Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Master's Thesis

Integrating a Verified Compiler and a Verified Garbage Collector

submitted by

Mikhail Kovalev

on July 28, 2009

Supervisor
Prof. Dr. Wolfgang J. Paul
Saarland University, Germany

Advisors
Elena Petrova, PhD
Saarland University, Germany

Dirk Leinenbach, PhD
Saarland University, Germany

Reviewers
Prof. Dr. Wolfgang J. Paul
Saarland University, Germany

Prof. Dr. Sebastian Hack
Saarland University, Germany

Contents

List of Figures

Chapter 1

Introduction

Formal verification is intended to use formal methods of mathematics in order to prove or disprove correctness of a component of some computer system with respect to its specification. The correctness of all components of a system, however, does not directly imply the correctness of the whole system. The interaction of the system components' specifications may become a tough challenge, especially if different levels of abstraction are involved. The goal of the pervasive system verification is the formal verification of the entire computer system as a whole. This includes hardware verification, verification of compiler, operating system, and applications on the source code level and developing a model which reflects the interaction between those levels.

In the frame of the Verisoft project [21], which aims at the pervasive formal verification of an entire computer system, the so called academic system was built and large parts of it were successfully verified. This system consists of hardware (32-bit RISC processor with the DLX instruction set and devices [20]) on top of which runs a micro kernel [9], a simple operating system [4], and applications. The top-level software of this system is implemented in the C-like language C0 [12]. Specifications and proofs in the Verisoft project are done in several abstraction levels (Fig. 1.1) and results are then transferred from the higher levels to the lower ones with the help of meta theorems [1, 18]. The interactive prover Isabelle/HOL [19] is used in the frame of the project to formalize proofs and definitions.

In order to be able to argue about correctness of the programs on the source code level, one needs to show that a generated binary is correctly compiled with respect to its C0 program. Correctness of a simple non-optimizing compiler of the C0 language in the scope of pervasive system verification has been shown in [12]. The goal of this thesis is to integrate garbage collection routines to the C0 compiler and establish its correctness with respect to new specification.

1.1 Goals and the Strategy

The C0 language allows dynamic memory allocation with the use of the *new* operator. However, for safety reasons it does not allow to free the allocated memory manually. Thus, memory usage of C0 programs without special memory management is not optimal. Integration of a garbage collector into the compiler

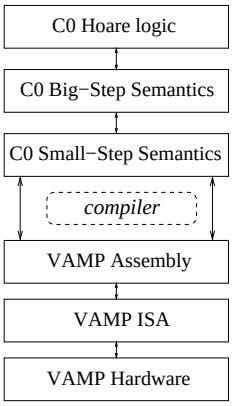

Figure 1.1: Semantic layers in Verisoft

can solve this problem.

Garbage collection, as a form of automatic memory management strategy, is a routine, which attempts to reclaim blocks of memory that will never be used by the application again. A garbage collector (GC) is able to distinguish such unused parts of memory and return them to the free memory. This process is completely invisible for the user program and allows to avoid dangers, hidden in the manual memory deallocation and explicit pointer handling.

GC itself is nothing else than a set of C0 procedures together with special data structures, stored in the global memory of the C0 program. These data structures (GC interface) contain information related to stack frames and data types of the program and form a communication interface, which is used by the garbage collector to obtain all the information it needs for correct memory deallocation.

By correct garbage collection we understand the following:

- memory blocks occupied by unreachable variables are freed (or marked to be free). Unreachable variables are the variables which cannot be accessed from the program by some pointer path;
- memory blocks occupied by reachable variables keep the data they contain;
- pointers in the program point to the same variables that they have pointed to before garbage collection.

The goal of this work is an integration of the garbage collection specification into the compiling specification of a simple, non-optimizing C0 compiler from [12] and proving correctness of the compiler with GC. Note, that in this thesis we deal only with the compiling specification of a compiler and do not address the issue of generating a trustworthy executable version of the compiler.

The specification of the garbage collector routine was done by E.Petrova in the Hoare logic environment for Isabelle/HOL [18]. Correctness of the C0 implementation of the garbage collector in the Hoare logic approach has also been shown. At the Hoare logic level the hardware memory is considered as a set of arrays of natural numbers and the program is not aware of the memory layout of the assembly machine. The compiler specification and verification, however,

has been done at the level of the small-step semantics. Thus, the garbage collection specification from the Hoare logic level (which is built upon big-step semantics) has to be transformed to the small-step semantics or directly to the assembly level. In the frame of this thesis we perform this transfer manually in paper-and-pencil manner. It could be done formally with the help of a meta-theorem, but this is presently still very hard to do [1].

In order to integrate GC into the compiling specification of the C0 compiler we have to:

- specify new transition function for the case of dynamic memory allocation at the C0 level,
- change the code generation algorithm of the compiler for the case of dynamic memory allocation,
- add the global variables of the GC to the list of program global variables,
- specify the result of the GC routine execution based on the Hoare logic specification done by E.Petrova at the machine level.

In order to prove correctness of the compiler with the integrated GC we need to:

- provide several new assumptions on the compiler and user program (restrictions),
- define a new consistency relation, which preserves properties used by GC (e.g. that node headers contain correct data),
- show that the step-by-step simulation relation still holds.

The major effort is to prove the step-by-step simulation theorem for the case of dynamic memory allocation. The garbage collector works directly with the memory of the hardware machine. It moves memory blocks in the heap memory from one place to another and changes the values of the pointers, which point to the heap. After all these operations pointers to the heap point to the same heap nodes as they did before and all program variables keep their old values. Thus, on the C0 level execution of the memory allocation statement with and without GC is almost the same.

1.2 Restrictions

For correct garbage collection the following restrictions have to be put on the compiler:

- no pointers to the heap in the list of GC global variables. The GC interface consists of variables located in the global memory just after the global variables of the program. During garbage collection, values of pointers to the heap located in the program global memory are changed. The values of the pointers located in the GC part of the global memory, however, cannot be changed by the garbage collector since this part of the memory is not "seen" by GC. If the GC would store pointers in the global memory, they could have wrong values after GC. However, our GC implementation does not store such pointers;

- all variables are initialized by default. In the compiler specification from [12] local variables are not automatically initialized by the compiler. However, the interface of the copying garbage collector, used in the frame of this thesis, has no information whether a variable has already been initialized or not. Uninitialized variables contain arbitrary data and uninitialized pointers my cause GC to see heap nodes which do not exist. Thus we have to require that all variables are initialized automatically;

- if in the program there is a pointer to the heap, it should point to the top-level variable (i.e. pointers to the element of an array or a structure on the heap are not allowed). The restriction is introduced due to chosen implementation of the GC algorithm.

1.3 Outline

The rest of the paper is organized in chapters:

- In Chapter 2 we briefly describe the formal semantics of the C0 language and the VAMP assembly semantics. We also introduce the simulation theorem, which is used to establish a connection between C0 and assembly levels in the compiler specification.

- In Chapter 3 we give the formal specification of a simple, copying garbage collector on the Hoare logic level. We also provide correctness criteria and describe the structure and some basic properties of GC interface.

- In Chapter 4 we integrate the garbage collector into the compiler. At first we give the memory layout for the program generated with the C0 compiler with the garbage collector. Then, we specify GC interface data structures inside small-step semantics and define a new transition function for the memory allocation statement. We continue with the extended simulation relation, where we add properties necessary for correct garbage collection. As a next step we define the low-level correctness of the code, generated with the new compiler based on the Hoare logic correctness from Chapter 3. Finally, we show high level correctness for the compiling specification of the compiler with GC.

- We conclude in Chapter 5 with summary and future work.

Chapter 2

Basics

In this chapter we introduce some notation and describe basic concepts which are used in the remainder part of the thesis. These concepts were developed by D.Leinenbach and W.J.Paul in the frame of the Verisoft project. More information about definitions and descriptions given in this chapter can be found in [12].

2.1 Notation

The set of boolean values is denoted by $\mathbb{B}$, the set of integer numbers by $\mathbb{Z}$, the set of all natural numbers (including zero) by $\mathbb{N}$, the set of identifiers by $\mathbb{S}$. By $\mathbb{Z}_w$ and $\mathbb{N}_w$ we denote sets of integer and natural numbers which can be represented with w bits, i.e. $\mathbb{Z}_w = \{-2^{w-1}, \ldots, 2^{w-1} - 1\}$ and $\mathbb{N}_w = \{0, \ldots, 2^w - 1\}$. The notation

$$\lceil n \rceil_d = \lceil n/d \rceil \cdot d$$

denotes the smallest multiple m of d such that m is greater than or equal to n (meaningful for $d > 0$).

Integer division is denoted by $\frac{a}{b}$, unsigned modulo operation by $a \ mod_u \ b$. The predicate $a \mid b$ checks whether a divides b.

Record component c of a record r is accessed by $r.c$ and updated with the value v by $r[c := v]$. A new record may also be composed on the fly by writing $[c_1 = v_1, c_2 = v_2, \ldots, c_n = v_n]$.

Logical implications of the form $(a \wedge b) \Rightarrow c$ are denoted by an inference rule

$$\frac{a \qquad b}{c}.$$

We use inference rules for definitions of predicates. Note, that everything which is not covered by rules explicitly does by default *not* fulfill the predicate.

In definitions and lemmas we often omit quantifiers $\forall$ and $\exists$ for free variables to keep formulas concise. For example, the definition

$$\frac{a \cdot i = b \qquad i \in \mathbb{Z}}{a \mid b}$$

should be read: for all a and b, a divides b if there exists $i \in \mathbb{Z}$, s.t. $a \cdot i = b$.

We use $\hat{\exists}$ to denote existence of the unique entity. Hilbert's choice operator is denoted by ε . The expression $\varepsilon\ x.\ P(x)$ returns an arbitrary value of x, s.t. the property $P(x)$ is satisfied. By $\hat{\varepsilon}$ we denote the unique Hilbert's choice, i.e. $\hat{\varepsilon}\ y.\ P(y)$ returns some value of y, such that $P(y)$ is satisfied and for all z not equal to y the property $P(z)$ does not hold.

Intervals of natural numbers are denoted as *ranges*. A range $(a : b)$ contains all numbers i, s.t. $a \leq i \leq b$. Notation $i \in (a : b)$ means $a \leq i \leq b$ and notation $b \subseteq a$ - that range b is completely contained in range a. A pair of values a and b is denoted as (a, b). We use $fst :: (t_1, t_2) \to t_1$ and $snd :: (t_1, t_2) \to t_2$ to extract first and second component of the pair respectively. Sometimes we also denote ranges by pairs of start address and length. Such ranges are called disjoint iff

$$r_1 \asymp r_2 = (fst(r_1) \geq fst(r_2) + snd(r_2) \vee fst(r_2) \geq fst(r_1) + snd(r_1)).$$

The predicate $bij :: (t_1 \to t2) \to \mathbb{B}$ tests whether an input function is bijective.

Abstract data types. Abstract data types are used to formalize terms which are built recursively by a set of so-called *constructors*. The abstract data type for lists whose elements are of type t is denoted by type $t\ list$ with two constructors $nil :: list$ and $cons :: \mathbb{Z} \times list \to list$.

$$t\ list = nil \mid cons(t, t\ list)$$

In order to find out whether specific constructor was used to build some element of an abstract data type we use a *recognizer*. For example recognizer $is\_nil(l)$ is a predicate which returns true iff l is built using the nil constructor.

In the specification of the compiler from [12] partial functions are widely used. They are simulated by the option type $t\ option$.

$$t\ option = None \mid Some(x),$$

where the constructor $None$ models an undefined value and $Some(x)$ a defined value $x :: t$. We use abbreviation $t_\perp$ for $t\ option$ and $\lfloor x \rfloor$ for $Some(x)$. In order to convert a value y of an option type back to a base type we use a function $the :: t\ option \to t$.

$$the(y) \quad = \quad \begin{cases} x & \text{if } y = \lfloor x \rfloor \\ undef & \text{if } y = None \end{cases}.$$

Lists. We abbreviate the empty list nil by $[\,]$ and $cons(h, t)$ by $h\#t$. To create a list on the fly we use notation $[a_1, a_2, \ldots, a_n]$. Functions $hd :: t\ list \to t$ and $tl :: t\ list \to t\ list$ compute head and tail of the list respectively. Function $l[p, n]$ returns n elements of the list l starting from position p. The length of list l is computed with $|l|$.

Function $map :: (t_1 \to t_2) \times t_1\ list \to t_2\ list$ maps function $f :: t_1 \to t_2$ on a list of type $t_1\ list$

$$\begin{aligned} map(f, [\,]) &= [\,] \\ map(f, h\#tl) &= f(h)\#map(f, tl) \end{aligned}$$

The set of all elements of list l is denoted by $\{l\}$. Access to the $i-$th element of the list l is denoted by $l!i$ and is defined only if $i < |l|$. If we use $l!i$ in

lemmas and proofs we automatically assume $i < |l|$. By $l[i := v]$ we denote update of the $i-$th element of the list l with the value v. Notations $a \in l$ and $a \notin l$ are used to state that a is or is not an element of the list l. The predicate $distinct(l)$ checks whether the elements of the list l are pairwise distinct and the function $replicate(n, a)$ generates a list consisting of n copies of element a. Concatenation of two lists l_1 and l_2 is denoted by $l_1 \circ l_2$. Concatenation of n lists - by $[l_1, l_2, \ldots, l_n]$. Function $concat :: t\ list\ list \to t\ list$ performs concatenation of an array of lists into one list. We overload the addition operator to add a single number to every element of the list.

$$\begin{aligned} op + &:: \mathbb{N}\ list \times \mathbb{N} \to \mathbb{N}\ list \\ [\,] + d &= [\,] \\ (h\#tl) + d &= (h + d)\#(tl + d) \end{aligned}$$

We define the function $map\text{-}of :: (t_1, t_2)\ list \times t_1 \to t_{2\perp}$ which finds the first element of the list of pairs, whose first component equals the second parameter of the function and returns the second component of this element.

$$\begin{aligned} map\text{-}of([\,], x) &= None \\ map\text{-}of((a, b)\#t, x) &= \begin{cases} \lfloor b \rfloor & \text{if } a = x \\ map\text{-}of(t, x) & \text{else} \end{cases} \end{aligned}$$

2.2 The C0 Programming Language

There exists a large variety of programming languages nowadays. Some of them have limited functionality. The scope of others is almost unlimited. Variety of language features makes the language more flexible and ease the program development process. Yet, they also make axiomatization and creation of the language model more complex or even impossible.

One of such widely used languages is ANSI C. However, because of the complex semantics of the C language the use of all C features encourages an error prone programming style. Thus, it is much easier to verify applications on the source code written in a safe and secure programming language similar to Pascal [13]. Such a language was developed by Leinenbach and Paul in the frame of the Verisoft project and is called the C0 programming language [12].

The simplicity of C0 allows to construct a formal model of the language and use it in the verification processes. More over, C0 shares with ANSI C a lot of features and syntax details.

2.2.1 Language Restrictions

The C0 programming language is a restricted version of the ANSI C language. Specification of C0 can be found in [13, 17]. Below is a sketch of the major restrictions :

- no pointer arithmetic;
- no function pointers;
- all functions must return a value, functions cannot return *void*;

- the last statement in a function is the *return* statement and it is the only *return* statement in the function;
- initialization of a variable during declaration is not possible;
- no declarations of variables after the first statement of a function (all declarations should be located in the very beginning of the function);
- function calls and side-effects inside expressions are prohibited;
- no pointers to local variables;
- the size of arrays must be statically fixed at the compile time;
- no *void* pointers, all the pointers should be typed.

2.2.2 Type System and Statements

The type system of C0 is limited to four basic ANSI C types (*int*, *unsigned int*, *char*, *bool*) and 3 complex types (array, pointer, and structure).

In the C0 language there are six basic types of statements: assignment, conditional, procedure call, loop, return, and allocation of dynamic memory. Additional information on the C0 language can be found in [16].

2.3 Formal C0 Small Step Semantics

2.3.1 Types and Type Name Environment

In C0 there are four basic types, a structure type, an array type, and a pointer type. We represent C0 types formally by the abstract data type ty.

$$\begin{aligned} ty = \quad & Bool_T \mid Int_T \mid Char_T \mid Unsigned_T \\ & \mid Str_T(\mathbb{S} \times ty\ list) \\ & \mid Arr_T(\mathbb{N} \times ty) \\ & \mid Ptr_T(\mathbb{S}) \mid Null_T \end{aligned}$$

To support self-referencing pointers (e.g. in structures for list elements which contain a pointer to list elements) we introduce a level of indirection for pointers. They do not point directly to a type but to a type name. Type names are mapped to their types using so-called *type-name environment*:

$$tenv \ = \ (\mathbb{S}, ty)\ list$$

2.3.2 Statements

In the C0 small-step semantics we model statements by the data type *stmt*. Statements are tagged with identifier *sid*, which is used to determine where the statement was originally placed in the program rest (we omit these identifiers if they are not explicitly used).

$$
\begin{aligned}
stmt = \quad & Skip \\
& \mid Comp(stmt, stmt) \\
& \mid Ass(expr, expr, sid) \\
& \mid PAlloc(expr, \mathbb{S}, sid) \\
& \mid SCall(\mathbb{S}, \mathbb{S}, expr\ list, sid) \\
& \mid Return(expr, sid) \\
& \mid Ifte(expr, stmt, stmt, sid) \\
& \mid Loop(expr, stmt, sid) \\
& \mid Asm(asm\ list, sid) \\
& \mid XCall(\mathbb{S}, expr\ list, expr\ list, sid)
\end{aligned}
$$

The first two statements are called *structural* statements. *Skip* is an empty statement and *Comp(stmt,stmt)* is a sequential composition of two statements. Assignment is represented by $Ass(expr_1, expr_2)$. $PAlloc(e, t_n)$ stands for dynamic memory allocation of an instance of the type t_n. The pointer to the newly allocated memory will be assigned to e. Function calls to a function f are represented by $SCall(vn, f, [e_1, \ldots, e_n])$, where $e_1, \ldots, e_n$ are passed parameters and vn is the variable where the return value of the function will be stored. Return statements with return expression e are represented by $Return(e)$. $Ifte(e, s1, s2)$ stands for conditional execution, $Loop(e, s1)$ - for a while loop. A call to a sequence of inline assembly instructions a is modeled by $Asm(a)$.

The so-called XCAlls [12] are used in order to change the additional *extended* state component, which was introduced to the C0 model in order to make it possible to deal with inline assembly code accessing devices. $XCall(f, [e_1, \ldots, e_n], [p_1, \ldots, p_n])$ represents the XCall f with input parameters $p_1, \ldots, p_n$ and left expressions $e_1, \ldots, e_n$ (the parts of the C0 state that can be changed by the XCall). In the C0 semantics the effect of the XCall is defined axiomatically.

2.3.3 Function Table

A function table contains information about all functions in a C0 program. A single function p is modeled by the record $funcT$ with four components:

- $p.body :: stmt$: the body of the function,
- $p.params :: (\mathbb{S} \times ty)\ list$: parameters of the function,
- $p.rtype :: ty$: the return type of the function,
- $p.lvars :: (\mathbb{S} \times ty)\ list$: the list of local variables of the function.

  A function table is modeled by the type $functableT = (S \times funcT)\ list$, where each pair contains a function name and a record of type $funcT$.

2.3.4 Configuration of the C0 Small-Step Semantics

In the small-step semantics configuration of the C0 program stores information about the current state of the program; it consists of the *memory configuration* and the *program rest*. The *memory configuration* stores information about the variables and program rest contains statements which still have to be executed.

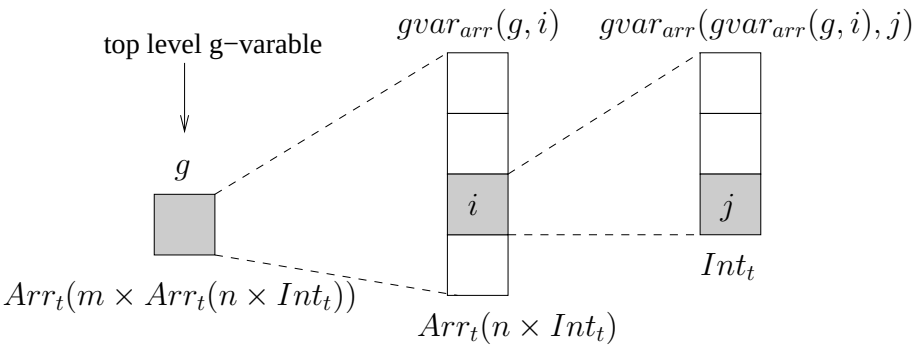

Figure 2.1: Structure of a g-variable of an array type.

Generalized Variables

Program variables are modeled by so-called *g-variables*. We distinguish local, global, and heap g-variables. Global and local g-variables are identified by their name (and the frame number for local g-variables). Heap g-variables are identified by their position in the corresponding symbol table. G-variables are represented by the following data type.

$$\begin{aligned} gvar = \quad & gvar_{gm}(\mathbb{S}) \\ & \mid gvar_{lm}(\mathbb{N}, \mathbb{S}) \\ & \mid gvar_{hm}(\mathbb{N}) \\ & \mid gvar_{arr}(gvar, \mathbb{N}) \\ & \mid gvar_{str}(gvar, \mathbb{S}) \end{aligned}$$

The inductive case defines g-variables for structure and array access. If g is a g-variable of structure type then $gvar_{str}(g, n)$ is also a g-variable, representing structure component with the name n. If g is a g-variable of array type then the i-th array element $gvar_{arr}(g, i)$ is also a g-variable. Note, that a g-variable with the simplest type has the most complicated structure (Fig. 2.1).

We call r the root of g-variable g iff it is the highest ancestor of g. The function $root_g :: gvar \mapsto gvar$ computes the root of a given g-variable.

$$\begin{aligned} root_g(gvar_{gm}(x)) &= gvar_{gm}(x) \\ root_g(gvar_{lm}(i,x)) &= gvar_{lm}(i,x) \\ root_g(gvar_{hm}(i)) &= gvar_{hm}(i) \\ root_g(gvar_{arr}(g,i)) &= root_g(g) \\ root_g(gvar_{str}(g,c)) &= root_g(g) \end{aligned}$$

The set $sub_g :: gvar \mapsto gvar$ contains all sub variables of a given g-variable. Initially, the set contains the g-variable itself.

$$\overline{g \in sub_g(g)}$$

For the cases of an array and a structure the set is constructed in the following

way.

$$\frac{h \in sub_g(g)}{gvar_{arr}(h,i) \in sub_g(g)}$$

$$\frac{h \in sub_g(g)}{gvar_{str}(h,c) \in sub_g(g)}$$

Memory Frames

We use a memory model in the style of [15], which stores the memory content as a mapping from addresses (natural numbers) to memory cells. A single memory cell stores the value of a variable of an elementary type.

$$mcell_{C0} \;=\; Int(\mathbb{Z}) \mid Nat(\mathbb{N}) \mid Char(\mathbb{Z}) \mid Bool(\mathbb{B}) \mid Ptr(gvar \cup \{\bot\})$$

Consecutive sequences of memory cells can store values of aggregate types.

Each memory frame contains not only memory content, but also list of variables of the frame together with their types and keeps track of the set of variables which are already initialized. We use the record $mframe$ to represent memory frame m.

- $m.ct :: \mathbb{N} \mapsto mcell_{C0}$: The content of the memory frame. Technically infinite. In the compiler without GC this mapping is used only for addresses smaller than the size of the memory (which can increase dynamically for the heap memory). However, in the presence of the GC the size of all variables really located in the heap memory might be smaller then size of variables located in the heap memory frame (variables are never removed from the memory frame). This mapping is defined only for variables, which are currently present in the heap memory.

- $m.st :: (\mathbb{S} \times ty)\ list$: A list of all variables of the memory frame together with their types (so called *symbol table* of the frame)

- $m.init :: \mathbb{S}\ set$: The set of variables which are already initialized. Our version of GC cannot distinguish initialized variables from uninitialized ones. Thus, we require that compiler never leaves any variables uninitialized and this set always contains all g-variables of the program.

The *base address* of a variable x in a symbol table is zero if x is the first variable in the symbol table. Otherwise, it is defined inductively as the sum of the size of the first variable and the base address of x in the tail of the symbol table. The function $ba_v :: (\mathbb{S} \times ty)\ list \times \mathbb{S} \mapsto \mathbb{N}_\bot$ computes the base address of a variable.

The type of a variable x in the symbol table st is computed by the function $type_v :: (\mathbb{S} \times ty)\ list \times \mathbb{S} \mapsto ty_\bot$.

Memory Configuration

The memory configuration of the C0 small-step semantics consists of three parts: a memory frame for the global variables, a memory frame for the heap variables,

and a list of memory frames and return destinations for the local variables. It stores information about the variables of a C0 program and their values. Formally memory configuration mc consists of three components.

- $mc.gm :: mframe$: The memory frame for the global variables.
- $mc.lm :: (mframe \times gvar)\ list$: Stack of local memories. Memory frames store values of local variables and g-variables store return destinations of stack frames (after execution of the procedure, return value will be assigned to its return destination g-variable).
- $mc.hm :: mframe$: The memory frame for nameless heap variables.

Local memory frames are abbreviated in the following way.

$$lm_{top}(mc) = mc.lm!(|mc.lm| - 1)$$
$$lm_i(mc) = mc.lm!i$$

For the symbol tables of the global memory, of the local memory frames, and of the heap memory we will use the following abbreviations.

$$gst(mc) = mc.gm.st$$
$$lst_{top}(mc) = lm_{top}(mc).st$$
$$lst_i(mc) = lm_i(mc).st$$
$$hst(mc) = mc.hm.st$$

Symbol Configuration

Frequently in this thesis it is sufficient to have only the symbol tables of all memory frames available. Thus, we introduce a record type *symbolconf* for the *symbol configuration* of a program.

- $sc.gst :: (\mathbb{S} \times ty)\ list$: The symbol table of the global memory frame.
- $sc.lst :: (\mathbb{S} \times ty)\ list\ list$: A list of symbol tables for the stack of local memories.
- $sc.hst :: (\mathbb{S} \times ty)\ list$: The symbol table of the heap memory frame.

We refer to the symbol configuration of the memory frame mc by $sc(mc)$. For every local memory frame there exists a corresponding function in the function table ft. Symbol tables of the local frame and the corresponding function in ft are equal. The function $stbl_f(ft, i)$ returns a symbol table of the $i-th$ function in ft.

Program Rest and Program Configuration

The program rest contains program statements which still have to be executed. It is initialized with the body of the main function and can shrink or grow during execution of the program. Formally we represent the program rest by a C0 statement of type *stmt*.

A configuration c of the C0 small-step semantics is formalized by the record $conf_{C0}$:

- $c.mem :: memconf$: The memory configuration of the program.
- $c.prog :: stmt$: The program rest.

Some Functions on G-Variables

Let te be a type name environment and sc a symbol configuration. For simplicity reasons we will often omit constant parameters like te and sc if their meaning is clear from the context. Note, that symbol configuration and type environment remain constant during program execution

The type of a g-variable is obtained by means of inductive function $ty_g :: symbolconf \times gvarT \mapsto ty$. For the base case we get the type directly from the corresponding symbol table.

$$\begin{aligned} ty_g(sc, gvar_{gm}(x)) &= type_v(sc.gst, x) \\ ty_g(sc, gvar_{lm}(i, x)) &= type_v(sc.lst!i, x) \\ ty_g(sc, gvar_{hm}(i)) &= snd(sc.hst!i) \end{aligned}$$

For the case of an array and a structure we use the following definitions.

$$ty_g(sc, gvar_{arr}(g, i)) = \begin{cases} t & \text{if } ty_g(g) = Arr_T(n, t), \\ undef & \text{otherwise} \end{cases}$$

$$ty_g(sc, gvar_{str}(g, x)) = \begin{cases} the(map\text{-}of(c, x)) & \text{if } ty_g(g) = Str_T(c), \\ undef & \text{otherwise} \end{cases}$$

The predicate $elem_g(sc, g)$ returns true iff a g-variable g is of a simple type. We also use predicates $ptr_{gvar}(sc, g)$, $arr_{gvar}(sc, g)$, $str_{gvar}(sc, g)$ to check whether g is of a pointer, an array, or a structure type respectively.

$$\begin{aligned} ptr_{gvar}(sc, g) &= \exists vn.\ ty_g(sc, g) = Ptr_T(vn) \\ arr_{gvar}(sc, g) &= \exists t\ n.\ ty_g(sc, g) = Arr_T(n, t) \\ str_{gvar}(sc, g) &= \exists scl.\ ty_g(sc, g) = Str_T(scl) \end{aligned}$$

We define the memory name of a g-variable by the type $memname$. A memory name is either gm for the global memory, hm for the heap memory, or $lm(i)$ for $i-$th frame of the local memory. Function $mem_g :: gvar \mapsto memname$ returns the memory name of a g-variable.

We distinguish *named* g-variables (global or local) and *nameless* g-variables (heap).

$$named_g(g) = (mem_g(g) = gm \vee \exists i.mem_g(g) = lm(i))$$

The base address of a g-variable is obtained by means of the function $ba_g :: symbolconf \times gvarT \mapsto \mathbb{N}$. For global and local top-level g-variables it calls function ba_v. For the case of a top level g-variable $gvar_{hm}(i)$ it returns the sum of sizes of all g-variables in the heap symbol table with indices smaller than i. For the array access $gvar_{arr}(g, i)$ it returns $ba_g(g) + i \cdot size_t(t)$ if g is of array type. For $gvar_{str}(g, x)$ it returns $ba_g(g) + the(ba_v(scl, x))$ if $ty_g(g) = Str_T(scl)$.

The value of a g-variable g is defined by

$$value_g(mc, g) = mc_{mem_g(g)}[ba_g(sc(mc), g), size_t(ty_g(sc(mc), g))],$$

where mc is a memory configuration and $mem_g(g)$ is memory name of g (i.e. gm, hm, or $lm(i)$). Function $mc_n[a, l]$ reads the content of the memory with name n in the range from a to $a + l - 1$.

2.3.5 Execution of C0 Programs

Execution of a C0 program starts from the *initial* configuration of the program. Values of global and heap variables are initialized with default values.

For updating the memory of a C0 configuration we use function

$$memupd :: memconf \times gvar \times (\mathbb{N} \mapsto mcell_{C0}) \mapsto memconf_{\perp},$$

which updates g-variable g with value $v :: \mathbb{N} \mapsto mcell_{C0}$. Partial updates of uninitialized variables are not allowed in C0. In the compiler with GC we require all g-variables to be initialized by default. Therefore, the memory update operation is always successful. Execution of a C0 program is modeled by the transition function of the C0 small-step semantics, which maps the current C0 configuration to its successor configuration or to $None$ (in the case of an error).

$$\delta_{C0} :: tenv \times functableT \times conf_{C0} \mapsto conf_{C0\perp}$$

The transition function is defined by induction on the program rest. If program rest consists of a compound $Comp(s1, s2)$ statement, transition function is applied recursively to statement $s1$. Otherwise, the program rest consists of a single indivisible statement and we apply δ_{C0} to this statement without recursion.

In this thesis we focus on the heap memory allocation statement. Semantics of this statement for the compiler with GC is given in Section 4.3.

2.3.6 Valid C0 Configurations

Not every possible C0 configuration, which could be built according to the definition from Section 2.3.4, is reasonable. I.e., every reasonable C0 configuration should satisfy a number of rules, which are built based upon language semantics.

Definition 2.1 (**Valid type**)**.** We call basic type t valid iff it belongs to a set of basic types. Pointer type is valid iff the type name is defined in the type name environment. Structure type is valid iff the list of components is not empty and all component names are pairwise distinct and components are of valid type. Array types are valid if they are not empty and the element type itself is a valid type.

$$\frac{t \in \{Bool_T, Int_T, Char_T, Unsigned_T, Null_T\}}{valid_{ty}(te, t)} \qquad \frac{tn \in map(fst, te)}{valid_{ty}(te, Ptr_T(tn))}$$

$$\frac{scl \neq [\,] \qquad distinct(map(fst, scl)) \qquad \forall sc \in scl.\ valid_{ty}(te, snd(sc))}{valid_{ty}(te, Str_T(scl))}$$

$$\frac{0 < n \qquad valid_{ty}(te, t)}{valid_{ty}(te, Arr_T(n, t))}$$

Definition 2.2 (**Valid type name environment**)**.** Type name environment te is valid iff all type names in it are pairwise distinct and types are valid.

$$\frac{distinct(map(fst, te)) \qquad \forall (tn, t) \in te.\ valid_{ty}(te, t)}{te \in valid_{tenv}}$$

Definition 2.3 (**Valid symbol table**). Symbol table st is valid iff all types of all variables in it are valid and variable names are pairwise distinct.

$$\frac{distinct(map(fst, st)) \qquad \forall (vn, t) \in st.\ valid_{ty}(te, t)}{st \in valid_{st}(te)}$$

Configurations which satisfy all validity rules are called *valid* configurations. In a valid configuration all symbol tables and the type environment are valid and a number of other properties hold. Function

$$conf_{\surd} :: tenv \times functableT \mapsto conf_{C0}\ set$$

defines the set of all possible valid C0 configurations. The formal definition for $conf_{\surd}$ can be found in [12].

The set of valid g-variables is computed by the function $gvars_{\surd}(sc)$. A top-level named g-variable is valid iff its name is present in the global or local symbol tables of the symbol configuration sc. A top-level heap g-variable $gvar_{hm}(i)$ is valid iff $i < |sc.hst|$. For an element of an array $gvar_{arr}(g, i)$ we require that the parent g-variable g is of an array type $Arr_T(n, t)$ and is itself a valid g-variable and the index of an element i is less than n. For the structure component $gvar_{str}(g, cn)$ we require that the parent g-variable g is of type $Str_T(scl)$, that the component name is defined in the structure and g is itself a valid g-variable.

2.4 VAMP Assembly Semantics

The target language of the verified C0 compiler is the assembly language of the microprocessor VAMP. The VAMP is a formally verified DLX-like processor [2, 3, 7, 8] based on the proofs from [14] and [11]. Correctness of the VAMP assembly model with respect to the VAMP Instruction Set Architecture (ISA) level for user programs has been shown by A.Tsyban. In this section we describe only the memory model of the assembly semantics and omit the ISA model.

2.4.1 Memory Model

Memory cells of the VAMP assembly semantics are modeled axiomatically with the type $mcell_{asm}$, functions $cell2int :: mcell_{asm} \to \mathbb{Z}$, $int2cell :: \mathbb{Z} \to mcell_{asm}$, and the following axioms:

$$\begin{aligned} cell2int(int2cell(m)) &= m \\ int2cell(cell2int(i)) &= i \end{aligned}$$

Data from a single memory cell is read as an integer number. Registers in the VAMP assembly model also store data as integers. In the frame of this thesis, however, we often want to interpret register and memory values as natural numbers. For this purpose we use conversion functions $i2n :: \mathbb{Z} \to \mathbb{N}$ and $n2i :: \mathbb{N} \to \mathbb{Z}$, which simulate numerically the conversion from integers to bit vectors and then to naturals and vice versa.

$$
\begin{aligned}
i2n(i) &= \begin{cases} i + 2^{32} & \text{if } i < 0 \\ i & \text{otherwise} \end{cases} \\
n2i(n) &= \begin{cases} n - 2^{32} & \text{if } 2^{31} \leq n < 2^{32} \\ n & \text{otherwise} \end{cases}
\end{aligned}
$$

Note, that these conversions make sense only if the converted number is in the range of 32-bit numbers. In the VAMP assembly model we require, that all numbers stored in the memory of assembly machine and in registers are in the range of 32-bit integers. For the functions given above the following properties follow from the definitions:

$$
\begin{aligned}
-2^{31} \leq i < 2^{31} \Longrightarrow n2i(i2n(i)) &= i \\
i2n(n2i(n)) &= n
\end{aligned}
$$

2.4.2 Configuration

A configuration of the VAMP assembly semantics is modeled with the record $d :: conf_{asm}$, which consists of the following components:

- program counters: $d.dpc :: \mathbb{B}_{32}$, $d.pcp :: \mathbb{B}_{32}$,
- register files for general purpose and special purpose registers $d.grp :: \mathbb{Z}_{32}\ list$, $d.spr :: \mathbb{Z}_{32}\ list$,
- word addressed memory: $d.mm :: \mathbb{N} \to mcell_{asm}$.

Note, that for registers we use types with bounded domains and therefore do not need to implicitly restrict their values. In Isabelle/HOL, however, we have to consider these additional restrictions.

The set of valid VAMP configurations is defined by the predicate $conf_{asm_\surd} :: conf_{asm} \to \mathbb{B}$:

$$
\frac{|d.gpr| = 32 \qquad |d.spr| = 32 \qquad \forall a.\ -2^{31} \leq cell2int(d.mm(a)) < 2^{31}}{conf_{asm_\surd}(d)}.
$$

We use predicates $unchngd_{mem}(m, m', a, b)$ and $memchngd_{mem}(m, m', a, l)$ to state that assembly memory in range (a, b) is not changed and that memory is changed only in the range $(a, a + l)$ respectively. Function $read_{data}(m, a, l)$ reads the content of l consecutive memory cells from memory m starting at word address a. We use one more notation for memory read function:

$$
m[b, l] = read_{data}(m, \frac{b}{4}, l).
$$

Transition function $\delta_{asm} :: conf_{asm} \to conf_{asm}$ is defined in the following way:

$$
\delta_{asm}(d) = exec(d, currinstr(d))
$$

where $exec :: conf_{asm} \times instr \to conf_{asm}$ is a function which executes single VAMP assembly instructions and $currinstr :: conf_{asm} \to instr$ calculates the next instruction to be executed in configuration d.

2.4.3 Execution of Assembly Code

Verification of the compiler specification is done by induction on the program rest on C0 level. On every step one single C0 statement is executed. However, for a single C0 statement there is a number of VAMP assembly instructions that are generated. Eventually, execution of these command will result in some new assembly configuration d'. We use the predicate

$$d \xrightarrow[range_c, range_a]{t, dest} d'$$

to state that the VAMP assembly machine started in configuration d reaches in t steps a configuration d' with $d'.dpc = dest$ and $d'.pcp = d'.dpc + 4$ using $range_c$ and $range_a$ as code range and address range, respectively. Moreover, in the predicate we require that the code stays unchanged and that no interrupt generating events occur.

By the predicate $asm_{pre}(d, range_c, il)$ we denote the preconditions for assembly execution, where $range_c$ is the code range and il is a list of assembly instructions. The predicate requires that d is a valid VAMP assembly configuration, that *dpc* is a multiple of four, that *pcp* points to the next instruction, that the instruction list il is stored in the memory of d starting at address $d.dpc$, that il is completely located within the code range, and that the code range fits into 32-bit addresses.

2.5 Simulation Theorem

Simulation theorem is the top level correctness theorem for the *compiling specification* of the C0 compiler. In this section we give only an overview of the simulation relation and do not provide the formal proof of the simulation theorem.

2.5.1 Simulation Relation

The simulation relation plays the role of the compiler correctness criteria. The compiler correctness proof ensures that the compiled code preserves the simulation relation between the C0 and the assembly machine and thus shows the same behavior as the original C0 program. In the compiler without garbage collection the simulation is defined via the predicate *consis*, which connects a C0 configuration c with an assembly configuration d. The predicate requires the variables of the C0 program to be stored correctly in the assembly memory and the control flow of the C0 program to be properly represented in the assembly machine. We give the formal definition of the consistency relation for the compiler with GC in Sec. 4.4. Here we give some basic definitions which are used in the simulation relation of the old compiler.

Reachable G-Variables

Reachability of g-variables is one of the key concepts in the compiler correctness definition. We require only for reachable g-variables that their data is stored correctly in the assembly memory of the machine.

Below we give another definition for a pointer g-variable. In the compiler without GC a pointer g-variable can have not only a pointer type, but also can be of the type $Null_T$. The garbage collector, however, does not consider variables of the $Null_T$ type to be pointers and treats them the same way as integer or natural g-variables (because their values are fixed to the null pointer). Nevertheless, our goal was to change the existing simulation relation as little as possible (keep old consistency criteria almost unchanged, while adding new relations concerning the garbage collector). Thus, we left the pointer definition in the simulation relation unchanged, while proving the correctness separately for the normal pointers and $Null_T$ pointers.

Definition 2.4 (**Pointer g-variables**)**.** Let te be a type name environment, sc a symbol configuration, and g a g-variable. We define the pointer g-variable in the following way.

$$is\_gvar_p(sc, g) = (ptr_{gvar}(sc, g) \vee ty_g(sc, g) = Null_T)$$

Definition 2.5 (**Reachable nameless g-variables**)**.** We introduce the predicate $reachable_g^{nameless} :: memconf \to gvar\ set$, which defines the set of reachable g-variables in the heap memory. In the base case heap g-variable g is reachable if there exists some named g-variable x, which points to g.

$$\frac{\begin{array}{c} x \in gvars_{\surd}(sc(mc)) \\ named_g(x) \qquad is\_gvar_p(sc(mc), x) \qquad initialized_g(mc, x) \\ g \in gvars_{\surd}(sc(mc)) \qquad \neg named_g(g) \qquad value_g(mc, x) = Ptr(g) \end{array}}{g \in reachable_g^{nameless}(mc)}$$

The induction case distinguishes two cases: when g is reachable via some other reachable nameless g-variable, or when g is a sub g-variable of a reachable nameless g-variable.

$$\frac{\begin{array}{c} h \in reachable_g^{nameless}(mc) \qquad is\_gvar_p(sc(mc), h) \qquad initialized_g(mc, h) \\ g \in gvars_{\surd}(sc(mc)) \qquad \neg named_g(g) \qquad value_g(mc, h) = Ptr(g) \end{array}}{g \in reachable_g^{nameless}(mc)}$$

$$\frac{h \in reachable_g^{nameless}(mc) \qquad g \in sub_g(h) \qquad g \in gvars_{\surd}(sc(mc))}{g \in reachable_g^{nameless}(mc)}$$

Definition 2.6 (**Reachable g-variables**)**.** We call a g-variable reachable, if it belongs either to a set of reachable nameless g-variables or is a valid named g-variable. Formally we define

$$\frac{g \in reachable_g^{nameless}(mc)}{g \in reachable_g(mc)}$$

$$\frac{g \in gvars_{\surd}(sc(mc)) \qquad named_g(g)}{g \in reachable_g(mc)}$$

Allocation Function

The simulation relation *consis* which we define in Sec. 4.4 is parameterized with the allocation function $alloc :: gvar \to (\mathbb{N} \times \mathbb{N})$, which maps a g-variable to a tuple of natural numbers. The first number represents the *allocated address* and the second - the *allocated size* of the g-variable. For global variables these numbers are constant during program execution; for local variables they are constant during local procedure execution. For heap variables we have to keep track of their actual position in the heap. The allocated address of heap variables may change with every execution of the memory allocation statement. The return values of the allocation function are fixed with the help of the simulation relation.

Structure of the Simulation Relation

Here we give the basic structure of the compiler simulation relation. The central statement of the simulation relation is that all reachable g-variables of the C0 configuration are properly stored at their allocated address in the assembly configuration. In order to make the compiler correctness statement inductive, however, we need a number of other properties to be added to the simulation relation. For example we require that the compiled code has not been changed and that all stack frames are properly represented. Integration of a garbage collector to the compiler also adds a number of properties to the simulation relation.

The top level definition of the simulation relation *consis* of the compiler without GC contains three predicates:

- $consis_{code}$: code consistency. Requires that the compiled code is stored at address *progbase* in the assembly configuration;
- $consis_c$: control consistency. Requires that the program counters of the assembly machine point to the start address of the code which has been generated for the first statement of the current program rest and that return addresses of all stack frames are correct;
- $consis_d$: data consistency. States that reachable g-variables are correctly stored in the assembly machine, some auxiliary information about stack and heap is stored correctly, and the allocation function for named g-variables is correct. This is the central statement of the simulation relation and showing that data consistency holds on the induction step is the major challenge in the frame of this thesis.

We give the formal definition for data consistency and introduce additional (garbage collector) top-level predicate to the simulation relation in Sec. 4.4. We do not focus on code and control consistency, since the proof of these predicates is largely based on the low-level garbage collection correctness, which is not covered in the thesis.

2.5.2 Simulation Theorem

The formal statement of the simulation theorem is quite complex and we do not present it here. Generally speaking, the simulation theorem states that for

every step of a C0 program the assembly machine executing the compiled code will reach an equivalent state, i.e. the state where the simulation relation *consis* holds.

The proof of the theorem is done by induction on the step number i of the C0 machine. For the base case $i = 0$ we have to show that the initialization code of the compiler correctly sets up an assembly configuration which is consistent with the initial C0 configuration. For the induction step we perform a case distinction on the next statement in the program rest. With the presence of the GC, the basic structure and the proof methodology of the simulation theorem has not changed. The only statement whose treatment has changed significantly is the dynamic memory allocation statement (*PAlloc*).

Chapter 3

Garbage Collector

In this chapter we provide specification for the copying garbage collector and data structures of the GC interface. Implementation of GC interface data structures was developed by M.A. Hillebrand and D. Leinenbach and formal specificaion of the data structures was done in the frame of this thesis (Sec. 4.2). Specification on the Hoare logic [5, 10, 17] level was developed by E.Petrova.

The copying garbage collection algorithm was proposed by C. J. Cheney in 1970 [6]. For this algorithm the heap of the program is divided into two disjoint parts: *to-space* and *from-space*. In every moment only from-space is observed and considered as a heap by the program. During garbage collection all reachable nodes are copied from the from-space to the to-space in BFS-fashion and the spaces are switched (from-space becomes to-space and vice versa).

3.1 GC Interface

GC interface is a set of variables and data structures located in the global memory of the program. The data structures are initialized by the compiler during the program compilation and remain constant during the program execution. Global memories of the program and the garbage collector do not overlap and the program does not access GC interface.

3.1.1 Data Structures Specification

For correct execution of the garbage collection routine GC needs to distinguish reachable nodes from unreachable ones and to know the location of the reachable nodes in the heap memory. The required information is obtained with the help of two implementation tables: the *type table* and the *pointer table*. Those tables contain static information about all pointers in the global and local memories of the program. In the type table we store allocated sizes of types and frames and number of pointers present in a type or a memory frame. In the pointer table we store displacements of pointers from memory frames and types.

A single element tt of the type table represents either a type from the program type environment or a memory frame (global or local). It is modeled with the record ty_{tt} with three components:

- $tt.st :: \mathbb{N}$: start index of the respective sub-array of tt in the pointer table. If there are no pointers in the type or in the frame, then the value of $tt.st$ is undefined,
- $tt.num :: \mathbb{N}$: the length of the respective sub-array of tt in the pointer table. Is equal to the number of pointers in the type or in the frame,
- $tt.asz :: \mathbb{N}$: allocated size of the type or of the frame in words.

An element pp of the pointer table represents a single pointer. It is modeled with the record ty_{pp} with two components:

- $pp.woff :: \mathbb{N}$: displacement of the pointer inside the memory frame or inside the memory allocated for the type (depending on whether the respective element tt of the type table represents a frame or a type) in words,
- $pp.idx_{ty} :: \mathbb{N}$: index of the type of the pointer in the type table. This field is not used in the garbage collection routine. It is only used on the initialization stage, when type and pointer tables are being constructed. The construction of TT and PP tables is not covered in this theses.

Note, that here and below displacements and allocated sizes of types and memory frames are given in words. In this model of the garbage collector we require that variables are correctly aligned in the memory. Thus, all allocated sizes and displacements are divisible by the word length (four bytes).

Type and pointer tables are modeled as lists of ty_{tt} and ty_{pp} elements respectively.

$$TT_{type} = ty_{tt}\ list$$
$$PP_{type} = ty_{pp}\ list$$

Later on in this thesis we denote instances of the types TT_{type} and PP_{type} by TT and PP. We also use notation PP_i and TT_j instead of $PP!i$ and $TT!j$.

The first element of the TT table represents global memory frame. The next $|ft|$ elements represent local frames from the function table $ft :: functableT$. The last $|te|$ elements of the TT table represent types from the type-name environment $te :: tenv$ (Fig. 3.1). Garbage collection is meaningful only if the function table is not empty (at least function *main* is present there) and the type environment is not empty (if type environment is empty, there are no pointers in the program, the heap memory is not used and garbage collector is never called).

3.1.2 GC Constants

In this section we define several data entities, which are considered as constants by the garbage collector. Some of these entities are program dependent and are fixed by the compiler on the initialization stage.

Definition 3.1 (**Frame header offsets**)**.** The garbage collector stores some data in the frame header of local memory frames. The displacement of the field in the frame header of the local frame, which keeps the index of the frame in the TT table (a link to the TT table) is represented by the constant $off_{TT} :: \mathbb{N}$

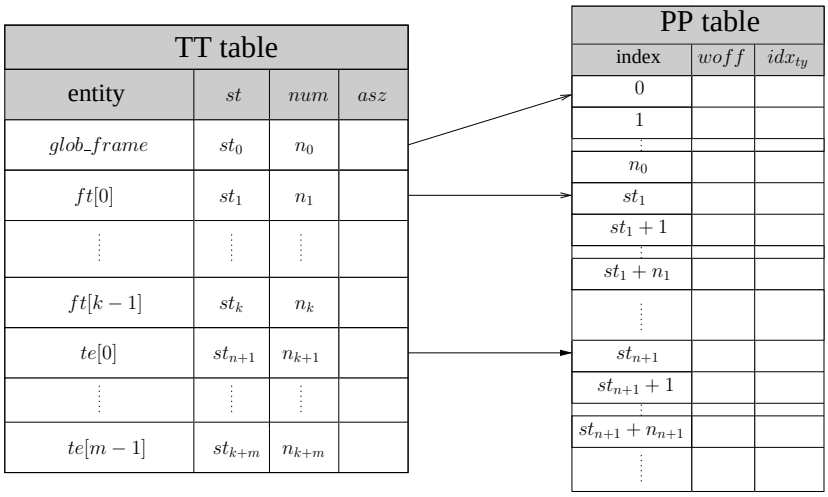

Figure 3.1: Structure of the TT and PP tables, where $k = |ft|$ and $m = |te|$.

. I.e. local frame l with the allocated word base address ba_w stores this link at address $ba_w + off_{TT}$. The displacement of the field in the header which keeps the start address of the previous stack frame (previous stack pointer) is denoted by $off_{psp} :: \mathbb{N}$. In the frame of this thesis we use the following definition.

$$off_{psp} = 2$$
$$off_{TT} = 4$$

Definition 3.2 (**Node header size**)**.** If the program is translated by compiler with the garbage collector each node on the heap has a header which stores information used by GC routines for correct garbage collection (Sec. 3.2.3). We represent the size of this header in words by the constant $AUX_{size} :: \mathbb{N}$. In the frame of this thesis we define

$$AUX_{size} = 2.$$

Definition 3.3 (**Number of functions**)**.** We represent the number of functions in the program, including the main function, by the constant $num_f :: \mathbb{N}$. For the function table ft we define

$$num_f = |ft| + 1$$

The value of the null pointer in memory is defined by the constant $NPtr = 0$.

Below is the list of other constants used by GC. We instantiate these constants in Sect. 4.2.

- $HHS :: \mathbb{N}$ - size of the half heap of the program memory in words; the size of the whole heap memory is defined as $HS = 2 \cdot HHS$,
- $MW :: \mathbb{N}$ - size of a word in bytes (fix it with 4 later in the thesis),
- $LMS :: \mathbb{N}$ - size of the local memory stack in words,
- $GMS :: \mathbb{N}$ - size of the global memory in words,
- $FHS :: \mathbb{N}$ - size of the local frame header in words.

3.1.3 Properties of GC Interface

In order for GC to perform correctly several properties of GC interface should hold. These properties were formulated by E.Petrova and are included in the precondition of the GC routine specification in Sec. 3.2.

Definition 3.4 (**Pointers inside frame**)**.** For every pointer from the PP table offset *woff* should be less than the allocated size of the frame where this pointer is located.

$$\begin{aligned} inframe_{ptrs}(TT, PP) \quad = \forall i\ j. \quad & j \in (TT_i.st : TT_i.st + TT_i.num - 1) \wedge \\ & i < |TT| \longrightarrow PP_i.woff < TT_j.asz \end{aligned}$$

Definition 3.5 (**Size is positive**)**.** Allocated size of all types and frames in the TT table cannot be negative. Moreover, for local frames and types it should be greater than zero.

$$\begin{aligned} asize_{pos}(TT, PP) = \quad \forall i. \quad & i = 0 \longrightarrow TT_i.asz \geq 0 \wedge \\ & i \in (1 : |TT| - 1) \longrightarrow TT_i.asz > 0 \end{aligned}$$

Definition 3.6 (**Link to pointer table is correct**)**.** The sub-array of pointers describing an element of the TT table has to fit in the PP table.

$$plink_{inside}(TT, PP) = \forall i.\ i < |TT| \longrightarrow TT_i.st + TT_i.num \leq |PP|$$

Definition 3.7 (**No pointers in frame header**)**.** The offset of the first pointer located in the local memory, as well as the allocated size of the local memory frame, should be greater than off_{TT}.

$$\begin{aligned} woff_{ge}(TT, PP) \quad = \forall i. \quad & i \in (1 : num_f - 1) \longrightarrow \\ & (TT_i.num > 0 \longrightarrow PP_{TT_i.st}.woff > off_{TT}) \wedge \\ & (TT_i.num = 0 \longrightarrow TT_i.asz > off_{TT}) \end{aligned}$$

Definition 3.8 (**Offsets are ordered**)**.** Offsets of pointers from one type or one memory frame are ordered.

$$\begin{aligned} woff_{ord}(TT, PP) \quad = \forall i\ j\ k. \quad & i < |TT| \wedge j \in (TT_i.st : TT_i.st + TT_i.num - 1) \wedge \\ & k \in (TT_i.st : TT_i.st + TT_i.num - 1) \wedge k < j \\ & \longrightarrow PP_k.woff < PP_j.woff \end{aligned}$$

Definition 3.9 (**Correct interface**)**.** Now we combine the above properties of the GC interface into one predicate. Additionally, we require that length of the TT table is greater than the number of functions in the program, i.e. that at least one type is present in the type environment.

$$\frac{\begin{array}{ll} inframe_{ptrs}(TT, PP) & asize_{pos}(TT, PP) \\ plink_{inside}(TT, PP) & woff_{ge}(TT, PP) \\ woff_{ord}(TT, PP) & |TT| > num_f \end{array}}{correct_{gci}(TT, PP)}$$

The above property follows from the interface construction. We present the implementation model for the type and pointer tables and prove predicate $correct_{gci}$ in Sec. 4.2.

3.2 GC Specification

In this section we present the specification of the C0 heap memory allocation procedure with garbage collection done by E.Petrova. This specification describes the behavior of a simple copying garbage collector. In the frame of this thesis we do not fix the exact implementation of the garbage collector and the use of any routine satisfying the given specification is appropriate.

3.2.1 Basics

The allocated address of the heap in bytes is denoted by $abase_{heap} :: \mathbb{N}$. We use $f :: \mathbb{B}$ to denote which half of the heap is currently utilized by the program. If $f = False$ then program is using the first part of the heap (from $abase_{heap}$ to $abase_{heap} + HHS \cdot MW$). Otherwise the second part of the heap is used (from $abase_{heap} + HHS \cdot MW$ to $abase_{heap} + HS \cdot MW$). The part of the heap used by the program is from-space. The remainder part is to-space.

Definition 3.10 (**Points to heap**)**.** We distinguish a pointer to the heap from other pointers with the help of the predicate $p2h :: \mathbb{N} \times \mathbb{N} \to \mathbb{B}$.

$$p2h(a) = abase_{heap} \leq a < abase_{heap} + HS \cdot MW$$

Definition 3.11 (**Half heap start**)**.** Displacement of the currently used half of the heap with respect to heap base address is calculated by the function $st_{idx} :: \mathbb{B} \to \mathbb{N}$.

$$st_{idx}(f) = \begin{cases} HHS & \text{if } f \\ 0 & \text{otherwise} \end{cases}$$

Definition 3.12 (**Index in half heap**)**.** To test whether some heap index hi belongs to the from- or to-space we use the following predicates.

$$in_{from}(f, hi) = \begin{cases} hi \in (HHS : HS - 1) & \text{if } f \\ hi < HHS & \text{otherwise} \end{cases}$$

$$in_{to}(f, hi) = \begin{cases} hi < HHS & \text{if } f \\ hi \in (HHS : HS - 1) & \text{otherwise} \end{cases}$$

We also define predicate in_{tol} to test whether a number belongs to the to-space including the border-index.

$$in_{tol}(f, hi) = \begin{cases} hi \leq HHS & \text{if } f \\ hi \in (HHS : HS) & \text{otherwise} \end{cases}$$

The garbage collector routine considers local, global, and heap memories of the compiled program as arrays of natural numbers. In order to distinguish a memory cell from other natural numbers in the frame of this thesis we introduce type $mcell_t$.

$$mcell_t = \mathbb{N}$$

We also use type mem_t to represent local, global, and heap memories of the program.

$$mem_t = mcell_t \; list$$

3.2.2 Abstract Stack

Definition 3.13 (**Displacement of the frame**). Local memory LM consists of local memory frames. The displacement of the local frame in the local memory LM is computed from the information stored in TT table. Function $displ_{frame} :: TT_{type} \times mem_t \times \mathbb{N} \to \mathbb{N}$ is defined by induction on the ordinal number of the frame in the local memory stack. Function $fid_{TT} :: TT_{type} \times mem_t \times \mathbb{N} \to \mathbb{N}$ returns a link to the TT table stored in the local memory frame (we define the function later in this section). For the induction case the displacement of the $(i+1)-$ th frame is calculated as the sum of the displacement of the $i-$the frame and the allocated size of the $i-$th frame.

$$\begin{aligned} displ_{frame}(TT, LM, 0) &= 0 \\ displ_{frame}(TT, LM, i+1) &= displ_{frame}(TT, LM, i) + \\ & \quad TT_{fid_{TT}(TT,LM,i)}.asz \end{aligned}$$

The memory cell in the local memory LM with displacement

$$displ_{frame}(TT, LM, i) + off_{TT}$$

stores a link to the TT table for the $i-$th memory frame. Note, that every memory frame in the program memory stack has a respective function in the function table ft. Thus, for every frame a link to a corresponding record in the TT table can be provided. The value of this link is denoted by $fid_{TT}(TT, LM, i)$.

$$fid_{TT}(TT, LM, i) = LM!(displ_{frame}(TT, LM, i) + off_{TT})$$

In an analogous way we use function $fid_{PSP}(TT, LM, i)$ to get the value of the previous stack pointer.

$$fid_{PSP}(TT, LM, i) = LM!(displ_{frame}(TT, LM, i) + off_{PSP})$$

We model the stack of the local memories of the program with help of the list of displacements of memory frames inside local memory:

$$stack :: \mathbb{N}\ list.$$

To establish the connection between *stack* and local memory LM we introduce an abstraction relation *Stack*, which fixes the following properties:

- *stack* contains at least one frame,
- the displacement of the last frame fits into $|LM|$,
- the program stack pointer points to the last frame from the *stack*,
- *stack*!i contains the displacement of the i-th frame from the local memory stack,
- the previous stack pointer of the $i-$th frame points to the $(i-1)-$th frame.

Definition 3.14 (**Stack**)**.** Here and below, ss and sp are stack start address and program stack pointer, respectively. We define predicate $Stack :: TT_{type} \times \mathbb{N} \times \mathbb{N} \times mem_t \times (\mathbb{N}\ list) \to \mathbb{B}$ in the following way:

$$\frac{\begin{array}{l} |stack| > 0 \\ displ_{frame}(TT, LM, |stack|) \leq |LM| \\ sp = ss + (displ_{frame}(TT, LM, |stack| - 1)) \cdot MW \\ \forall i < |stack|.\ stack!i = displ_{frame}(TT, LM, i) \wedge \\ \qquad fid_{TT}(TT, LM, i) \in (1 : num_f - 1) \wedge \\ \qquad (i > 0 \longrightarrow fid_{PSP}(TT, LM, i) = ss + stack!(i-1) \cdot MW) \end{array}}{Stack(TT, ss, sp, LM, stack)}$$

3.2.3 Abstract Heap

Analogously to Def. 3.13 we define the displacement of a node in the heap memory HM.

Definition 3.15 (**Displacement of a node**)**.** Function $displ_{node} :: TT_{type} \times mem_t \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is defined by induction on the ordinal number of the node in the heap memory. Analogously to the frame displacement definition, function $fid_{TT} :: TT_{type} \times mem_t \times \mathbb{N} \to \mathbb{N}$ returns a link to the TT table stored in the node header. The size of the node header is not included into the allocated size of the node and we have to add it explicitly. Offset st denotes the start index of the current half of the heap (either 0 or HHS).

$$\begin{array}{rcl} displ_{node}(TT, HM, st, 0) & = & st \\ displ_{node}(TT, HM, st, i+1) & = & displ_{node}(TT, HM, i) + AUX_{size} \\ & & +TT_{nid_{TT}(TT,HM,i)}.asz \end{array}$$

The link to the TT table for the $i-$th node is obtained by the function nid_{TT}.

$$nid_{TT}(TT, HM, i) = HM!displ_{node}(TT, HM, i)$$

The second word in the node header is called the forward pointer. It is used by the garbage collector implementation to store some auxiliary information during execution of the GC routines. The function $nid_{fwd}(TT, HM, i)$ obtains the forward pointer.

$$nid_{fwd}(TT, HM, i) = HM!(displ_{node}(TT, HM, i) + 1)$$

The program heap is modeled with the help of the list of displacements of nodes inside the heap memory (Fig. 3.2), which we call an *abstract heap*. Here and below in this thesis we denote the abstract heap by

$$xs :: \mathbb{N}\ list.$$

The connection between abstract heap xs and the heap memory HM is established by the abstract relation *HalfHeap*.

Definition 3.16 (**HalfHeap**)**.** Let $nhi :: \mathbb{N}$ be the last index in the heap memory used by the program. We require that the $i-$th element in the abstract

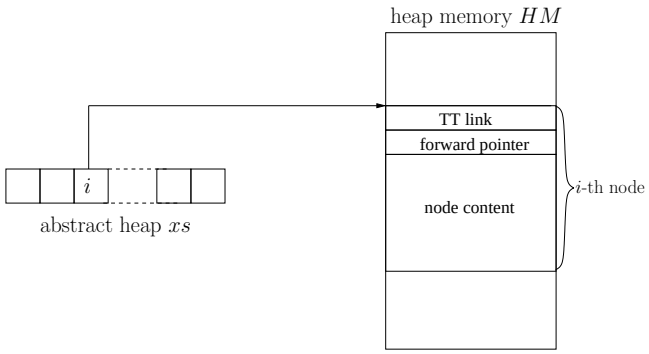

Figure 3.2: Single node in the program and abstract heap.

heap xs contains the displacement of the $i-$th node in the heap memory HM, the TT link of the $i-$th node points to some element in the type table, and the forward pointer in the node points to some address inside the heap memory (technical requirement of the used GC implementation). Moreover, we require that nhi equals the displacement of the last node plus the size of the node and this address lies in the to-space of the heap memory. Formally, we define the predicate $\mathit{HalfHeap} :: TT_{type} \times mem_t \times \mathbb{B} \times \mathbb{N} \times (\mathbb{N}\ list) \to \mathbb{B}$ in the following way:

$$\frac{\begin{array}{l} nhi = \mathit{displ}_{node}(TT, HM, \mathit{st}_{idx}(f), |xs|) \\ \mathit{in}_{tol}(f, nhi) \\ \forall i < |xs|.\ xs!i = \mathit{displ}_{node}(TT, HM, \mathit{st}_{idx}(f), i) \wedge \\ \qquad \mathit{nid}_{TT}(TT, HM, i) \in (1 : |TT| - 1) \wedge \\ \qquad \mathit{nid}_{fwd}(TT, HM, i) \leq |HM| \end{array}}{\mathit{HalfHeap}(TT, HM, f, nhi, xs)}$$

3.2.4 Roots and Reachable Nodes

The garbage collector uses information from the PP table to find all reachable variables in the program. Pointers located in the local or global memory are called root pointers or *roots*. GC reads displacements of the root pointers directly from the PP table. After that, if a pointer in *roots* points to some heap node n, GC finds a link to the TT table stored in the node header of n. From the TT table GC gets access to all pointers of node n and performs the next step of pointer traversal. If some pointer points to a global variable v (pointers to local variables are not supported by C0 language), then GC does not need to handle it explicitly since all pointers in v are already present in the *roots* set.

In this section and later in this thesis we omit constant parameters TT and PP in the definitions.

Definition 3.17 (Pointer offsets extraction). Function $\mathit{ptrs}_{\mathit{off}}(i)$ extracts displacements of all pointers for the $i-$th element of the TT table.

$$\mathit{ptrs}_{\mathit{off}}(i) = \mathit{map}(\mathit{off}, PP[TT_i.\mathit{st}, TT_i.\mathit{num}])$$

Definition 3.18 (**Pointer content extraction**)**.** The content of pointers for the $i-$th element of the TT table from memory M is extracted by the function $ptrs_{cnt}(M, i, st)$. Extraction is performed starting from the memory index st (e.g. st can be equal to the start index of some local frame inside the local memory LM) and is based on the pointer offsets information.

$$ptrs_{cnt}(M, i, st) = map((\lambda x.\ M[st, |M| - st]!x), ptrs_{off}(i))$$

Definition 3.19 (**Global roots**)**.** We define function $roots_{gm} :: TT_{type} \times PP_{type} \times mem_t \to \mathbb{N}\ list$ which extracts the content of the global roots from the memory.

$$roots_{gm}(GM) = ptrs_{cnt}(GM, 0, 0)$$

Definition 3.20 (**Local roots**)**.** Function $roots_{lmi}(TT, PP, LM, st)$ extracts content of the roots of a single memory frame with displacement st.

$$roots_{lmi}(LM, st) = ptrs_{cnt}(LM, LM!(st + off_{TT}), st)$$

The content of the roots of the whole local memory stack is obtained by the function $roots_{lm}(LM, stack)$.

$$roots_{lm}(LM, stack) = concat(map(\lambda i.\ roots_{lmi}(LM, stack!i), (0 : |stack|-1))$$

Later on in this thesis we denote the set of program roots by $roots_m$.

$$roots_m = roots_{gm}(GM) \circ roots_{lm}(LM, stack)$$

Definition 3.21 (**Content of heap pointers**)**.** The following function extracts the content of pointers for the node i in the abstract heap xs.

$$ncnt_{ptrs}(HM, i, xs) = ptrs_{cnt}(HM, HM!(xs!i), xs!i + AUX_{size})$$

Definition 3.22 (**Address node**)**.** The predicate $node_{addr}(xs, a, i)$ states that address a is the start address of the node i in xs.

$$\frac{i < |xs| \qquad a = abase_{heap} + (xs!i + AUX_{size}) \cdot MW}{node_{addr}(xs, a, i)}$$

Definition 3.23 (**Node edge**)**.** Predicate $edge_{node}$ states that there exists a pointer in the node i which points to the node j.

$$\frac{a \in set(ncnt_{ptrs}(HM, i, xs)) \qquad p2h(a) \qquad node_{addr}(xs, a, i)}{edge_{node}(HM, xs, i, j)}$$

Now we can define the set of all reachable heap nodes in the program.

Definition 3.24 (**Reachable nodes**)**.** The set of reachable nodes is defined inductively. For the base case node n is reachable iff some pointer from the list of $roots$ points to n.

$$\frac{x \in roots \qquad p2h(x) \qquad node_{addr}(xs, x, n)}{n \in RN(HM, roots, xs)}$$

For the induction case node with index j is reachable iff there exists some node i which contains a pointer to node j.

$$\frac{i \in RN(HM, roots, xs) \qquad edge_{node}(HM, xs, i, j)}{j \in RN(HM, roots, xs)}$$

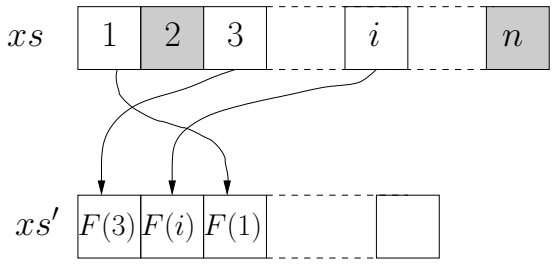

Figure 3.3: Abstract heap before and after garbage collection.

Definition 3.25 (**Roots**)**.** We use predicates

$$Roots(roots, xs)$$
$$heap\_Roots(HM, xs)$$

to denote that every pointer to the heap points to some node from the abstract heap xs, i.e. there are no pointers to fields of structures or elements of the array. The first predicate establishes the given above property for pointers in $roots$. The second one for all pointers to the heap located in the heap nodes. We omit formal definition of these predicates here.

3.2.5 Isomorphism

After garbage collection the non-pointer content of the local and global memories is not changed. The non-pointer content of a reachable heap node also remains unchanged, but the node itself is moved to a new place. We denote the abstract heap after garbage collection by xs'. The order of the nodes in the new heap in general differs from the order of the nodes in the initial one. This happens because the garbage collection algorithm is not intended to preserve order of the nodes on the heap. The new order is defined with the help of a bijective permutation function F. If some node with index i in the abstract heap is reachable, then after garbage collection this node has index $F(i)$ on the heap xs' (Fig. 3.3). In this section we define the isomorhpism relation on the heap which describes the impact of the garbage collection to the heap and to the roots.

We introduce two functions which extract some content for the $i-$th element of the TT table from the memory M starting from the offset st.

$$nhptrs_{cnt}(M, i, st) :: mcell_t\ list$$
$$noptrs_{cnt}(M, i, st) :: mcell_t\ list$$

The first one extracts pointers which do not point to the heap. The second extracts the whole content from st to $TT_i.asz$ except pointers. The formal definition of these function involves filtering and is analogues to Def. 3.18. We omit it here.

Definition 3.26 (**Content extraction**)**.** The following two functions extract pointers which do not point to the heap and the non-pointer content from the

node i respectively.

$$ncnt_{nhptrs}(HM, i, xs) = nhptrs_{cnt}(HM, HM!(xs!i), xs!i + AUX_{size})$$
$$ncnt_{noptrs}(HM, i, xs) = noptrs_{cnt}(HM, HM!(xs!i), xs!i + AUX_{size})$$

We define analogous functions for the global and local memory frames.

$$gmcnt_{nhptrs}(GM) = nhptrs_{cnt}(GM, 0, 0)$$
$$gmcnt_{noptrs}(GM) = noptrs_{cnt}(GM, 0, 0)$$
$$lmicnt_{nhptrs}(LM, st) = nhptrs_{cnt}(LM, LM!(st + off_{TT}), st)$$
$$lmicnt_{noptrs}(LM, st) = noptrs_{cnt}(LM, LM!(st + off_{TT}), st)$$

The functions which perform the given above extractions for the whole local memory LM are defined as follows.

$$lmcnt_{nhptrs}(LM, stack) = concat(map(\lambda i.\ lmicnt_{nhptrs}(LM, stack!i), (0 : |stack| - 1))$$
$$lmcnt_{noptrs}(LM, stack) = concat(map(\lambda i.\ lmicnt_{noptrs}(LM, stack!i), (0 : |stack| - 1))$$

Now we have all we need to establish equality of the node content before and after garbage collection. Here and below by HM', LM', GM', xs', f', etc. we denote memories and other entities after garbage collection is performed. Note, that some entities, e.g. abstract stack *stack*, remain unchanged during garbage collection.

Definition 3.27 (**Node content equality**)**.** The following predicate establishes content equality of the heap nodes before and after garbage collection (everything except pointers to the heap remains unchanged).

$$\frac{\begin{array}{c}\forall i \in RN(HM, roots, xs).\ HM!(xs!i) = HM'!(xs!i) \wedge \\ ncnt_{nhptrs}(HM, i, xs) = ncnt_{nhptrs}(HM, F(i), xs') \wedge \\ ncnt_{noptrs}(HM, i, xs) = ncnt_{noptrs}(HM, F(i), xs')\end{array}}{content_{eq}(TT, PP, heap_{base}, roots, HM, HM', xs, xs', F)}$$

Definition 3.28 (**Reachable equality**)**.** Let $roots'$ be the set of program roots after garbage collection. One of the basic properties of the garbage collector ensures that the set of the reachable nodes remains unchanged during garbage collection. I.e. if some node i is reachable before garbage collection, then node $F(i)$ is reachable after it. Formally we define it in the following way.

$$\frac{\begin{array}{c}RN(HM', roots', xs') = \\ map(F, RN(HM, roots, xs))\end{array}}{reachable_{eq}(roots, roots', HM, HM', xs, xs', F)}$$

Isomorphism on the set of the program roots is defined by the predicate

$$isomorph_{roots}(roots, roots', xs, xs', F).$$

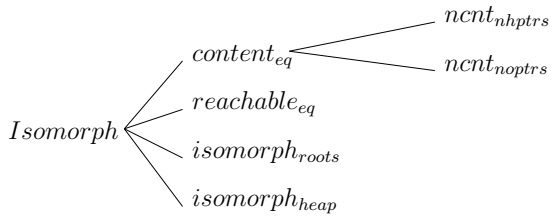

Figure 3.4: Structure of the isomorphic relation.

It requires that every pointer in *roots* which points to some heap node before garbage collection points to the same heap node (w.r.t. F) after garbage collection. Predicate

$$isomoprh_{heap}(roots, roots', HM, HM', xs, xs', F)$$

establishes an analogues property for pointers to the heap located in reachable heap nodes. We do not give formal definition for the given above predicates in this thesis.

Definition 3.29 (**Isomorph**)**.** We combine the given above isomorphism definitions in one predicate. Generally speaking, this predicate describes the permutation function F, which defines the order of the nodes on the heap after garbage collection (Fig. 3.4).

$$\frac{\begin{array}{l} content_{eq}(roots, HM, HM', xs, xs', F) \\ reachable_{eq}(roots, roots', HM, HM', xs, xs', F) \\ isomorph_{roots}(roots, roots', xs, xs', F) \\ isomoprh_{heap}(roots, roots', HM, HM', xs, xs', F) \end{array}}{Isomorph(roots, roots', HM, HM', xs, xs', F)}$$

3.2.6 Garbage Collection

First we define the predicate which establishes the equality of the content of local and global memories after garbage collection.

Definition 3.30 (**Memories equality**)**.** Everything in local and global memories except pointers to the heap is not changed.

$$\frac{\begin{array}{c} gmcnt_{noptrs}(GM) = gmcnt_{noptrs}(GM') \\ gmcnt_{nhptrs}(GM) = gmcnt_{nhptrs}(GM') \\ lmcnt_{nhptrs}(LM, stack) = lmcnt_{nhptrs}(LM', stack) \\ lmcnt_{nhptrs}(LM, stack) = lmcnt_{nhptrs}(LM', stack) \end{array}}{memcont_{eq}(GM, GM', LM, LM', stack)}$$

From these definitions we can also derive the equality of local frame headers. However, for simplicity reasons, we introduce a separate predicate which states the mentioned property.

Definition 3.31 (**Stable frame headers**).

$$\frac{\forall\, i < |stack|.\ LM[stack!i, FHS] = LM'[stack!i, FHS]}{fheaders_{stable}(LM, LM', stack)}$$

Let us denote the set of memory components of the program by m_σ.

$$m_\sigma = \{GM, LM, HM\}$$

Further, we denote the set of variables of the GC interface, which are not changed during garbage collection by inf_σ

$$inf_\sigma = \{TT, PP, ss, sp, heap_{base}\},$$

where ss is the start address of the stack of local memories and sp is a current program stack pointer.

Now we can define the result of the garbage collection routine. The sketch of the heap memory transformation is shown on Fig. 3.5. After garbage collection the following properties hold:

- lengths of the memories remain unchanged,
- all pointers in *roots* and in the reachable heap nodes point to some element in the new abstract heap xs' (Def. 3.25),
- the abstract heap xs' is connected to the new heap memory HM' via the *HalfHeap* predicate (Def. 3.16),
- the abstract stack *stack* is connected to the new local memory LM' via the *Stack* predicate (Def. 3.14),
- function F is a bijection and isomorphic relation on the heap holds with respect to F (Def. 3.29),
- all nodes in the new heap are reachable,
- to-space and from-space are switched,
- the content of local and global memories which does not contain pointers to the heap remains unchanged (Def. 3.30),
- the content of local frame headers remains unchanged (Def. 3.31),
- forward pointers in the heap nodes do not point to to-space (technical requirement for the used implementation of the garbage collector).

Definition 3.32 (**GC result**). Let xs and *stack* be the abstract heap and the stack of the program respectively . Further, let $nhi :: nat$ be the value of the pointer to the next free heap location (next heap index pointer) and $F :: \mathbb{N} \to \mathbb{N}$ be a permutation relation. By $roots'_m$ we denote the set of program roots

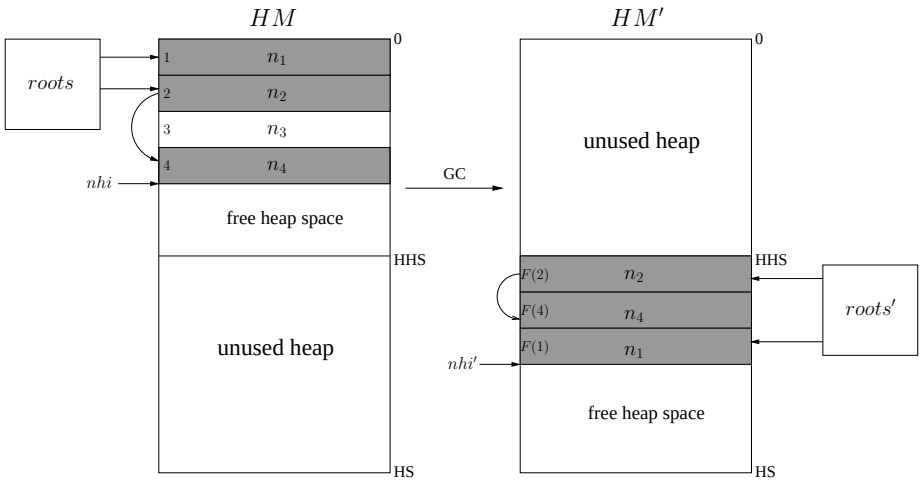

Figure 3.5: Heap memory before and after garbage collection.

after garbage collection, i.e. $roots'_m = roots_{gm}(GM') \circ roots_{lm}(LM', stack)$. We define the result of the GC execution in the following predicate.

$$\frac{\begin{array}{c} |HM| = |HM'| \qquad |LM| = |LM'| \qquad |GM| = |GM'| \\ Roots(roots_{gm}(GM'), xs') \qquad Roots(roots_{lm}(GM', stack), xs') \\ heap\_Roots(HM', xs') \\ HalfHeap(TT, HM', f', nhi', xs') \qquad Stack(TT, ss, sp, LM', stack) \\ bij(F) \qquad Isomorph(roots_m, roots'_m, HM, HM', xs, xs', F) \\ RN(HM', roots'_m, xs') = (0 : |xs'| - 1) \qquad f' = (\neg f) \\ memcont_{eq}(GM, GM', LM, LM', stack) \qquad fheaders_{stable}(LM, LM', stack) \\ \forall i < |xs'|.\ \neg in_{to}(f', nid_{fwd}(TT, HM', i)) \end{array}}{gc\_performed(inf_\sigma, m_\sigma, m'_\sigma, nhi, nhi', f, f', stack, xs, xs', F)}$$

3.2.7 Memory Allocation

In this Section we formalize the result of the redefined memory allocation procedure (*new* function). We distinguish three cases:

- there is enough memory on the heap for allocation of the new variable and no garbage collection is needed,
- before garbage collection there is not enough memory on the heap, but after the call of GC routine the freed space is enough for the allocation of the new variable,
- even after garbage collection there is not enough heap space for the new heap variable allocation.

The GC routine is called in the second and the third cases. Thus, garbage collection is performed even if the new heap variable cannot be allocated after its run.

Precondition

Here we define the precondition for the correct execution of the garbage collection and redefined memory allocation routines.

Definition 3.33 (**Collect garbage precondition**)**.** The precondition for the garbage collection routine is defined in the following way.

$$\frac{\begin{array}{cc} Roots(roots_{gm}(GM), xs) & Roots(roots_{lm}(LM, stack), xs) \\ heap\_Roots(heap_{base}, HM, xs) & HalfHeap(TT, HM, f, nhi, xs) \\ Stack(TT, ss, sp, LM, stack) & \forall i < |xs|.\ \neg in_{to}(f, nid_{fwd}(TT, HM, i)) \end{array}}{collect_{pre}(inf_\sigma, m_\sigma, nhi, f, xs, stack)}$$

Definition 3.34 (**Memory allocation precondition**)**.** Precondition for the redefined *new* function includes predicates defined in Def. 3.33 and Def. 3.9. Additionally, we set requirements on the valid ranges of some addresses and lengths of the memories. We require that the size of the first frame in the TT table equals the size of the GM memory and the last local frame is located inside LM memory. Type and pointer tables need $(|TT| \cdot 3 + |PP| \cdot 2) \cdot MW$ bytes in the global memory for storage. We want stack start ss to be greater or equal than the interface data storage size. Let $asz :: nat$ be the size of the new variable in words, t_{id} be the index of the type of the new variable (i.e. variable to be allocated) in the TT table. Formally we define the memory allocation precondition in the following way.

$$\frac{\begin{array}{c} correct_{gci}(TT, PP) \qquad collect_{pre}(inf_\sigma, m_\sigma, nhi, f, xs, stack) \\ (asz + AUX_{size}) \in \mathbb{N}_{32} \qquad t_{id} < |TT| \qquad TT_{t_{id}}.asz = asz \\ (abase_{heap} + HS \cdot MW) \in \mathbb{N}_{32} \qquad ss + LM \cdot MW \leq abase_{heap} \qquad ss \leq sp \\ |GM| = GMS \qquad |LM| = LMS \qquad |HM| = HS \qquad MW \mid ss \qquad MW \mid sp \\ TT_0.asz = |GM| \qquad \frac{(sp - ss)}{4} + TT_{fid_{TT}(TT, LM, |stack|-1)}.asz \leq |LM| \\ (|TT| \cdot 3 + |PP| \cdot 2) \cdot MW \leq ss \end{array}}{new_{pre}(inf_\sigma, m_\sigma, nhi, f, asz, t_{id}, xs, stack)}$$

Postcondition

The result of the call of the modified *new* procedure in the case of memory allocation without garbage collection is expressed in a predicate

$$alloc_{post}(inf_\sigma, m_\sigma, m'_\sigma, f, nhi, nhi', xs, asz, res, t_{id}).$$

In this case local and global memories remain unchanged, and the next heap index pointer is increased by the size of the new variable (asz). The region in the heap memory which corresponds to the newly allocated variable is filled with zeros. The link to the TT table t_{id} is written to the header of the new node. The remaining part of the heap is unchanged. The resulting pointer res equals the start address of the new allocated node, i.e.

$$res = heap_{base} + (nhi + AUX_{size}) \cdot MW.$$

We do not focus on the case of memory allocation without garbage collection in this thesis and omit formal definition of the above predicate.

Definition 3.35 (**Unsuccessful gc**)**.** In the case of unsuccessful garbage collection the new heap variable cannot be allocated and we return a null pointer as the result of memory allocation.

$$\frac{gc\_performed(inf_\sigma, m_\sigma, m'_\sigma, nhi, nhi', f, f', stack, xs, xs', F) \quad res = NPtr}{gc\_failure(inf_\sigma, m_\sigma, m'_\sigma, nhi, nhi', f, f', stack, xs, xs', F, res)}$$

Definition 3.36 (**Successful gc**)**.** In case of a successful garbage collection the new heap variable is allocated. We denote the state of memory M and variable a after allocation of the new heap node by M'' and a''. Thus, we distinguish three states: the initial one, the state after garbage collection is performed, and the state when the new variable is allocated. The value of the next heap index pointer nhi'' is increased by the size of the new variable (asz). The new heap memory region is filled with zeroes and the address of the new variable is assigned to res. Additionally, we require that all relations included in Def. 3.32 also hold for the new memories and new abstract heap xs''. Let t_{id} be a link to the TT table for the new heap variable. We define the result of successful garbage collection in the following way.

$$\frac{\begin{array}{c} gc\_performed(inf_\sigma, m_\sigma, m'_\sigma, nhi, nhi', f, f', stack, xs, xs', F) \\ res = heap_{base} + (nhi + AUX_{size}) \cdot MW \\ H''[(nhi + AUX_{size}), asz] = replicate(asz, 0) \qquad H''!nhi' = t_{id} \\ ncnt_{eq}(HM', HM'', xs') \qquad xs'' = xs' \circ [nhi'] \qquad nhi'' = nhi + asz + AUX_{size} \\ Roots(roots_{gm}(GM''), xs'') \qquad Roots(roots_{lm}(GM'', stack), xs'') \\ heap\_Roots(HM'', xs'') \qquad HalfHeap(HM'', f', nhi'', xs'') \\ Isomorph(roots_m, roots''_m, HM, HM'', xs, xs'', F) \\ RN(HM', roots', xs') = (0 : |xs'| - 1) \\ LM'' = LM' \qquad GM'' = GM' \qquad |HM''| = |HM'| \\ \forall i < |xs''|.\ \neg in_{to}(f', nid_{fwd}(TT, HM'', i)) \end{array}}{gc\_ok(inf_\sigma, m_\sigma, m'_\sigma, m''_\sigma, nhi, nhi', nhi'', f, f', stack, xs, xs', xs'', F, asz, ti, res)}$$

Definition 3.37 (**Size of the nodes**)**.** In order to distinguish cases when garbage collection is or is not successful we define the following predicate, which calculates the size of all reachable nodes on the heap.

$$rnodes_{size}(TT, PP, heap_{base}, HM, roots, xs) = \sum_{i \in RN(TT, PP, heap_{base}, HM, roots, xs)} TT_{nid_{TT}(HM, xs, i)}.asz + AUX_{size}$$

Definition 3.38 (**Memory allocation postcondition**)**.** Now we can define

the postcondition of the memory allocation routine.

$$nhi + AUX_{size} + asz \leq st_{idx}(f) + HHS \longrightarrow xs'' = xs \circ [nhi] \wedge f' = f \wedge$$
$$alloc_{post}(inf_\sigma, m_\sigma, m''_\sigma, f, nhi, nhi'', xs, asz, res, t_{id})$$

$$nhi + AUX_{size} + asz > st_{idx}(f) + HHS \wedge$$
$$rnodes_{size}(TT, PP, heap_{base}, HM, roots, xs) + AUX_{size} + asz \leq HHS \longrightarrow$$
$$gc\_ok(inf_\sigma, m_\sigma, m'_\sigma, m''_\sigma, nhi, nhi', nhi'', f, f', stack, xs, xs', xs'', F, asz, ti, res)$$

$$\frac{rnodes_{size}(TT, PP, heap_{base}, HM, roots, xs) + AUX_{size} + asz > HHS \longrightarrow \quad gc\_failure(inf_\sigma, m_\sigma, m''_\sigma, nhi, nhi'', f, f', stack, xs, xs', F, res)}{new_{post}(inf_\sigma, m_\sigma, m'_\sigma, m''_\sigma, nhi, nhi', nhi'', f, f', stack, xs, xs', xs'', asz, ti, res)}$$

Correctness criteria

Correctness criteria for the redefined memory allocation routine is expressed in Theorem 3.1.

Theorem 3.1 (**Correct garbage collection**)**.** Let m_σ represent the program memories before execution of the *new* procedure and m''_σ represent memories after execution of the procedure. Further, let nhi, nhi'' be values of the program next heap index pointers, f and f' denote the half heap used by the program before and after procedure execution. Then, the following relation holds.

$$new_{pre}(inf_\sigma, m_\sigma, nhi, f, asz, t_{id}, xs, stack) \Longrightarrow$$
$$\exists xs'\ xs''\ nhi'\ m'_\sigma\ .$$
$$new_{post}(inf_\sigma, m_\sigma, m'_\sigma, m''_\sigma, nhi, nhi', nhi'', f, f', stack, xs, xs', xs'', asz, ti, res)$$

Proof. The lemma has been proven in the Hoare logic environment by E. Petrova for a particular implementation of GC. We do not provide the proof in the frame of this thesis. □

Chapter 4

Compiler with a Garbage Collector

In this chapter we consider the formal specification and verification of a simple compiler with garbage collector. The basic definitions and concepts about the compiler are taken from [12]. We mainly focus on the integration of the garbage collector into the compiler and do not provide the full compiler specification here.

The garbage collector is nothing else than a set of procedures together with some specific variables located in the global memory of the program. During compilation, these procedures are compiled in the same way as regular program functions and are added to the program's function table. Variables of the garbage collector are placed in the global memory just after regular global variables of the program. Assembly code, generated for the memory allocation statement, is replaced with a function call to the main GC routine.

4.1 Memory Layout

The memory layout for the program generated by the C0 compiler with the garbage collector is inherited from the memory layout of the original compiler. It consists of four major areas: the part with the program code, global memory frame, stack of local memories, and the heap. The basic memory layout is depicted on Fig. 4.1.

The program code part of the memory starts from the address *progbase*, which is a parameter of the compiler and remains constant during compilation. The size of the program code is computed with the function $csize_{prog}(te, gst, ft)$, which calculates the size of all functions from the function list (including the main function) and the size of the initialization code in words. Behind the program code there is an unused space of size $bubble_{code}$, which is reserved for later use.

The global memory frame starts at the address $abase_{gm}(te, ft, gst(m))$, which is calculated in the following way.

$$abase_{gm}(te, ft, gst) = progbase + 4 \cdot csize_{prog}(te, gst, ft) + 4 \cdot bubble_{code}$$

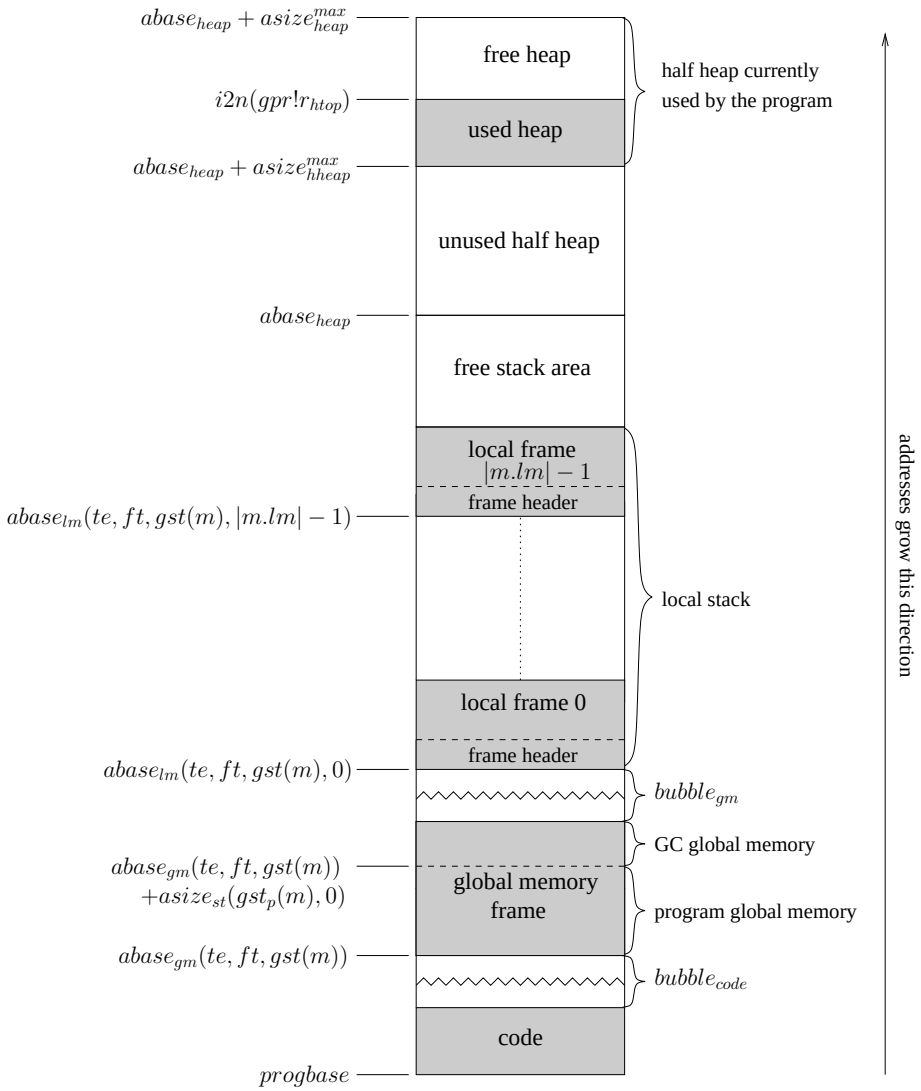

Figure 4.1: Memory layout of the compiler with GC.

We divide the global memory into two parts: one containing the program's global variables and one for the global variables of the GC. From the compiler view all variables located in the program or GC part of the global memory are treated equally. Thus, we have to require that a user program does not change the GC part of the global memory.

Formally, we divide the symbol table for the global memory frame gst (Sec. 2.3.4) into two parts:

- $gst_p(gst) :: (\mathbb{S} \times ty)\ list$ represents variables located in the program part of the global memory,
- $gst_{gc}(gst) :: (\mathbb{S} \times ty)\ list$ represents variables of the GC interface.

Let gci_{length} be a constant, denoting the number of variables belonging to the GC interface and gst be the symbol table of the global memory frame. Then we define

$$gst_p(gst) = gst[0, |gst| - gci_{length}]$$
$$gst_{gc}(gst) = gst[|gst| - gci_{length}, gci_{length}]$$

For the program symbol configuration sc and memory m we also use the following notation.

$$sc.gst_p = gst_p(sc.gst)$$
$$sc.gst_{gc} = gst_{gc}(sc.gst)$$
$$gst_p(m) = gst_p(gst(m))$$
$$gst_{gc}(m) = gst_{gc}(gst(m))$$

Behind the global memory a free space of the size $bubble_{gm}$ follows. The local memory stack starts at the address $abase_{lm}(te, ft, gst(m), 0)$. We give the formal definition for this predicate later in this section. Each local memory frame starts with the *frame header*, which occupies five words (in the original compiler without GC the frame header occupies 3 words). The meaning of these fields is given in Table 4.1. The program heap starts at the address $abase_{heap}$ (another input parameter to the compiler). The local stack grows from the bottom up and its size is limited (only) by the start address of the heap. In the compiler with the garbage collector the heap is divided into two parts of the equal size. All variables are allocated only in one half of the heap, the second stays inactive until the next garbage collection is performed. The size of the active half of the heap is bounded by the constant $asize^{max}_{hheap}$. The size of the whole heap equals

$$asize^{max}_{heap} = asize^{max}_{hheap} \cdot 2.$$

The used part of the heap is composed of the set of heap *nodes*. By a heap node we understand a top-level heap g-variable together with its node header.

The base address of the global memory, the first unused address on the heap, and the base address of the current stack frame are stored by the compiler in the registers $r_{sbase}, r_{htop}, r_{lframe}$, respectively.

Table 4.1: Frame Header Layout

Offset	Field Name	Field Meaning
0	return address	address, where to jump after execution of the function is finished
4	return destination	address, where to store the result of the function
8	previous stack pointer	start address of the previous stack frame
12	frame size	allocated size of the frame
16	TT link	stores the index of the corresponding entry in the TT table of the GC

4.1.1 Allocated Size

We define the function which calculates the number of bytes needed to store an instance of the type in the target machine, i.e. the *allocated size* of a type.

The function $algn :: ty \to \mathbb{N}$ calculates alignment of a type in the memory. This function was introduced to the compiler specification in order to enable compatibility with the memory layout, where some basic types occupy a single byte. In our model, however, all simple types occupy one word and, thus, are automatically aligned. Alignment of any valid type in the compiler specification with GC equals four.

The function $asize_t :: ty \to \mathbb{N}$ calculates the allocated size of a type in bytes. For all simple types the allocated size equals four. For an array and a structure the allocated size is defined in the following way.

$$\begin{aligned} asize_t(Arr_T(n,t)) &= n \cdot \lceil asize_t(t) \rceil_{algn(t)} \\ asize_t(Str_t(scl)) &= displ_v(0, scl, fst(last(scl))) + asize_t(snd(last(scl))) \end{aligned}$$

For the structure case the allocated size is calculated as the sum of the displacement of the last component and its allocated size. Function

$$displ_v :: \mathbb{N} \times (\mathbb{S} \times ty)\ list \times \mathbb{S} \to \mathbb{N}$$

calculates recursively the displacement of a component (a structure component or a variable from the symbol table) starting from some initial displacement (first parameter to the function). Note, that functions $displ_v$ and $asize_t$ are defined simultaneously and recursively call each other. For details see [12].

The size of a symbol table (local or global) is calculated by the function $asize_{st} :: (\mathbb{S} \times ty)\ list \times \mathbb{N} \to \mathbb{N}$ simply as the sum of the allocated size of the last variable and its displacement. The second parameter of the function denotes the size of the frame header (0 for a global frame and 20 for a local frame), i.e. initial displacement of the first variable in the frame.

The allocated size of a memory is computed in the following way. Let hs be the frame header size of the memory m.

$$\begin{aligned} asize_{mem} &::mframe \times \mathbb{N} \to \mathbb{N} \\ asize_{mem}(m, hs) &= asize_{st}(m.st, hs) \end{aligned}$$

The given above function calculates the allocated size of the local or global memory frames. Computing the allocated size of the heap is more complex.

Due to the garbage collection unreachable heap nodes are (sometimes) removed from the heap. The heap symbol table at the C0 small-step layer, however, remains unchanged during garbage collection, i.e. variables are never removed from the symbol table. Thus, we need to distinguish the g-variables present in the heap memory of the assembly machine from others. We call such variables *alive* and introduce a predicate which tests whether a g-variable is alive.

$$aliveT = gvar \to \mathbb{B}$$
$$alive :: aliveT$$

The compiler specification from [12] does not allow to define the *alive* predicate based on the C0 configuration. Thus, we keep it as a parameter to the simulation relation just like we do with the allocation function (Sec. 2.5).

Definition 4.1 (**The set of alive variables**)**.** Let *hst* be the heap symbol table and *alive* be the alive function. We define the set of indices of alive top-level heap variables.

$$alive_{idx}(hst, alive) = \{i \in \mathbb{N} \mid i < |hst| \wedge alive(gvar_{hm}(i))\}$$

Definition 4.2 (**Allocated size of the heap**)**.** The allocated size of the heap equals the sum of sizes of all alive heap variables and their headers.

$$asize^{alive}_{heap}(sc, alive) = \sum i \in alive_{idx}(sc.hst, alive).\ 4 \cdot AUX_{size} + \lceil asize_t(snd(sc.hst!i)) \rceil_4$$

Lemma 4.1 (**Allocated size of the heap is a multiple of four**)**.** Let *sc* be the program symbol configuration. The allocated size of the heap is a multiple of four.

$$4 \mid asize^{alive}_{heap}(sc, alive)$$

Proof. The proof follows directly from the definition. □

4.1.2 Displacements of Variables and G-Variables

The displacement of an element of an array is defined as the displacement of the array plus the allocated size of the preceding array elements. For the displacement of a part of a structure we use the function $displ_v$ from the previous section.

Definition 4.3 (**Displacement of a g-variable**)**.** The function $displ_g :: symbolconf \times gvar \to \mathbb{N}$ computes the displacement of some g-variable with respect to its root g-variable.

$$displ_g(sc, gvar_{gm}(x)) = 0$$
$$displ_g(sc, gvar_{lm}(i, x)) = 0$$
$$displ_g(sc, gvar_{hm}(i)) = 0$$
$$displ_g(sc, gvar_{arr}(g, i)) = displ_g(sc, g) + i \cdot asize_t(ty_g(sc, gvar_{arr}(g, i)))$$

For the structure case let type $ty_g(sc, gvar_{str}(g, cn))$ be some structure $Str_T(scl)$.

$$displ_g(sc, gvar_{str}(g, cn)) = displ_g(sc, g) + displ_v(0, scl, cn)$$

Displacement of the top-level variable with the name n in the symbol table sc with the frame header size hs is computed by the function $displ_v(hs, sc, n)$. Note, that only named variables' displacements can be computed in the mentioned way. For the heap variables we do not compute displacements in the symbol table at all.

4.1.3 Base Address

Definition 4.4 (**Base address of the local frame**)**.** We define the base address of a local memory frame by induction on the ordinal number of the frame in the local memory stack. Let te be a type name environment, ft be a function table, and sc the current symbol configuration.

$$\begin{aligned} abase_{lm}(te, ft, sc, 0) =& abase_{gm}(te, ft, sc.gst) + \lceil asize_{st}(sc.gst, 0)\rceil_4 \\ &+ 4 \cdot bubble_{gm} \\ abase_{lm}(te, ft, sc, i+1) =& abase_{lm}(te, ft, sc, i) + \lceil asize_{st}(sc.lst!i, 20)\rceil_4 \end{aligned}$$

For simplicity reasons we use notation $abase_{lm}(te, ft, sc.gst)$ for the base address of the first local memory frame $abase_{lm}(te, ft, sc, 0)$.

Now we can define the allocated base address of the named g-variables in the memory. Here and below, we sometimes skip constant parameters te and ft in the definitions.

Definition 4.5 (**Base address of g-variables**)**.** The allocated base address of a g-variable g is defined by case distinction on the location of the root g-variable of g.

$$\begin{aligned} abase_g(sc, gvar_{gm}(x)) =& abase_{gm}(te, ft, sc.gst) + displ_v(0, sc.gst, x) \\ abase_g(sc, gvar_{lm}(i, x)) =& abase_{lm}(te, ft, sc, i) + displ_v(20, sc.lst!i, x) \\ abase_g(sc, gvar_{arr}(g, i)) =& abase_g(sc, g) + i \cdot asize_t(ty_g(sc, gvar_{arr}(g, i))) \end{aligned}$$

For the structure case let $ty_g(sc, gvar_{str}(g, cn))$ be equal $Str_T(scl)$.

$$abase_g(sc, gvar_{arr}(g, i)) = abase_g(sc, g) + displ_v(0, scl, cn)$$

4.1.4 GC Constants

We instantiate the constants introduced in Sec. 3.1. The size of a word is fixed to 4.

$$MW = 4$$

The size of the local frame header FHS equals 5 words (Tab. 4.1).

$$FHS = 5$$

The size of the half heap HHS equals $asze^{max}_{hheap}$ divided by the word length. By the size of the local memory stack LMS we understand the maximal possible size of the stack (remains constant during the program execution). The size of the global memory frame is also constant and is equal to the size of the symbol

table of the global frame. Let te be a type name environment, ft be function table, and sc the current symbol configuration.

$$HHS = \frac{asize^{max}_{hheap}}{4}$$
$$LMS = \frac{(abase_{heap} - abase_{lm}(te, ft, sc.gst))}{4}$$
$$GMS = \lceil asize_{st}(sc.gst, 0) \rceil_4$$

4.1.5 Assembly Memory: Arrays Representation

On the Hoare logic level, memory is represented as a set of arrays of natural numbers (Sec. 3.2). On the assembly level, however, we have a single mapping $d.mm :: \mathbb{N} \to mcell_{asm}$. Thus, we have to divide this single machine memory into parts, corresponding to global, local, and heap memories of the program. Moreover, the global memory is represented by two arrays: one for the program global memory (seen by the garbage collector as GM array) and one for the GC global memory (which is not processed by the GC explicitly). Additionally, we have an array for the code part of the assembly memory, which is not accessed by GC and is not changed during garbage collection. Note, that in spite of the fact that the GC part of the global memory is not processed by the garbage collector explicitly, it is still changed during garbage collection. This occurs, because some of the variables in the GC interface are not constant and are changed every time when the garbage collector is called.

Definition 4.6 (**Memories extraction**)**.** We present the set of functions which extract certain portions of memory cells from the assembly memory: gm^p_m, gm^{gc}_m, $code_m$, lm_m, hm_m. All the functions, with the exception of $code_m$, return arrays of natural numbers and are of type

$$conf_{asm} \times symbolconf \times tenv \times functableT \to \mathbb{N}\ list.$$

For the code extraction we do not need to convert memory cells to natural numbers since this part remains completely unchanged during garbage collection and memory allocation.

$$code_m :: conf_{asm} \times symbolconf \times tenv \times functableT \to \mathbb{Z}\ list$$

We define the extraction functions in the following way.

$$\begin{aligned}
gm^p_m(d, sc) =& map(i2n, d.mm[abase_{gm}(te, ft, sc.gst), asize_{st}(sc.gst_p, 0)]) \\
gm^{gc}_m(d, sc) =& map(i2n, d.mm[abase_{gm}(te, ft, sc.gst) \\
&+ asize_{st}(sc.gst_p, 0), asize_{st}(sc.gst_{gc}, 0)]) \\
code_m(d, sc) =& d.mm[progbase, 4 \cdot csize_{prog}(te, sc.gst, ft)] \\
lm_m(d, sc) =& map(i2n, d.mm[abase_{lm}(te, ft, sc.gst), \\
&abase_{heap} - abase_{lm}(te, ft, sc.gst)]) \\
hm_m(d) =& d.mm[abase_{heap}, asize^{max}_{heap}]
\end{aligned}$$

Definition 4.7 (**Local memory extraction**)**.** The part of the memory containing the $i-$th local memory frame is extracted by the function $lmi_m(d, sc, i)$.

$$lmi_m(d, sc, i) = map(i2n, d.mm[abase_{lm}(te, ft, sc.gst, i), asize_{st}(sc.lst!i, 20)])$$

Definition 4.8 (**Heap g-variable extraction**)**.** Let *alloc* be the current g-variable allocation function. We define the function $hmg_m(d, g, alloc)$, which extracts that part of the heap memory which contains g-variable g.

$$hmg_m(d, g, alloc) = map(i2n, d.mm[fst(alloc(g)), snd(alloc(g))])$$

In the remaining part of the thesis we use the following notation for memory extractions.

$$GM^d = gm_m^p(d, sc)$$
$$GCM^d = gm_m^{gc}(d, sc)$$
$$LM^d = lm_m(d, sc)$$
$$HM^d = hm_m(d)$$

Now, we define functions which extract some values from frame and node headers.

Definition 4.9 (TT **link for frames**)**.** The following function extracts the index of the i-th local frame in the TT table from the frame header.

$$read_{fid}(d, sc, i) = i2n(d.mm(abase_{lm}(te, ft, sc.gst, i) + off_{TT} \cdot 4))$$

Note that the order of the local frames in the type table is defined by the order of respective functions in the function table ft. Thus, if the index of the i-th local frame in the TT table is j then the respective function has index $j-1$ in ft. We obtain this index by the function $read_{fnum}$

$$read_{fnum}(d, sc, i) = read_{fid}(d, sc, i) - 1$$

Definition 4.10 (TT **link for a node**)**.** Let *alloc* be the current g-variable allocation function. The following function extracts a TT link for a top-level g-variable g from the node header. This link is stored at the beginning of the node header. The allocation function applied to g returns the start address of the content of the node. Thus, we have to subtract the size of the node header from the returned value to get address where the TT link is stored.

$$read_{nid}(d, g, alloc) = i2n(d.mm(alloc(g) - AUX_{size} \cdot 4))$$

From the TT link we can easily get the index of the type of the node (type of the root g-variable) in the type environment te. The function $read_{tid}$ returns this index.

$$read_{tid}(d, g, alloc) = read_{nid}(d, g, alloc) - num_f$$

4.2 GC Interface

On the C0 level GC interface (Sec. 3.1) is nothing else than a set of variables located at the end of the global memory of the program.

4.2.1 GC Variables

The value of the constant gci_{length} depends on the implementation of the GC and is not fixed in the frame of this thesis. However, we require that all variables described in this section are present in the global symbol table. This gives a lower bound on the value of gci_{length}. The garbage collector implementation used in the frame of the thesis requires the following variables to be present in the GC part of the global memory:

- $gc_{gm} :: \mathbb{N}$: pointer to the global memory of the program. C0 semantics supports pointers to g-variables only. Thus, we use natural type $\mathbb{N}$ to model pointers to some memory regions located in the GC interface. Note, that use of natural numbers instead of pointers here also allows us to distinguish "real" pointers, which are considered by the garbage collector, from "fake" pointers, which point to some memory regions and do not produce any reachable heap nodes;
- $gc_{heap} :: \mathbb{N}$: pointer to the program heap;
- $gc_{stack} :: \mathbb{N}$: pointer to the stack of local memories of the program;
- $gc_{TT} :: TT_{type}$: type table;
- $gc_{PP} :: PP_{type}$: pointer table;
- $gc_f :: \mathbb{B}$: from-space indicator (flag);
- $gc_{nhi} :: \mathbb{N}$: value of the next heap index pointer of the program. Our implementation of the GC reads the next heap index from the global memory rather than directly from the next heap index register of the assembly machine. Thus, we have to keep next heap index values in memory and in the register consistent.

First we define functions which test whether some g-variable with index i in the GC global memory of the program symbol table sc is one of the GC variables. The functions test whether the name of the g-variable in the symbol table matches the searched variable name. Since variable names in the global memory are unique, there can be only one index i which satisfies the requirement.

$$
\begin{aligned}
is\_gm_{ptr}(sc, te, i) &= (fst(sc.gst_{gc}!i) = "gci\_gm") \\
is\_heap_{ptr}(sc, te, i) &= (fst(sc.gst_{gc}!i) = "gci\_heap") \\
is\_stack_{ptr}(sc, te, i) &= (fst(sc.gst_{gc}!i) = "gci\_stack") \\
is\_TT(sc, te, i) &= (fst(sc.gst_{gc}!i) = "gci\_TT") \\
is\_PP(sc, te, i) &= (fst(sc.gst_{gc}!i) = "gci\_PP") \\
is\_f(sc, te, i) &= (fst(sc.gst_{gc}!i) = "gci\_f") \\
is\_nhi(sc, te, i) &= (fst(sc.gst_{gc}!i) = "gci\_nhi")
\end{aligned}
$$

Now we define functions of the type $(symbolconf \times tenv \to gvarT)$ which return g-variables belonging to the GC interface from the global symbol table

of the program.

$$\begin{aligned}
gv_{gm}(sc, te) &= sc.gst_{gc}!(\varepsilon\ i.\ is_gm_{ptr}(sc, te, i)) \\
gv_{heap}(sc, te) &= sc.gst_{gc}!(\varepsilon\ i.\ is_heap_{ptr}(sc, te, i)) \\
gv_{stack}(sc, te) &= sc.gst_{gc}!(\varepsilon\ i.\ is_stack_{ptr}(sc, te, i)) \\
gv_{TT}(sc, te) &= sc.gst_{gc}!(\varepsilon\ i.\ is_TT(sc, te, i)) \\
gv_{PP}(sc, te) &= sc.gst_{gc}!(\varepsilon\ i.\ is_PP(sc, te, i)) \\
gv_{f}(sc, te) &= sc.gst_{gc}!(\varepsilon\ i.\ is_f(sc, te, i)) \\
gv_{nhi}(sc, te) &= sc.gst_{gc}!(\varepsilon\ i.\ is_nhi(sc, te, i))
\end{aligned}$$

However, an entity of the type $gvarT$ does not immediately give us the value of the variable. Thus, we define a set of functions which read values of GC variables from the memory. Such a function for the global memory pointer is given below.

$$\begin{aligned}
val_{gm} &:: memconf \times tenv \times functableT \to \mathbb{N} \\
val_{gm}(mc) &= i2n(cell2int(value_g(mc, gv_{gm}(sc(mc), te))!0))
\end{aligned}$$

For the case of TT and PP arrays, entities of type TT_{type} and PP_{type} are constructed inductively, i.e. every single value from the tables is read separately and then the values are combined into variables of the mentioned types.

In the remainder part of the thesis we use the following notation for the values of GC variables in the C0 configuration c.

$$\begin{aligned}
TT &= val_{TT}(c.mc) \\
PP &= val_{PP}(c.mc) \\
nhi^c &= val_{nhi}(c.mc)
\end{aligned}$$

Type and pointer tables remain constant during garbage collection and memory allocation. Thus, by TT and PP we understand their values in the initial configuration c.

4.2.2 Program Pointers

Here we give definitions used in the formalization of the type and pointer tables of the GC interface in the next section. We define functions, which extract displacements of pointers in words from a type or a symbol table and count the number of pointers.

By the displacement of a pointer inside some complex type t we understand the displacement of the pointer variable in the variable of type t allocated in some memory m. The displacement is computed as the sum of the allocated sizes of all types before the respective pointer type in the structure of the complex type t. For example in a type

$$t = Str([(n_1, Int_T), (n_2, Arr_T(m \times Bool_T)), (n_3, Ptr_T(n_4))])$$

the displacement of the only pointer with the type $Ptr_T(n_4)$ is computed as the sum of the allocated sizes of an integer and m boolean variables. Note, that a complex type may contain several instances of the type $Ptr_T(n_4)$ with different displacements.

Definition 4.11 (**Displacements of pointers in a type**)**.** The function $ptrs_t :: ty \to \mathbb{N}\ list$ extracts pointer displacements from a single type. A variable of the $Null_T$ type is considered as a non-pointer variable. For the base case we define

$$\begin{aligned} ptrs_t(Ptr_T(tn)) &= [0] \\ ptrs_t(t) &= [\,], \end{aligned}$$

where $t \in \{Bool_T, Int_T, Char_T, Unsigned_T, Null_T\}$.

For an array we extract displacements from the type of a single element and add the allocated size of the preceding part of the array.

$$ptrs_t(Arr_T(n,t)) = concat(map(\lambda x.\ ptrs_t(t) + x \cdot \frac{asize_t(t)}{4}, (0 : n-1)))$$

For the structure case we define an additional function $ptrs_{str} :: \mathbb{N} \times (\mathbb{S} \times ty)\ list \to \mathbb{N}\ list$, which extracts pointers from a list of structure components. In this function we use an additional parameter $i :: \mathbb{N}$, which accumulates the size of all components before the current one in the structure.

$$\begin{aligned} ptrs_{str}(i, [\,]) &= [\,] \\ ptrs_{str}(i, (cn,t)\#xs) &= (ptrs_t(t) + i) \circ ptrs_{str}(i + \frac{asize_t(t)}{4}, xs) \\ ptrs_t(Str_T(cts)) &= ptrs_{str}(0, cts) \end{aligned}$$

Definition 4.12 (**Displacements of pointers in a symbol table**)**.** We define the function, which extracts pointer displacements from a symbol table st of some memory frame with the frame header size hs. For this purpose we use the function $ptrs_{str}$ to extract pointers from a list of components of a symbol table.

$$ptrs_{st}(st, hs) = ptrs_{str}(\frac{hs}{4}, st)$$

Arrays of pointers, which are produced by the functions $ptrs_{st}(st, hs)$ and $ptrs_t(t)$ are called *pointer displacement arrays* of symbol table st and type t, respectively.

Definition 4.13 (**Number of pointers in a type**)**.** The function $cptrs_t :: ty \to \mathbb{N}$ calculates number of pointers in a type. Analogously, the function $cptrs_{st} :: (\mathbb{S} \times ty)\ list \to \mathbb{N}$ calculates number of pointers in a symbol table of the memory frame.

$$\begin{aligned} cptrs_t(t) &= |ptrs_t(t)| \\ cptrs_{st}(st) &= |ptrs_{st}(st, 0)| \end{aligned}$$

Below we give some basic properties of the $ptrs_t$ and $ptrs_{st}$ functions.

Lemma 4.2 (**Number of pointers in an array**)**.** The number of pointers in a type $Arr_T(n,t)$ is equal to the number of pointers in a type t times length of the array.

$$|ptrs_t(Arr_T(n,t))| = n \cdot |ptrs_t(t)|$$

Proof. We omit the simple proof of this lemma here. □

Now we give several properties on the pointer displacement arrays of a given type or a symbol table. Note, that here and below access of an element in the list $xs!i$ is valid iff condition $i < xs$ holds.

Lemma 4.3 (**Pointer displacement less size of type**)**.** Displacement of a pointer in a type t is less than the allocated size of the type.

$$\forall j.\ valid_{ty}(te, t) \Longrightarrow ptrs_t(t)!j < \frac{asize_t(t)}{4}$$

Proof. This lemma is proved by induction on the type t. The proof in Isabelle is somewhat complicated, since we have to deal with the mutual recursion in the definition of $ptrs_{str}$ and formulate separate condition for the structure case, but nevertheless straightforward. □

Lemma 4.4 (**Pointer displacement less size of symbol table**)**.** Displacement of a pointer in a symbol table st is less than allocated size of the symbol table.

$$\forall j.\ st \in valid_{st}(te) \wedge hs \mid 4 \Longrightarrow ptrs_{st}(st, hs)!j < \frac{asize_{st}(st, hs)}{4}$$

Proof. The proof is done by the structural induction on the symbol table st. The base case is trivial. For the induction step we have $st = (n_1, t_1)\#xs$.

Case $(j < cptrs_t(t_1))$. We use Lemma 4.3 to conclude the goal.

Case $(j \geq cptrs_t(t_1))$. Use induction hypothesis, definitions of $asize_{st}$, and $ptrs_{str}$ functions to conclude the proof. The correctness follows from the fact that the allocated size of the symbol table equals the sum of the allocated sizes of all variables present in the symbol table.

□

Lemma 4.5 (**Ordered pointer displacements**)**.** Displacements of pointers in a type t are ordered, i.e. displacement with index i is less then displacement with index j if i is less than j.

$$\forall i\ j.\ valid_{ty}(te, t) \wedge i < j \Longrightarrow ptrs_t(t)!i < ptrs_t(t)!j$$

Proof. This lemma is proved by induction on the type t. The correctness follows from the inductive construction of $ptrs_t$. □

Lemma 4.6 (**Ordered pointer displacements**)**.** Displacements of pointers in a symbol table st are ordered.

$$\forall i\ j.\ st \in valid_{st}(te) \wedge hs \mid 4 \wedge i < j \Longrightarrow ptrs_{st}(st, hs)!i < ptrs_{st}(st, hs)!j$$

Proof. The proof is done by structural induction over the symbol table st. We use Lemma 4.5 and properties of the $ptrs_{st}$ recursive construction to conclude the goal on the induction step. □

Now we present a set of lemmas, which connect displacements in complex types with displacements in the respective sub-types of the complex types. We use these lemmas in the proofs in Sec. 4.2.3.

By the following lemma we establish a connection between some pointer displacement in an array and in a single element of the array.

Lemma 4.7 (**Single element - array**)**.** Displacement of a pointer in an array of the type $Arr_T(n,t)$ equals displacement of a pointer in a type t plus the sum of the allocated sizes of all elements before the one, where the pointer is located. Let p be the the index of some pointer displacement in the $ptrs_t(Arr_T(n,t))$ array, k be the number of elements in the array before the one where the pointer is located, and j be displacement of the pointer inside an element of the array.

$$\begin{aligned} &p = k \cdot cptrs_t(t) + j \wedge j < cptrs_t(t) \wedge p < cptrs_t(Arr_T(n,t)) \\ &\Longrightarrow ptrs_t(Arr_T(n,t))!p = ptrs_t(t)!j + k \cdot \frac{asize_t(t)}{4} \end{aligned}$$

Proof. The proof is simple and follows directly from the definition of the $ptrs_t$ and properties of *concat* function. □

Lemma 4.8 (**Array - single element**)**.** For every pointer displacement with index p in the array type $Arr_T(n,t)$, which is equal to the size of k array elements plus some displacement m less than the size of a single element, there exists a respective displacement with index j in a single element of the array and this displacement is equal to m. This lemma is the the opposite direction for Lemma 4.7.

$$\begin{aligned} &ptrs_t(Arr_T(n,t))!p = m + k \cdot \frac{asize_t(t)}{4} \wedge m < \frac{asize_t(t)}{4} \wedge valid_{ty}(te,t) \\ &\Longrightarrow \exists j.\ m = ptrs_t(t)!j \wedge p = k \cdot cptrs_t(t) + j \end{aligned}$$

Proof. From $p < |ptrs_t(Arr_T(n,t))|$ we use Lemma 4.2 to find i and k_1, s.t.

$$p = k_1 \cdot cptrs_t(t) + i \wedge k_1 < n \wedge i < cptrs_t(t).$$

We instantiate the existential quantifier in the goal with i and assume $k_1 = k$. We conclude the goal by applying Lemma 4.7. The subgoal $k_1 = k$ is shown by contradiction analyzing cases $k_1 < k$ and $k_1 > k$. □

Lemma 4.9 (**Structure component - structure**)**.** Displacement with index p of a pointer in a structure of the type $Str_T(cts)$ equals displacement with index j of a pointer in a component of the structure, where this pointer is located, plus allocated size of all components before this one in the structure.

$$\begin{aligned} &cts = ct \circ [(n,t)] \circ xs \wedge p = cptrs_{str}(ct) + j \wedge j < cptrs_t(t) \\ &\Longrightarrow ptrs_t(Str_T(cts))!p = ptrs_t(t)!j + asize_t(Str_T(ct)) \end{aligned}$$

Proof. The proof is done by induction on the list of structure components cts. The correctness follows from the properties of the $ptrs_t$ and $ptrs_{str}$ functions. □

Lemma 4.10 (**Structure - structure component**)**.** For every pointer displacement with index p in the structure type $Str_T(cts)$ there exists a respective displacement with index j in some structure component. This lemma is the opposite direction for Lemma 4.9.

$$\begin{aligned} &cts = ct \circ [(n,t)] \circ xs \wedge ptrs_t(Str_T(cts))!p = asize_t(Str_T(ct)) + m \\ &\wedge\ m < cptrs_t(t) \wedge valid_{ty}(te,t) \\ &\Longrightarrow \exists j.\ m = ptrs_t(t)!j \wedge p = ptrs_{str}(ct) + j \end{aligned}$$

Proof. The proof is analogues to Lemma 4.8 and we omit it here. □

Lemma 4.11 (**Symbol table - type displacements**)**.** For every pointer displacement with index k in type t there exists a respective displacement with index m in the $ptrs_{st}(st, hs)$ array of a symbol table st containing type t.

$$\begin{aligned} &c \in conf_{\surd}(te, ft) \wedge st \in valid_{st}(te) \wedge hs \mid 4 \wedge st!i = (n,t) \wedge k < cptrs_t(t) \\ &\Longrightarrow \exists m.\ ptrs_{st}(st, hs)!m = cptrs_t(t)!k + \frac{displ_v(hs, st, n)}{4} \end{aligned}$$

Proof. The lemma is proven by induction on the symbol table st. The base case is trivial. In the induction step $st = (n_1, t_1)\#xs$.

Case $(i = 0)$. We instantiate the existential quantifier with k and using definition of $ptrs_{st}$ derive the goal.

Case $(i \neq 0)$. We use the induction hypothesis for the $(i-1)$-th element of xs and the frame header size $hs + asize_t(t_1)$. We get

$$\exists m_1.\ ptrs_{st}(xs, hs{+}asize_t(t_1))!m_1 = cptrs_t(t)!k{+}\frac{displ_v(hs + asize_t(t_1), xs, n)}{4}.$$

We instantiate the existential quantifier with $m_1 + cptrs_t(t_1)$ and derive

$$\begin{aligned} &ptrs_{st}((n_1,t_1)\#xs, hs)!(m_1 + cptrs_t(t_1)) \\ &= (ptrs_t(t_1) + \frac{hs}{4}) \circ ptrs_{st}(xs, hs + asize_t(t_1))!(cptrs_t(t_1) + m_1) && \text{Def. 4.12} \\ &= ptrs_{st}(xs, hs + asize_t(t_1))!m_1 \\ &= cptrs_t(t)!k + \frac{displ_v(hs + asize_t(t_1), xs, n)}{4} && \text{Ind. hyp.} \\ &= cptrs_t(t)!k + \frac{displ_v(hs, (n_1,t_1)\#xs, n)}{4} && \text{Def. of } displ_v \end{aligned}$$

□

4.2.3 Displacements: Types vs. G-Variables

In order to be able to argue about program g-variables using information about pointer displacements in some type we need to connect displacement of a pointer in a type with the displacement of a respective pointer in the allocated top-level g-variable (or a symbol table). Though the connection seems to be obvious, it is not easy to establish it formally. Recursion is used both in the definition of the g-variable and the type of a g-variable. The direction in which recursion

is performed, however, is opposite. (Sec. 2.3.4 and Fig. 2.1). This creates additional difficulties in the induction proofs.

The following lemma connects the displacement of some pointer g-variable in its root with the displacement in $ptrs_t$ array of the type of the root g-variable.

Lemma 4.12 (**G-variable - type displacement**)**.** For every displacement of a pointer g-variable x (w.r.t. its root g-variable) there exists a respective displacement in the pointer displacement array of the root g-variable of x.

$$\begin{aligned} &c \in conf_{\surd}(te, ft) \wedge x \in gvars_{\surd}(sc) \wedge ptr_{gvar}(x) \\ &\Longrightarrow \exists m.\ ptrs_t(ty_g(sc, root_g(x)))!m = \frac{displ_g(sc, x)}{4} \end{aligned}$$

Proof. The proof of the lemma is done by structural induction on x. We prove an auxiliary lemma to make the induction proof possible (Lemma 4.13). □

Lemma 4.13. For every displacement of a pointer g-variable x (w.r.t. its root g-variable) there exists a respective displacement in the pointer displacement array of the root g-variable of x. Moreover, if x is an array or a structure, for every displacement with index k in the pointer displacement array of the type of x there exists a respective displacement in the pointer displacement array of the root g-variable of x.

$$\begin{aligned} &c \in conf_{\surd}(te, ft) \wedge x \in gvars_{\surd}(sc) \Longrightarrow \\ &(ptr_{gvar}(x) \longrightarrow \exists m.\ ptrs_t(ty_g(sc, root_g(x)))!m = \frac{displ_g(sc, x)}{4}) \wedge \\ &(arr_{gvar}(x) \vee str_{gvar}(x) \longrightarrow \exists m.\ ptrs_t(ty_g(sc, root_g(x)))!m \\ &\qquad = ptrs_t(ty_g(sc, x))!k + \frac{displ_g(sc, x)}{4}) \end{aligned}$$

Proof. The proof is done by induction on x. In the base case x is a top-level g-variable and $root_g(x) = x$. It follows to $displ_g(sc, x) = 0$ and we conclude the goal. In the induction step x is either an element of the array g or a structure component of the structure g. We perform a case split on the type of x.

Case $(x = gvar_{arr}(g, i))$. The first conjunct of the goal is proved with the help of the second conjunct in the induction hypothesis with g in place of x. We get

$$\forall k.\ \exists m.\ ptrs_t(ty_g(sc, root_g(g)))!m = ptrs_t(ty_g(sc, g))!k + \frac{displ_g(sc, g)}{4}$$

Both x and g have the same root g-variable, i.e. $root_g(g) = root_g(x)$. Moreover, x is a pointer g-variable, thus its displacement is present in array $ptrs_t(ty_g(sc, g))$ at the position i. We derive

$$\begin{aligned} &\exists m.\ ptrs_t(ty_g(sc, root_g(x)))!m \\ &\quad = ptrs_t(ty_g(sc, g))!(i \cdot cptrs_t(ty_g(sc, x))) + \frac{displ_g(sc, g)}{4} && \text{Ind. hyp.} \\ &\quad = ptrs_t(ty_g(sc, x))!0 + i \cdot asize_t(ty_g(sc, x)) + \frac{displ_g(sc, g)}{4} && \text{Lemma. 4.7} \\ &\quad = 0 + i \cdot asize_t(ty_g(sc, x)) + \frac{displ_g(sc, g)}{4} && x \text{ is a pointer} \\ &\quad = \frac{displ_g(sc, x)}{4} && \text{Def. of } displ_g \end{aligned}$$

In the second conjunct we have a case when x is itself an array or a structure. To prove it we also use the second conjunct from the induction hypothesis. We instantiate the universal quantifier with $i \cdot cptrs_t(ty_g(sc, x)) + k$ and derive

$$\begin{aligned}
&\exists m.\ ptrs_t(ty_g(sc, root_g(x)))!m = \\
&\quad ptrs_t(ty_g(sc, g))!(i \cdot cptrs_t(ty_g(sc, x)) + k) + \frac{displ_g(sc, g)}{4} && \text{Ind. hyp.} \\
&\quad = ptrs_t(ty_g(sc, x))!k + i \cdot asize_t(ty_g(sc, x)) + \frac{displ_g(sc, g)}{4} && \text{Lemma 4.7} \\
&\quad = ptrs_t(ty_g(sc, x))!k + \frac{displ_g(sc, x)}{4} && \text{Def. of } displ_g
\end{aligned}$$

Case $(x = gvar_{str}(g, i))$. The proof is analogues to the first case. We use Lemma 4.9 in place of Lemma 4.7 and find structure components sc and xs which fulfill the equality $scl = sc \circ [(n, t)] \circ xs$ where $[(n, t)]$ is the structure component of the g-variable x in the structure g with the type $ty_g(sc, g) = Str_T(scl)$. The universal quantifier is instantiated with $cptrs_{st}(ct) + k$.

□

Lemma 4.12 states that if there exists some pointer g-variable x, then its displacement is present in the pointer displacement array of the type of the root g-variable of x. We also need to formulate the other direction of this lemma. I.e. for every displacement in the $ptrs_t$ array there exists a respective pointer g-variable. The proof of the lemma (Lemma 4.16) is a bit tricky and requires an auxiliary Lemma 4.14.

Lemma 4.14. Let some pointer displacement d, belonging to array $ptrs_t$ of the root g-variable, be in allocation range of the g-variable x. Then in the pointer displacement array of the type of x there exists an element equal to the difference between d and displacement of x (w.r.t. to its root g-variable).

$$\begin{aligned}
&c \in conf_{\surd}(te, ft) \wedge x \in gvars_{\surd}(sc) \wedge d \in ptrs_t(ty_g(sc, root_g(x))) \\
&d - \frac{displ_g(sc, x)}{4} < \frac{asize_t(ty_g(sc, x))}{4} \\
&\implies d - \frac{displ_g(sc, x)}{4} \in ptrs_t(ty_g(sc, x))
\end{aligned}$$

Proof. The proof is done by the structural induction on x. In the base case $root_g(x) = x$ and the proof is trivial.

For the induction step we have two cases.

Case $(x = gvar_{arr}(g, i))$. We use the induction hypothesis for displacement d

and derive

$$
\begin{aligned}
& d - \frac{displ_g(sc,g)}{4} \in ptrs_t(ty_g(sc,g)) && \text{Ind. hyp.}\\
\equiv\ & d - \frac{displ_g(sc,g)}{4} + i \cdot \frac{asize_t(t)}{4} \in ptrs_t(ty_g(sc,g)) && \text{Def. of } displ_g\\
\equiv\ & \exists m.\ ptrs_t(ty_g(sc,g))!m = d - \frac{displ_g(sc,g)}{4} + i \cdot \frac{asize_t(t)}{4} &&\\
\equiv\ & \exists j.\ d - \frac{displ_g(sc,g)}{4} = ptrs_t(ty_g(sc,x))!j && \text{Lemma 4.8}\\
\equiv\ & d - \frac{displ_g(sc,g)}{4} \in ptrs_t(ty_g(sc,x)) &&
\end{aligned}
$$

Case $(x = gvar_{str}(g,i))$. The proof is very similar to the first case with Lemma 4.10 in place of Lemma 4.8.

□

Lemma 4.15 (**Zero displacement**)**.** If the pointer displacement array of an elementary g-variable g contains only a single zero element then g is a pointer.

$$elem_g(g) \wedge ptrs_t(ty_g(sc,g)) = [0] \Longrightarrow ptr_{gvar}(g)$$

Proof. The proof follows directly from the definition of $ptrs_t$. □

Lemma 4.16 (**Type - g-variable displacement**)**.** If the displacement of an elementary g-variable x is present in the $ptrs_t$ array of the root g-variable then x is a pointer. Note, that this lemma does not hold for complex g-variables, because a complex g-variable can have the same displacement as the pointer sub g-variable.

$$
\begin{aligned}
& c \in conf_{\surd}(te,ft) \wedge x \in gvars_{\surd}(sc) \wedge elem_g(g)\\
& \wedge \frac{displ_g(sc,x)}{4} \in ptrs_t(ty_g(sc,root_g(x)))\\
& \Longrightarrow ptr_{gvar}(x)
\end{aligned}
$$

Proof. We use Lemma 4.14 with $d = \frac{displ_g(sc,x)}{4}$ and get

$$0 \in ptrs_t(ty_g(sc,x))$$

Finally we apply Lemma 4.15 and use the fact, that the $ptrs_t$ array of an elementary g-variable contains at most one element. □

Now we formulate analogous lemmas to Lemma 4.12 and Lemma 4.16 for the $ptrs_{st}$ function. We omit the detailed proof of these lemmas here. Let sc be the program symbol configuration and st be some symbol table from the symbol configuration containing g-variable x.

Lemma 4.17 (**G-variable - symbol table displacement**)**.** For every displacement of a pointer g-variable x there exists a respective displacement with index m in the displacement array of the symbol table containing x.

$$
\begin{aligned}
& c \in conf_{\surd}(te,ft) \wedge x \in gvars_{\surd}(sc) \wedge ptr_{gvar}(x) \wedge fs \mid 4\\
& \wedge\ st \in valid_{st}(te) \wedge st!i = (fst(root_g(x)), ty_g(sc,root_g(x)))\\
& \Longrightarrow \exists m.\ ptrs_{st}(st,hs)!m = \frac{displ_g(sc,x) + displ_v(hs,sc,fst(root_g(x)))}{4}
\end{aligned}
$$

Proof. The proof of this lemma requires application of Lemmas 4.12 and Lemma 4.11. □

Lemma 4.18 (**Symbol table - g-variable displacement**)**.** If the displacement of an elementary g-variable x (w.r.t. to the start of the symbol table) is present in the $ptrs_{st}$ array of the symbol table containing x, then x is a pointer.

$$\begin{aligned} & c \in conf_{\surd}(te, ft) \wedge x \in gvars_{\surd}(sc) \wedge elem_g(g) \wedge fs \mid 4 \\ & \wedge\ st \in valid_{st}(te) \wedge st!i = (fst(root_g(x)), ty_g(sc, root_g(x))) \\ & \wedge\ \frac{displ_g(sc, x)}{4} + displ_v(hs, sc, fst(root_g(x))) \in ptrs_{st}(st, hs) \\ & \Longrightarrow ptr_{gvar}(x) \end{aligned}$$

Proof. The proof of this lemma requires application of Lemma 4.16 and the connection between $ptrs_{st}$ and $ptrs_t$ arrays. We omit the proof here. □

4.2.4 Data Structures Formalization

In order to prove correctness of the compiler, we need to argue about the structure of the TT and PP tables introduced in Sec. 3.1. In this section we develop a formal model of these tables based on the information from the function table, symbol table of the global memory frame, and the type environment of the program.

Correctness Criteria

Definition 4.14 (**Length of the pointer table**)**.** The length of the pointer table can be computed from the given type table. We define the function which adds values of all fields num in the TT table.

$$length_{PP}(TT) = \sum_{i=0}^{|TT|-1} TT_i.num$$

Definition 4.15 (**Start of the TT element in the PP table**)**.** The function $start_{PP} :: TT_{type} \times \mathbb{N} \to \mathbb{N}$ computes the start index of the $i-$th TT element in the pointer table.

$$start_{PP}(TT, i) = length_{PP}(TT[0, i])$$

The following definitions state that type and pointer tables contain correct information about a single memory frame. Let $hs :: \mathbb{N}$ be the frame header size. Further, let $st :: (S \times ty)\ list$ be the symbol table of the $i-$th memory frame in table TT.

Definition 4.16 (**Correct TT data for a frame**)**.** We require that field asz of a TT record is equal to the allocated size of the frame in words, field num contains the number of pointers in a frame and field st is equal to the start index of the corresponding part of the PP table.

$$\frac{\begin{array}{l} TT_i.asz = \dfrac{\lceil asize_{st}(st, hs) \rceil_4}{4} \\ TT_i.num = cptrs_{st}(st) \\ (TT_i.num > 0 \longrightarrow TT_i.st = start_{PP}(TT, i)) \end{array}}{frame_{TT\surd}(i, st, TT, PP, hs)}$$

Definition 4.17 (**Correct PP data for a frame**)**.** We require that the k-th record in the respective part of the PP table equals the k-th pointer displacement in the $i-$th frame.

$$\frac{\forall k < cptrs_{st}(st).\ PP_{TT_i.st+k}.woff = ptrs_{st}(st, hs)!k}{frame_{PP\surd}(i, st, TT, PP, hs)}$$

Now we use Def. 4.16 and Def. 4.17 to obtain the interface specification for the local and global memory frames.

Definition 4.18 (**Correct global frame**)**.** Let $gst :: (S \times ty)\ list$ be the symbol table of the global memory frame in the current program configuration.

$$\frac{frame_{TT\surd}(0, gst, TT, PP, 0) \qquad frame_{PP\surd}(0, gst, TT, PP, 0)}{frame_{g\surd}(gst, TT, PP)}$$

Definition 4.19 (**Correct local frames**)**.** We require that $frame_{TT\surd}$ and $frame_{PP\surd}$ properties hold for symbol tables of all functions from the program function table ft.

$$\frac{\begin{array}{r}\forall i < |ft|.\ frame_{TT\surd}(i, stbl_f(ft, i), TT, PP, 20)\\ frame_{PP\surd}(i, stbl_f(ft, i), TT, PP, 20)\end{array}}{frame_{s\surd}(ft, TT, PP)}$$

In an analogues way we define the correctness criteria for the parts of type and pointer tables corresponding to the program type environment te.

Definition 4.20 (**Correct TT data for a type**)**.** Let i be the index of the type t in the type table TT.

$$\frac{\begin{array}{l}TT_i.asz = \dfrac{\lceil asize_t(t)\rceil_4}{4}\\ TT_i.num = cptrs_t(t)\\ (TT_i.num > 0 \longrightarrow TT_i.st = start_{PP}(TT, i))\end{array}}{type_{TT\surd}(i, t, TT, PP)}$$

Definition 4.21 (**Correct PP data for a type**)**.** Let i be the index of the type t in the type table TT. We require that the k-th record in the respective part of the array PP equals the k-th pointer displacement in the type t.

$$\frac{\forall k < cptrs_t(t).\ PP_{TT_i.st+k}.woff = ptrs_t(t)!k}{type_{PP\surd}(i, t, TT, PP)}$$

Definition 4.22 (**Correct type environment**)**.** The number of frames in the program equals $|ft| + 1$. We require that the next $|te|$ elements of the array TT fulfil correctness criteria for the types.

$$\frac{\begin{array}{r}\forall i < |te|.\ type_{TT\surd}(i + |ft| + 1, snd(te!i), TT, PP)\\ type_{PP\surd}(i + |ft| + 1, snd(te!i), TT, PP)\end{array}}{type_{te\surd}(te, ft, TT, PP)}$$

Finally, we combine Def. 4.18, Def. 4.19, and Def. 4.22 into one predicate.

Definition 4.23 (**Correct GC interface**)**.** For the global symbol table gst correct GC interface is defined in the following way.

$$\frac{frame_{g\surd}(gst, TT, PP) \qquad frame_{s\surd}(ft, TT, PP) \qquad type_{te\surd}(te, ft, TT, PP) \qquad |TT| = |ft| + |te| + 1 \qquad |PP| = length_{PP}(TT)}{gci_{\surd}(te, ft, gst, TT, PP)}$$

Note, that all input parameters to the predicate $gci_{\surd}$ are fixed during program compilation and are not changed during program execution.

Correct Interface Proof

In this section we use the correctness criteria from Def. 4.23 to derive predicate $correct_{gci}$ (Def. 3.9) from Sec. 3.1.

Lemma 4.19 (**Correct interface**)**.**

$$c \in conf_{\surd}(te, ft) \wedge gci_{\surd}(te, ft, gst, TT, PP) \wedge te \neq [\,] \implies correct_{gci}(TT, PP)$$

Proof. We divide this lemma into 5 sub-lemmas (one for each predicate in $correct_{gci}(TT, PP)$) and prove them separately.

1. $inframe_{ptrs}(TT, PP)$: use Lemma 4.3 and Lemma 4.4 to conclude the proof;
2. $woff_{ge}(TT, PP)$: to prove this property we formulate an additional lemma[1] which states, that every displacement in $ptrs_{st}$ is greater or equal than frame header size divided by four;
3. $plink_{inside}(TT, PP)$: correctness follows from the definition of function $start_{PP}$ and the fact that $|PP| = length_{PP}(TT)$;
4. $asize_{pos}(TT, PP)$: from the valid configuration and definitions of allocated size from Sec. 4.1.1 we derive that allocated size of a type or a local frame is greater than zero and allocated size of the global frame is greater or equal zero;
5. $woff_{ord}(TT, PP)$: use Lemma 4.5 and Lemma 4.6 to conclude the property.

The last conjunct in $correct_{gci}$ is $|TT| > num_f$, i.e. $|TT| > |ft|+1$. This follows directly from the definition of $gci_{\surd}(te, ft, gst, TT, PP)$.

□

Note, that we require the type environment to contain at least one element. If the type environment is empty, then the program does not contain any pointers, the heap is always empty and heap memory allocation is never performed.

[1] Lemma frame_header_size_le_ptrs_displ_stbl

4.3 Transition Function

In this section we define the new C0 transition function for the compiler with the garbage collector. We are interested only in the heap memory allocation (i.e. execution of the *PAlloc* statement), since definition for transition function for all other C0 statements is identical to the definition in the original compiler [12].

4.3.1 Memory Allocation

In order to distinguish the case when there is not enough heap memory available in the hardware machine (in the C0 machine heap memory is infinite), we introduce predicate $avail_{heap} :: (tenv \times memconf \times ty) \mapsto \mathbb{B}$. It returns true, if there is enough heap memory available to allocate a new object of a given type. We do not interpret $avail_{heap}$ inside C0 semantics, i.e. it is an input parameter to the C0 transition function. Thus, the C0 machine is independent of the amount of available memory in a concrete machine. Later in this section we give a concrete instance of this predicate.

The transition function for the new heap memory allocation statement consists of two parts. In the first part we assign new values to the global variables of the GC and in the second part we perform memory allocation itself. We introduce an additional transition function to cover the first part of memory allocation and integrate it into the C0 transition function δ_{C0} (Sec. 2.3.5).

Available heap

Analogously to Def. 4.2 we define the size of all reachable nodes on the heap.

Definition 4.24 (**The set of reachable heap variables**)**.** Let $hst(mc)$ be the heap symbol table of the memory configuration mc. We define the set of indices of the reachable top-level heap variables.

$$reach_{idx}(mc) = \{i \in \mathbb{N} \mid i < |hst(mc)| \wedge gvar_{hm}(i) \in reachable_g(mc)\}$$

Definition 4.25 (**Reachable heap size**)**.** The size of the reachable part of the heap equals the sum of sizes of all reachable heap variables and their headers.

$$asize_{heap}^{reachable}(mc) = \sum_{i \in reach_{idx}(mc)} 4 \cdot AUX_{size} + \lceil asize_t(snd(hst(mc)!i)) \rceil_4$$

Definition 4.26 (**Available heap**)**.** The predicate $avail_{heap}(te, mc, t)$ returns true iff the size of the reachable heap plus the size of the type t and the size of the node header fits into the half heap size.

$$\frac{asize_{heap}^{reachable}(mc) + asize_t(t) + AUX_{size} \cdot 4 \leq asize_{hheap}^{max}}{avail_{heap}(te, mc, t)}$$

Definition 4.27 (**Reachable g-variables alive**)**.** The predicate $reach_{alive}$ holds iff all alive g-variables in the memory configuration mc are reachable. This property is always satisfied after garbage collection is performed.

$$reach_{alive}(mc, alive) = \forall g.\ alive(g) \longrightarrow g \in reachable_g(mc)$$

GC Transition Function

Let $memupd :: memconf \times gvar \times (mcell_{C0}\ list) \mapsto memconf_{\perp}$ be a function which updates a g-variable with a new value (we skip the definition of *memupd* function here).

Definition 4.28 (**GC variables update**)**.** We introduce the function

$$update_{gci} :: memconf \times \mathbb{B} \times \mathbb{N} \mapsto memconf$$

which sets the new values for the variables located in the GC part of the global memory. Namely, the next heap index and the from-space flag are updated with $nhi_{val} :: \mathbb{N}$ and $f_{val} :: \mathbb{B}$ respectively. Let mc' be a memory configuration after update of the *nhi* variable, i.e.

$$mc' = \lfloor memupd(mc, gvar_{nhi}(sc(mc), te), [Nat(nhi_{val})]) \rfloor.$$

We define the function $update_{gci}$ in the following way.

$$update_{gci}(mc, f_{val}, nhi_{val}) = \lfloor memupd(mc', gvar_f(sc(mc), te), [Bool(f_{val})]) \rfloor$$

Now we need to calculate the new values of the next heap index and the from-space flag from the current C0 configuration and the size of the type we want to allocate. The value of f remains unchanged in the case when the garbage collection is not performed (i.e. there is enough space for allocation without garbage collection) and is reversed in all other cases. The value of the next heap index *nhi* is changed in one of the following ways:

- is increased by the size of the type we allocate: if there is enough heap space for allocation without garbage collection;
- is equal to the size of the reachable heap before allocation plus the size of the type we allocate: if there is not enough heap space for allocation without garbage collection, but enough after garbage collection is performed;
- is equal to the size of the reachable heap before allocation: if there is not enough heap space for allocation even after garbage collection is performed.

Definition 4.29 (**New values of GC variables**)**.** Let $f = gvar_f(sc(mc), te)$ be the current from-space indicator and $avail_{heap}$ be the predicate which tests whether a new variable can be allocated on the heap. Functions upd_f and upd_{nhi} calculate the new values of the next heap index and from-space variables, respectively. By predicate $no\_gc\_needed(mc, t, f)$ let us denote the case when

$$val_{nhi}(mc) + AUX_{size} \cdot 4 + asize_t(t) \leq st_{idx}(f) \cdot 4 + asize^{max}_{hheap}.$$

We also use the following notation:

$$nhi_1 = st_{idx}(\neg f) \cdot 4 + asize^{reachable}_{heap}(mc) + AUX_{size} \cdot 4 + asize_t(t)$$
$$nhi_2 = st_{idx}(\neg f) \cdot 4 + asize^{reachable}_{heap}(mc)$$

Update functions are defined in the following way.

$$upd_f(mc,t) = \begin{cases} f & no\_gc\_needed(mc,t,f) \\ \neg f & \text{otherwise} \end{cases}$$

$$upd_{nhi}(mc,t,avail_{heap}) = \begin{cases} nhi + asize_t(t) & no\_gc\_needed(mc,t,f) \\ nhi_1 & avail_{heap}(te,mc,t) \\ nhi_2 & \text{otherwise} \end{cases}$$

Definition 4.30 (**GC transition function**)**.** Let us denote the value of $upd_f(c.mem,t)$ by f_{val} and by nhi_{val} the value of $upd_{nhi}(c.mem,t,avail_{heap})$. Then we define the GC delta transition function as an update of the memory configuration of the C0 machine c with the values f_{val} and nhi_{val}. The program rest of the configuration is not changed.

$$\delta_{gc}(te,ft,c,t) = \lfloor c\,[mem := update_{gci}(c.mem, f_{val}, nhi_{val})] \rfloor .$$

C0 transition function

Now we consider the second part of the $PAlloc(e_l,t_n)$ statement - allocation of the new heap variable. In the case if there is enough heap memory, allocation of the new variable is done in two steps:

1. Heap memory is extended by new (nameless) variable of the given type;
2. a pointer to the newly allocated variable is created and is assigned to the left expression e_l.

In the case if there is not enough heap memory for allocation, we skip the first step and assign a null pointer in the second. The requirement in both cases is that the left expression e_l can be evaluated to some g-variable and that the type name t_n is defined in the type name environment.

Definition 4.31 (**Heap extension**)**.** We introduce the function $extend_{heap} :: (memconf \times ty \to \mathbb{B}) \times memconf \times ty \mapsto memconf$ which adds a new initialized variable to the heap, i.e. performs step 1 of the allocation. Let m be the old memory configuration and t the type of newly allocated variable. Then we define the new memory configuration $m' = extend_{heap}(avail_{heap}, m, t)$ in the following way.

The base address of the new variable equals the abstract size of the old heap memory of the C0 machine, i.e $b = \sum_{j=0}^{|hst(m)|-1} size_t(snd(hst(m)!j))$. If there is enough heap memory available, i.e. $avail_{heap}(te,m,t) = true$, the new heap memory is defined by

$$hm' = m.hm \left[\begin{array}{l} st := hst(m) \circ [(undef,t)] \\ ct := m.hm.ct([b, size_t(t)] := init_{val}(t)[0, size_t(t)]) \end{array} \right],$$

where $init_{val}(t)$ returns some initial value of the given type. Otherwise, if $avail_{heap}(te,m,t) = false$, we set $hm' = m.hm$.

The second step of memory allocation is defined as follows.

Definition 4.32 (**Memory allocation**)**.** Let p be a pointer, which points to the newly allocated variable, i.e.

$$p = \begin{cases} Ptr(gvar_{hm}(|hst(c.mem)|)) & \text{if } avail_{heap}(te, c.mem, t) \\ Ptr(\bot) & \text{otherwise} \end{cases}$$

Let $\bar{c}$ be a C0 configuration with the updated values of GC variables, i.e. $\bar{c} = \delta_{gc}(te, ft, c, t)$.The new memory configuration mc' is defined in the following way.

$$\lfloor mc' \rfloor = memupd(extend_{heap}(avail_{heap}, \bar{c}.mem, t), g, [p]).$$

Let the program rest of the C0 configuration consists of a statement for allocation of new heap memory, i.e. $c.prog = PAlloc(e_l, t_n)$, where t_n is type name of the type t. The new C0 configuration is defined in the following way.

$$\delta_{C0}(te, ft, c) = \left\lfloor c \left[\begin{array}{l} prog := Skip \\ mem := mc' \end{array} \right] \right\rfloor .$$

4.3.2 Extended Alive Function

During garbage collection unreachable variables are removed from the heap. After memory allocation (if it is successful) another reachable top-level g-variable is added to the heap. In both cases we have to change the current alive function. The possible changes to the alive function are:

- the new alive function returns true for the new heap variable and does not change its value for old variables: if allocation is successful without garbage collection;
- the new alive function returns true only for the new heap variable and all variables reachable in the old configuration: if there is not enough heap space for allocation without garbage collection but enough after garbage collection is performed;
- the new alive function returns true only for variables reachable in the old configuration: if there is not enough heap space for allocation even after garbage collection is performed.

To model these situations we introduce two functions.

$$alive_{gc} :: memconf \to aliveT$$
$$alive_{xt} :: memconf \times aliveT \times \mathbb{N} \to aliveT$$

The first one reflects the impact of the garbage collection on the alive function. The second - extends the current alive function with the new heap variable (heap extension occurs only if allocation is successful).

Definition 4.33 (**Alive function after GC**)**.** After the garbage collection all reachable nameless g-variables are alive. Unreachable variables are removed from the heap.

$$alive_{gc}(mc)(g) = g \in reachable_g^{nameless}(mc)$$

Definition 4.34 (**Extended alive function**). Let i be the index of the newly allocated heap variable in the heap symbol table.

$$alive_{xt}(mc, alive, i)(g) = \begin{cases} true & g = gvar_{hm}(i) \\ alive(g) & \text{otherwise} \end{cases}$$

In case of successful garbage collection (when the variable is allocated after it) we need to apply $alive_{gc}$ and $alive_{xt}$ functions consequently to get the new alive function function.

4.3.3 Extended Allocation Function

Analogously to the alive function we have to distinguish three different updates of the allocation function *alloc*:

- if the memory is allocated with the garbage collector then *alloc* is just extended with the new heap g-variable;
- if the garbage collection is performed but the variable is not allocated, then the second component of the allocation function for reachable heap g-variables remains unchanged and the first component is assigned some new value, which denotes the new position of the g-variable on the heap;
- if the garbage collection is performed and a new variable allocated, then both described changes are applied to *alloc* function.

In order to define the new allocation function formally we introduce a function which computes the displacement of some top-level g-variable in the new heap of the assembly machine (after garbage collection is performed). In order to find out the position of some heap node in the new heap memory, we need to find a position of the respective node in the abstract heap xs and apply the permutation function F (Sec. 3.2.5), which defines the order of the nodes after garbage collection. This gives us the position of the $i-$th node in the new abstract heap xs' and the new displacement of the node in the heap is equal to $xs'!F(i)$. The new abstract heap xs' and the permutation relation F are obtained from the postcondition of the heap allocation routine (Sec. 3.2.6).

First, we introduce a function which finds the element in the abstract heap which corresponds to some heap g-variable.

Definition 4.35 (**G-variable in the abstract heap**). Let xs be the program abstract heap and *alloc* be the program allocation function. The function $gvar_{hid}(xs, alloc, g)$ finds an element in xs with the same displacement as displacement of $root_g(g)$. Note, that in the abstract heap we store displacement of a node (i.e. top level g-variable *with* the header). The allocation function *alloc*, however, returns address of a g-variable *without* node header, i.e. address of the first node content word. Thus, we have to subtract AUX_{size} from the displacement of the top-level g-variable in order to get the displacement of the heap node.

$$gvar_{hid}(xs, alloc, g) = \hat{\varepsilon}\ i.\ xs!i = \frac{fst(alloc(root_g(g))) - abase_{heap}}{4} - AUX_{size}$$

Definition 4.36 (**G-variable in the new heap**)**.** Let xs and xs' be abstract heaps before and after garbage collection respectively and F be a permutation function. The function $displ_{gc}$ computes the displacement of the root g-variable $root_g(g)$ in the heap after garbage collection is performed.

$$displ_{gc}(xs, xs', alloc, F)(g) = xs'!F(gvar_{hid}(xs, alloc, g)) + AUX_{size}$$

Definition 4.37 (**Allocation function after GC**)**.** In order to keep GC specific parameters (xs and F) out of the top-level specification we introduce the function $d_{gc} :: gvar \to \mathbb{N}$ as an input parameter. This function returns the displacement of a g-variable in the new heap. Later on we instantiate it with $displ_{gc}(xs, xs', alloc, F)$. Let $alloc$ be the allocation function before garbage collection. We define the allocation function after GC in the following way.

$$alloc_{gc}(sc, alloc, d_{gc})(g) = \begin{cases} alloc(g) & named_g(g) \\ (abase_{heap} + d_{gc}(g) \cdot 4 + displ_g(sc, g),\ asize_t(ty_g(sc, g))) & \text{otherwise} \end{cases}$$

Now we extend the allocation function with an entry for the new heap variable which contains its allocated base address and size.

Definition 4.38 (**Extended allocation function**)**.** Let i be the index of the newly allocated heap variable in the heap symbol table and b the allocated base address of the new heap variable in the heap memory. Let $alloc$ be the allocation function before the variable allocation (i.e. after execution of the last C0 statement or after garbage collection). Then, we define the new allocation function in the following way.

$$alloc_{xt}(sc, alloc, i, b)(g) = \begin{cases} (b + displ_g(sc, g),\ asize_t(ty_g(sc, g))) & root_g(g) = gvar_{hm}(i) \\ alloc(g) & \text{otherwise} \end{cases}$$

In case of successful garbage collection (when the variable is allocated after it) we need to apply $alloc_{gc}$ and $alloc_{xt}$ functions consequently to get the new allocation function.

4.4 Simulation Relation

In this section we give the formal definition for the modified simulation relation of the compiler simulation theorem (Sec. 2.5). The simulation relation presented here is a modified and extended version of the relation from [12]. In the frame of the thesis we want to keep the structure of the original simulation relation unchanged as much as possible in order to reuse lemmas and proofs done in [12]. Thus, we leave all three basic predicates (code consistency, control consistency, data consistency) unchanged with the exception of the small change in the register consistency relation (part of the data consistency) and add a new predicate for the garbage collector consistency (GC consistency). In the global sense, however, this might not be the best solution, because some conditions in the GC consistency predicate are the modified versions of conditions in data

consistency. Thus, applying more changes to the data consistency might make the structure of the GC consistency predicate simpler.

In the compiler with the garbage collector the consistency relation *consis* has four new parameters:

- $alive :: aliveT$: the heap alive function, which distinguishes variables currently present in the heap memory from the deleted ones;
- $f :: \mathbb{B}$: from-space flag. Indicates which part of the heap is currently in use by the program;
- $xs :: \mathbb{N}\ list$: abstract heap of the program. Contains displacements of nodes inside the heap memory;
- $stack :: \mathbb{N}\ list$: abstract stack of the program. Contains displacements of memory frames inside local memory.

All four introduced entities are initialized with some values at the first step of execution of the C0 machine and are then extended and modified on every step of the machine execution.

The only indispensable parameter here is the alive function. All the other parameters, in principal, can be removed from the simulation relation. We keep them for simplicity reasons due to the Hoare logic definition of the GC result (Chapter 3). The from-space flag f is stored in the GC part of the global memory and can be obtained from there. The abstractions xs and *stack* are nothing else than lists of displacements of some entities in the memory of the assembly machine, and can be constructed on every step of the C0 machine execution with the help of other parameters of the simulation relation. However, these manipulations would make the proofs much harder, while not making the specification much simpler.

We start from the top-level definition and then go down in order to show the structure of the relation before we present its formal definition. The overall structure of the simulation relation is depicted on Fig. 4.2.

Definition 4.39 (**Consistency**)**.** Let te be a type name environment, ft a function table, c a C0 configuration, d a configuration of the VAMP assembly machine, *alloc* an allocation function, *alive* an alive function, and f be the from-space indicator. We say that c and d are consistent if they fulfill the predicate *consis* which requires code consistency, control consistency, data consistency, and GC consistency.

$$\frac{\begin{array}{cc} consis_{code}(te, ft, c, d) & consis_{d}(te, ft, c, alloc, alive, f, d) \\ consis_{c}(te, ft, c, d) & consis_{gc}(te, ft, c, alloc, alive, f, d, xs, stack) \end{array}}{consis(te, ft, c, alloc, d, alive, f, xs, stack)}$$

Code consistency requires that the compiled code is stored at the address *progbase* in the assembly configuration. Control consistency specifies values of program counters and return address of stack frames. The latter requirement is formally defined with the predicate $consis_{ra}(te, ft, c, d)$. We do not give the formal definitions for code and control consistencies here, but focus on the data and GC consistency relations instead.

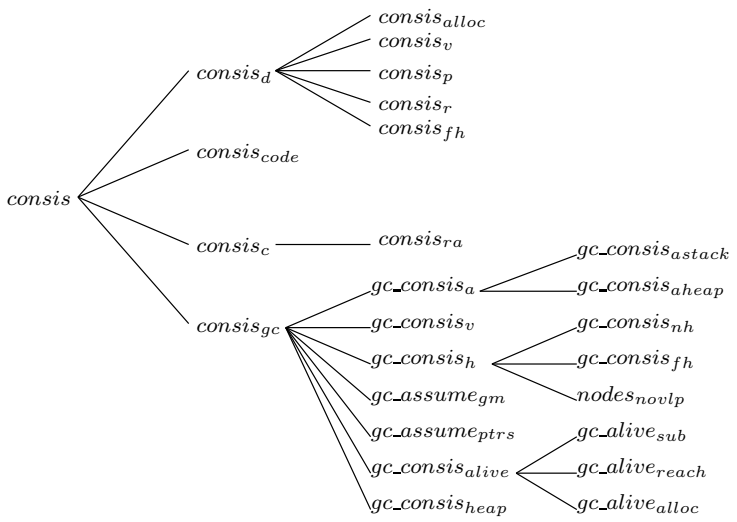

Figure 4.2: Overall structure of the simulation relation.

4.4.1 Data Consistency

Below we define data consistency using several auxiliary definitions.

Allocation Consistency. Here we fix properties of the allocation function. We require that the second component of the allocation function returns the allocated size of a g-variable and the first component returns the base address of a named g-variable. We cannot fix the absolute position of the heap g-variable in the memory, instead we require that variables are allocated above heap base, sub g-variables are properly placed relatively to their root g-variables, and that nameless g-variables with different roots do not overlap in the assembly memory. The last condition is not automatically fulfiled with the presence of the garbage collector, since it does not argue about node headers (in the compiler without GC heap nodes do not have headers). The predicate $nodes_{novlp}$, which is defined later, fixes the problem.

Definition 4.40 (**Named allocation consistency**)**.** Predicate $consis^{named}_{alloc}$ tests whether a given allocation function contains the correct data for named g-variables.

$$\frac{\begin{array}{l}\forall g \in gvars_{\surd}(sc(c.mem)).\ named_g(g) \longrightarrow \\ \qquad\qquad fst(alloc(g)) = abase_g(sc(c.mem), g)\ \wedge \\ \qquad\qquad snd(alloc(g)) = asize_t(ty_g(sc(c.mem), g))\end{array}}{consis^{named}_{alloc}(te, ft, c, alloc)}$$

Definition 4.41 (**Heap allocation consistency**)**.** The following predicate tests whether a given allocation function returns the correct data for nameless g-variables. Additionally, we require that the end of the top most local frame is

below the heap base and that the root nameless g-variables are properly aligned.

$$\frac{\begin{array}{l} abase_{lm}(te, ft, sc(c.mem), |c.mem.lm|) \leq abase_{heap} \\ \forall g \in gvars_{\surd}(sc(c.mem)).\ \neg named_g(g) \longrightarrow \\ \quad snd(alloc(g)) = asize_t(ty_g(sc(c.mem), g)) \\ \forall g' \in reachable_g^{nameless}(c.mem). \\ \quad fst(alloc(g')) = fst(alloc(root_g(g'))) + displ_g(sc(c.mem), g') \\ \quad \wedge\ align(ty_g(sc(c.mem), root_g(g'))) \mid fst(alloc(root_g(g'))) \\ \quad \wedge\ abase_{heap} \leq fst(alloc(g')) \\ \quad \wedge\ \forall h \in reachable_g^{nameless}(c.mem). \\ \qquad root_g(g') \neq root_g(h) \longrightarrow alloc(g') \asymp alloc(h) \end{array}}{consis_{alloc}^{heap}(te, ft, c, alloc)}$$

With the presence of a garbage collector we want to have a more accurate bound on the position of heap g-variables in the assembly memory. Depending on the current value of the from-space indicator f, all heap g-variables are located in one half of the heap or the other. This bound in fixed in the predicate gc_consis_{aheap}.

Definition 4.42 (**Allocation consistency**)**.** We combine the two previous definitions into one predicate

$$\frac{consis_{alloc}^{named}(te, ft, c, alloc) \qquad consis_{alloc}^{heap}(te, ft, c, alloc)}{consis_{alloc}(te, ft, c, alloc)}$$

Value Consistency. Value consistency requires that values of reachable non-pointer g-variables are correctly stored in the memory of the assembly machine. The predicate $consis_v^g(mc, alloc, d, g)$ sets the following property on a reachable, non-pointer, elementary g-variable g (for other g-variables it returns $true$):

$$vmatch(value_g(mc, g)(0), cell2int(d.mm(\frac{fst(alloc(g))}{4}))).$$

The predicate $vmatch :: mcell_{C0} \times \mathbb{Z} \to \mathbb{B}$ tests whether its input parameters contain equivalent values.

Definition 4.43 (**Value consistency**)**.** A C0 configuration c is value consistent with the assembly configuration d in the context of allocation function $alloc$ iff all g-variables in c are value consistent (the test for reachability and pointer g-variables occurs inside predicate $consis_v^g$).

$$consis_v(c, alloc, d) = \forall g.\ consis_v^g(c.mem, alloc, d, g)$$

Pointer Consistency. Pointer consistency states that the values of reachable pointer g-variables are properly represented in the assembly configuration. Analogously to value consistency, the predicate $consis_p^g(mc, alloc, d, g)$ tests whether the following property holds for a reachable, pointer g-variable g (for other g-variables it returns $true$):

$$vmatch_{ptr}(alloc, g, cell2int(d.mm(\frac{fst(alloc(g))}{4}))).$$

The predicate $vmatch_{ptr}(alloc, p, v_{asm})$ with a pointer p to a g-variable g returns true iff the assembly value of the pointer (interpreted as a natural number) equals the allocated base address of g.

Definition 4.44 (**Pointer consistency**)**.** A C0 configuration c is pointer consistent with the assembly configuration d in the context of allocation function *alloc* iff all g-variables in c are pointer consistent.

$$consis_p(c, alloc, d) = \forall g.\ consis_p^g(c.mem, alloc, d, g)$$

Register Consistency. Register consistency argues about the content of certain special registers of the assembly machine. It requires, that registers r_{sbase}, r_{lframe}, and r_{htop} have values according to their intended meaning, that no reachable nameless g-variables are allocated above the top most heap address r_{htop}, and that the value of r_{htop} is properly aligned. In order to formulate the correct value of the r_{htop} register we have to use the *alive* function. We require that the value of r_{htop} is equal to the start of the currently used half heap plus the size of all alive variables on the heap.

Definition 4.45 (**Register consistency**)**.** Formally we define register consistency in the following way.

$$\frac{\begin{array}{c} 4 \mid d.gpr!r_{htop} \\ i2n(d.gpr!r_{sbase}) = abase_{gm}(te, ft, gst(c.mem)) \\ i2n(d.gpr!r_{lframe}) = abase_{lm}(te, ft, sc(c.mem), |c.mem.lm| - 1) \\ i2n(d.gpr!r_{htop}) = abase_{heap} + st_{isx}(f) \cdot 4 + asize_{heap}^{alive}(sc(c.mem), alive) \\ \forall g \in reachable_g^{nameless}(c.mem).\ fst(alloc(g)) + snd(alloc(g)) \leq i2n(d.gpr!r_{htop}) \end{array}}{consis_r(te, ft, c, alloc, d, alive, f)}$$

Frame Header Consistency. Frame header consistency requires the values in the frame headers of the local frames to be consistent with their intended meaning. Here, we set the values only for the return destination and the previous stack pointer. The value of the return address is fixed in the control consistency and the value of the link to the pointer table is fixed in the GC consistency. The predicate $consis_{fh}(te, ft, c, alloc, d)$ requires for the i-th local frame (excluding 0-th frame) the following properties to hold:

$$\begin{array}{l} fh_{psp}(te, ft, sc(c.mem), d.mm, i) = abase_{lm}(te, ft, sc(c.mem), i - 1) \\ fh_{rd}(te, ft, sc(c.mem), d.mm, i) = fst(alloc(snd(c.mem.lm!i))) \end{array}$$

The functions fh_{psp} and fh_{rd} read from the assembly memory values of the previous stack pointer and the return destination from the frame header.

Data Consistency. We combine the given above definitions into one predicate.

Definition 4.46 (**Data consistency**)**.** The following predicate checks whether c and d are data consistent with respect to allocation function *alloc*, alive func-

tion *alive*, and from-space indicator f.

$$\frac{\begin{array}{c} consis_{alloc}(te, ft, c, alloc) \qquad consis_v(c, alloc, d) \qquad consis_p(c, alloc, d) \\ consis_r(te, ft, c, alloc, d, alive, f) \qquad consis_{fh}(te, ft, c, alloc, d) \end{array}}{consis_d(te, ft, c, alloc, d, alive, f)}$$

4.4.2 GC Consistency

GC consistency states properties of the configuration which are necessary for correct execution of the garbage collector routine and correct memory allocation of the new heap variable. We define the relation using several auxiliary definitions.

Abstractions Consistency. Abstractions consistency states a number of properties for the abstract heap xs and the abstract stack *stack*.

For the abstract stack we simply require that every element of the *stack* contains the displacement of the respective frame in the local memory stack $c.mem.lm$ and that the length of the *stack* equals the length of the local stack.

Definition 4.47 (**Abstract stack consistency**)**.**

$$\frac{\begin{array}{c} \forall l < |stack|.\ stack!l = \dfrac{abase_{lm}(te, ft, sc(c.mem), l) - abase_{lm}(te, ft, sc(c.mem))}{4} \\ |stack| = |c.mem.lm| \end{array}}{gc\_consis_{astack}(te, ft, c, stack)}$$

For the abstract heap xs we require the following properties:

- for every alive node from the heap symbol table $hst(c.mem)$ there exists a respective unique element in xs;
- for every element in the abstract heap xs there exists a respective unique alive node in the heap symbol table;
- the relation *HalfHeap* (Def. 3.16) holds for xs and heap memory HM^d of the assembly machine d;
- forward pointers of all nodes on the heap point to the from-space.

Definition 4.48 (**Abstract heap consistency**)**.** Formally we define abstract heap consistency in the following way.

$$\frac{\begin{array}{l} \forall i < |hst(c.mem)|.\ alive(gvar_{hm}(i)) \longrightarrow \\ \qquad \hat{\exists} j < |xs|.\ xs!j = \dfrac{fst(alloc(gvar_{hm}(i))) - abase_{heap}}{4} - AUX_{size} \\ \forall j < |xs|.\ \hat{\exists} i < |hst(c.mem)|.\ alive(gvar_{hm}(i)) \\ \qquad \wedge\ xs!j = \dfrac{fst(alloc(gvar_{hm}(i))) - abase_{heap}}{4} - AUX_{size} \\ HalfHeap(TT, HM^d, f, nhi^c, xs) \qquad \forall i < |xs|.\ \neg in_{to}(f, nid_{fwd}(TT, HM^d, i)) \end{array}}{gc\_consis_{aheap}(te, ft, c, alloc, alive, f, xs, d)}$$

Definition 4.49 (**Abstractions consistency**)**.** We combine the given above two definitions into abstractions consistency.

$$\frac{gc\_consis_{astack}(te, ft, c, stack) \qquad gc\_consis_{aheap}(te, ft, c, alloc, alive, f, xs, d)}{gc\_consis_{a}(te, ft, c, alloc, alive, f, xs, stack, d)}$$

Headers Consistency. Here, we set some requirements on the frame and node headers. Note, that some properties of the heap nodes are already covered in the abstractions consistency. In the headers consistency we focus on the value of the TT link stored in the headers. Additionally we formulate the full version of the last condition in Def. 4.41, i.e. we state that nameless g-variables with different roots and their headers do not overlap in the assembly memory.

Definition 4.50 (**Node headers consistency**)**.** We require that the value of the TT index of the i-th node is greater than the number of frames in the program, but less than the number of functions plus the number of types in the type environment. The index of the type of a g-variable in te (obtained by $read_{tid}$ function) points to the type of this g-variable (the type of the heap node). Formally we define the relation in the following way.

$$\frac{\begin{array}{l}\forall i < |hst(c.mem)|.\ alive(gvar_{hm}(i)) \longrightarrow \\ \quad read_{nid}(d, gvar_{hm}(i), alloc) \in (num_f : num_f + |te| - 1) \\ \quad \wedge\ snd(te!read_{tid}(d, gvar_{hm}(i), alloc)) = ty_g(sc(c.mem), gvar_{hm}(i))\end{array}}{gc\_consis_{nh}(te, ft, c, alloc, alive, d)}$$

Definition 4.51 (**Node headers do not overlap**)**.** We formulate the property only for top-level heap g-variables. We require that all root, valid, alive g-variables (i.e. unreachable, but alive g-variables are also considered) together with their headers do not overlap. To simplify the definition we introduce a new predicate $\asymp_n$, which checks whether two ranges extended by the size n are disjoint.

$$r_1 \asymp_n r_2 = (fst(r_1) - n \geq fst(r_2) + snd(r_2) \vee fst(r_2) - n \geq fst(r_1) + snd(r_1))$$

Formally, we define the main predicate in the following way.

$$\frac{\begin{array}{l}\forall g \in gvars_{\surd}(sc). \forall h \in gvars_{\surd}(sc).\ \neg named_g(g) \wedge \neg named_g(h) \\ \quad \wedge\ alive(g) \wedge alive(h) \wedge root_g(g) = g \wedge root_h(h) = h \longrightarrow \\ \quad alloc(g') \asymp_8 alloc(h)\end{array}}{nodes_{novlp}(sc, alloc, alive)}$$

Definition 4.52 (**Frame headers consistency**)**.** We require that the value of the TT index of the l-th frame is greater than zero and less or equal than the number of functions in the program configuration. The index of the function in the function table ft (obtained by $read_{fnum}(d, sc, l)$) points to the function which corresponds to the frame l, i.e. to the function with the same symbol table as the symbol table of l.

$$\frac{\begin{array}{l}\forall l < |c.mem.lm|.\ read_{fid}(d, sc(c.mem), l) \in (1 : |ft|) \\ \quad \wedge\ stbl_f(ft, read_{fnum}(d, sc(c.mem), l)) = c.mem.lm!l.st\end{array}}{gc\_consis_{fh}(te, ft, c, alloc, alive, d)}$$

Definition 4.53 (**Headers consistency**)**.** We combine the given above three definitions into one predicate.

$$\frac{gc\_consis_{nh}(te, ft, c, alloc, alive, d) \quad nodes_{novlp}(sc(c.mem), alloc, alive) \quad gc\_consis_{fh}(te, ft, c, alloc, alive, d)}{gc\_consis_{h}(te, ft, c, alloc, alive, d)}$$

Extended Heap Consistency. Here we define the position of a heap g-variable in the assembly heap memory. In the compiler without garbage collector this position is fixed with the predicate $consis^{heap}_{alloc}$. Now we have to set a tighter bound: depending on the value of f, a reachable g-variable is located in one half of the heap or the other.

Definition 4.54 (**Extended heap consistency**)**.** Depending on the value of the from-space indicator f, g-variable g is allocated either in one half of the heap or the other.

$$\frac{\begin{array}{l}\forall g \in reachable^{nameless}_{g}(c.mem).\\ abase_{heap} + (AUX_{size} + st_{idx}(f)) \cdot 4 \leq fst(alloc(g)) \wedge \\ fst(alloc(g)) + snd(alloc(g)) \leq abase_{heap} + (AUX_{size} + st_{idx}(f) + HHS) \cdot 4\end{array}}{gc\_consis_{heap}(te, c, alloc, f)}$$

Alive Consistency. Here we set some basic properties on the heap alive function.

Definition 4.55 (**Alive sub g-variables**)**.** If some g-variable g is alive then its root g-variable $root_g(g)$ is also alive. Analogously, the property holds in the opposite direction.

$$\frac{\forall g \in gvars_{\surd}(sc).\ alive(root_g(g)) \wedge \neg named_g(g) \longrightarrow alive(g) \quad \forall g \in gvars_{\surd}(sc).\ alive(g) \wedge \neg named_g(g) \longrightarrow alive(root_g(g))}{gc\_alive_{sub}(sc, alive)}$$

Definition 4.56 (**Reachable g-variables are alive**)**.** This is the main property of the alive function. If some nameless g-variable is reachable, then it is alive (i.e. present in the heap memory of the assembly machine).

$$\frac{\forall g \in reachable^{nameless}_{g}(c.mc).\ alive(g)}{gc\_alive_{reach}(c, alive)}$$

Lemma 4.20 (**Reachable heap equals alive heap**)**.** If both gc_alive_{reach} and $reach_{alive}$ predicates hold at the same time, then sizes of reachable and alive heaps are equal. I.e. in this case the heap contains only reachable alive nodes.

$$\begin{array}{l}gc\_alive_{reach}(c, alive) \wedge reach_{alive}(c.mc, alive)\\ \Longrightarrow asize^{reachable}_{heap}(c.mc, alive) = asize^{alive}_{heap}(sc(c.mc), alive)\end{array}$$

Proof. We omit the (simple) proof of this lemma here. □

Definition 4.57 (**Allocation function for alive g-variables**)**.** If some g-variable g is alive, then $fst(alloc(g))$ is divisible by four and is greater than $abase_{heap}$ plus the size of the node header. For reachable g-variables these properties can be derived from the heap allocation consistency (Def. 4.41). However, we want them to hold not only for reachable nameless g-variables, but for all heap g-variables currently present on the heap.

$$\frac{\forall g \in gvars_{\surd}(sc).\ alive(g) \wedge \neg named_g(g) \longrightarrow \\ fst(alloc(g)) \mid 4 \wedge fst(alloc(g)) \geq abase_{heap} + AUX_{size} \cdot 4}{gc\_alive_{alloc}(sc, alloc, alive)}$$

Definition 4.58 (**Alive consistency**)**.** We combine the given above properties into one predicate.

$$\frac{gc\_alive_{sub}(sc(c.mem), alive) \qquad gc\_alive_{reach}(c, alive) \qquad gc\_alive_{alloc}(sc(c.mem), alloc, alive)}{gc\_consis_{alive}(c, alloc, alive)}$$

GC Interface Consistency. In the GC interface consistency we define correct values of the variables located in the global memory of the garbage collector. First, we require that each of the nine GC variables defined in Sec. 4.2 is present in the symbol table of the global memory. The predicate

$$gc\_vars_{stbl}(sc, te)$$

states the property formally (we do not give the definition here).

Definition 4.59 (**GC interface consistency**)**.** We fix the values of GC variables in the C0 configuration c and assembly configuration d with the from-space indicator f. First, we require that all GC variables are present in the global symbol table of the program. Then we require that values of the global memory pointer, stack pointer, and heap pointer are equal to $abase_{gm}$, $abase_{lm}$, and $abase_{heap}$, respectively. The value of the next heap index is fixed with the help of the r_{htop} register and the values of type and pointer tables are restricted by the means of $gci_{\surd}$ predicate (Sec. 4.2.4).

$$\frac{\begin{array}{c} gc\_vars_{stbl}(sc(c.mem), te) \\ val_{gm}(c.mem) = abase_{gm}(te, ft, gst(c.mem)) \\ val_{heap}(c.mem) = val_{heap\_st}(c.mem) = abase_{heap} \\ val_{stack}(c.mem) = val_{stack\_st}(c.mem) = abase_{lm}(te, ft, gst(c.mem)) \\ val_f(c.mem) = f \qquad abase_{heap} + 4 \cdot val_{nhi}(c.mem) = i2n(d.gpr!r_{htop}) \\ gci_{\surd}(te, ft, gst(c.mem), val_{TT}(c.mem), val_{PP}(c.mem)) \end{array}}{gc\_consis_v(c, te, ft, f, d)}$$

GC Assumptions. We define two additional predicates which represent some basic assumptions on the program introduced with the integration of the garbage collector to the compiler. These properties are required by the simple garbage collector used in the frame of this thesis.

Definition 4.60 (**Heap pointers assumption**)**.** If some pointer g-variable x points to the heap, then it should point to the heap node (top-level g-variable). Pointers to sub-variables (e.g. element of the array or a structure component) on the heap are not allowed.

$$\frac{\forall x \in reachable_g(mc).\ \forall g.\ ptr_{gvar}(sc(mc), x) \wedge \neg named_g(g) \\ \wedge\ value_g(mc, x) = Ptr(g) \longrightarrow root_g(g) = g}{gc\_assume_{ptrs}(mc)}$$

We require that the GC interface does not contain any pointers to g-variables. Actually, we need this assumption to hold only for heap pointers, since these pointers located in the GC part of the global memory may have wrong values after garbage collection is performed (the garbage collector doesn't "see" pointers located in the GC part of the global memory and cannot handle them correctly). However, since the presence of pointers to variables in the GC global memory is not essential (and usually is not needed) in the implementation of a simple copying garbage collector, we decided to put a restriction on all GC pointers (both pointers to the heap and not to the heap). As mentioned before, pointers to memory addresses are not allowed in the C0 language and we use natural numbers to model them.

Definition 4.61 (**No pointers in GC interface**)**.** Formally we define the described property in the following way.

$$\frac{\forall g \in reachable_g(mc).\ root_g(g) = gvar_{gm}(x) \wedge \\ ptr_{gvar}(sc(mc), g) \longrightarrow \forall i.\ fst(gst_{gc}(gst(mc))!i) \neq x}{gc\_assume_{gm}(mc)}$$

GC Consistency. Now we have all we need to define GC consistency relation

Definition 4.62 (**GC consistency**)**.** Let c and d be C0 and VAMP assembly configurations respectively. Let *alloc* be the current allocation function and *alive* the current alive function of the program. Further, let f be a from-space indicator, xs be an abstract heap, and *stack* be an abstract stack respectively. We define GC consistency formally in the following way.

$$\frac{gc\_consis_{heap}(te, c, alloc, f) \qquad gc\_consis_h(te, ft, c, alloc, alive, d) \\ gc\_consis_{alive}(c, alloc, alive) \qquad gc\_consis_a(te, ft, c, alloc, alive, f, xs, stack, d) \\ gc\_consis_v(c, te, ft, f, d) \qquad gc\_assume_{ptrs}(c.mem) \qquad gc\_assume_{gm}(c.mem)}{consis_{gc}(te, ft, c, alloc, alive, f, d, xs, stack)}$$

4.5 Low-Level Correctness

The low-level correctness deals with the code generation algorithm of the compiler. It guarantees, that execution of the compiled code on the VAMP assembly machine starting from some configuration d will eventually result in a configuration d' with a number of fixed properties. In the high-level correctness proof we use the low-level correctness lemma in order to show that the simulation relation holds for configuration d' and c', where c' is the resulting C0 configuration after execution of the C0 transition function for the compiled statement.

Code generation algorithms for most statements in the compiler with and without garbage collector are identical. The only difference are statements *PAlloc* and *SCall*. For the dynamic memory allocation in place of a simple memory allocation code (present in the compiler without GC [12]) we now have a function call to the garbage collector routine. Note, that depending on implementation we also might need to add assembly code which sets the new value for the r_{htop} register here. For the statement *SCall* the code generation algorithm is extended with the code which fills the new frame header fields with correct data. In this paper we focus only on the correctness of the memory allocation statement. The proof of other statements, as well as changing of the code generation algorithm for a function call, remains as future work.

In this section we formulate low level correctness for the memory allocation statement. In order to do that we manually transform Theorem 3.1 from the Hoare logic to the assembly level. The correspondence between Theorem 3.1 and the resulting low-level correctness lemma is rather straightforward. However, the formal proof of this correspondence is not possible inside small step or Hoare logic semantics. It requires the use of a meta-theorem and remains as future work. Note also, that we do not explicitly define the code generation algorithm for the memory allocation statement, but we rather specify the result of the execution of the generated code on the VAMP assembly machine.

In this section by $stack_{st}$ we understand the allocated address of the stack computed in the C0 configuration c, by sp - the value of the stack pointer in the initial VAMP configuration d.

$$stack_{st} = abase_{lm}(te, ft, gst(c.mem)) \qquad sp = d.gpr!r_{htop}$$

The g-variables $gvar_f(sc(c.mem), te)$ and $gvar_{nhi}(sc(c.mem), te)$ are denoted by g_f and g_{nhi} respectively. Displacements of these g-variables in the GC part of the global memory are denoted by d_f and d_{nhi}. Let sc and gst be the program symbol configuration $sc(c.mem)$ and the global symbol table $gst(c.mem)$ respectively.

$$d_f = \frac{abase_g(sc, g_f) - (abase_{gm}(te, ft, gst) + asize_{st}(gst_p(gst), 0))}{4}$$
$$d_{nhi} = \frac{abase_g(sc, g_{nhi}) - (abase_{gm}(te, ft, gst) + asize_{st}(gst_p(gst), 0))}{4}$$

The set inf_σ is instantiated in the following way.

$$inf_\sigma = \{TT, PP, stack_{st}, sp, abase_{heap}\}$$

Remember, that by TT and PP we understand the values of type and pointer tables in the configuration c and by nhi^c the value of the next heap index variable in the GC interface.

4.5.1 Unsuccessful GC

We start from the case, when even after garbage collection is performed there is not enough heap space for allocation of a new variable. In Lemma 3.1 this case is covered with the predicate

$$gc\_failure(inf_\sigma, m_\sigma, m'_\sigma, nhi^c, nhi', f, f', stack, xs, xs', F, res).$$

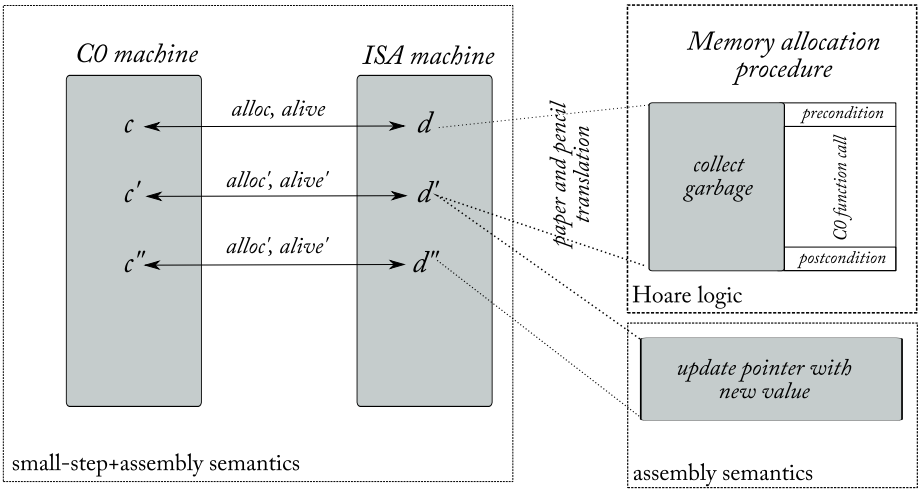

Figure 4.3: Integration of GC specification into small step semantics: unsuccessful garbage collection.

Let *alloc* be the program allocation function before execution of the *PAlloc* statement. By $alloc'$ we denote the modified allocation function as described in Sec. 4.3.3.

$$alloc' = alloc_{gc}(sc(c.mem), alloc, displ_{gc}(xs, xs', alloc, F))$$

We denote three states of the VAMP assembly machine: d - state before execution of *PAlloc* statement, d' - state after garbage collection is performed and values of the GC global variables changed, and d'' - state after a null value is assigned to a pointer. Integration of the Hoare logic specification into small step semantics for the case of unsuccessful GC is shown in Fig. 4.3.

We require the following properties to hold for the configurations d' and d'':

- execution of the memory allocation routine starting from configuration d eventually finishes after t cycles in the configuration d'';
- the value of the program counter *dpc* in d'' is equal to the value of *dpc* in d increased by the size of the generated code $size_{gc}^{code}$ (the size of the code generated by the compiler for the memory allocation statement)
- the predicate $gc\_failure$ holds for memories in configurations d and d';
- the new values of the next heap index and the from-space indicator are correctly stored in the GC part of the global memory of the machine d';
- the code memory region remains unchanged in d';
- a new value of the top of the heap is assigned to the r_{htop} register of the machine d';
- registers r_{sbase}, r_{lframe}, and the interrupt register r_{jal} are unchanged in configuration d'. Note, that we assume that execution of garbage collection routines does not produce any interrupts;

- all special general purpose registers are unchanged between d' and d'';
- memory of the configuration d'' equals memory of d', where the result pointer is updated with the null value. Note, that the result pointer might be located on the heap and, thus, be moved to another location;
- special purpose registers are unchanged between d and d''.

Definition 4.63 (**Result of unsuccessful GC**)**.** The predicate $failure_{gc}$ denotes the result of the execution of the compiled code on the VAMP assembly machine for the memory allocation statement in the case of unsuccessful garbage collection. Let m_σ be the set of initial memories $\{GM^d, LM^d, HM^d\}$ and m'_σ be the set of memories after garbage collection is performed. Input parameters for the predicate are: C0 configuration c, assembly configuration d, allocation function $alloc$, from-space indicator f, abstract heap xs and abstract stack $stack$, pointer g-variable g_{ptr} (where the address of the allocated variable has to be stored), and the set of initial memories. Output parameters are: assembly configurations d' and d'', new from-space indicator f', new abstract heap xs', and permutation function F. The function $b2n :: \mathbb{B} \to \mathbb{N}$ converts a boolean value to a natural value (either 1 or 0). We define the GC result in the following way.

$$\frac{\begin{array}{c}\exists t.\ d \xrightarrow[range_c, range_a]{t, d''.dpc} d'' \qquad d''.dpc = d.dpc + 4 \cdot size^{gc}_{code} \\ gc\_failure(inf_\sigma, m_\sigma, m'_\sigma, nhi^c, nhi', f, f', stack, xs, xs', F, res) \\ GCM^{d'} = GCM^d[d_f := b2n(f'), d_{nhi} := nhi'] \\ code_m(d, sc, te, ft) = code_m(d', sc, te, ft) \\ i2n(d'.gpr!r_{htop}) = abase_{heap} + 4 \cdot nhi' \\ \forall r \in [r_{sbase}, r_{lframe}, r_{jal}].\ d'.gpr!r = d.gpr!r \\ \forall r \in [r_{sbase}, r_{lframe}, r_{jal}, r_{htop}].\ d''.gpr!r = d'.gpr!r \qquad d''.spr = d.spr \\ d.mm'' = d.mm' \left[\frac{fst(alloc'(g_{ptr}))}{4} := int2cell(n2i(res))\right]\end{array}}{failure_{gc}(c, d, d', d'', alloc, f, f', xs, xs', stack, g_{ptr}, F)}$$

4.5.2 Successful GC

The case of successful garbage collection in Lemma 3.1 is covered with the predicate

$$gc\_ok(inf_\sigma, m_\sigma, m'_\sigma, m''_\sigma, nhi^c, nhi', nhi'', f, f', stack, xs, xs', xs'', F, asz, ti, res)$$

In this case we denote 4 states of the VAMP assembly machine: d - state before execution of *PAlloc* statement, d' - state after garbage collection is performed and values of the GC global variables changed, d'' - state after the newly allocated variable on the heap is filled with zeroes, and d''' - state after a value is assigned to a pointer. Integration of the Hoare logic specification into small step semantics for the case of successful GC is shown on Fig. 4.4.

We require the following properties to hold for configurations d', d'', and d''':

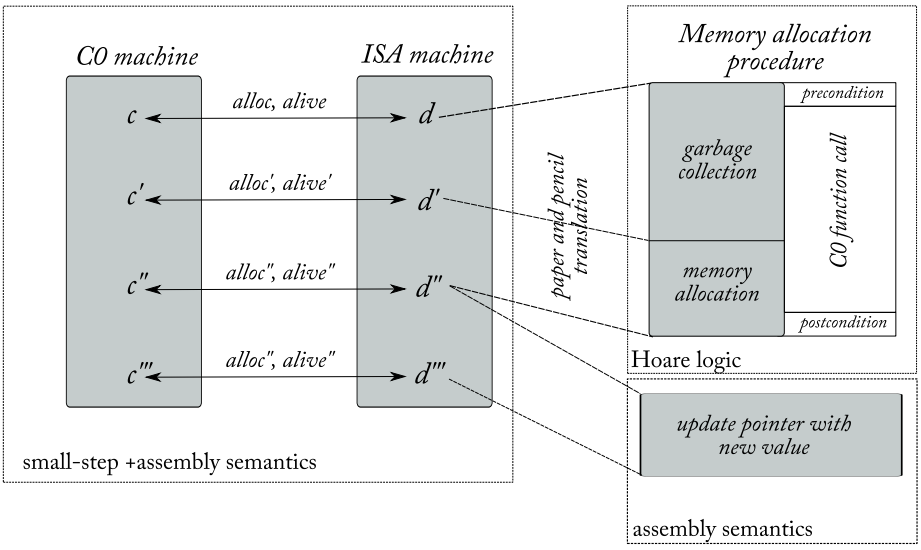

Figure 4.4: Integration of GC specification into small step semantics: successful garbage collection.

- execution of the memory allocation routine starting from configuration d eventually finishes after t cycles in the configuration d''';
- the value of the program counter dpc in d''' is equal to the value of dpc in d increased by the size of the generated code;
- the predicate gc_ok holds for memories in configurations d, d' and d'';
- the new values of the next heap index and the from-space indicator are correctly stored in the GC part of the global memory of the machine d'';
- code memory region remains unchanged in d'';
- a new value of the top of the heap is assigned to the r_{htop} register of the machine d'';
- registers r_{sbase}, r_{lframe}, and the interrupt register r_{jal} are unchanged in configuration d'';
- all special general purpose registers are unchanged between d'' and d''';
- memory of the configuration d''' equals memory of d'', where the result pointer is updated with the new value. Note, that the result pointer might be located on the heap and, thus, be moved to another location;
- special purpose registers are unchanged between d and d'''.

Definition 4.64 (Result of successful GC). The predicate $success_{gc}$ denotes the result of the execution of the compiled code on the VAMP assembly machine for the memory allocation statement in case of successful garbage collection. Let m_σ be the set of initial memories $\{GM^d, LM^d, HM^d\}$, m'_σ be the set of memories after garbage collection is performed, i.e. $\{GM^{d'}, LM^{d'}, HM^{d'}\}$, and

m''_σ be the set of memories after the new heap variable is initialized with zeroes. Input parameters for the predicate are the same as in Def. 4.63 plus the size of the variable we want to allocate (asz) and index of the type in the type name environment (t_{id}). Output parameters are: assembly configurations d', d'', and d''', new from-space indicator f', new abstract heaps xs' and xs'' (heap after garbage collection and initializing new variable, respectively), and permutation function F.

$$\frac{\begin{array}{c}\exists t.\ d \xrightarrow[range_c, range_a]{t, d'''.dpc} d''' \qquad d'''.dpc = d.dpc + 4 \cdot size^{gc}_{code} \\ gc\_ok(inf_\sigma, m_\sigma, m'_\sigma, m''_\sigma, nhi^c, nhi', nhi'', f, f', stack, xs, xs', xs'', F, asz, ti, res) \\ GCM^{d''} = GCM^{d'} = GCM^{d}[d_f := b2n(f'), d_{nhi} := nhi'] \\ code_m(d, sc, te, ft) = code_m(d'', sc, te, ft) \\ i2n(d''.gpr!r_{htop}) = abase_{heap} + 4 \cdot nhi'' \\ \forall r \in [r_{sbase}, r_{lframe}, r_{jal}].\ d''.gpr!r = d.gpr!r \\ \forall r \in [r_{sbase}, r_{lframe}, r_{jal}, r_{htop}].\ d'''.gpr!r = d''.gpr!r \qquad d'''.spr = d.spr \\ d.mm''' = d.mm'' \left[\frac{fst(alloc'(g_{ptr}))}{4} := int2cell(n2i(res))\right]\end{array}}{success_{gc}(c, d, d', d'', d''', alloc, f, f', xs, xs', xs'', stack, t_{id}, asz, g_{ptr}, F)}$$

4.5.3 Allocation Without GC

In the case when there is enough heap space for allocation of a new variable in the currently used half of the heap, garbage collection is not performed. In Lemma 3.1 this case is covered with the predicate

$$alloc_{post}(inf_\sigma, m_\sigma, m'_\sigma, f, nhi^c, nhi', xs, asz, res, t_{id}).$$

The result of this case is very similar to the original memory allocation statement in the compiler without garbage collector. The only difference is that we have to update the next heap index variable in the GC interface with the new value. Analogously to the previous cases we distinguish three states: d - state before execution of *PAlloc* statement, d' - state after we update the value of the next heap index variable and fill the newly allocated variable with zeroes, d'' - state after a value is assigned to a pointer.

Definition 4.65 (Result of allocation without GC). Predicate $alloc_{no\_gc}$ denotes the result of the execution of the compiled code on the VAMP assembly machine for the memory allocation statement in case of allocation without garbage collection. We require that $alloc_{post}$ predicate holds, that the next heap index variable is correctly updated with the new value, the result pointer is updated with the address of the new variable, and the heap top register is updated

with the new heap top value.

$$\frac{\begin{array}{c}\exists t.\ d \xrightarrow[range_c, range_a]{t, d''.dpc} d'' \qquad d''.dpc = d.dpc + 4 \cdot size^{gc}_{code} \\ alloc_{post}(inf_\sigma, m_\sigma, m'_\sigma, f, nhi^c, nhi', xs, asz, res, t_{id}) \\ GCM^{d'} = GCM^d[d_{nhi} := nhi'] \\ d.mm'' = d.mm'\left[\frac{fst(alloc'(g_{ptr}))}{4} := int2cell(n2i(res))\right] \\ i2n(d''.gpr!r_{htop}) = abase_{heap} + 4 \cdot nhi' \\ \forall r \in [r_{sbase}, r_{lframe}, r_{jal}].\ d''.gpr!r = d.gpr!r \qquad d''.spr = d.spr\end{array}}{alloc_{no\_gc}(c, d, d', d'', alloc, f, xs, stack, t_{id}, asz, g_{ptr})}$$

4.5.4 Precondition

Precondition to Lemma 3.1 is expressed with the predicate

$$new_{pre}(inf_\sigma, m_\sigma, nhi^c, f, asz, t_{id}, xs, stack)$$

from Def. 3.33. In the next lemma we derive the above predicate from the simulation relation.

Lemma 4.21 (**Allocation precondition**)**.** Let t be the type of the heap variable we want to allocate, t_{id} be the index of this type in the type name environment. Further, let t be a valid type and the size of the type be in a range of 32-bit numbers, the maximal address on the heap be also in the range of 32-bit numbers. By m_σ we denote the set of memories of the assembly machine d. Then the following lemma holds.

$$\begin{array}{l} c \in conf_{\surd}(te, ft) \wedge consis(c, alloc, d, alive, f, xs, stack) \\ \wedge\ snd(tenv!t_{id}) = t \wedge valid_{ty}(te, t) \\ \wedge\ (asize_{ty}(t) + 4 \cdot AUX_{size}) \in \mathbb{N}_{32} \wedge (abase_{heap} + asize^{max}_{heap}) \in \mathbb{N}_{32} \\ \Longrightarrow new_{pre}(inf_\sigma, m_\sigma, nhi^c, f, \frac{asize_{ty}(t)}{4}, t_{id}, xs, stack) \end{array}$$

Proof. First we use Lemma 4.19 to conclude $correct_{gci}(TT, PP)$. In order to show that predicate *Roots* holds for local and global roots we need to show that every heap pointer in the *roots* array points to some node in the abstract heap. To do so we first find a respective pointer g-variable g in the local or global memory of the C0 program. Then we use pointer consistency to show that g points to another g-variable x. Finally we use abstractions consistency (Def. 4.49) to show that g points to the node with the index $gvar_{hid}(xs, alloc, x)$ (Def. 4.35). For the heap pointers in the memory HM^d we proceed in analogues way and derive the predicate $heap\_Roots(HM^d, xs)$.

All the other conjuncts in new_{pre} predicate follow from the preconditions to the lemma and from the GC interface consistency (Def. 4.2.4). □

4.5.5 Correctness Lemmas

Now we formulate the low-level correctness lemmas. We formulate three lemmas, one for each of the possible cases.

Lemma 4.22 (**Low-level correctness: successful GC**)**.** Let s be the first statement in the program rest of the configuration c (and be a memory allocation statement) and t_n be the name of the type t in the type name environment te. The function $leval(te, mc, e)$ calculates the g-variable corresponding to the left expression of the C0 statement (in our case it is a result pointer). Note, that we calculate the result pointer after we perform an update of the GC variables in the C0 memory. Further, let $code_{gc}$ be the code, generated by the compiler for the memory allocation statement, asz be the allocated size of the type t in words, and c' be C0 memory configuration after GC variables are updated with new values. Successful garbage collection occurs iff the predicate $no\_gc\_needed(mc, t, f)$ from Def. 4.29 does not hold and the predicate $avail_{heap}(te, mc, t)$ from the Def. 4.26 holds. We formulate the first low level correctness lemma in the following way.

$$
\begin{aligned}
&asm_{pre}(d, range_c, code_{gc}) \wedge new_{pre}(inf_\sigma, m_\sigma, nhi^c, f, asz, t_{id}, xs, stack) \\
&\wedge s = PAlloc(e, t_n) \wedge \lfloor c' \rfloor = \delta_{gc}(c, t) \wedge g_{ptr} = leval(te, c'.mem, e) \\
&\wedge \neg no\_gc\_needed(c.mem, t, f) \wedge avail_{heap}(te, c.mem, t) \\
&\Longrightarrow \exists d', d'', d''', f', xs', xs'', F. \\
&\quad success_{gc}(c, d, d', d'', d''', alloc, f, f', xs, xs', xs'', stack, t_{id}, asz, g_{ptr}, F)
\end{aligned}
$$

Lemma 4.23 (**Low-level correctness: unsuccessful GC**)**.** Let us assume the same parameters as in Lemma 4.22. Unsuccessful GC occurs iff both predicates $no\_gc\_needed(mc, t, f)$ and $avail_{heap}(te, mc, t)$ do not hold.

$$
\begin{aligned}
&asm_{pre}(d, range_c, code_{gc}) \wedge new_{pre}(inf_\sigma, m_\sigma, nhi^c, f, asz, t_{id}, xs, stack) \\
&\wedge s = PAlloc(e, t_n) \wedge \lfloor c' \rfloor = \delta_{gc}(c, t) \wedge g_{ptr} = leval(te, c'.mem, e) \\
&\wedge \neg no\_gc\_needed(c.mem, t, f) \wedge \neg avail_{heap}(te, c.mem, t) \\
&\Longrightarrow \exists d', d'', f', xs', F.\ failure_{gc}(c, d, d', d'', alloc, f, f', xs, xs', stack, g_{ptr}, F)
\end{aligned}
$$

Lemma 4.24 (**Low-level correctness: no GC**)**.** Let us assume the same parameters as in Lemma 4.22. No garbage collection is done in the case when there is enough heap memory for a variable allocation in the free part of the currently used half of the heap. In this case the predicate $no\_gc\_needed(mc, t, f)$ holds.

$$
\begin{aligned}
&asm_{pre}(d, range_c, code_{gc}) \wedge new_{pre}(inf_\sigma, m_\sigma, nhi^c, f, asz, t_{id}, xs, stack) \\
&\wedge s = PAlloc(e, t_n) \wedge \lfloor c' \rfloor = \delta_{gc}(c, t) \wedge g_{ptr} = leval(te, c'.mem, e) \\
&\wedge no\_gc\_needed(c.mem, t, f) \\
&\Longrightarrow \exists d', d''.\ alloc_{no\_gc}(c, d, d', d'', alloc, f, xs, stack, t_{id}, asz, g_{ptr})
\end{aligned}
$$

In the frame of the thesis we do not prove the above three lemmas. However, in the next section we present some ideas about how the formal proof of the low level correctness can be obtained.

4.5.6 Low-Level Correctness Proof

In the thesis we do not present a formal proof of the low-level compiler correctness for memory allocation statement. The statements of the low-level correctness lemmas are obtained by manual transformation of the Hoare logic

correctness directly to the assembly level. However, in order for the compiler correctness to be complete, a formal proof of the low-level correctness would be required. The task is non trivial. Below we present some ideas about the way how low-level correctness can be formally shown.

Generally speaking, our goal is to show that execution of the assembly code compiled for the *PAlloc* statement starting from assembly configuration d will result in some assembly configuration d' (d'', d'''), which satisfies the conditions stated in Def. 4.63, Def. 4.64, and Def. 4.65. Normally, one would provide the exact code generated for the statement and would show that execution of this code results in d' with the desired properties. That's how the low level correctness of the assembly code for all other C0 statements is shown [12]. In our case, however, we want to use the results of the GC correctness on the Hoare logic level (Sec. 3). Thus, the approach from [12] is not directly applicable here.

A different solution of the problem is to show compiler correctness for C0 programs without *PAlloc* statement. Then we could replace all *PAlloc* statements in the C0 program with a function call to the GC routine (the GC routine itself does not contain any *PAlloc* statements), compile this code, and obtain the corresponding configuration d'. The major problem here is that the GC routine operates with arrays of natural numbers (LM, HM, GM), which are considered to be regular C0 variables by the routine, but practically are aliases for portions of VAMP assembly memory, which are also accessible via normal C0 variables. Thus, we cannot use the regular compiler correctness theorem to argue about properties of these variables. We need a special compiler correctness theorem, which takes care of those aliased arrays.

Basically, we would proceed in the following way (Fig. 4.5):

1. use a meta theorem to bring results of the GC routine execution (Theorem 3.1) from the Hoare logic to the small-step semantics level [1,18]. Aliased arrays remain regular C0 variables here;

2. show correctness of the compiler for C0 programs without memory allocation statement. We do not access or use aliased variables in the program outside the GC routine, so we can use a regular compiler correctness theorem [12];

3. in the C0 source code program, which has to be compiled, replace all occurrences of the statement *PAlloc* with a function call to the GC routine together with some inline assembly code (which updates the value of the register r_{htop}). Use results from (1) to get C0 configuration c^* with a number of properties obtained from the Hoare logic GC specification (Fig. 4.5).

4. define a special compiler simulation relation which takes care of aliased variables. Show compiler correctness with the new simulation relation for the program without memory allocation statement. Note, that all global variables used by GC routines are allocated in the GCM part of the assembly memory and cannot be accessed through aliased arrays. The only memory region which can be accessed through both C0 variables and aliased array is the local memory frame corresponding to one of the procedures included in the GC routine. However, this local frame (or several local frames) is located on the top of the memory stack and is

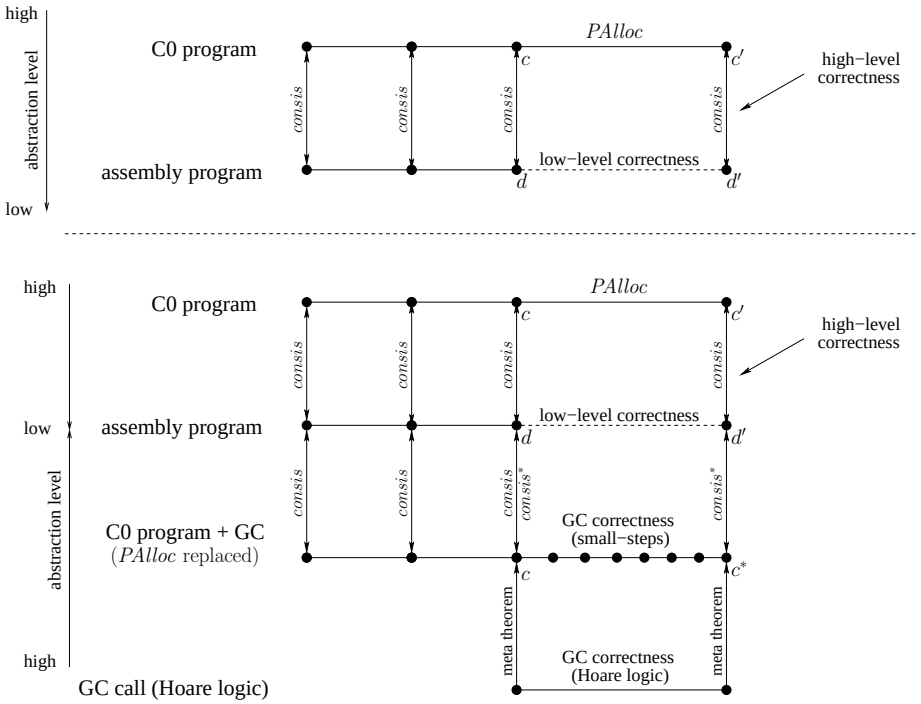

Figure 4.5: Computations of the C0 and assembly programs without and with the presence of GC.

not actually modified by the GC routine. Thus, aliasing is done in a "friendly" way, i.e. in one part of the program (the normal C0) *only* the "normal" C0 variables are used and in the other (the garbage collector implementation) *only* the arrays are used.

5. use the special compiler correctness theorem from (4) to show that execution of compiled code from (3) results in desired configuration d' (with the help of the new simulation theorem we can transfer required properties from C0 configuration c^* to assembly configuration d') (Fig. 4.5).

Step (4) of the suggested algorithm is the most time consuming, because special compiler correctness has to be shown for every C0 statement (except *PAlloc*). The low-level correctness proof remains as a possible future work.

4.6 High-Level Correctness

The high-level correctness deals with the program simulation relation. Our goal is to show that the low-level behavior of the assembly code achieves consistency.

We start from the formulation of the high-level correctness theorem for the memory allocation statement and then proceed with its proof. The major effort is to show that data and GC consistencies hold after execution of the *PAlloc* statement.

Lemma 4.25 (**Data and GC consistency**). Let s be the first statement in the program rest of the configuration c, t_n be the name of the type t in the type name environment te, $size^{gc}_{code}$ be the size of the assembly code generated for the statement *PAlloc*. Further, let c' be C0 memory configuration after *PAlloc* is executed. The function $avail_{mem}$ checks whether the maximal possible heap address fits into the program address space bounded by the address $addr_{max}$. The set $xltbl_{prog}$ denotes the set of translatable C0 programs (i.e. programs where all expressions can be evaluated and there are enough free registers for statements' execution). The function $\#ret_{top}(s2l(c.prog))$ calculates the number of return statements in the program rest *c.prog*. We state that execution of the compiled code for the statement *PAlloc* on the assembly machine will eventually result in a configuration d', which is GC and data consistent with the new C0 configuration c'.

$$
\begin{aligned}
& c \in conf_{\surd}(te, ft) \wedge consis(c, alloc, d, alive, f, xs, stack) \\
& \wedge\ s = PAlloc(e, t_n) \wedge (te, ft, gst(c.mem)) \in xltbl_{prog} \wedge \lfloor c' \rfloor = \delta_{C0}(c, t) \\
& \wedge\ abase_{heap} + asize^{max}_{heap} < addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \\
& \wedge\ \#ret_{top}(s2l(c.prog)) + 1 = |c.mem.lm| \wedge asm_{pre}(d, range_c, code_{gc}) \\
& \wedge\ (asize_{ty}(t) + 4 \cdot AUX_{size}) \in \mathbb{N}_{32} \\
& \wedge\ snd(tenv!t_{id}) = t \wedge valid_{ty}(te, t) \\
& \Longrightarrow \exists t, d', alloc', alive', xs', f'.\ d \xrightarrow[range_c, range_a]{t, d'.dpc} d' \\
& \qquad\qquad \wedge\ consis_d(te, ft, c', alloc', d', alive', f') \\
& \qquad\qquad \wedge\ consis_{ra}(te, ft, c', d') \\
& \qquad\qquad \wedge\ consis_{gc}(te, ft, c', alloc', alive', f', d', xs', stack) \\
& \qquad\qquad \wedge\ d.spr' = d.spr \wedge d''.dpc = d.dpc + 4 \cdot size^{gc}_{code}
\end{aligned}
$$

Proof. The proof is divided into three cases: successful GC, unsuccessful GC and no GC. Here we give the proof of the most complicated case - successful garbage collection. Condition for the case is

$$\neg no\_gc\_needed(c.mem, t, f) \wedge avail_{heap}(te, c.mem, t).$$

In order to prove the goal, we model the execution of the memory allocation statement and construct assembly configurations d', d'', and d''' as in Sec. 4.5.2. For this purpose we use Lemma 4.22 which gives us predicates

$$
\begin{aligned}
& success_{gc}(c, d, d', d'', d''', alloc, f, f', xs, xs', xs'', stack, t_{id}, asz, g_{ptr}, F) \\
& gc\_ok(inf_\sigma, m_\sigma, m'_\sigma, m''_\sigma, nhi^c, nhi', nhi'', f, f', stack, xs, xs', xs'', F, asz, ti, res)
\end{aligned}
$$

We distinguish four C0 configurations: c - configuration before execution of *PAlloc* statement, c' - configuration after GC variables are updated, c'' - configuration after the heap is extended with the new variable, and c''' - the final configuration (where the pointer value is assigned to the result pointer). To establish the connection between C0 and assembly configurations on different stages of the memory allocation we distinguish allocation functions $alloc'$ and

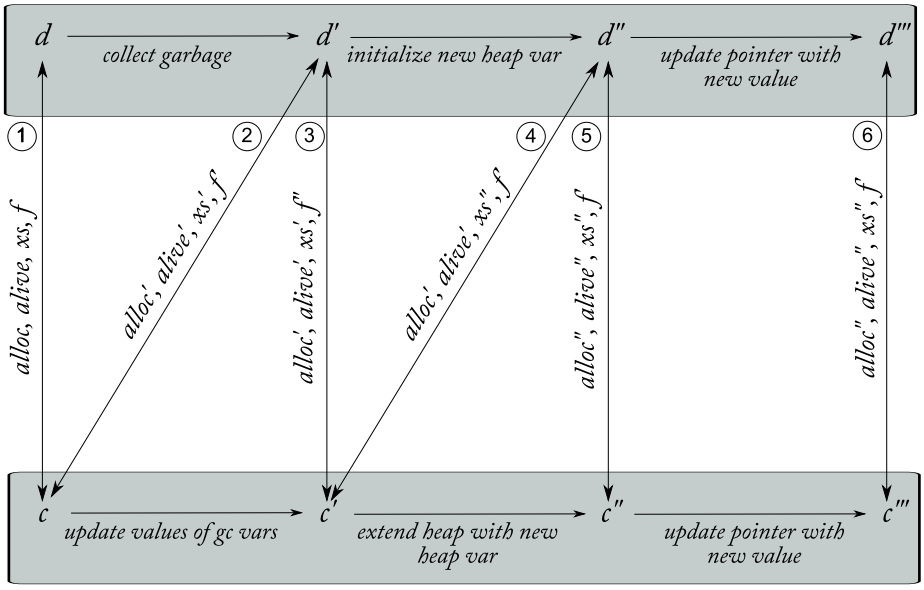

Figure 4.6: Proof strategy for the memory allocation statement.

$alloc''$ and alive functions $alive'$ and $alive''$.

$$
\begin{aligned}
alloc' &= alloc_{gc}(sc(c.mem), alloc, displ_{gc}(xs, xs', alloc, F)) \\
alloc'' &= alloc_{xs}(sc(c.mem), alloc_{gc}, |hst(c.mem)|, abase_{heap} + 4 \cdot nhi') \\
alive' &= alive_{gc}(c.mem) \\
alive'' &= alive_{xs}(c.mem, alive_{gc}, |hst(c.mem)|)
\end{aligned}
$$

We prove different parts of the simulation relation in different stages of memory allocation. The proof process is shown on Fig. 4.6

In the frame of the thesis it is impossible to present the full formal correctness proof for the memory allocation statement (it required more than 580 lemmas with about 40.000 proof steps in Isabelle). Here we briefly describe all stages of the proof process and later in this section focus on the most interesting aspects.

1. Initial stage. Consistency relation holds, next statement in the program rest of the configuration c is the memory allocation statement.

2. Garbage collection on the assembly machine has been performed, values of the global GC variables are changed on the assembly level, but remain unchanged on the C0 level. Abstract heap xs' represents the heap after garbage collection has been performed. Most parts of the simulation relation (Def. 4.4) hold for c and d' in the context of $alloc'$, $alive'$, f', and xs'. Value consistency (Def. 4.43) holds for all variables except the ones located in the GC part of the global memory. The proof of the value consistency is shown in Sec. 4.6.3. Pointer consistency (Def. 4.44) holds for all g-variables, since GC interface does not contain any pointers to variables. The proof of pointer and (heap) data consistencies at this stage is not trivial, since we have to build a bridge between g-variables in the

C0 configuration c and new values of pointers in the assembly memory d', which is specified by the *gc_ok* predicate (Def. 3.36). Another non-trivial proof is required for the heap allocation consistency (Def. 4.41) and extended heap consistency (Def. 4.54). The new allocation function $alloc'$ uses permutation relation F to define the new order of the nodes on the heap. To prove heap allocation consistency, we first have to argue about reachable nodes and show that if g is a reachable g-variable, then the index $gvar_{hid}(xs, alloc, g)$ belongs to a set $RN(HM, roots, xs)$ (proof in Sec. 4.6.1). Then we have to show that isomorphism holds on the indices of g-variables in the heaps xs and xs', i.e.

$$F(gvar_{hid}(xs, alloc, g)) = gvar_{hid}(xs', alloc', g).$$

After that we can use *gc_ok* to find the new location of the g-variable on the heap and prove the properties of the allocation function $alloc'$. However, the proof of reachability and isomorphism itself requires some properties stated in heap allocation and extended heap allocation consistencies. Thus, we have to introduce weaker versions of these predicates and prove them first. We also introduce a stronger version of the $reachable_g^{nameless}$ predicate to make the induction proofs simpler. Proof of the isomorphism is done together with the pointer consistency in one lemma and is shown in Sec. 4.6.2. The proof of the heap allocation consistency is given in Sec. 4.6.4.

Register consistency (Def. 4.45) does not hold at this stage since we specify the new value of the r_{htop} register only in the configuration d''. However, in order to be able to show register consistency on later stages, we have to formulate the part of the relation concerning reachable nameless g-variables and prove it on this stage (in place of the register r_{htop} we use index nhi'). Analogously, GC interface consistency does not fully hold since it also uses the value of r_{htop}.

3. At this stage we update values of GC variables at the C0 level (values at the assembly level were updated on the previous stage). All parts of the simulation relation which hold at the stage 2 are shown for the new C0 configuration c'. Now we can also show value consistency for all variables. At this stage we also prove a property, that the set of reachable g-variables remains unchanged, i.e.

$$reachable_g(c.mem) = reachable_g(c'.mem).$$

We still do not set the new value of r_{htop} register and cannot show register consistency and GC interface consistency.

4. We initialize the allocated memory with zero values. The heap at the C0 level is not yet extended and the set of reachable variables remains unchanged. Because the heap is not extended, allocation and alive functions are also not changed. However, the abstract heap xs'' (which represents the heap assembly memory $HM^{d''}$) is already extended with the new heap variable, i.e. $xs'' = xs' \circ [nhi']$. Thus, abstract heap consistency does not hold at this stage. The r_{htop} register is updated with the new value $(abase_{heap} + 4 \cdot nhi')$ and we can now show register and GC interface consistency.

5. We extend the heap memory of the C0 machine c' with the new variable. All the simulation relations except pointer and value consistency are derived from the previous stage. Since we update allocation and alive functions with $alive''$ and $alloc''$, abstractions consistency also holds at this stage. Pointer and value consistencies are proved in a similar manner as in [12], i.e. they are shown only for old C0 variables (for which $root_g(g) \neq gvar_{hm}(|hst(c''.mem)| - 1)$).

6. At this stage we update the value of the result pointer at the C0 and assembly levels. The proof of GC consistency is divided into two parts: first we show that it holds after assembly memory update (it is true since we do not change any data in the frame or node headers during pointer update), then we extend it for the C0 pointer update (during C0 pointer update we do not change any variables in GC interface, thus GC consistency still holds).

   For the data consistency we proceed in the following way. At first we show that pointer and value consistencies hold for the newly allocated g-variable. Then we show that for all g-variables – except the result pointer – assembly and C0 memories are unchanged. Thus, value and pointer consistency still hold for all g-variables except the result pointer. The memory cell for the pointer is filled with the allocated address of the new heap variable, thus data and pointer consistencies also hold for the result pointer. The new heap g-variable is allocated below the value stored in the $d'''.gpr(r_{htop})$ register (which is equal to nhi'') and register consistency holds. Frame header consistency holds, because we do not change data in the headers of the frames. Finally, allocation consistency cannot be affected by the C0 or assembly variable update and, obviously, holds.

□

Theorem 4.1 (**High-level correctness**)**.** Assume the same preconditions as in Lemma 4.25. The function $csize_s(te, ft, gst, f)$ calculates the size of the generated assembly code for the first statement in the program rest. We state that execution of the compiled code for the statement *PAlloc* on the assembly machine will eventually result in a configuration d', which is completely consistent with the next C0 configuration c'.

$$
\begin{aligned}
& c \in conf_{\surd}(te, ft) \wedge consis(c, alloc, d, alive, f, xs, stack) \\
& \wedge\ s = PAlloc(e, t_n) \wedge (te, ft, gst(c.mem)) \in xltbl_{prog} \wedge \lfloor c' \rfloor = \delta_{C0}(c, t) \\
& \wedge\ abase_{heap} + asize_{heap}^{max} < addr_{max} \leq 2^{32} \wedge avail_{mem}(addr_{max}, te, ft, c) \\
& \wedge\ \#ret_{top}(s2l(c.prog)) + 1 = |c.mem.lm| \wedge asm_{pre}(d, range_c, code_{gc}) \\
& \wedge\ (asize_{ty}(t) + 4 \cdot AUX_{size}) \in \mathbb{N}_{32} \\
& \wedge\ snd(tenv!t_{id}) = t \wedge valid_{ty}(te, t) \\
& \wedge\ the(csize_s(te, ft, gst(c.mem), f)) = size_{code}^{gc} \\
& \Longrightarrow \exists t, d', alloc', alive', xs', f'.\ d \xrightarrow[range_c, range_a]{t, d'.dpc} d' \\
& \qquad\qquad \wedge\ consis(te, ft, c, alloc, d, alive, f, xs, stack) \\
& \qquad\qquad \wedge\ d.spr' = d.spr
\end{aligned}
$$

Proof. To conclude the goal we use Lemma 4.25 and the proof of control consistency from [12] (Theorem 10.5) extended for the new simulation relation. [2] □

4.6.1 Reachable nodes

The set of reachable heap g-variables at the C0 level is denoted with the set $reachable_g^{nameless}(c.mem)$. On the Hoare logic level the set of reachable heap nodes is denoted by $RN(HM^d, roots, xs)$, where $roots$ is the set of pointers located in global and local memories of the C0 program, constructed with the help of type and pointer tables from the GC interface. In order to show that some heap g-variable g, which is reachable before garbage collection, is reachable after it too, we need to establish the connection between g and the corresponding node in the set RN. We establish this connection with the help of the function $gvar_{hid}(xs, alloc, g)$ (Def. 4.35).

In order to be able to use the following lemma in the proof of the allocation consistency, we introduce a weaker predicate for the heap allocation consistency.

$$\frac{consis_{alloc}^{named}(te, ft, c, alloc) \qquad consis_{alloc}^{wheap}(te, ft, c, alloc)}{consis_{alloc}^{weak}(te, ft, c, alloc)}$$

The definition of the predicate $consis_{alloc}^{wheap}(te, ft, c, alloc)$ is the same as the original heap allocation consistency definition (Def. 4.41) with the last condition (overlapping g-variables) removed. Without the last conjunct, we can show heap allocation consistency without arguing about position of the node on the heap after GC. Thus, we can use these properties to show isomorphism for g-variables, which then will be used to show the last conjunct in $consis_{alloc}^{heap}$. In a similar way we introduce a weaker version of the extended allocation consistency (Def. 4.54).

$$gc\_consis_{aheap}^{weak}(te, c, alloc, f)$$

Lemma 4.26 (**Reachable g-variable in RN**)**.** Let g be a reachable heap g-variable. Further, let $roots$ denote the set of pointers in the global and local memories of the program, i.e.

$$roots = roots_{gm}(TT, PP, GM^d) \circ roots_{lm}(TT, PP, LM^d, stack)$$

Then the index of the corresponding heap node obtained with $gvar_{hid}(xs, alloc, g)$ is present in the set of reachable nodes.

$$\begin{aligned}
&c \in conf_{\surd}(te, ft) \wedge gci_{\surd}(te, ft, gst(c.mem), TT, PP) \\
&\wedge consis_{alloc}^{weak}(te, ft, c, alloc) \wedge consis_p(c, alloc, d) \\
&\wedge gc_consis_{aheap}^{weak}(te, c, alloc, f) \wedge gc_consis_a(te, ft, c, stack) \\
&\wedge gc_consis_{fh}(te, ft, c, alloc, alive, d) \wedge gc_alive_{reach}(c, alive) \\
&\wedge gc_assume_{gm}(c.mem) \wedge g \in reachable_g^{nameless}(c.mem) \\
&\Longrightarrow gvar_{hid}(xs, alloc, g) \in RN(TT, PP, HM^d, roots, xs)
\end{aligned}$$

[2] Lemma: control_consistency_no_return_gc

Proof. We prove the lemma by induction according to the definition of reachable nameless g-variables (Def. 2.5).

Case (Base case). In this case g is reachable directly from some pointer g-variable x located in the global or local memory. We find the displacement i of g-variable x in its memory frame M^d and show that the value of this pointer belongs to the program roots: $M^d!i \in roots$. Then we show that $M^d!i$ points to the heap and that it points exactly to the heap node with index $gvar_{hid}(xs, alloc, g)$. Then by Def. 3.24 we conclude the goal.

Case (Induction step). In this case g is reachable via some heap g-variable x. Here two situations are possible. In the first case, g is a sub-variable of x. Then $gvar_{hid}(xs, alloc, g) = gvar_{hid}(xs, alloc, x)$ and by the induction hypothesis we conclude the proof. In the other case x is a reachable nameless g-variable and by the induction hypothesis we get

$$gvar_{hid}(xs, alloc, x) \in RN(TT, PP, HM^d, roots, xs).$$

From the inductive case in Def. 3.24 we need to show that the predicate

$$edge_{node}(HM^d, xs, gvar_{hid}(xs, alloc, x), gvar_{hid}(xs, alloc, g))$$

holds. For this purpose we find displacement i of the g-variable x in the assembly memory corresponding to the node $root_g(x)$ obtained by the predicate $hmg_m(d, root_g(x), alloc)$. Then we show that $hmg_m(d, root_g(x), alloc)!i$ points to a heap node with index $gvar_{hid}(xs, alloc, g)$ and belongs to extracted set of node pointers $ncnt_{ptrs}(HM^d, gvar_{hid}(xs, alloc, x), xs)$. This concludes the proof according to Def. 3.23.

□

Now we formulate the other direction of Lemma 4.26. First we prove an auxiliary lemma, which constructs a pointer g-variable from some value in the *roots* set. Here we provide such lemma only for the case of a pointer located in the global memory (other cases are also considered in Isabelle/HOL proofs).

Lemma 4.27 (**Pointer g-variable existence**)**.** If some value a belongs to the set of global roots, then there exists a corresponding pointer g-variable g with displacement k in the global memory GM^d of the assembler configuration d.

$$\begin{aligned}
&c \in conf_{\surd}(te, ft) \wedge gci_{\surd}(te, ft, gst(c.mem), TT, PP) \\
&\wedge\ consis_{alloc}(te, ft, c, alloc) \wedge gc_assume_{gm}(c.mem) \\
&\wedge\ gc_alive_{alloc}(sc(c.mem), alloc, alive) \wedge roots_{gm}(TT, PP, GM^d)!i = a \\
&\Longrightarrow \exists g\ k.\ ptr_{gvar}(sc(c.mem), g) \wedge g \in gvars_{\surd}(sc(c.mem)) \wedge \\
&\qquad\qquad mem_g(g) = gm \wedge GM^d!k = a \wedge \\
&\qquad\qquad fst(alloc(g)) = abase_{gm}(te, ft, gst(c.mem)) + 4 \cdot k
\end{aligned}$$

Proof. To prove this lemma we first show, that for every displacement inside the global memory frame there exists a valid elementary g-variable (this follows from the g-variable construction and from the way, how we define the base address of global g-variables). Then, by the construction of $roots_{gm}(TT, PP, GM^d)$ (Def. 3.19) we know that for every root there exists some offset k in the $ptrs_{off}(TT, PP, 0)$

array. From $gci_{\surd}$ we can easily derive that k is less than the allocated size of the global memory frame. Thus, we can construct an elementary global g-variable g with the displacement k. From Def. 3.19 and Def. 3.18 we conclude that $GM^d!k = a$. We use Lemma 4.18 to show that g is a pointer g-variable and conclude the proof. □

Lemma 4.28 (**G-variable in RN is reachable**)**.** Let d and c be the VAMP assembly and the C0 configurations respectively, xs be the current abstract heap and *stack* be the abstract program stack. Let *roots* denote the set of pointers in the global and local memories of the program as in Lemma 4.26. If the index $gvar_{hid}(xs, alloc, g)$ of some alive valid nameless g-variable g is present in the set of reachable nodes, then g is a reachable g-variable.

$$
\begin{aligned}
& c \in conf_{\surd}(te, ft) \wedge gci_{\surd}(te, ft, gst(c.mem), TT, PP) \\
& \wedge consis^{weak}_{alloc}(te, ft, c, alloc) \wedge consis_p(c, alloc, d) \\
& \wedge gc\_consis^{weak}_{aheap}(te, c, alloc, f) \wedge gc\_consis_{fh}(te, ft, c, alloc, alive, d) \\
& \wedge gc\_consis_a(te, ft, c, alloc, alive, f, xs, stack, d) \\
& \wedge gc\_assume_{gm}(c.mem) \wedge gc\_consis_{alive}(c, alloc, alive) \\
& \wedge gvar_{hid}(xs, alloc, g) \in RN(TT, PP, HM^d, roots, xs) \\
& \wedge g \in gvars_{\surd}(sc(c.mem)) \wedge alive(g) \\
& \Longrightarrow g \in reachable^{nameless}_g(c.mem)
\end{aligned}
$$

Proof. We prove the lemma by induction according to Def. 3.24.

Case (Induction base). In this case there exists some address $a \in roots$, that points to the heap node $gvar_{hid}(xs, alloc, g)$. Depending on the position in the array *roots*, a either belongs to a set of global or local roots. Let us consider the case $a \in roots_{gm}(TT, PP, GM^d)$ (the second case is proven analogously). By Lemma 4.27 we obtain a respective pointer g-variable x with displacement k and value a. By definition of a pointer g-variable we find some g', such that $x = Ptr(g')$. Since a is a heap address, g' is a nameless g-variable. Moreover, both g and g' are top-level (root) g-variables, since pointers to the heap may point only to root g-variables (gc_assume_{gm}). It remains to show that $g = g'$, i.e. x points exactly to the g-variable g. To prove this we first show that g' is a reachable and alive g-variable. Since x points to g' and the value of x is equal to a, a is equal to the allocation address of g'. At the same time a points to the heap node $gvar_{hid}(xs, alloc, g)$. We get

$$
\begin{aligned}
a &= fst(alloc(g')) \\
&= abase_{heap} + (xs!(gvar_{hid}(xs, alloc, g')) - AUX_{size}) \cdot 4 \\
&= abase_{heap} + (xs!(gvar_{hid}(xs, alloc, g)) - AUX_{size}) \cdot 4
\end{aligned}
$$

From *HalfHeap* (Def. 4.48 and Def.
refdef:HalfHeap) we know that xs is distinct, thus $gvar_{hid}(xs, alloc, g') = gvar_{hid}(xs, alloc, g)$. But according to gc_consis_{aheap} (Def. 4.48) for every element in the abstract heap xs there exists exactly one top-level heap g-variable. Thus, $g = g'$ and we conclude the proof.

Case (Induction step).

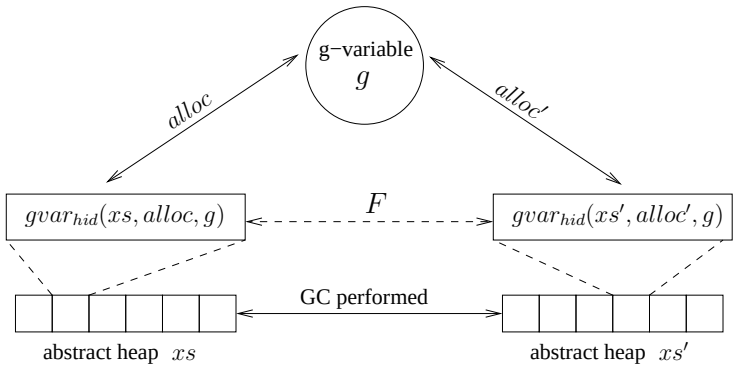

Figure 4.7: G-variable, node and an abstract heap: before and after garbage collection.

In this case there exists another node j which is reachable and which contains a pointer to the node $gvar_{hid}(xs, alloc, g)$. For a node j we find a corresponding top-level heap g-variable x, s.t. $j = gvar_{hid}(xs, alloc, x)$. By the induction hypothesis x is reachable. Moreover, we know that the array

$$ncnt_{ptrs}(HM^d, gvar_{hid}(xs, alloc, x), xs)$$

contains a pointer to the node $gvar_{hid}(xs, alloc, g)$. We need to show that there exists a corresponding nameless pointer g-variable x' with $root_g(x') = x$ and $x' = Ptr(g)$. We proceed with the proof in the same way as in the induction base case and conclude the goal. □

4.6.2 Isomorphism and pointer consistency

The following lemma plays a crucial role in the proof of the simulation relation after the garbage collection is performed. Basically, it establishes the connection between a "node" concept, used in the GC specification on the Hoare logic level and the "g-variable" concept used on the C0 level. When we prove some property for a heap g-variable g after garbage collection is performed, we have to find a respective node in the abstract heap xs, then find the new position of this node in the abstract heap xs', derive the property for the node from the Hoare logic specification and then transform it to the property of a C0 variable (Fig. 4.7). Lemma 4.29 establishes a connection between positions of the node corresponding to the g-variable g in abstract heaps xs and xs'. Using this connection we can derive all the properties we need after the garbage collection is performed (from the Hoare logic specification (Sec. 3.2.6)).

The proof of the lemma is done by induction on the number of steps required to get a heap g-variable g starting from some named pointer g-variable. Note, that according to definition of $reachable_g^{nameless}$ if some g-variables is reachable in i steps, then it is also reachable in $i + k, k \in \mathbb{N}$ steps, since the relation $g \in sub_g(g)$ always holds. To remove this ambiguity we introduce a stronger predicate

$$reachable_g^{steps}(mem, i, x, g),$$

which returns true iff a heap g-variable g is reachable exactly in i steps without self-looping (i.e. $g \in sub_g(g)$ cannot produce an additional edge in the reachability path), and the last variable in the path before g is x. This means that x is a pointer g-variable (local or global) and points to g. Note, that g might also be reachable in $i+k$ or $i-k$ steps, but this will require two different paths containing (some) different variables in it.

Lemma 4.29 (**Isomorphism and pointer consistency**)**.** Let d and d' be VAMP assembly machine states before and after garbage collection, respectively. Further, let $alloc'$ be an allocation function after GC. Let the simulation relation hold in the initial state and partially hold in the state after GC. Then for every reachable g-variable g the isomorphism permutation relation F being applied to the initial index of g in the abstract heap xs returns the new index of g in xs'. Moreover, the g-variable x which points to g is pointer consistent.

$$
\begin{aligned}
& c \in conf_{\surd}(te, ft) \wedge consis^{weak}_{alloc}(te, ft, c, alloc) \\
& \wedge consis_{gc}(te, ft, c, alloc, alive, f, d, xs, stack) \wedge consis_p(c, alloc, d) \\
& \wedge consis_{alloc}(te, ft, c, alloc') \wedge gc\_consis^{weak}_{aheap}(te, c, alloc', f') \\
& \wedge gc\_consis_a(te, ft, c, alloc', alive', f', xs', stack, d') \\
& \wedge gc\_consis_{fh}(te, ft, c, alloc', alive', d') \wedge gc\_alive_{reach}(c, alive') \\
& \wedge gc\_alive_{alloc}(sc(c.mem), alloc', alive') \\
& \wedge gc\_performed(inf_\sigma, m_\sigma, m'_\sigma, nhi^c, nhi', f, f', stack, xs, xs', F) \\
& \wedge reachable^{steps}_g(c.mem, i, x, g) \\
& \Longrightarrow consis^g_p(c.mem, alloc', d', x) \wedge \\
& \quad F(gvar_{hid}(xs, alloc, g)) = gvar_{hid}(xs', alloc', g)
\end{aligned}
$$

Proof. Let $roots$ denote the set of pointers in the global and local memories of the program as in Lemma 4.26 and $roots'$ denote the same set after the garbage collection. We prove the lemma by induction on the number of steps in a reachability path.

Case ($j = 0$). In this case g is reachable directly from pointer g-variable x located in global or local memory. We find the displacement i of g-variable x in its memory frame M^d and show that the value of this pointer belongs to the program roots: $M^d!i \in roots$. We use the predicate $isomorph_{roots}$ from $Isomorph$ (Def. 3.29) to show that x points after garbage collection to the same heap node as it pointed before. After some computations this concludes the second conjunct of the goal. From the definition of $alloc'$ and using properties from $gc\_performed$ we derive

$$
\begin{aligned}
fst(alloc'(g)) &= abase_{heap} + (xs'!F(gvar_{hid}(xs, alloc, g)) + AUX_{size}) \cdot 4 \\
&= abase_{heap} + (xs'!gvar_{hid}(xs', alloc', g) + AUX_{size}) \cdot 4 \\
&= M^{d'}!i = i2n(cell2int(d'.mm(\frac{fst(alloc(g))}{4})))
\end{aligned}
$$

From this follows that x contains the correct allocation address of g and is pointer consistent. The third step of the derivation is not trivial and requires some additional steps, involving finding the element in $roots$ and $roots'$ arrays, which corresponds to pointer x, and different properties from $Isomorph$ and

Roots predicates. We do not give the detailed derivation here, since it would require a number of additional definitions and several auxiliary lemmas.

Case $(j \rightarrow j+1)$. In this case g is reachable via heap g-variable x. Here two situations are possible. In the first case g is a sub-variable of x. Then $gvar_{hid}(xs, alloc, g) = gvar_{hid}(xs, alloc, x)$ and by the induction hypothesis we conclude the proof. In the other case x is a reachable nameless g-variable and by the induction hypothesis we get

$$F(gvar_{hid}(xs, alloc, x)) = gvar_{hid}(xs', alloc', x)$$

We find displacement i of the pointer g-variable x in the assembly memory corresponding to the node $root_g(x)$ obtained by the predicate $hmg_m(d, root_g(x), alloc)$. Analogously to the base case, we find displacement i of the g-variable x in the heap memory corresponding to the node of x and use parts of the $gc\_performed$ predicate to derive

$$hmg_m(d', root_g(x), alloc')!i = abase_{heap} + (xs'!gvar_{hid}(xs', alloc', g) + AUX_{size}) \cdot 4,$$

and, thus, to conclude pointer consistency of x. We use pointer consistency of x and $Isomorph$ predicate to show the second conjunct of the goal.

□

4.6.3 Value consistency

In order to show value consistency for the named program variables we simply show that the assembly values of these variables have not changed after garbage collection. The following lemma establishes value equivalence for global non-pointer g-variables.

Lemma 4.30 (**Values unchanged: global variables**)**.** Assume the same parameters as in Lemma 4.29. After garbage collection assembly values for elementary global non-pointer g-variables remain unchanged.

$$\begin{aligned}
&c \in conf_{\surd}(te, ft) \wedge gci_{\surd}(te, ft, gst(c.mem), TT, PP) \\
&\wedge\ consis(te, ft, c, alloc, d, alive, f, xs, stack) \\
&\wedge\ gc_performed(inf_\sigma, m_\sigma, m'_\sigma, nhi^c, nhi', f, f', stack, xs, xs', F) \\
&\wedge\ g \in gvars_{\surd}(sc(c.mem)) \wedge mem_g(g) = gm \\
&\wedge\ elem_g(sc(c.mem), g) \wedge \neg ptr_{gvar}(sc(c.mem), g) \\
&\Longrightarrow d'.mm(fst(alloc'(g))) = d.mm(fst(alloc(g)))
\end{aligned}$$

Proof. The allocation function for named variables after GC remains the same, i.e. $alloc'(g) = alloc(g)$. From $gc\_performed$ we derive the equality of non-pointer global memory regions.

$$gmcnt_{noptrs}(GM^d) = gmcnt_{noptrs}(GM^{d'}) \tag{4.1}$$

We find the displacement i of g in the global memory GM.

$$GM^d!i = d.mm(fst(alloc(g))) \quad \text{and} \quad GM^{d'}!i = d.mm(fst(alloc(g'))) \tag{4.2}$$

From definition of $gmcnt_{noptrs}$ and (4.1) we get

$$i < TT_0.asz \wedge i \notin ptrs_{off}(TT, PP, 0) \longrightarrow GM^d!i = GM^{d'}!i. \qquad (4.3)$$

The first conjunct in the precondition of (4.3) follows from $gci_{\surd}$. To conclude the second conjunct we use Lemma 4.17 and $gci_{\surd}$. □

To show value consistency for nameless variables we formulate an analogues lemma. The proof of the lemma, however, is more complicated and requires the node isomorphism property from Lemma 4.29.

Lemma 4.31 (**Values unchanged: nameless variables**)**.** Let d and d' be VAMP assembly machine states before and after garbage collection respectively, and $alloc'$ be an allocation function after GC. Let the simulation relation hold in the initial state and partially hold in the state after GC. After garbage collection assembly values for elementary heap non-pointer g-variable remain unchanged (with respect to a new allocation function).

$$\begin{aligned}
&c \in conf_{\surd}(te, ft) \wedge gci_{\surd}(te, ft, gst(c.mem), TT, PP)\\
&\wedge consis(te, ft, c, alloc, d, alive, f, xs, stack)\\
&\wedge gc_performed(inf_\sigma, m_\sigma, m'_\sigma, nhi^c, nhi', f, f', stack, xs, xs', F)\\
&\wedge consis_{alloc}(te, ft, c, alloc') \wedge gc_consis^{weak}_{aheap}(te, c, alloc', f')\\
&\wedge gc_consis_a(te, ft, c, alloc', alive', f', xs', stack, d')\\
&\wedge gc_consis_h(te, ft, c, alloc', alive', d') \wedge gc_consis_{alive}(c, alloc', alive')\\
&\wedge g \in gvars^{nameless}_{\surd}(sc(c.mem)) \wedge elem_g(sc(c.mem), g) \wedge \neg ptr_{gvar}(sc(c.mem), g)\\
&\Longrightarrow d'.mm(fst(alloc'(g))) = d.mm(fst(alloc(g)))
\end{aligned}$$

Proof. The pointer g-variable g is located in the node $gvar_{hid}(xs, alloc, g)$. Using Lemma 4.29 and $gc\_performed$ we derive the equality of non-pointer parts of the heap node before and after garbage collection.

$$\begin{aligned}
&ncnt_{noptrs}(HM^d, gvar_{hid}(xs, alloc, g), xs)\\
&= ncnt_{noptrs}(HM^{d'}, F(gvar_{hid}(xs, alloc, g)), xs') && \text{From } gc_performed\\
&= ncnt_{noptrs}(HM^{d'}, gvar_{hid}(xs', alloc', g), xs') && \text{Lemma 4.29}
\end{aligned}$$

Let k be the displacement of the pointer g in the node $gvar_{hid}(xs, alloc, g)$ and n_{id} be the index of the node in the TT table, i.e. $n_{id} = read_{nid}(d, g, alloc)$. We calculate the displacement i of g in the $ncnt_{noptrs}$ array of the node in the following way.

$$i = |map(\lambda x.\ x \notin ptrs_{off}(TT, PP, n_{id}), (0 : k-1))|$$

Using $gc\_performed$ we can derive that n_{id} doesn't change its value after garbage collection and the position of the pointer in the node also remains unchanged. We proceed with the following derivation.

$$
\begin{aligned}
& d.mm(fst(alloc(g))) \\
& = hmg_m(d, root_g(g), alloc)!k && \text{Def. } hmg_m \\
& = ncnt_{noptrs}(HM^d, gvar_{hid}(xs, alloc, g), xs)!i && \text{Def. } hmg_m,\ ncnt_{noptrs} \\
& = ncnt_{noptrs}(HM^{d'}, gvar_{hid}(xs', alloc', g), xs')!i \\
& = hmg_m(d', root_g(g), alloc')!k && \text{Def. } hmg_m,\ ncnt_{noptrs} \\
& = d'.mm(fst(alloc'(g))) && \text{Def. } hmg_m
\end{aligned}
$$

This concludes the goal. □

4.6.4 Allocation consistency

The proof of the named allocation consistency is rather trivial and we do not show it here. The proof of heap allocation consistency is done in Lemma 4.32.

Lemma 4.32 (**Heap allocation consistency**)**.** Let d and d' be VAMP assembly machine states before and after garbage collection, respectively. Further, let $alloc'$ be an allocation function after GC. Let the simulation relation hold in the initial state and partially hold in the state after GC. The heap allocation consistency holds for configuration c and new allocation function $alloc'$.

$$
\begin{aligned}
& c \in conf_{\surd}(te, ft) \wedge gci_{\surd}(te, ft, gst(c.mem), TT, PP) \\
& \wedge\ consis(te, ft, c, alloc, d, alive, f, xs, stack) \\
& \wedge\ gc\_performed(inf_\sigma, m_\sigma, m'_\sigma, nhi^c, nhi', f, f', stack, xs, xs', F) \\
& \wedge\ consis_{alloc}(te, ft, c, alloc') \wedge gc\_consis^{weak}_{aheap}(te, c, alloc', f') \\
& \wedge\ gc\_consis_a(te, ft, c, alloc', alive', f', xs', stack, d') \\
& \wedge\ gc\_consis_h(te, ft, c, alloc', alive', d') \wedge gc\_consis_{alive}(c, alloc', alive') \\
& \Longrightarrow consis^{heap}_{alloc}(te, ft, c, alloc')
\end{aligned}
$$

Proof. We proceed in the following proof steps.

- From the definition of the function $alloc'$ we show $consis^{wheap}_{alloc}(te, ft, c, alloc')$. This step is rather simple and we do not show it here. It remains to show the last statement from heap allocation consistency, i.e. that heap g-variables do not overlap.
- From the definition of the function $alloc'$ and from $gc\_performed$ we show that $gc\_consis^{weak}_{aheap}(te, c, alloc', f')$ holds. Again, the step is simple and we do not give details about it here.
- For reachable nameless g-variables x and g we have to show the following property.
$$root_g(x) \neq root_g(h) \longrightarrow alloc'(g) \asymp alloc'(h)$$
We unfold the definition of $alloc'$ and get the following goal. By sc we understand the symbol configuration of the memory $c.mem$.
$$
\begin{aligned}
& (displ_{gc}(xs, xs', alloc, F)(g) \cdot 4 + displ_g(sc, g), asize_t(ty_g(sc, g))) \\
& \asymp (displ_{gc}(xs, xs', alloc, F)(x) \cdot 4 + displ_g(sc, x), asize_t(ty_g(sc, x))
\end{aligned}
$$

Since the allocation region of a g-variable is completely contained in the allocated region of its root g-variable, we can claim a stronger goal. Let $g' = root_g(g)$ and $x' = root_g(x)$. Then our goal is

$$\begin{aligned}&(displ_{gc}(xs, xs', alloc, F)(g') \cdot 4, asize_t(ty_g(sc, g')))\\ \asymp\ &(displ_{gc}(xs, xs', alloc, F)(x') \cdot 4, asize_t(ty_g(sc, x')).\end{aligned}$$

From the correctness of GC interface we know that the size of the type is stored in the respective element of the TT table. The index of this element is stored in the first word of the node header and is obtained by $read_{nid}(d, g', alloc)$ function. From the properties of the abstract stacks xs and xs' (Def. 4.47), definition of $alloc'$, and the fact that node headers are not changed during garbage collection we derive the following.

$$\begin{aligned}asize_t(ty_g(sc, g')) &= TT_{read_{nid}(d,g',alloc)}.asz\\ &= TT_{HM^d!(xs!gvar_{hid}(xs,alloc,g'))}.asz\\ &= TT_{HM^{d'}!(xs'!gvar_{hid}(xs',alloc',g'))}.asz\end{aligned}$$

Unfolding $displ_{gc}$ and applying Lemma 4.29 we get the following goal.

$$\begin{aligned}&(xs'!gvar_{hid}(xs', alloc', g'), TT_{HM^{d'}!(xs'!gvar_{hid}(xs',alloc',g'))}.asz)\\ \asymp\ &(xs'!gvar_{hid}(xs', alloc', x'), TT_{HM^{d'}!(xs'!gvar_{hid}(xs',alloc',x'))}.asz)\end{aligned}$$

Using definitions of *HalfHeap* (Def. 3.16) and $displ_{node}$ (Def. 3.15) we show (by induction) that for all HM, f, nhi, xs the following property holds[3].

$$\begin{aligned}&HalfHeap(TT, HM, f, nhi, xs) \wedge i \neq j\\ \Longrightarrow\ &(xs!i, TT_{HM!(xs!i)}.asz) \asymp (xs!j, TT_{HM!(xs!j)}.asz)\end{aligned}$$

We conclude the goal using $HalfHeap(TT, HM^{d'}, f', nhi', xs')$ and the above property .

□

[3]Lemma HalfHeap_range_not_overlap_aux

Chapter 5

Summary and Future Work

In this thesis we have presented the formal verification of a simple, non-optimizing compiler from [12] with the integrated garbage collector; this includes the formal specification of the garbage collector routines on the Hoare logic level done by E.Petrova. The work was done in the context of pervasive system verification in the frame of the Verisoft project [21]. The results of verification have been formalized in the interactive theorem prover Isabelle/HOL (Table 5.1).

We used the garbage collector specification for a simple copying garbage collector and formally defined the impact of the GC routine on the memory of the assembly machine. We have also given the formal specification for the interface of the garbage collector and modeled it on the small step-semantics level.

As the next step we have updated the specification for the compiler and have given an extended simulation relation, which includes program properties needed for correct garbage collection. We have discovered a missing requirement for the C0 semantics in presence of the garbage collector, namely that all g-variables are automatically initialized. We proved the correctness of the compiling specification by the induction on the number of the steps in a C0 computation. In the induction step we performed a case distinction by the type of the next C0 statement to be executed. In the thesis we considered the most interesting and complicated case - memory allocation statement. We have introduced several intermediate assembly and C0 configurations for the case and have shown, that simulation relation partially holds for intermediate configurations and fully holds for the final configurations. The proof of all other C0 statements, as well as the proof of the base case, does not differ much from the proof of the original compiler without garbage collector. However, these proofs remain as a future work.

Another interesting direction of future work is the formal transfer of the correctness criteria from the Hoare logic to the assembly level (Sec. 4.5.6).

To the best of our knowledge, our work is the first compiler correctness result for a compiler with an integrated garbage collector done in the context of pervasive system verification.

Table 5.1: Verification summary in Isabelle/HOL

Theory name	# Lemmas	# Commands
GC interface + properties	68	5292
Reachability + isomorphism	20	5107
Auxiliary properties	240	7994
Consistency	245	21336
Total	~ 580	~ 40000

Appendix A

Mapping to Lemmas in Isabelle/HOL

This section gives a mapping from lemmas and theorems in this thesis to the corresponding formal proofs in the Isabelle/HOL theories.

Name	Page	Name in Isabelle
Theorem 3.1	40	correct_gc
Lemma 4.1	45	asize_heap_alive_dvd_4
Lemma 4.2	51	length_ptrs_displ_t_Arr
Lemma 4.3	52	ptrs_displ_t_less_asize_type
Lemma 4.4	52	ptrs_displ_stbl_less_asize_symboltable
Lemma 4.5	52	ptrs_displ_t_ordered
Lemma 4.6	52	ptrs_displ_stbl_ordered
Lemma 4.7	53	nth_ptrs_displ_t_Arr
Lemma 4.8	53	displ_t_Arr_nth_index_exists
Lemma 4.9	53	nth_ptrs_displ_t_Struct
Lemma 4.10	54	displ_t_Struct_nth_index_exists
Lemma 4.11	54	ptrs_displ_t_plus_displ_var_in_ptrs_displ_stbl
Lemma 4.12	55	gvar_displ_in_ptrs_displ$_t$
Lemma 4.13	55	gvar_displ_in_ptrs_displ_t_aux
Lemma 4.14	56	displ_of_pointer_in_set_ptrs_displ_t_aux
Lemma 4.15	57	ptrs_displ_t_zero_is_Ptr
Lemma 4.16	57	elementary_gvar_in_ptrs_displ_t_is_Ptr
Lemma 4.17	57	gvar_displ_in_ptrs_displ_stbl
Lemma 4.18	58	gvar_displ_in_ptrs_displ_stbl_is_Ptr
Lemma 4.19	60	gci_correct_interface
Lemma 4.20	73	asize_heap_reachable_eq_alive
Lemma 4.21	81	consistent_impl_PAlloc_pre_gc
Lemma 4.22	82	gc_palloc_code_correct
Lemma 4.23	82	gc_palloc_code_correct
Lemma 4.24	82	gc_palloc_code_correct
Lemma 4.25	84	gc_palloc_correct_aux_not_enough_heap, gc_palloc_correct_aux_enough_heap_gc_needed, gc_palloc_correct_aux_enough_heap_no_gc_needed

Name	Page	Name in Isabelle
Theorem 4.1	88	gc_palloc_correct_not_enough_heap , gc_palloc_correct_enough_heap_gc_needed, gc_palloc_correct_enough_heap_no_gc_needed
Lemma 4.26	89	reachable_gvar_in_RN
Lemma 4.27	90	gm_cnt_ptrs_pointer_gvar_exists
Lemma 4.28	91	gvar_in_RN_reachable_nameless_gvars
Lemma 4.29	93	isomorph_reachable_heap_nodes_aux
Lemma 4.30	94	gc_mm_d_not_ptrs_gm_eq
Lemma 4.31	95	gc_heap_not_ptrs_eq
Lemma 4.32	96	gc_update_alloc_heap_heap_consistent_invariant

Glossary

AUX_{size}	size of the node header, 26
FHS	size of the local frame header in words, 26
GCM^d	shortcut for $gm^{gc}_m(d, sc)$ function, 48
GM^d	shortcut for $gm^{p}_m(d, sc)$ function, 48
HH	size of the heap memory, 26
HHS	size of the half heap memory, 26
HM^d	shortcut for $hm_m(d)$ function, 48
$Isomorph$	memories isomorphism (predicate), 35
LM^d	shortcut for $lm_m(d, sc)$ function, 48
MW	size of the word in bytes, 26
PP	pointer table (GC interface), 50
RN	set of reachable nodes, 32
Roots	states some properties for global and local roots, 33
Stack	abstraction relation for the abstract stack of the program, 29
TT	type table (GC interface), 50
δ_{C0}	C0 transition function, 17
δ_{gc}	GC C0 transition function, 63
GMS	size of the global memory in words, 26
HalfHeap	abstraction relation for the abstract heap, 30
LMS	size of the local memory stack in words, 26
off_{psp}	PSP field displacement in the frame header, 25
off_{TT}	TT link displacement in the frame header, 25
$ptrs_{off}$	extracts displacements of pointers for some element of TT table, 31
$woff_{ge}$	no pointers in frame header (predicate), 27
$woff_{ord}$	offsets are ordered (predicate), 27
$abase_g$	allocated base address of a g-variable, 46
$abase_{gm}$	allocation base for global memory, 41
$abase_{lm}$	allocated address of the local frame, 46
$algn$	calculates alignment of a type in the memory, 43
$aliveT$	type for alive function, 44
$alive_{gc}$	computes alive function after GC, 64
$alive_{idx}$	set of alive variables, 45
$alive_{xt}$	computes alive function after new heap variable allocation, 64

Bibliography

[1] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. The Verisoft approach to systems verification. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments Second International Conference, VSTTE 2008, Toronto, Canada, October 6–9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*, pages 209–224, Toronto, Canada, October 2008. Springer.

[2] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In D. Geist and E. Tronci, editors, *Proc. of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2003.

[3] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang Paul. Putting it all together: Formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 8(4–5):411–430, August 2006.

[4] Sebastian Bogan. *Formal Specification of a Simple Operating System*. PhD thesis, Saarland University, Computer Science Department, August 2008.

[5] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland, 1972.

[6] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[7] Iakov Dalinger. *Formal Verification of a Processor with Memory Management Units*. PhD thesis, Saarland University, Computer Science Department, July 2006.

[8] Iakov Dalinger, Mark Hillebrand, and Wolfgang Paul. On the verification of memory management mechanisms. In D. Borrione and W. Paul, editors, *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *Lecture Notes in Computer Science*, pages 301–316. Springer, 2005.

[9] Matthias Daum, Jan Drrenbcher, and Sebastian Bogan. Model stack for the pervasive verification of a microkernel-based operating system. In Bernhard Beckert and Gerwin Klein, editors, *5th International Verification Workshop (VERIFY'08)*, volume 372 of *CEUR Workshop Proceedings*, pages 56–70. CEUR-WS.org, 2008.

[10] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[11] Daniel Kroening. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Computer Science Department, 2001.

[12] D. Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Computer Science Department, Saarbrücken, Germany, 2008.

[13] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler. In *3rd International Conference on Software Engineering and Formal Methods(SEFM 2005)*, Koblenz, Germany, September 2005.

[14] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[15] Michael Norrish. *C Formalised in HOL*. PhD thesis, University of Cambridge, Computer Laboratory, December 1998.

[16] E. Petrova. *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, Computer Science Department, Saarbrücken, Germany, 2007.

[17] N. Schirmer. *Verifcation of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[18] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technical University of Munich, 2006.

[19] L.C. Paulson T. Nipkow and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[20] Sergey Tverdyshev and Andrey Shadrin. Formal verification of gate-level computer systems. In Kristin Yvonne Rozier, editor, *LFM 2008: Sixth NASA Langley Formal Methods Workshop*, NASA Scientific and Technical Information (STI), pages 56–58. NASA, 2008.

[21] The Verisoft project. `http://www.verisoft.de/`, 2003.