r5
56         add r6, r5, r1
57         pack α_tag
58         roll α_tag
59         apply r3[α_mem + α'_size − α_size, α_tag/α, α_free]
60         jmp r3

61 ret_t ≡ ∀ α, α_free. | · |[{α ↦ ∃β. ⟨β⟩ [α_size]}⊗
62         {α_free ↦ FreeMem(α_free)} ⊗ {α_stk ↦ γ} ⊗ ϵ]
63           (r1 : α, r2 : α_free, r4 : α_stk)
```

Figure 15: Simple malloc implementation in TALK

list (line 33) and returns it as a new free memory list (from line 34 to 38) through the register `r2`. Of course, the array allocated by the recursive call is also returned through the register `r1`. Here the stack type $\{\alpha_{stk} - 2 \mapsto ret\_t :: \alpha_{tag} :: \gamma\}$ in the memory type (line 26) represents a stack whose top element has the type ret_t and the next element has type α_{tag} and the rest is unknown (γ).

The code of `malloc_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 (line 47). 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` (line 40). Then, it rewrites the information about the unused array and its size in the second element (from line 43-45) and returns the allocated array (from line 54 to 60).

4.3 Implementation of free

Fig. 16 is a simple implementation of free. First, the code converts the first two elements of the array to be freed into a tuple (from line 9 to 13). Then, the code concatenates the tuple to the given free memory list along with the rest of the array (from line 15 to 19). The label type of `free` indicates that the freed array cannot be used any more because the array is deleted from the memory type after the function return (line 5).

```
1  ∀ α_mem, α_free, α_size, ϵ. |α_size > 2|
2  [{α_mem ↦ ∃ β. ⟨β⟩ [α_size]}⊗
3    {α_free ↦ FreeMem(α_free)} ⊗ ϵ]
4     (r1 : α_mem, r2 : α_free, r4 : α_size,
5      r3 : ∀ α.| · |[{α ↦ FreeMem(α)} ⊗ ϵ](r1 : α))
6  free:
7        # create a tag from
8        # the memory to be freed
9        split α_mem, 2
10       split α_mem, 1
11       unpack α_mem
12       unpack α_mem + 1
13       tuple_concat α_mem, α_mem + 1
14       # initialize the tag
15       sub 2, r4, r4
16       st r4, [r1 + 1]
17       # link the tag to
18       # the free memory list
19       st r2, [r1]
20       pack α_mem
21       roll α_mem
22       apply r3[α_mem/α]
23       jmp r3
```

Figure 16: Simple free implementation in TALK

5. IMPLEMENTATION

We implemented a TALK assembler and a TALK type checker for the IA-32 [10] 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 [16].

Using the TALK assembler, we implemented a prototype OS kernel in TALK[1]. The kernel provides a memory management facility, a multi-thread management facility and a very basic device control facility. For booting the kernel, the GNU GRUB boot loader [8] is used. In addition, some peculiar boot procedures (e.g., segment preparation) of IA-32 are not typed. 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 (3GHz) machine.

The TALK assembler, the TALK type checker and the prototype OS kernel are available from our web site [23].

6. RELATED WORK

Linear type systems [24] 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 [2, 1]. 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 [20, 25] 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 Sec. 3, our TALK is based on the alias types and extended to support variable-length arrays and integer constraints. Thus, we are able to implement practical memory management in TALK.

Hawblitzel et al. [9] extends the alias type system for implementing flexible memory management. The similarity between our approach 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 our type system because it directly supports the variable-length arrays as language primitives. Thus, our type system is simpler than theirs and yet powerful enough to implement memory management code.

DTAL [26] is a typed assembly language extended with the dependent type. As our type system, DTAL also introduced 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-

[1]'K' of TALK stands for 'Kernel'

check array bound-checking.

In region-based memory management [21, 22, 7], 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 [3] 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 [5, 6, 19] 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 a data structure can be deallocated safely, but programmers cannot reuse the data structure explicitly.

As for verifying the correctness of existing memory management programs, Marti et al. [12] proved the correctness of the heap manager of the Topsy operating system [18] using separation logic [17].

7. CONCLUSION

We designed and implemented a new strictly and statically typed assembly language (TALK) which is powerful enough to implement practical memory management (e.g., malloc/free). The type system of our TALK supports variable-length arrays as language primitives. Therefore, our TALK is able to efficiently handle free memory of systems whose size is not known until runtime. In addition, The type system of our TALK keep track of aliases of pointers explicitly. Therefore, programmers are able to reuse memory regions safely because the type system allows them to change the types of the regions. We implemented the assembler and the type checker of our TALK for IA-32. We also implemented a prototype OS kernel for the IA-32 architecture in TALK. The kernel provides memory management facilities and a multi-thread management facilities.

8. REFERENCES

[1] D. Aspinall and A. Compagnoni. Heap bounded assembly language. *Automated Reasoning*, 31:261–302, 2003.

[2] J. Cheney and G. Morrisett. A linearly typed assembly language. Technical report, Department of Computer Science, Cornell University, 2003.

[3] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, 1999.

[4] C#. http://msdn.microsoft.com/net/ecma.

[5] 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*, pages 230–241, 1994.

[6] 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*, pages 1–15, 1996.

[7] 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*, pages 282–293, 2002.

[8] GNU GRUB. http://www.gnu.org/software/grub/.

[9] C. Hawblitzel, E. Wei, H. Huang, E. Krupski, and L. Wittie. Low-level linear memory management. In *SPACE 2004*, 2004.

[10] IA-32 Intel Architecture. http://developer.intel.com.

[11] Java. http://java.sun.com.

[12] N. Marti, R. Affedlt, and A. Yonezawa. Verification of the heap manager of an operating system using separation logic. In *SPACE 2006*, Jan. 2006.

[13] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, 1999.

[14] 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.

[15] Objective Caml. http://caml.inria.fr.

[16] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[17] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[18] L. Ruf, C. Jeker, B. Lutz, and B. Plattner. Topsy v3: A nodeos for network processors. In *ANTA 2003*, 2003.

[19] 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.

[20] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 366–381. Springer-Verlag, 2000.

[21] 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*, pages 188–201, 1994.

[22] M. Tofte and J. P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[23] TOS project. http://web.yl.is.s.u-tokyo.ac.jp/~tosh/tos/.

[24] D. Turner, P. Wadler, and C. Mossion. Once upon a type. In *ACM International Conference on Functional Programming and Computer Architecture*, 1995.

[25] D. Walker and G. Morrisett. Alias types for recursive data structures. In *Types in Compilation*, 2000.

[26] H. Xi and R. Harper. A dependently typed assembly language. In *ICFP*, 2001.

[27] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, January 1999.

APPENDIX

A. TYPING DERIVATION OF MALLOC

The typing derivation of the code in Fig. 15 is as follows. First, the type of the initial memory is

$$\{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon.$$

Next, after the `unroll` and `unpack` instructions (line 5 and 6), the memory type becomes

$$\{\alpha_{free} \mapsto \langle \alpha_{next}, \alpha'_{size} \rangle\} \otimes \{\alpha_{mem} \mapsto \exists \beta.\, \langle \beta \rangle\, [\alpha'_{size}]\} \otimes \{\alpha_{next} \mapsto \mathit{FreeMem}(\alpha_{next})\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon,$$

where $\alpha_{mem} = \alpha_{free} + 2$. Here the argument for the `unpack` instruction is `with` $\alpha_{next}, \alpha'_{size}, \alpha_{mem}$. Then, after the `ld` instruction at line 7, the type of the register `r5` is α'_{size}. Thus, the `ble` instruction at line 11 checks whether $\alpha_{size} \leq \alpha'_{size}$ or not. Then, the argument for the `apply` instruction at line 10 is

$$[\alpha_{size}, \alpha_{free}, \alpha_{stk}, \alpha_{next}, \alpha_{mem}, \alpha'_{size}, \gamma, \epsilon / \alpha_{size}, \alpha_{tag}, \alpha_{stk}, \alpha_{free}, \alpha_{mem}, \alpha'_{size}, \gamma, \epsilon].$$

Accordingly, the type of the register `r10` becomes

$$\forall. |\alpha_{size} \leq \alpha'_{size} \wedge \alpha_{mem} = \alpha_{free} + 2| \; [\{\alpha_{free} \mapsto \langle \alpha_{next}, \alpha'_{size} \rangle\} \otimes \{\alpha_{mem} \mapsto \exists \beta.\, \langle \beta \rangle\, [\alpha'_{size}]\} \otimes \{\alpha_{next} \mapsto \mathit{FreeMem}(\alpha_{next})\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon] \; (\mathtt{r1} : \alpha_{size}, \mathtt{r2} : \alpha_{free}, \mathtt{r3} : \mathit{ret\_t}, \mathtt{r4} : \alpha_{stk}).$$

Here the memory type, the registers type and the integer constraints satisfy the precondition specified by the above label type, because the `ble` instruction adds a new integer constraint ($\alpha_{size} \leq \alpha'_{size}$). Thus, the `ble` instruction at line 11 is type checked successfully.

Next, after the two `push` instructions (line 12 and 13), the memory type becomes

$$\{\alpha_{next} \mapsto \mathit{FreeMem}(\alpha_{next})\} \otimes \{\alpha_{stk} - 2 \mapsto \mathit{ret\_t} :: \alpha_{free} :: \gamma\} \otimes \epsilon',$$

where

$$\epsilon' \equiv \{\alpha_{free} \mapsto \langle \alpha_{next}, \alpha'_{size} \rangle\} \otimes \{\alpha_{mem} \mapsto \exists \beta.\, \langle \beta \rangle\, [\alpha'_{size}]\} \otimes \epsilon.$$

In addition, the type of the register `r4` becomes $\alpha_{stk} - 2$. Then, after the `ld` instruction at line 16, the type of the register `r2` becomes α_{next}. Here the argument for the `apply` instruction at line 18 is

$$[\alpha_{size}, \alpha_{free}, \alpha_{stk}, \alpha_{next}, \alpha_{mem}, \alpha'_{size}, \gamma, \epsilon / \alpha_{size}, \alpha_{tag}, \alpha_{stk}, \alpha_{junk}, \alpha_{mem}, \alpha'_{size}, \gamma, \epsilon].$$

Then, the type of the register `r3` becomes

$$\forall \alpha, \alpha'_{free}. |\alpha_{mem} = \alpha_{free} + 2| \; [\{\alpha \mapsto \exists \beta.\, \langle \beta \rangle\, [\alpha_{size}]\} \otimes \{\alpha'_{free} \mapsto \mathit{FreeMem}(\alpha'_{free})\} \otimes \{\alpha_{stk} - 2 \mapsto \mathit{ret\_t} :: \alpha_{free} :: \gamma\} \otimes \epsilon'] \; (\mathtt{r1} : \alpha, \mathtt{r2} : \alpha'_{free}, \mathtt{r4} : \alpha_{stk} - 2).$$

(Here the bound integer variable α_{free} is renamed to α'_{free}.) In addition, the argument for the `apply` instruction at line 20 is

$$[\alpha_{size}, \alpha_{next}, (\alpha_{stk} - 2), (\mathit{ret\_t} :: \alpha_{free} :: \gamma), \epsilon' / \alpha_{size}, \alpha_{free}, \alpha_{stk}, \gamma, \epsilon].$$

Then, the type of the register `r10` becomes

$$\forall. |\cdot| \; [\{\alpha_{next} \mapsto \mathit{FreeMem}(\alpha_{next})\} \otimes \{\alpha_{stk} - 2 \mapsto \mathit{ret\_t} :: \alpha_{free} :: \gamma\} \otimes \epsilon'] \; (\mathtt{r1} : \alpha_{size}, \mathtt{r2} : \alpha_{next}, \mathtt{r3} : \mathit{ret\_t'}, \mathtt{r4} : \alpha_{stk} - 2),$$

where

$$\mathit{ret\_t'} \equiv \forall \alpha, \alpha'_{free}. |\cdot| \; [\{\alpha \mapsto \exists \beta.\, \langle \beta \rangle\, [\alpha_{size}]\} \otimes \{\alpha'_{free} \mapsto \mathit{FreeMem}(\alpha'_{free})\} \otimes \{\alpha_{stk} - 2 \mapsto \mathit{ret\_t} :: \alpha_{free} :: \gamma\} \otimes \epsilon'] \; (\mathtt{r1} : \alpha, \mathtt{r2} : \alpha'_{free}, \mathtt{r4} : \alpha_{stk} - 2).$$

Now, the `jmp` instruction at line 21 is type checked because the current memory and registers type satisfies the precondition specified in the label type of the register `r10`. Please note that the integer constraint ($\alpha_{mem} = \alpha_{free} + 2$) specified in the label type of the register `r3` is satisfied by the current integer constraints. Thus, the type system ignores the constraint when checking the equality of the label types of the register `r3` and *ret_t'*.

Next, the typing derivation of the instructions of the label `malloc_cont` is as follows. First, the type of the initial memory is

$$\{\alpha_{tag} \mapsto \langle \alpha_{junk}, \alpha'_{size} \rangle\} \otimes \{\alpha_{mem} \mapsto \exists \beta.\, \langle \beta \rangle\, [\alpha'_{size}]\} \otimes \{\alpha \mapsto \exists \beta.\, \langle \beta \rangle\, [\alpha_{size}]\} \otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \{\alpha_{stk} - 2 \mapsto \mathit{ret\_t} :: \alpha_{tag} :: \gamma\} \otimes \epsilon,$$

where $\alpha_{mem} = \alpha_{tag} + 2$. Next, after the two *pop* instructions (line 29 and 30), the memory type becomes

$$\{\alpha_{tag} \mapsto \langle \alpha_{junk}, \alpha'_{size} \rangle\} \otimes \{\alpha_{mem} \mapsto \exists \beta.\, \langle \beta \rangle\, [\alpha'_{size}]\} \otimes \{\alpha \mapsto \exists \beta.\, \langle \beta \rangle\, [\alpha_{size}]\} \otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon.$$

In addition, the type of the register `r4` becomes α_{stk}, the type of the register `r3` becomes *ret_t*, and the type of the register `r5` becomes α_{tag}. Then, after the `st` instruction at line 33, the memory type becomes

$$\{\alpha_{tag} \mapsto \langle \alpha_{free}, \alpha'_{size} \rangle\} \otimes \{\alpha_{mem} \mapsto \exists \beta.\, \langle \beta \rangle\, [\alpha'_{size}]\} \otimes \{\alpha \mapsto \exists \beta.\, \langle \beta \rangle\, [\alpha_{size}]\} \otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon.$$

In addition, after the `mov` instruction at line 34, the type of the register `r2` becomes α_{tag}. Next, after the `pack` instruction at line 35, the memory type becomes

$$\{\alpha_{tag} \mapsto \tau\} \otimes \{\alpha \mapsto \exists \beta.\, \langle \beta \rangle\, [\alpha_{size}]\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon,$$

where

$$\tau \equiv \exists \alpha_{next}, \alpha_{size}, \alpha_{mem}. |\alpha_{mem} = \alpha_{tag} + 2| \; [\{\alpha_{mem} \mapsto \exists \beta.\, \langle \beta \rangle\, [\alpha_{size}]\} \otimes \{\alpha_{next} \mapsto \mathit{FreeMem}(\alpha_{next})\}] \; \langle \alpha_{next}, \alpha_{size} \rangle\,.$$

Here the argument for the `pack` instruction is

$$[\alpha_{free}, \alpha'_{size}, \alpha_{mem} | \; \{\alpha_{mem} \mapsto \exists \beta.\, \langle \beta \rangle\, [\alpha'_{size}]\} \otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\}] \; \mathit{as}\ \tau.$$

Next, after the `roll` instruction at line 36, the memory type becomes

$$\{\alpha_{tag} \mapsto \mathit{FreeMem}(\alpha_{tag})\} \otimes \{\alpha \mapsto \exists \beta.\, \langle \beta \rangle\, [\alpha_{size}]\} \otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon.$$

Then, after the `apply` instruction at line 37, the type of the register `r3` becomes

$$\forall.|\cdot|[\{\alpha \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size}]\} \otimes \{\alpha_{tag} \mapsto \mathit{FreeMem}(\alpha_{tag})\}\otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon] (\mathtt{r1} : \alpha, \mathtt{r2} : \alpha_{tag}, \mathtt{r4} : \alpha_{stk}).$$

Here the current memory and registers type satisfies the precondition specified by the above label type. Thus, the `jmp` instruction at line 38 is type checked successfully.

Last, the typing derivation of the instructions of the label `malloc_success` is as follows. First, the type of the initial memory is

$$\{\alpha_{tag} \mapsto \langle\alpha_{free}, \alpha'_{size}\rangle\} \otimes \{\alpha_{mem} \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha'_{size}]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\}\otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon,$$

where $\alpha_{size} \leq \alpha'_{size} \wedge \alpha_{mem} = \alpha_{tag} + 2$. Then, after the `split` instruction at line 47, the memory type becomes

$$\{\alpha_{tag} \mapsto \langle\alpha_{free}, \alpha'_{size}\rangle\} \otimes \{\alpha_{mem} \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha'_{size} - \alpha_{size}]\}\otimes \{\alpha_{mem} + \alpha'_{size} - \alpha_{size} \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size}]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\}\otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon.$$

Here the `split` instruction passes the type check because the type checker knows that $\alpha_{size} \leq \alpha'_{size}$. Then, after the `ld` and `sub` instruction at line 49 and 50, the type of the register `r6` becomes $\alpha'_{size} - \alpha_{size}$. Next, after the `st` instruction at line 51, the memory type becomes

$$\{\alpha_{tag} \mapsto \langle\alpha_{free}, \alpha'_{size} - \alpha_{size}\rangle\}\otimes \{\alpha_{mem} \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha'_{size} - \alpha_{size}]\}\otimes \{\alpha_{mem} + \alpha'_{size} - \alpha_{size} \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size}]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\}\otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon.$$

Then, after the instructions from line 54 to 56, the type of the register `r1` becomes $\alpha_{tag} + 2 + \alpha'_{size} - \alpha_{size}$. Next, after the `pack` instruction at line 57, the memory type becomes

$$\{\alpha_{tag} \mapsto \tau\} \otimes \{\alpha_{mem} + \alpha'_{size} - \alpha_{size} \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size}]\}\otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon,$$

where

$$\tau \equiv \exists\alpha_{next}, \alpha_{size}, \alpha_{mem}.|\alpha_{mem} = \alpha_{tag} + 2| [\{\alpha_{mem} \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size}]\} \otimes \{\alpha_{next} \mapsto \mathit{FreeMem}(\alpha_{next})\}] \langle\alpha_{next}, \alpha_{size}\rangle\,.$$

Here the argument for the `pack` instruction is

$$[\alpha_{free}, \alpha'_{size} - \alpha_{size}, \alpha_{mem}| \{\alpha_{mem} \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha'_{size} - \alpha_{size}]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\}]\ \mathit{as}\ \tau.$$

Next, after the `roll` instruction, the memory type becomes

$$\{\alpha_{tag} \mapsto \mathit{FreeMem}(\alpha_{tag})\}\otimes \{\alpha_{mem} + \alpha'_{size} - \alpha_{size} \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size}]\}\otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon.$$

Then, after the `apply` instruction at line 59, the type of the register `r3` becomes

$$\forall.|\cdot|[\{\alpha_{mem} + \alpha'_{size} - \alpha_{size} \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size}]\}\otimes \{\alpha_{tag} \mapsto \mathit{FreeMem}(\alpha_{tag})\}\otimes \{\alpha_{stk} \mapsto \gamma\} \otimes \epsilon] (\mathtt{r1} : \alpha_{mem} + \alpha'_{size} - \alpha_{size}, \mathtt{r2} : \alpha_{tag}, \mathtt{r4} : \alpha_{stk}).$$

Now, the current memory and registers type satisfies the precondition specified in the above label type. Thus, the `jmp` instruction at line 60 passes the type check. As for the type of the register `r1`, please note that the type checker knows that $\alpha_{mem} = \alpha_{tag} + 2$.

B. TYPING DERIVATION OF FREE

The typing derivation of the code in Fig. 16 is as follows. First, the type of the initial memory is

$$\{\alpha_{mem} \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size}]\} \otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \epsilon,$$

where $\alpha_{size} > 2$. Next, after the two `split` instructions (line 9 and 10), the memory type becomes

$$\{\alpha_{mem} \mapsto \exists\beta.\,\langle\beta\rangle\} \otimes \{\alpha_{mem} + 1 \mapsto \exists\beta.\,\langle\beta\rangle\}\otimes \{\alpha_{mem} + 2 \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size} - 2]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \epsilon.$$

Then, after the two `unpack` instructions (line 11 and 12), the memory type becomes

$$\{\alpha_{mem} \mapsto \langle\beta_1\rangle\} \otimes \{\alpha_{mem} + 1 \mapsto \langle\beta_2\rangle\}\otimes \{\alpha_{mem} + 2 \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size} - 2]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \epsilon,$$

Then, after the `tuple_concat` instruction (line 13), the memory type becomes

$$\{\alpha_{mem} \mapsto \langle\beta_1, \beta_2\rangle\} \otimes \{\alpha_{mem} + 2 \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size} - 2]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \epsilon.$$

Thus, a tuple of size 2 was created at the top of the memory (array) to be freed.

Next, we initialize the created tuple. First, we store the size of the array to be freed in the second element of the tuple (line 15 and 16). Thus, the memory type becomes

$$\{\alpha_{mem} \mapsto \langle\beta_1, \alpha_{size} - 2\rangle\} \otimes \{\alpha_{mem} + 2 \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size} - 2]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\} \otimes \epsilon.$$

Then, we link the tuple to the free memory list (line 19). Now, the memory type becomes

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

Then, after the `pack` instruction (line 20), the memory types becomes $\{\alpha_{mem} \mapsto \tau\} \otimes \epsilon$, where

$$\tau \equiv \exists\alpha_{next}, \alpha_{size}, \alpha'_{mem}.|\alpha'_{mem} = \alpha_{mem} + 2| [\{\alpha'_{mem} \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size}]\} \otimes \{\alpha_{next} \mapsto \mathit{FreeMem}(\alpha_{next})\}] \langle\alpha_{next}, \alpha_{size}\rangle\,.$$

Here the hidden argument of the `pack` instruction is

$$[\alpha_{free}, (\alpha_{size} - 2), (\alpha_{mem} + 2)| \{\alpha_{mem} + 2 \mapsto \exists\beta.\,\langle\beta\rangle\,[\alpha_{size} - 2]\}\otimes \{\alpha_{free} \mapsto \mathit{FreeMem}(\alpha_{free})\}]\ \mathit{as}\ \tau.$$

Next, after the `roll` instruction (line 21), the memory type becomes $\{\alpha_{mem} \mapsto \mathit{FreeMem}(\alpha_{mem})\} \otimes \epsilon$. The omitted argument of the `roll` instruction is $\mathit{FreeMem}(\alpha_{mem})$. Then, after the `apply` instruction at line 22, the type of the register `r3` becomes

$$\forall.|\cdot|[\{\alpha_{mem} \mapsto \mathit{FreeMem}(\alpha_{mem})\} \otimes \epsilon](\mathtt{r1} : \alpha_{mem}).$$

Last, the `jmp` instruction passes the type check because the precondition specified in the above label type is satisfied by the current memory and registers type.