Heap Slicing Using Type Systems

Mohamed A. El-Zawawy

College of Computer and Information Sciences, Al-Imam M. I.-S. I. University
Riyadh, Kingdom of Saudi Arabia
and
Department of Mathematics, Faculty of Science, Cairo University
Giza 12613, Egypt
maelzawawy@cu.edu.eg

Abstract. Using type systems, this paper treats heap slicing which is a technique transforming a program into a new one that produces the same result while working on a heap sliced into independent regions. Heap slicing is a common approach to handle the problem of modifying the heap layout without changing the program semantics. Heap slicing has applications in the areas of performance optimization and security.

Towards solving the problem of heap slicing, this paper introduces three type systems. The first type system does a pointer analysis and annotates program points with pointer information. This type system is an augmentation of a previously developed type system by the author. The second type system does a region analysis and refines the result of the first type system by augmenting the pointer information with region information. The region information approximately specifies at each program point for each memory cell the region where the cell exists. The third type system uses the information gathered by the region type system to do the principal transformation of heap slicing.

The paper also presents two operational semantics; one for single-region heap scenario and the other for multi-regions heap scenario. These semantics are used to prove the soundness of the type systems.

Keywords: heap slicing, type systems, semantics of programming languages, operational semantics, region analysis, pointer analysis.

1 Introduction

Heap slicing [28,31] is a technique that transforms a program into a new one that produces the same result while working on a heap sliced into independent regions. This transforation enables an optimizing compiler to figure out memory cells that must lie in different slices of the heap. The input to this technique is a program in which integer argument-expressions in statements allocating memory cells are annotated with slice (region) names. Every slice only contains data that was annotated with the slice name. Arithmetic and Boolean operations are allowed only between arguments in the same slice. Usually, it is assumed that no cell in a slice is allowed to point to a cell in a different slice.

$$\begin{array}{lcl}
x := cons(1:R_1, 2:R_2); & & x := cons'(1:R_1, 2:R_2);\\
y := cons(x, 3:R_2, 4:R_3, 5:R_1); & \hookrightarrow & y := cons'(x:\{1\}, 3:R_2, 4:R_3, 5:R_1);\\
z := cons(y, 6:R_3, 7:R_2); & & z := cons'(y:\{1\}, 6:R_3, 7:R_2);\\
w := [x+1]; & & w :=_{\{2\}} [x+1];\\
t := [y+2]; & & t :=_{\{3\}} [y+2];\\
[z+1] := t; & & [z+1] :=_{\{3\}} t;
\end{array}$$

Fig. 1. A motivating example

Very often while maintaining a large software, it becomes apparent that a change to the heap layout (e.g. adding arguments to an allocation statement) is necessary. The amount of code depending on the heap layout can make the process of introducing such a change, even when it is very little, very tricky. Introducing changes in such situations can be time-consuming and it scarifies the software correctness as it may call bugs. The heap slicing techniques are good tools to address the problem of altering the heap layout without changing the program semantics.

Heap slicing has applications in the areas of performance optimization and security [29,3]. The instance interleaving optimization is a static analysis [20] technique that rearranges the memory cells (or fields of different data structures) to improve cache performance via letting frequently-accessed fields (or cells) belong to the same cache line. Heap slicing techniques provide good implementations for instance interleaving optimization [20]. In security, heap slicing can be used to hide function pointers in a heap slice (region) preventing attackers from accessing them.

Motivating Example

Figure 1 shows a motivating example of our work. Consider the program on the l.h.s. of the figure. The integer-expressions of the allocation statements are annotated with their region names. For example the first allocation statement allocates an array of length two: the first of which belongs to region 1 and the second of which belongs to region 2. The goal of our research is to automatically transform such a program into the program on the r.h.s. of the figure. In the new program: (a) the address expressions (expressions evaluates to addresses) of allocation statements are annotated with their region names, and (b) mutation and look-up statements are annotated with reign names where the statements are allowed to be executed.

While the original program is assumed to be executed on a one-slice heap, the new program is executed on a heap that physical sliced into 3 regions. The number of the regions is fixed in the programming language. Figures 2 and 3 show the heaps of the original and new programs, respectively, after executing the allocation statements.

Moreover, we want to associate each of such program transformation with a proof that original and new programs have the same semantics: compute the same result. This proof is required in many application like *proof-carrying code* [19,22].

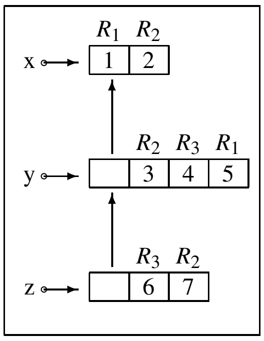

Fig. 2. One-slice heap

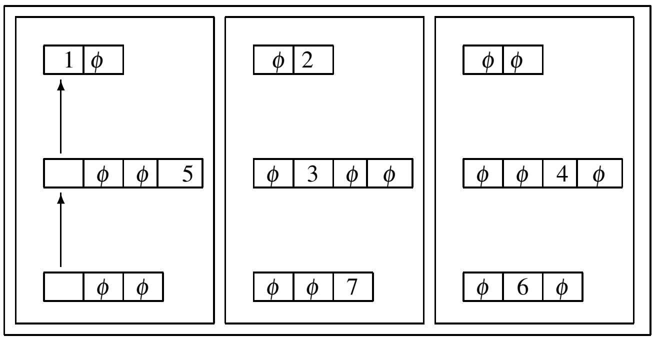

Fig. 3. Sliced heap

Algorithm

The transformation process described above on the motivating example is achieved in this paper using a 3-steps algorithm. Each of the 3 steps is accomplished by a type system. The first step is a pointer analysis to the input program. This analysis results in annotating the program points with points-to information in the form of types. The points-to information at a given point specifies approximately for each store (a variable or a memory cell) the address that has a chance of going into that store. The second step is a region analysis to the program resulting from the first step. This analysis results in augmenting the pointer information with region information in the form of types. The region information at a given point specifies approximately the region for each memory cell. Also the region information at a given point specifies approximately for each variable the source region of the variable's content. The third step does the transformation step using the information gathered in the previous steps.

The justification (proof) that the source and the new programs are semantically equivalent takes the form of a type derivation.

$$
\begin{array}{rcl}
 & & \oplus \in \{+,-,\times\}, x \in \mathit{Var}, \text{ and } \{i,j\}, \mathit{Rs} \subseteq \{1,\ldots,\gamma\} \\
e \in \mathit{Aexprs} & ::= & x \mid n \mid e_1 \oplus e_2 \mid \mathit{Cast}(R_i \hookrightarrow R_j)e \\
d \in \mathit{Allo\text{-}exprs} & ::= & e : R_i \mid e \\
b \in \mathit{Bexprs} & ::= & \mathit{true} \mid \mathit{false} \mid \neg b \mid e_1 = e_2 \mid e_1 \leq e_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\
S \in \mathit{Stmts} & ::= & x := e \mid x := \mathit{cons}(d_1,\ldots,d_n) \mid x := [e] \mid [e_1] := e_2 \mid \mathit{dispose}(e) \mid \\
 & & \mathit{skip} \mid S_1; S_2 \mid \mathit{if}\ b\ \mathit{then}\ S_t\ \mathit{else}\ S_f \mid \mathit{while}\ b\ \mathit{do}\ S_t \\
S' \in \mathit{Stmts}' & ::= & x := e \mid x := \mathit{cons}'(e_1 : \mathit{Rs}_1,\ldots,e_n : \mathit{Rs}_n) \mid x :=_{\mathit{Rs}} [e] \mid [e_1] :=_{\mathit{Rs}} e_2 \\
 & & \mid \mathit{dispose}'(e) \mid \mathit{skip} \mid S'_1; S'_2 \mid \mathit{if}\ b\ \mathit{then}\ S'_t\ \mathit{else}\ S'_f \mid \mathit{while}\ b\ \mathit{do}\ S'_t
\end{array}
$$

Fig. 4. Our language for studying heap slicing

Contributions

Contributions of this paper are the following.

1. A type system for pointer analysis of the language presented in this paper. This type system is an augmented version of that we presented in [13].
2. A novel approach for region analysis (in the form of a type system as well).
3. An original technique for heap slicing.
4. Two new operational semantics; one for single-region heap scenario and the other for multi-regions heap scenario.

Organization

The rest of the paper is organized as follows. A toy programming language together with two operational semantics (one for single-region heap scenario and the other for multi-regions heap scenario) are presented in Section 2. Type systems for flow-sensitive pointer and region analyses are presented in Sections 3 and 4, respectively. The type system carrying program optimization is introduced in Section 5. A brief survey of related work and future work are presented in Section 6.

2 Programming Language and Two Operational Semantics

This section presents the programming language used to study heap slicing. The section also presents an operational semantics [23] for a one-slice heap executions and another operational semantics for γ-slices heap executions. The number of regions or slices in memory is fixed in our language and is denoted by γ.

We have two memory models; one for the single-slice heap scenario and the other for the γ-slices heap scenario. In our single-heap model, we assume that for any $m \in \mathbb{N}^+$ the memory has an infinite number of arrays of length m with addresses $\{a^1_{m,1}, a^1_{m,2}, \ldots, a^1_{m,m}, a^2_{m,1}, a^2_{m,2}, \ldots, a^2_{m,m}, \ldots\}$. Therefore the set of address, *Addrs*, has the form presented in Figure 5. In order to facilitate evaluating inequalities we assume

$$\begin{aligned}
\mathit{Atoms} &\subseteq \mathit{Integers}.\\
\mathit{Addrs} &= \{a^i_{j,k} \mid i,j,k \in \mathbb{N}^+, k \leq j\}\\
&= \{a^1_{1,1}, a^1_{2,1}, a^1_{2,2}, a^1_{3,1}, a^1_{3,2}, a^1_{3,3}, \ldots, a^2_{1,1}, a^2_{2,1}, a^2_{2,2}, a^2_{3,1}, a^2_{3,2}, a^2_{3,3}, \ldots\}.\\
\mathscr{R} &= \{1, \ldots, \gamma\}.\\
\mathit{Values} &= \mathbb{Z} \cup \mathit{Addrs}.\\
\mathit{Values}^+ &= \mathit{Values} \cup \{\phi\}.
\end{aligned}$$

Fig. 5. Entities of our memory model

that the set *Values* is equipped with an order. We assume that our $\gamma-$slices memory model consists of γ separated regions each of which has the single-slice model. The value ϕ in the set *Values*$^+$ goes into cells that are inactive in a region. Arithmetic and Boolean operations are only allowed between arguments of the same region.

The language (Figure 4) that we study is based on the programming language usually used to introduce separation logic [24]. There are two additions to the separation logic language. The first addition is that the arithmetic expression is extended with a cast statement permitting handling a value that we obtained form region i as it is obtained from region j. This is useful in many situations like if the programmer is interested in copying a value from a private slice of a memory to a public slice. The other addition is to annotate arguments (the ones evaluates to integers) of the allocation statement with region names. *Stmt*$'$ presents the syntax of transformed programs. A clue to meaning of *Stmt*$'$-commands is given by the motivating example above and a precise meaning is given below by operational semantics.

The states of our operational semantics are defined as follows.

Definition 1. *1.* $s \in \mathit{Stacks} = \{(s_v, s_r) \mid s_v : \mathit{Var} \rightarrow \mathit{Values} \text{ and } s_r : \mathit{Var} \rightarrow \mathscr{R} \cup \{\bot\}\}$.

2. $h \in \mathit{Heaps} = \{(h_v, h_r) \mid h_v : A \rightarrow \mathit{Values}, h_v : A \rightarrow \mathscr{R}, \text{ and } A \subseteq_{fin} \mathit{Addrs}\}$.
3. *A sliced heap* $\tilde{h}$ *is a* γ*-tuple* $(\tilde{h}_1, \ldots, \tilde{h}_\gamma)$ *of finite partial maps from Addrs to Values*$^+$ *such that:*
   (a) *these maps share the same domain, and*
   (b) *for any* $a \in dom(\tilde{h}_1)$ *there is a unique* $i \in [1, \gamma]$ *such that* $\tilde{h}_i(a) \neq \phi$.

Definition 2. *1. A state is an abort or a pair of a stack and a heap* (s, h).

2. *A sliced state is an abort or a pair of a stack and a sliced heap* $(s, \tilde{h})$.

2.1 One-Slice Heap Semantics

This section presents an operational semantics for the input program of our transformation technique. The states of the semantics are defined in Definition 2.1.

The semantics of arithmetic and Boolean expressions are defined as follows:

$$[\![d]\!] \in \mathit{States} \rightharpoonup \mathit{Values} \times (\mathscr{R} \cup \{\bot\})$$

$$[\![n]\!](s,h) = (n, \bot) \quad [\![x]\!](s,h) = (s_v(x), s_r(x)) \quad [\![e_1 \oplus e_2]\!](s,h) = \eta_h([\![e_1]\!](s,h) \oplus [\![e_1]\!](s,h))$$

where,

$$\eta_h(\alpha,\beta)\begin{cases}(\alpha,\beta), & \text{if } \alpha\in\mathbb{Z};\\ (\alpha,h_r(\alpha)), & \text{if } \alpha\in dom(h_r);\\ \mathit{undefined}, & \text{otherwise.}\end{cases}\qquad \llbracket e:R_i\rrbracket(s,h)=\begin{cases}(n,i) & \text{if } \llbracket e\rrbracket(s,h)\in\{(n,i),(n,\bot)\}\\ \mathit{undefined} & \text{otherwise.}\end{cases}$$

$$\llbracket cast(R_i\hookrightarrow R_j)e\rrbracket(s,h)=\begin{cases}(n,j) & \text{if } \llbracket e\rrbracket(s,h)\in\{(n,j),(n,i)\},\\ \mathit{undefined} & \text{otherwise.}\end{cases}$$

The semantics of the operation $\oplus$ is defined as usual if both of its operands are integers and otherwise as follows:

$$v_1\oplus v_2=\begin{cases}(n\oplus m,\bot), & \text{if } v_1=(n,\bot) \text{ and } v_2=(m,\bot);\\ (n\oplus m,i), & \text{if } v_1=(n,i) \text{ and } (v_2=(m,i) \text{ or } v_2=(m,\bot));\\ (a^r_{s,t\oplus n},i), & \text{if } v_1=(a^r_{s,t},i),\ (v_2=(n,i) \text{ or } v_2=(n,\bot)), \text{ and } 1\le t\oplus n\le s;\\ \mathit{undefined}, & \text{otherwise.}\end{cases}$$

Boolean operations are only allowed between values from the same region.

The inference rules of the semantics are defined as follows.

$$\frac{}{skip:(s,h)\to(s,h)}\qquad \frac{\llbracket e\rrbracket(s,h)\text{ is undefined}}{x:=e:(s,h)\to abort}\qquad \frac{\llbracket e\rrbracket(s,h)=(\alpha,\beta)}{x:=e:(s,h)\to([s_v\mid x:\alpha],[s_r\mid x:\beta],h)}$$

$$\frac{u=min\{t\mid\{a^t_{n,1},\dots,a^t_{n,n}\}\cap dom(h)=\emptyset\}\quad \forall 1\le i\le n(\llbracket d_i\rrbracket(s,h)=(\alpha_i,\beta_i))}{\begin{array}{c}x:=cons(d_1,\dots,d_n):(s,h)\to\\ ([s_v\mid x:a^u_{n,1}],[s_r\mid x:\beta_1],[h_v\mid a^u_{n,1}:\alpha_1\mid\dots\mid a^u_{n,n}:\alpha_n],[h_r\mid a^u_{n,1}:\beta_1\mid\dots\mid a^u_{n,n}:\beta_n])\end{array}}$$

$$\frac{\exists 1\le i\le n\ (\llbracket d_i\rrbracket(s,h)\text{ is undefined})}{x:=cons(d_1,\dots,d_n):(s,h)\to abort}\qquad \frac{\begin{array}{l}\llbracket e\rrbracket(s,h)\text{ is undefined, or}\\ \llbracket e\rrbracket(s,h)=(\alpha,\_)\wedge\alpha\notin dom(h)\end{array}}{dispose(e):(s,h)\to abort}$$

$$\frac{}{x:=[e]:(s,h)\to\begin{cases}([s_v\mid x:h_v(\alpha)],[s_r\mid x:\beta)],h), & \text{if } \llbracket e\rrbracket(s,h)=(\alpha,\beta)\text{ and }\alpha\in dom(h);\\ abort, & \text{otherwise.}\end{cases}}$$

$$\frac{}{[e_1]:=e_2:(s,h)\to\begin{cases}(s,[h_v\mid\alpha_1:\alpha_2],[h_r\mid\alpha_1:\beta]), & \text{if}\llbracket e_i\rrbracket(s,h)=(\alpha_i,\beta)\text{ and }\alpha_1\in dom(h);\\ abort, & \text{otherwise.}\end{cases}}$$

$$\frac{\llbracket e\rrbracket(s,h)=(\alpha,\_)\wedge\alpha\in dom(h)}{dispose(e):(s,h)\to(s,h_v\rceil(dom(h)\setminus\{\alpha\}),h_r\rceil(dom(h_r)\setminus\{\alpha\}))}\qquad \frac{\begin{array}{l}S_1:(s,h)\to(s',h')\\ S_2:(s',h')\to st\end{array}}{S_1;S_2:(s,h)\to st}$$

$$\frac{\begin{array}{l}S_1:(s,h)\to abort\\ S_2\in Stmts\end{array}}{S_1;S_2:(s,h)\to abort}\qquad \frac{\llbracket b\rrbracket(s,h)\text{ is undefined}}{if\ b\ then\ S_t\ else\ S_f:(s,h)\to abort}\qquad \frac{\begin{array}{c}\llbracket b\rrbracket(s,h)=false\\ S_f:(s,h)\to st\end{array}}{if\ b\ then\ S_t\ else\ S_f:(s,h)\to st}$$

$$\frac{\begin{array}{c}\llbracket b\rrbracket(s,h)=true\\ S_t:(s,h)\to st\end{array}}{if\ b\ then\ S_t\ else\ S_f:(s,h)\to st}\qquad \frac{\llbracket b\rrbracket(s,h)\text{ is undefined}}{while\ b\ do\ S_t:(s,h)\to abort}\qquad \frac{\begin{array}{c}\llbracket b\rrbracket(s,h)=true\\ S_t:(s,h)\to abort\end{array}}{while\ b\ do\ S_t:(s,h)\to abort}$$

$$\frac{\llbracket b\rrbracket(s,h)=false}{while\ b\ do\ S_t:(s,h)\to(s,h)}\qquad \frac{\llbracket b\rrbracket(s,h)=true\quad \begin{array}{l}S_t:(s,h)\to(s',h')\\ while\ b\ do\ S_t:(s',h')\to st\end{array}}{while\ b\ do\ S_t:(s,h)\to st}$$

If f is a map and A is a set, $f\rceil A$ denotes the restriction of f on A and $[f\mid x:A]$ denotes the function whose domain is $dom(f)\cup\{x\}$ and whose definition is $\lambda y.$ if $y=x$ then A else $f(y)$.

Lemma 1. *Suppose $\llbracket e\rrbracket(s,h)=(\alpha,\beta)$. If $\alpha\in Addrs$ then $\beta=h_r(\alpha)$.*

2.2 γ-Slices Heap Semantics

This section presents an operational semantics for programs resulted by our proposed transformation technique. The semantics uses a memory model where the memory is physically sliced into γ regions. The states of the semantics are introduced in Definition 2.2.

The semantics of arithmetic and Boolean expressions is defined similarly to the one-heap semantics except that η_h is replaced with $\eta_{\tilde{h}}$:

$$\llbracket d \rrbracket \in \mathit{Sliced\ States} \rightharpoonup \mathit{Values} \times (\mathscr{R} \cup \{\bot\})$$

$$\eta_{\tilde{h}}(\alpha,\beta) = \begin{cases} (\alpha,\beta), & \text{if } \alpha \in \mathbb{Z}; \\ (\alpha,i), & \text{if } \alpha \in dom(\tilde{h}_1) \text{ and } \exists! i \in \{1,\dots,\gamma\}.\ \tilde{h}_i(\alpha) \neq \phi; \\ \mathit{undefined}, & \text{otherwise.} \end{cases}$$

The inference rules of the semantics are defined as follows.

$$\frac{}{\mathit{skip} : (s,\tilde{h}) \rightsquigarrow (s,\tilde{h})} \qquad \frac{\llbracket e \rrbracket(s,\tilde{h}) \text{ is undefined}}{x := e : (s,\tilde{h}) \rightsquigarrow \mathit{abort}} \qquad \frac{\llbracket e \rrbracket(s,\tilde{h}) = (\alpha,\beta)}{x := e : (s,\tilde{h}) \rightsquigarrow ([s_v \mid x : \alpha],[s_r \mid x : \beta],\tilde{h})}$$

$$\frac{u = \mathit{min}\{t \mid \{a^t_{n,1},\dots,a^t_{n,n}\} \cap dom(\tilde{h}_1) = \emptyset\} \qquad \zeta_i(\alpha_j,\beta_j) = \begin{cases} \alpha_j, & \text{if } i = \beta_j; \\ \phi, & \text{otherwise.} \end{cases}}{\begin{array}{l} x := \mathit{cons}'(d_1 : Rs_1,\dots,d_n : Rs_n) : (s,\tilde{h}) \rightsquigarrow \\ \begin{cases} ([s_v \mid x : a^u_{n,1}],[s_r \mid x : \beta_1],\dots,[\tilde{h}_i \mid a^u_{n,1} : \zeta_i(\alpha_1,\beta_1) \mid \dots \mid a^u_{n,n} : \zeta_i(\alpha_n,\beta_n)],\dots), \\ \qquad\qquad \text{if } \llbracket d_i \rrbracket(s,\tilde{h}) = (\alpha_i,\beta_i) \quad \beta_i \in Rs_i; \\ \mathit{abort}, \qquad \mathit{otherwise}. \end{cases} \end{array}}$$

$$\frac{}{x :=_{Rs} [e] : (s,\tilde{h}) \rightsquigarrow \begin{cases} ([s_r \mid x : \tilde{h}_\beta(\alpha)],[s_r \mid x : \beta],\tilde{h}), & \text{if } \llbracket e \rrbracket(s,\tilde{h}) = (\alpha,\beta) \quad \beta \in Rs; \\ \mathit{abort}, & \text{otherwise.} \end{cases}}$$

$$\frac{}{\begin{array}{l} [e_1] :=_{Rs} e_2 : (s,\tilde{h}) \rightsquigarrow \\ \begin{cases} (s,\dots,\tilde{h}_1,[\tilde{h}_\beta \mid \alpha_1 : \alpha_2],\dots,\tilde{h}_\gamma), & \text{if} \llbracket e_i \rrbracket(s,h) = (\alpha_i,\beta),\ \tilde{h}_\beta(\alpha_1) \neq \phi, \text{ and } \beta \in Rs; \\ \mathit{abort}, & \mathit{otherwise.} \end{cases} \end{array}}$$

$$\frac{}{\mathit{dispose}(e) : (s,\tilde{h}) \rightsquigarrow \begin{cases} (s,\dots,\tilde{h}_i \rceil (dom(\tilde{h}_i) \setminus \{\alpha\}),\dots), & \text{if } \llbracket e \rrbracket(s,\tilde{h}) = (\alpha,\beta) \text{ and } \alpha \in dom(\tilde{h}_1); \\ \mathit{abort}, & \text{otherwise.} \end{cases}}$$

$$\frac{\begin{array}{c} S_1 : (s,\tilde{h}) \rightsquigarrow (s',\tilde{h}') \\ S_2 : (s',\tilde{h}') \rightsquigarrow st \end{array}}{S_1;S_2 : (s,\tilde{h}) \rightsquigarrow st} \qquad \frac{\begin{array}{c} S_1 : (s,\tilde{h}) \rightsquigarrow \mathit{abort} \\ S_2 \in \mathit{Stmts} \end{array}}{S_1;S_2 : (s,\tilde{h}) \rightsquigarrow \mathit{abort}} \qquad \frac{\llbracket b \rrbracket(s,\tilde{h}) \text{ is undefined}}{\mathit{if\ b\ then\ } S_t \mathit{\ else\ } S_f : (s,\tilde{h}) \rightsquigarrow \mathit{abort}}$$

$$\frac{\begin{array}{c} \llbracket b \rrbracket(s,\tilde{h}) = \mathit{true} \\ S_t : (s,\tilde{h}) \rightsquigarrow st \end{array}}{\mathit{if\ b\ then\ } S_t \mathit{\ else\ } S_f : (s,\tilde{h}) \rightsquigarrow st} \qquad \frac{\begin{array}{c} \llbracket b \rrbracket(s,\tilde{h}) = \mathit{false} \\ S_f : (s,\tilde{h}) \rightsquigarrow st \end{array}}{\mathit{if\ b\ then\ } S_t \mathit{\ else\ } S_f : (s,\tilde{h}) \rightsquigarrow st} \qquad \frac{\llbracket b \rrbracket(s,\tilde{h}) \text{ is undefined}}{\mathit{while\ b\ do\ } S_t : (s,\tilde{h}) \rightsquigarrow \mathit{abort}}$$

$$\frac{\begin{array}{c} \llbracket b \rrbracket(s,\tilde{h}) = \mathit{true} \\ S_t : (s,\tilde{h}) \rightsquigarrow \mathit{abort} \end{array}}{\mathit{while\ b\ do\ } S_t : (s,\tilde{h}) \rightsquigarrow \mathit{abort}} \qquad \frac{\llbracket b \rrbracket(s,\tilde{h}) = \mathit{false}}{\mathit{while\ b\ do\ } S_t : (s,\tilde{h}) \rightsquigarrow (s,\tilde{h})} \qquad \frac{\begin{array}{l} \llbracket b \rrbracket(s,\tilde{h}) = \mathit{true} \\ S_t : (s,\tilde{h}) \rightsquigarrow (s',\tilde{h}') \\ \mathit{while\ b\ do\ } S_t : (s',\tilde{h}') \rightsquigarrow st \end{array}}{\mathit{while\ b\ do\ } S_t : (s,\tilde{h}) \rightsquigarrow st}$$

Lemma 2. *Suppose $\llbracket e \rrbracket(s,\tilde{h}) = (\alpha,\beta)$. If $\alpha \in \mathit{Addrs}$ then $\tilde{h}_\beta(\alpha) \neq \phi$.*

Lemma 3. *The semantics introduced in this section are well defined.*

3 Pointer Analysis

This section presents a type system for pointer analysis [13,11,16,12,14,9] which is a flow-sensitive forward analysis. The analysis presented in this section is an augmented version of the type system we presented in [13]. We include the system here for the following reasons; (a) to make the current manuscript self-contained, (b) to show how differences between the language of this paper and that of [13] are treated, and (c) the following sections are built on this type system. The proof of the soundness for the type system presented here can be built by revising that presented in [13] pearing in mind that the operational semantics used in both cases are different. The augmentation mentioned above is related to arithmetic expressions. The analysis annotates program points with partial maps (types of our type system) that approximatively specifies for each store the addresses that can go into the store.

The set of points-to types, *PTS*, and the sub-typing relation are defined as follows.

Definition 3. *1. $pts = \{pts \mid pts : Var \cup A \rightarrow 2^{Addrs} \mid A \subseteq Addrs\}$. The bottom type is denoted by $\perp$.*

2. $pts \leq pts' \overset{\text{def}}{\iff} dom(pts) \subseteq dom(pts')$ and $\forall t \in dom(pts).\ pts(t) \subseteq pts'(t)$.

3. A state (s,h) has type pts, denoted by $(s,h) \models pts$, if
  - *$dom(h) \subseteq dom(pts)$,*
  - *$\forall x \in Var.\ s_v(x) \in Addrs \Longrightarrow s_v(x) \in pts(x)$, and*
  - *$\forall a \in dom(h).\ h_v(a) \in Addrs \Longrightarrow h_v(a) \in pts(a)$.*

The pointer analysis of a program takes the form of a post-type derivation for a given pre-type. Typically $\perp$, the bottom type, is the pre-type.

The judgement of an arithmetic expression e has the form $e : pts \rightarrow V$. The set V is either a set of addresses or a singleton of an integer. The intended meaning, which is formalized in Lemma 4, of this judgement is that V captures any address that e evaluates to in a state of type *pts*. In particular if V is a set of addresses, then e is either an address from V or any integer.

The judgement of a statement S has the form $S : pts \rightarrow pts'$. The intuition, which is formalized in Theorem 1, of this judgement is that if S is executed in a state of type *pts*, then any state (rather than *abort*) where the execution ends is of type pts'.

The inference rules of our type system for pointer analysis are the following:

$$\frac{}{n : pts \rightarrow \{n\}} \quad \frac{}{x : pts \rightarrow pts(x)} \quad \frac{}{cast(R_i \hookrightarrow R_j)e : pts \rightarrow \emptyset} \quad \frac{}{e : R_i : pts \rightarrow \emptyset}$$

$$\frac{e_1 : pts \rightarrow V_1 \qquad e_2 : pts \rightarrow V_2}{e_1 \oplus e_2 : pts \rightarrow \begin{cases} \{n \oplus m\} & \text{if } V_1 = \{n\} \wedge V_2 = \{m\}, \\ \{a^m_{i,j\oplus n} \mid a^m_{i,j} \in V_2 \wedge 1 \leq j \oplus n \leq i\} & \text{if } V_1 = \{n\} \wedge V_2 \subseteq Addrs, \\ \{a^m_{i,j\oplus n} \mid a^m_{i,j} \in V_1 \wedge 1 \leq j \oplus n \leq i\} & \textit{if } V_2 = \{n\} \wedge V_1 \subseteq Addrs, \\ \{a^m_{i,j} \mid j = 1, \ldots i \textit{ and for some } j, a^m_{i,j} \in V_1 \cup V_2\} & \textit{otherwise.} \end{cases}}$$

In the rest of the paper when $e : pts \rightarrow V$, we let V' denotes $V \cap Addrs$.

$$\frac{}{skip : pts \rightarrow pts} \qquad \frac{e : pts \rightarrow V}{x := e : pts \rightarrow [pts \mid x : V']} \ (ass^p)$$

$$\frac{v = min\{t \mid \{a^t_{n,1},\ldots,a^t_{n,n}\} \cap dom(pts) = \emptyset\} \qquad \forall 1 \leq i \leq n.\ d_i : pts \to V_i}{x := cons(d_1,\ldots,d_n) : pts \to \cup_{1\leq i\leq v}[pts \mid x : \{a^i_{n,1}\} \mid a^i_{n,1} : V'_1 \mid \ldots \mid a^i_{n,n} : V'_n]}\ (con^p)$$

$$\frac{e : pts \to V}{x := [e] : pts \to [pts \mid x : \cup_{a\in V'} pts(a)]}\ (lok^p) \qquad \frac{\forall 1 \leq i \leq 2.\ e_i : pts \to V_i}{[e_1] := e_2 : pts \to \cup_{a\in V'_1}[pts \mid a : V'_2]}\ (mut^p)$$

$$\frac{}{dispose(e) : pts \to pts}\ (dis^p) \qquad \frac{\begin{array}{c} S_1 : pts \to pts'' \\ S_2 : pts'' \to pts' \end{array}}{S_1;S_2 : pts \to pts'}\ (seq^p) \qquad \frac{\begin{array}{c} S_t : pts \to pts' \\ S_f : pts \to pts' \end{array}}{if\ b\ then\ S_t\ else\ S_f : pts \to pts'}\ (if^p)$$

$$\frac{S_t : pts \to pts}{while\ b\ do\ S_t : pts \to pts}\ (whl^p) \qquad \frac{pts'_1 \leq pts_1 \quad S : pts_1 \to pts_2 \quad pts_2 \leq pts'_2}{S : pts'_1 \to pts'_2}\ (csq^p)$$

Lemma 4. *Suppose that* $(s,h) \models pts$, $[\![d]\!](s,h) = (\alpha,\beta)$ *and* $d : pts \to V$. *Then*

1. $V \subseteq Addrs$ *or* $\exists n \in \mathbb{Z}.\ V = \{n\}$,
2. $\forall n \in \mathbb{Z}.\ V = \{n\} \Longrightarrow \alpha = n$, *and*
3. $\alpha \in Addrs \Longrightarrow \alpha \in V$.

The soundness of the type system is stated in the following theorem whose proof can be driven from the corresponding theorem in [13].

Theorem 1. *1.* $pts \leq pts' \Longleftrightarrow (\forall (s,h),\ (s,h) \models pts \Longrightarrow (s,h) \models pts')$.

2. *Suppose that* $S : pts \to pts'$ *and* $S : (s,h) \to (s',h')$. *Then* $(s,h) \models pts$ *implies* $(s',h') \models pts'$.

4 Region Analysis

In this section, we introduce a type system for region analysis which is a flow-sensitive, forward, and may analysis. The analysis annotates program points with region information in the form of partial maps from variables and memory locations to the power set of regions. Under these maps, the image of an address is an over-approximate set of regions where this address may exist. The image of a variable is an over-approximate set of regions from which the variable gets its value. We recall that the set of regions $\mathscr{R} = \{1,\ldots,\gamma\}$.

The set of region types, *PTS-REG*, and the sub-typing relation are defined as follows.

Definition 4. *1.* $REG = \{reg \mid reg : Var \cup A \to 2^{\mathscr{R}} \mid A \subseteq Addrs\}$.

2. $PTS\text{-}REG = \{(pts, reg) \in pts \times reg \mid dom(pts) = dom(reg)\}$.
3. $reg \leq reg' \overset{\text{def}}{\Longleftrightarrow} dom(reg) \subseteq dom(reg')$ *and* $\forall t \in dom(reg).\ reg(t) \subseteq reg'(t)$.
4. $(pts, reg) \leq (pts', reg') \overset{\text{def}}{\Longleftrightarrow} pts \leq pts'$ *and* $reg \leq reg'$.
5. *A state* (s,h) *has type reg, denoted by* $(s,h) \models reg$, *if*
   - $dom(h_r) \subseteq dom(reg)$,
   - $\forall t \in Var.\ s_r(t) = \beta \Longrightarrow \beta \in reg(t)$, *and* $s_r(t) = \bot \Longrightarrow reg(t) = \{1,\ldots,\gamma\}$, *and*
   - $\forall t \in dom(h_r).\ h_r(t) = \beta \Longrightarrow \beta \in reg(t)$.

6. *A state (s,h) has type (pts,reg), denoted by $(s,h) \models (pts,reg)$, if*
   - $dom(pts) = dom(reg)$,
   - $(s,h) \models pts$, *and*
   - $(s,h) \models reg$.

The inference rules of our type system for region analysis are the following:

$$\frac{}{n : (pts,reg) \rightarrow \{1,\dots,\gamma\}} \quad \frac{}{x : (pts,reg) \rightarrow reg(x)} \quad \frac{}{Cast(R_i \hookrightarrow R_j)e : (pts,reg) \rightarrow \{j\}}$$

$$\frac{e_1 : (pts,reg) \rightarrow Rs_1 \quad e_2 : (pts,reg) \rightarrow Rs_2 \quad e_1 \oplus e_2 : pts \rightarrow V}{e_1 \oplus e_2 : (pts,reg) \rightarrow (Rs_1 \cap Rs_2) \cup (\cup_{a \in V'} reg(a))} \quad \frac{}{e : R_i : (pts,reg) \rightarrow \{i\}}$$

$$\frac{x := e : pts \rightarrow pts' \quad e : (pts,reg) \rightarrow Rs}{x := e : (pts,reg) \rightarrow (pts',[reg \mid x : Rs])}\,(ass^R) \quad \frac{}{dispose(e) : (pts,reg) \rightarrow (pts,reg)}\,(dis^R)$$

$$\frac{}{skip : (pts,reg) \rightarrow (pts,reg)} \quad \frac{x := [e] : pts \rightarrow pts' \quad e : (pts,reg) \rightarrow Rs}{x := [e] : (pts,reg) \rightarrow (pts',[reg \mid x : Rs])}\,(lok^R)$$

$$\frac{\begin{array}{c} v = min\{t \mid \{a^t_{n,1},\dots,a^t_{n,n}\} \cap dom(reg) = \emptyset\} \\ \forall 1 \leq i \leq n.\ d_i : (pts,reg) \rightarrow Rs_i \end{array} \quad x := cons(d_1,\dots,d_n) : pts \rightarrow pts'}{x := cons(d_1,\dots,d_n) : (pts,reg) \rightarrow (pts',\cup_{1 \leq i \leq v}[reg \mid x : Rs_1 \mid a^i_{n,1} : Rs_1 \mid \dots \mid a^i_{n,n} : Rs_n])}\,(con^R)$$

$$\frac{\begin{array}{c} [e_1] := e_2 : pts \rightarrow pts' \\ e_2 : (pts,reg) \rightarrow Rs \end{array} \quad e_1 : pts \rightarrow V}{[e_1] := e_2 : (pts,reg) \rightarrow (pts',\cup_{a \in V'}[reg \mid a : Rs])}\,(mut^R) \quad \frac{\begin{array}{c} S_1 : (pts,reg) \rightarrow (pts'',reg'') \\ S_2 : (pts'',reg'') \rightarrow (pts',reg') \end{array}}{S_1;S_2 : (pts,reg) \rightarrow (pts',reg')}\,(seq^R)$$

$$\frac{\begin{array}{c} S_t : (pts,reg) \rightarrow (pts',reg') \\ S_f : (pts,reg) \rightarrow (pts',reg') \end{array}}{if\ b\ then\ S_t\ else\ S_f : (pts,reg) \rightarrow (pts',reg')}\,(if^R) \quad \frac{S_t : (pts,reg) \rightarrow (pts,reg)}{while\ b\ do\ S_t : (pts,reg) \rightarrow (pts,reg)}\,(whl^R)$$

$$\frac{(pts'_1,reg'_1) \leq (pts_1,reg_1) \quad S : (pts_1,reg_1) \rightarrow (pts_2,reg_2) \quad (pts_2,reg_2) \leq (pts'_2,reg'_2)}{S : (pts'_1,reg'_1) \rightarrow (pts'_2,reg'_2)}\,(csq^R)$$

The following lemma is needed in the proof of the following theorem which proves the soundness of the type system.

Lemma 5. *Suppose that $(s,h) \models (pts,reg)$, $[\![d]\!] = (\alpha,\beta)$, and $d : (pts,reg) \rightarrow Rs$. Then*

1. $\beta \in \mathscr{R} \Longrightarrow \beta \in Rs$.
2. $\beta = \perp \Longrightarrow Rs = \mathscr{R} = \{1,\dots,\gamma\}$.

Proof. The proof is by induction on the structure of d as follows:

1. If $d = n$, then by definition $\beta = \perp$ and $Rs = \mathscr{R}$ as required.
2. If $d = x$, then $\beta = s_r(x)$ and the required holds because $(s,h) \models reg$.
3. If $d = e : R_j$ or $d = Cast(R_i \hookrightarrow R_j)\ e$ then by definition $\beta = \{j\}$ and $Rs = \{j\}$ as required.
4. If $d = e_1 \oplus e_2$, then there are three subcases:
   (a) α is an integer and $\beta = \perp$. In this case $[\![\ e_1]\!] = (\alpha_1,\perp)$, $[\![\ e_2]\!] = (\alpha_2,\perp)$, and $\alpha = \alpha_1 \oplus \alpha_2$, where α_1 and α_2 are integers. Therefore by the induction hypothesis $V_1 = V_2 = \mathscr{R}$. Hence $\mathscr{R} \subseteq Rs \subseteq \mathscr{R}$ implying $Rs = \mathscr{R}$.

(b) α is an integer and $\beta \in \mathscr{R}$. In this case $[\![\ e_1]\!] = (\alpha_1, \beta)$, $[\![\ e_2]\!] = (\alpha_2, \perp)$, and $\alpha = \alpha_1 \oplus \alpha_2$, where α_1 and α_2 are integers. Therefore by the induction hypothesis $\beta \in V_1 \cap V_2 \subseteq Rs$.

(c) α is address. Then by Lemma 1, $\beta \in \mathscr{R}$ and $\beta = h_r(\alpha)$. In this case, $\beta \in reg(\alpha)$ because $(s,h) \models reg$ and $\alpha \in V'$ because $(s,h) \models pts$. Therefore $\beta \in \cup_{a \in V'} reg(a) \subseteq RS$.

Theorem 2. *1. $(pts, reg) \leq (pts', reg') \implies (\forall (s,h),\ (s,h) \models (pts, reg) \implies (s,h) \models (pts', reg'))$.*
2. $(S : (pts, reg) \rightarrow (pts', reg')) \implies (S : pts \rightarrow pts')$.
3. Suppose that $S : (pts, reg) \rightarrow (pts', reg')$ and $S : (s,h) \rightarrow (s', h')$. Then $(s,h) \models (pts, reg)$ implies $(s', h') \models (pts', reg')$.

Proof. The first two items are obvious. For the last item and by (2), it is enough to prove that $(s', h') \models reg'$. This is proved by induction on the structure of type derivation as follows:

1. The type derivation has the form (ass^R). In this case, $reg' = [reg \mid x : Rs]$ and $(s', h') = ([s_v \mid x : \alpha], [s_r \mid x : \beta], h)$, where $[\![e]\!](s,h) = (\alpha, \beta)$. By 2 and Theorem 1, $(s', h') \models pts'$. By Lemma 5, $(s', h') \models reg'$. Clearly $dom(pts') = dom(reg')$ and hence $(s', h') \models (pts'.reg')$.
2. The type derivation has the form (con^R). In this case, $reg' = \cup_{1 \leq i \leq v} [reg \mid x : Rs_1 \mid a^i_{n,1} : Rs_1 \mid \ldots \mid a^i_{n,n} : Rs_n]$ and $(s', h') = ([s_v \mid x : a^u_{n,1}], [s_r \mid x : \beta_1], [h_v \mid a^u_{n,1} : \alpha_1 \mid \ldots \mid a^u_{n,n} : \alpha_n], [h_r \mid a^u_{n,1} : \beta_1 \mid \ldots \mid a^u_{n,n} : \beta_n])$. Clearly, $1 \leq u \leq v$. For every $1 \leq i \leq n$ by Lemma 5, if $\beta_i \in \mathscr{R}$ then $\beta_i \in Rs_i$ and if $\beta_i = \perp$ then $Rs_i = \mathscr{R}$. We have $s'_r(x) = \beta_1 \in Rs_1 = reg'(x)$. We also have that $dom(h') \subseteq dom(reg')$ because $dom(h) \subseteq dom(reg)$ ($(s,h) \models reg$) and $1 \leq u \leq v$. It is obvious that for any $x \neq y \in Var$ and $a \in dom(h') \setminus \{a^u_{n,1}, \ldots, a^u_{n,n}\}$,
   - $s'_r(y) \in \mathscr{R}$ implies $s'_r(y) \in reg'(y)$,
   - $s'_r(y) = \perp$ implies $reg'(y) = \mathscr{R}$, and
   - $h'_r(a) \in \mathscr{R}$ implies $h'_r(a) = h_r(a) \in reg(a) \subseteq reg'(a)$.

   For every $1 \leq i \leq n$, if $h_r(a^u_{n,i}) \in \mathscr{R}$, then $h_r(a^u_{n,i}) = \beta_i \in Rs_i \subseteq reg'(a^u_{n,i})$. Hence $(s', h') \models reg'$.
3. The type derivation has the form (lok^R). In this case, $reg' = [reg \mid x : Rs]$ and $(s', h') = ([s_v \mid x : h_v(\alpha)], [s_r \mid x : \beta)], h)$, where $[\![e]\!](s,h) = (\alpha, \beta)$. By Lemma 5, $\beta \in Rs$. Also we have $\alpha \in Addrs \cap dom(h)$ and hence $\alpha \in V'$ by Lemma 4.
4. The type derivation has the form (mut^R). In this case, $reg' = \cup_{a \in V'} [reg \mid a : Rs]$ and $(s', h') = (s, [h_v \mid \alpha_1 : \alpha_2], [h_r \mid \alpha_1 : \beta])$, where $[\![e_i]\!](s,h) = (\alpha_i, \beta)$. We have $\alpha_1 \in dom(h) \cap V_1$ and $\beta \in Rs$ by Lemma 5. Therefore $h_r(\alpha_1) \in reg'(\alpha_1)$.

The remaining cases are straightforward to check.

5 Data Slicing

This section presents a technique for solving the principal problem, heap slicing, motivating the paper. The basic instrument of the technique is a type system which is an enrichment of the type system for region analysis with a transformation component. This transformation is that of heap slicing. In this section it is also shown that the transformation presented by the type system is sound in the sense that the original program and that results from the transformation produce the same result.

Definition 5. *A sliced heap* $(s,\tilde{h})$ *is a valid slicing of a state* (s,h)*, denoted by* $(s,h) \sim (s,\tilde{h})$*, if*

1. $dom(h) = dom(\tilde{h}_1)$*, and*
2. $(\forall a \in dom(h))\ (h_v(a), h_r(a)) = (\alpha,\beta) \Longrightarrow \tilde{h}_\beta(a) = \alpha$ *and* $(\forall i \neq \beta)\ h_i(a) = \phi$.

Definition 6. 1. $Slice : Heaps \to Sliced\ Heaps :\ h \mapsto (h_1,\dots,h_\gamma)$*, where for every* $i \in [1,\gamma]$,

$$h_i : dom(h) \to Values^+ : a \mapsto \begin{cases} h_v(a), & \text{if } h_r(a) = i; \\ \phi, & \text{otherwise.} \end{cases}$$

2. $Con : Sliced\ Heaps \to Heaps :\ \tilde{h} \mapsto (h_v, h_r)$*, where*

$$h_v : dom(\tilde{h}) \to Values : a \mapsto \tilde{h}_{i_a}(a) \qquad h_r : dom(\tilde{h}) \to \mathscr{R} : a \mapsto i_a,\ \textit{where}$$

i_a *is the unique index such that* $\tilde{h}_{i_a}(a) \neq \phi$.

3. $Slice_S : States \to Sliced\ States :\ (s,h) \mapsto (s, Slice(H))$.
4. $Con_S : SlicedStates \to States :\ (s,\tilde{h}) \mapsto (s, Con(\tilde{H}))$

Lemma 6. *The maps of the previous definitions are well-defined. Moreover* $Slice_S$ *and* Con_S *are inverses to each other.*

The inference rules of our type system are the following:

$$\rho(d_i,Rs_i) = \begin{cases} d_i : Rs_i, & \text{if } d_i = e_i; \\ d_i, & \text{otherwise.} \end{cases} \qquad \frac{}{x := e : (pts,reg) \to (pts',reg') \hookrightarrow x := e} \qquad \frac{}{skip : (pts,reg) \to (pts,reg) \hookrightarrow skip}$$

$$\frac{x := cons(d_1,\dots,d_n) : (pts,reg) \to (pts',reg') \quad \forall 1 \le i \le n.\ d_i : (pts,reg) \to Rs_i}{x := cons(d_1,\dots,d_n) : (pts,reg) \to (pts',reg') \hookrightarrow x := cons'(\rho(d_1,Rs_1),\dots,\rho(d_n,Rs_n))}$$

$$\frac{\begin{array}{l} x := [e] : (pts,reg) \to (pts',reg') \\ e : (pts,reg) \to Rs \end{array}}{x := [e] : (pts,reg) \to (pts',reg') \hookrightarrow x :=_{Rs} [e]} \qquad \frac{}{dispose(e) : (pts,reg) \to (pts,reg') \hookrightarrow dispose'(e)}$$

$$\frac{\begin{array}{l} [e_1] := e_2 : (pts,reg) \to (pts',reg') \\ e_1 : (pts,reg) \to Rs_1 \qquad e_2 : (pts,reg) \to Rs_2 \end{array}}{[e_1] := e_2 : (pts,reg) \to (pts',reg') \hookrightarrow [e_1] :=_{Rs_1 \cap Rs_2} e_2}$$

$$\frac{\begin{array}{l} S_1 : (pts,reg) \to (pts'',reg'') \hookrightarrow S_1' \\ S_2 : (pts'',reg'') \to (pts',reg') \hookrightarrow S_2' \end{array}}{S_1;S_2 : (pts,reg) \to (pts',reg') \hookrightarrow S_1';S_2'} \qquad \frac{S_t : (pts,reg) \to (pts,reg) \hookrightarrow S_t'}{while\ b\ do\ S_t : (pts,reg) \to (pts,reg') \hookrightarrow while\ b\ do\ S_t'}$$

$$\frac{\begin{array}{l} S_t : (pts,reg) \to (pts',reg') \hookrightarrow S_t' \\ S_f : (pts'',reg'') \to (pts',reg') \hookrightarrow S_f' \end{array}}{if\ b\ then\ S_t\ else\ S_f : (pts,reg) \to (pts',reg') \hookrightarrow if\ b\ then\ S_t'\ else\ S_f'}$$

$$\frac{(pts_1',reg_1') \le (pts_1,reg_1) \quad S : (pts_1,reg_1) \to (pts_2,reg_2) \hookrightarrow S' \quad (pts_2,reg_2) \le (pts_2',reg_2')}{S : (pts_1',reg_1') \to (pts_2',reg_2') \hookrightarrow S'}$$

Theorem 3. *(Soundness) Suppose that $S : (pts, reg) \to (pts', reg') \hookrightarrow S'$ and $(s,h) \sim (s,\tilde{h})$. Then*

1. *If $S : (s,h) \to (s',h')$, then there exists a state $(s',\tilde{h}')$ such that $S' : (s,\tilde{h}) \rightsquigarrow (s',\tilde{h}')$ and $(s',h') \sim (s',\tilde{h})$.*
2. *If $S' : (s,\tilde{h}) \rightsquigarrow (s',\tilde{h}')$, then there exists a state (s',h') such that $S : (s,h) \to (s',h')$ and $(s',h') \sim (s',\tilde{h}')$.*

Proof. The proof is by induction on the structure of type derivation. For the base cases in the proof of (1), take $(s',\tilde{h}') = Slice_s((s',h'))$. For the base cases in the proof of (2), take $(s',h') = Con_s((s',\tilde{h}'))$.

6 Related and Future Work

In [6], Condit et al. present data slicing [28,31], a program transformation which divides the heap into separate regions, for a C-like language. The basic idea in [6] is to syntactically slice structures defined in a given program. Then, the slicing of the program commands is calculated using sliced versions of program structures. The physical slicing of the program heap follows upon executing the sliced program.

Related concepts to data slicing are program slicing, intentional polymorphism, structure splitting. Program slicing [1,18,4,26] finds the program portions that contribute to evaluating the value of a given variable at a given program point. In other words, program slicing [25] is a practicable technique to bound the focus of a job to certain part of a program. Program slicing is used in program comprehension, testing, restructuring, debugging, and optimizing. A technique to compile polymorphism while still being able to use types information at run time is intentional polymorphism [7,17,8]. The similarity to data slicing comes from the fact that intentional polymorphism enables the compiler of preserving type safety and efficiently representing types. An alternative approach to data slicing, is structure splitting [2,5]. This approach marks the non-active fields of data structures by adding new pointers to data structures. Clearly this pointer addition does sacrifices the backward compatibility. Therefore data slicing is advantageous over structure splitting.

Among advantages of data slicing is preserving backward compatibility. As an alternative, splay trees [30,21,27] can be used to preserve backward compatibility. However some research like [6] concludes that the use of splay trees is more expensive in terms of time and complexity of the system used in implementation.

A typical approach for heap slicing is the algorithmic style. However the use of type systems in program analysis (in general) [13,11,16,12], rather than classical algorithms, and in data slicing (in particular) is very useful for applications like certified code or proof-carrying code. The catch of the type systems approach is that type derivations serve as proofs for the technique result.

Programs and data structures can mathematically be represented by mathematical domains and maps between domains. This representation is called denotational semantics of programs. An important direction for future research is to transfer concepts of data and program slicing to the side of denotational semantics [15,10]. This enables us to mathematically study in deep heap slicing and translates back obtained results to the side of programs and data structures.

References

1. Barraclough, R.W., Binkley, D., Danicic, S., Harman, M., Hierons, R.M., Kiss, Á., Laurence, M., Ouarbya, L.: A trajectory-based strict semantics for program slicing. Theor. Comput. Sci. 411(11-13), 1372–1386 (2010)
2. Carrillo, S., Siegel, J., Li, X.: A control-structure splitting optimization for gpgpu. In: Johnson, G., Trinitis, C., Gaydadjiev, G., Veidenbaum, A.V. (eds.) Conf. Computing Frontiers, pp. 147–150. ACM (2009)
3. Chen, C.-L., Lin, S.-H.: Formulating and solving a class of optimization problems for high-performance gray world automatic white balance. Appl. Soft Comput. 11(1), 523–533 (2011)
4. Cheney, J.: Program slicing and data provenance. IEEE Data Eng. Bull. 30(4), 22–28 (2007)
5. Chilimbi, T.M., Davidson, B., Larus, J.R.: Cache-conscious structure definition. In: PLDI, pp. 13–24 (1999)
6. Condit, J., Necula, G.C.: Data Slicing: Separating the Heap into Independent Regions. In: Bodik, R. (ed.) CC 2005. LNCS, vol. 3443, pp. 172–187. Springer, Heidelberg (2005)
7. Crary, K., Weirich, S., Gregory Morrisett, J.: Intensional polymorphism in type-erasure semantics. J. Funct. Program. 12(6), 567–600 (2002)
8. Duggan, D.: Dynamic typing for distributed programming in polymorphic languages. ACM Trans. Program. Lang. Syst. 21(1), 11–45 (1999)
9. El-Zawawy, M., Daoud, N.: New error-recovery techniques for faulty-calls of functions. Computer and Information Science 4(3) (May 2012)
10. El-Zawawy, M.A.: Semantic spaces in Priestley form. PhD thesis, University of Birmingham, UK (January 2007)
11. El-Zawawy, M.A.: Flow Sensitive-Insensitive Pointer Analysis Based Memory Safety for Multithreaded Programs. In: Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2011, Part V. LNCS, vol. 6786, pp. 355–369. Springer, Heidelberg (2011)
12. El-Zawawy, M.A.: Probabilistic pointer analysis for multithreaded programs. ScienceAsia 37(4) (December 2011)
13. El-Zawawy, M.A.: Program optimization based pointer analysis and live stack-heap analysis. International Journal of Computer Science Issues 8(2) (March 2011)
14. El-Zawawy, M.A.: Dead code elimination based pointer analysis for multi-threaded programs. Journal of the Egyptian Mathematical Society (January 2012), doi:10.1016/j.joems.2011.12.011
15. El-Zawawy, M.A., Jung, A.: Priestley duality for strong proximity lattices. Electr. Notes Theor. Comput. Sci. 158, 199–217 (2006)
16. El-Zawawy, M.A., Nayel, H.A.: Partial redundancy elimination for multi-threaded programs. IJCSNS International Journal of Computer Science and Network Security 11(10) (October 2011)
17. Harper, R., Gregory Morrisett, J.: Compiling polymorphism using intensional type analysis. In: POPL, pp. 130–141 (1995)
18. Gaikovina Kula, R., Fushida, K., Kawaguchi, S., Iida, H.: Analysis of Bug Fixing Processes Using Program Slicing Metrics. In: Ali Babar, M., Vierimaa, M., Oivo, M. (eds.) PROFES 2010. LNCS, vol. 6156, pp. 32–46. Springer, Heidelberg (2010)
19. George, C.: Proof-carrying code. In: Henk, C., van Tilborg, H.C.A., Jajodia, S. (eds.) Encyclopedia of Cryptography and Security, 2nd edn., pp. 984–986. Springer (2011)
20. Nielson, F., Nielson, H.R., Hankin, C.L.: Principles of Program Analysis. Springer (1999); second printing (2005)
21. Pettie, S.: Splay trees, davenport-schinzel sequences, and the deque conjecture. In: Teng, S.-H. (ed.) SODA, pp. 1115–1124. SIAM (2008)

22. Pfenning, F., Caires, L., Toninho, B.: Proof-Carrying Code in a Session-Typed Process Calculus. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 21–36. Springer, Heidelberg (2011)
23. Prasad, S., Arun-Kumar, S.: Introduction to operational semantics. In: The Compiler Design Handbook, pp. 841–890 (2002)
24. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Symposium on Logic in Computer Science, p. 55 (2002)
25. Sasirekha, N., Edwin Robert, A., Hemalatha, M.: Program slicing techniques and its applications. CoRR, abs/1108.1352 (2011)
26. Tip, F.: A survey of program slicing techniques. J. Prog. Lang. 3(3) (1995)
27. Weiser, M.: Program slicing. IEEE Trans. Software Eng. 10(4), 352–357 (1984)
28. Xin, B., Zhang, X.: Memory slicing. In: Rothermel, G., Dillon, L.K. (eds.) ISSTA, pp. 165–176. ACM (2009)
29. Ye, X., Li, P.: Parallel program performance modeling for runtime optimization of multi-algorithm circuit simulation. In: Sapatnekar, S.S. (ed.) DAC, pp. 561–566. ACM (2010)
30. Zhang, S., Cui, Z., Gong, S.-R., Liu, Q., Fan, J.-X.: A data aggregation algorithm based on splay tree for wireless sensor networks. JCP 5(4), 492–499 (2010)
31. Zhang, X., Gupta, R., Zhang, Y.: Cost and precision tradeoffs of dynamic data slicing algorithms. ACM Trans. Program. Lang. Syst. 27(4), 631–661 (2005)