Program Optimization Based Pointer Analysis and Live Stack-Heap Analysis

Mohamed A. El-Zawawy

**Department of Mathematics, Faculty of Science, Cairo University
Giza, 12316, Egypt**

Abstract

In this paper, we present type systems for flow-sensitive pointer analysis, live stack-heap (variables) analysis, and program optimization. The type system for live stack-heap analysis is an enrichment of that for pointer analysis; the enrichment has the form of a second component being added to types of the latter system. Results of pointer analysis are proved useful via their use in the type system for live stack-heap analysis. The type system for program optimization is also an augmentation of that for live stack-heap analysis, but the augmentation takes the form of a transformation component being added to inference rules of the latter system. The form of program optimization being achieved is that of dead-code elimination. A form of program correction may result indirectly from eliminating faulty code (causing the program to abort) that is dead. Therefore program optimization can result in program correction. Our type systems have the advantage of being compositional and relatively-simply structured.

The novelty of our work comes from the fact that our type system for program optimization associates the optimized version of a program with a justification (in the form of a type derivation) for the optimization. This justification is pretty much appreciated in many research areas like certified code (proof-carrying code) which is the motivation of this work.

Keywords*: Pointer analysis, Live variables analysis, Live stack-heap analysis, Program optimization, Type systems, Certified code.*

1. Introduction

Rather than dynamic code analysis concerned with analyzing programs during execution time, static code analysis (statics analysis) [14] is a concept describing analyzing programs without actually executing them. Static analysis can result in improving the quality of the code in different ways (including correcting and optimizing the code) or in verifying industrial standards of the code. Data-flow analysis, one of the techniques used in static analysis, is useful for collecting information for each program point. An analysis whose results do not change due to permuting a statement sequence $S_1;S_2$ into $S_2;S_1$ is described as flow-insensitive; otherwise it is described as flow-sensitive. For a flow-sensitive analysis, if the program is traversed forwardly (backwardly) to collect information, the technique is called a forward (backward) analysis. If the collected information may (must) be true, the technique is described as may (must). Examples of forward-may and backward-may analyses are pointer and live variables analyses [14], respectively. Pointer analysis calculates for each store (a variable or a memory location) at every program point the set of addresses that have a chance of being contained in that store at that program point. Roughly speaking, live variables analysis calculates for every program point the set of variables used later in the program. In case of pointer programs, we call this analysis live stack-heap analysis and it calculates the set of variables and memory locations that are used later in the program. Results of live variables analysis can be used to eliminate unnecessary code in a technique called dead-code elimination.

Although static analysis is usually treated in an algorithmic style [14], there are other frameworks that can be used to successfully achieve static analysis. One of such frameworks is type systems [9, 20, 2, 15] that has proved so far to be a very convenient tool for this job. In the algorithmic fashion, the work is done on an annotated form of the program control- flow graph. However in the type-systems manner the work is done on the program using its phrase structure. This fact is advantageous to the use of the type-systems framework when it comes to optimizing programs. This is so because the algorithmic style usually produces only the optimized version of the program. However the type-systems style is conveniently-capable of producing the optimized version together with a justification (in the form of a type derivation) for the optimization. This justification is necessary in applications like certified code. Also the relative simplicity of inference rules of type systems makes their framework auspicious.

Motivation

The program on the left-hand side of Figure 1 is a motivating example of our work. Suppose that *y* is the only variable whose value concerns us at the end of the program. Then the last statement is unnecessary (dead code). Also the assignment statement in line 6 is a dead code and it causes the program to aborts because the value of *i* is not in the domain of the heap. Therefore removing theses statements optimizes the program and in the same time removes a cause for abortion.

```
1. i := 1;                                  i := 1;
2. x := cons(10,22,53);                     x := cons(10,22,53);
3. while(i < 3) do                          while(i < 3)do
4.     {x := x + 1;                             {x := x + 1;
5.      i := i + 1;}              ⇒              i := i + 1;}
6.     k := [i];                            skip;
7.     y := [x];                             y := [x];
8.     [x] := 5;                             skip
```

Fig.1. A motivating example.

The objective of this paper is to develop a technique that transforms a program like this one into an optimized version like that in the right-hand side of Figure 1 and also produces a proof or justification for the transformation process.

All in all this paper tackles the problem of transforming pointer programs into optimized and possibly corrected versions and producing justifications for the transformation process. The importance of producing the justification comes from the area of certified code which is the motivation of our work and which provides good applications for the work as well. The program optimization, meant here, is dead-code elimination. The optimized version of a program is possibly a corrected version as well; this is the case if reasons for abortions in the program are included in dead code and hence gets removed with the dead code. In other words, program optimization can result in program correction. Our tool for solving this problem is type systems. Up to our knowledge, our paper is the first to tackle this problem (using type systems) for pointer programs.

Contributions

1. An original type system for flow-sensitive pointer analysis.
2. A novel type system for live stack-heap (variables) analysis for pointer programs. This type system utilizes results of our type system for pointer analysis and is an enrichment of it.
3. The third contribution is a new type system for optimizing and possibly correcting pointer programs. This type system serves also as a tool for obtaining a justification (in the form of a type derivation) for every individual transformation and is an augmentation of our type system for live stack-heap analysis.

$$n \in \mathbb{Z}, \oplus \in \{+,-,\times\}, \mathit{and}\ x \in \mathit{Var}$$

$$\begin{aligned}
e \in \mathit{Aexprs} &::= x \mid n \mid e_1 \oplus e_2\\
b \in \mathit{Bexprs} &::= \mathit{true} \mid \mathit{false} \mid \mathit{not}(b) \mid e_1 = e_2 \mid e_1 \le e_2\\
&\quad\ \mid b_1 \wedge b_2 \mid b_1 \vee b_2\\
S \in \mathit{Stmts} &::= x := e \mid x := \mathit{cons}(e_1,\dots,e_n) \mid x := [e] \mid [e_1] := e_2\\
&\quad\ \mid \mathit{dispose}(e) \mid \mathit{skip} \mid S; S_2\\
&\quad\ \mid \mathit{if}\ b\ \mathit{then}\ S_t \mathit{else}\ S_f \mid \mathit{while}\ b\ \mathit{do}\ S_t
\end{aligned}$$

Fig.2. Grammar of the *while$_p$* language.

Organization

The rest of the paper is organized as follows. The language *while$_p$* (the while language enriched with pointer commands) and an operational semantics for its constructs are presented in Section 2. Our proposed type systems for flow-sensitive pointer analysis and live stack-heap analysis 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 is presented in Section 6**.**

2. The Programming Language

The programming language that we are using is usually used to introduce separation logic like in [19] and its operational semantics is a slightly-modified version of that in [19]. The language is an imperative one that is enriched with commands dealing with pointers. We call this language *while$_p$*. This section presents the language *while$_p$* with an operational semantics to its constructs. The grammar of the *while$_p$* language is shown in Figure 2, where *Var* is a finite set of program variables.

$$n \in \mathbb{Z}, \oplus \in \{+,-,\times\}, \mathit{and}\ x \in \mathit{Var}$$

$$\begin{aligned}
&\mathit{Atoms} \subseteq \mathit{Integers}\\
&\mathit{Addrs} = \{a^i_{j,k} \mid i,j,k \in \mathbb{N}^+, k \le j\}\\
&= \{a^1_{1,1}, a^1_{2,1}, a^1_{2,1}, a^1_{2,2}, a^1_{3,1}, a^1_{3,2}, a^1_{3,3}, \dots, a^2_{1,1}, a^2_{2,1}, a^2_{2,1}, a^2_{2,2}, a^2_{3,1}, \dots, \}\\
&\mathit{Values} = \mathit{Integers} \cup \mathit{Addrs} \cup \{\mathit{nil}\}
\end{aligned}$$

Fig.3. Atoms, addresses, and values of *while$_p$* semantics.

For any $m \in N^+$, we assume that the memory has an infinite number of arrays of length *m* with addresses $\{a^1_{m,1}, a^1_{m,2}, \dots, a^1_{m,1}, a^2_{m,1}, a^2_{m,2}, \dots, a^2_{m,m}, \dots\}$ Therefore the set of address, *Addrs*, has the form presented in Figure 3. This memory model, rather than letting addresses to be a subset of integers, has the advantage of reducing the chance of messing with the memory. This is so because a number which is intended to be used as a numerical value (not as an address) can be an address as well and therefore it can be accidently used to access un-allowed or unintended memory cells. In order to facilitate evaluating inequalities we assume that *Values* is equipped with an order.

$$v_1 \oplus v_2 = \begin{cases} a^m_{i,j\oplus n} & \mathit{if}\ \big((v_1 = a^m_{i,j} \wedge v_2 = n \in \mathbb{Z})\ \mathit{or}\ (v_2 = a^m_{i,j} \wedge v_1 = n \in \mathbb{Z})\big)\ \mathit{and}\ 1 \le j \oplus n \le i, \\ \mathit{nil} & \mathit{otherwise.} \end{cases}$$

$$\frac{}{\mathit{skip} : (s,h) \to (s,h)} \qquad \frac{}{x := [e] : (s,h) \to ([s \mid x{:}\llbracket e\rrbracket s], h)} \qquad \frac{\llbracket e\rrbracket s \in \mathit{dom}(h)}{x := e{:}\,(s,h) \to (s, h\rceil(\mathit{dom}(h) \setminus \{\llbracket e\rrbracket s\}))}$$

$$\frac{\llbracket e\rrbracket s \notin \mathit{dom}(h)}{x := e{:}\,(s,h) \to \mathit{abort}} \qquad \frac{\llbracket e\rrbracket s \in \mathit{dom}(h)}{x := e{:}\,(s,h) \to ([s \mid x{:}\,h(\llbracket e\rrbracket s)], h)} \qquad \frac{\llbracket e_1\rrbracket s \notin \mathit{dom}(h)}{[e_1] := e_2 : (s,h) \to \mathit{abort}} \qquad \frac{\llbracket e\rrbracket s \notin \mathit{dom}(h)}{\mathit{dispose}\,(e){:}\,(s,h) \to \mathit{abort}}$$

$$\frac{u = \min\{t \mid \{a^t_{n,1}, \dots, a^t_{n,n}\} \cap \mathit{dom}(h) = \emptyset\}}{x := \mathit{cons}\,(e_1,\dots,e_n){:}\,(s,h) \to ([s \mid x{:}\,a^u_{n,1}], [h \mid a^u_{n,1} \mid \llbracket e_1\rrbracket s \mid \cdots \mid a^u_{n,n}{:}\,\llbracket e_n\rrbracket s])} \qquad \frac{\llbracket e_1\rrbracket s \in \mathit{dom}(h)}{[e_1] := e_2 : (s,h) \to (s, [h \mid \llbracket e_1\rrbracket s{:}\,\llbracket e_2\rrbracket s])}$$

$$\frac{S_1 : (s,h) \to (s'',h'') \qquad S_2 : (s'',h'') \to (s',h')}{S_1;\, S_2{:}\,(s,h) \to (s',h')} \qquad \frac{S_1 : (s,h) \to \mathit{abort} \qquad S_2 \in \mathit{Stmt}}{S_1;\, S_2{:}\,(s,h) \to \mathit{abort}} \qquad \frac{\llbracket b\rrbracket s = \mathit{true} \qquad S_t{:}\,(s,h) \to \mathit{st}}{\mathit{if}\ b\ \mathit{then}\ S_t\ \mathit{else}\ S_f : (s,h) \to \mathit{st}}$$

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

Fig.4. Operation semantics for *while*$_p$.

Definition 1.
A stack (heap) is a map (finite partial map) from Var (Addrs) to Values. A state is an abort or a pair of a stack and a heap.
Arithmetic and Boolean expressions have the same semantics as in the case of the *while* language except for the operation $\oplus$. The semantics of this operation on *Values* is defined as usual if both of its operands are integers and otherwise as in Figure 4.
The semantics of *whilep* statements is given by an operational semantics whose transition relation is denoted by $\longrightarrow$ and whose configurations (nonterminal and terminal) are defined in Definition 1. In the inference rules of the semantics (Figure 4), *st* denotes a state.
The *cons* allocates the array $a^u_{n,1}, \dots, a^u_{n,n}$ with the minimum dimension, u, of all available arrays of length n. The allocation takes place by storing the address $a^u_{n,1}$ in x and semantics of expressions $e_1, \dots, e_n$ in addresses $a^u_{n,1}, \dots, a^u_{n,n}$, respectively. 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 $\mathit{dom}(f) \cup \{x\}$ and whose definition is $\lambda y.$ if $y = x$ then A else $f(y)$.

3. Pointer Analysis

In this section, we introduce a type system for flow-sensitive pointer analysis which is a forward-may analysis that assigns to each program point a partial map from variables and memory locations to the power set of addresses. Under this map, the image of an element is an over-approximate set of addresses that the element may contain (point to) at this program point. The set of points-to types, *PTS*, and the sub-typing relation are defined as follows.

Definition 2.
1. $PTS = \{pts \mid pts\colon Var \cup d \longrightarrow 2^{Addrs} \mid d \subseteq Addrs\}$. The bottom type is denoted by $\perp$.
2. $pts \le pts' \Leftrightarrow dom(pts) \subseteq dom(pts')$ 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
– $dom(h) \subseteq dom(pts)$,
– $\forall x \in Var.\ s(x) \in Addrs \Rightarrow s(x) \in pts(x)$, and
– $\forall a \in dom(h).\ h(a) \in Addrs \Rightarrow h(a) \in pts(a)$.

Given a points-to type *pts*, the pointer analysis is achieved for a statement *S* via a post-type derivation for *S* and *pts* as the pre-type. Typically the pre-type *pts* is the bottom type $\perp$. The judgment of an arithmetic expression *e* has the form $e\colon pts \longrightarrow V$. The set *V* is either a set of addresses or a singleton of an integer. The intended meaning, which is formalized in Lemma 1, of this judgment 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*, any integer, or *nil*. The judgment of a statement *S* has the form $S\colon pts \longrightarrow pts'$. The intuition, which is formalized in Theorem 1, of this judgment 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'*. In the rest of the paper when $e\colon pts \longrightarrow V$, we let V' denotes $V \cap Addrs$. The inference rules of our type system for pointer analysis are presented in Figure 5.

Lemma 1.
Suppose that $(s, h) \models pts$ and $e\colon pts \longrightarrow V$. Then

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

$$\frac{}{skip : pts \rightarrow pts} \quad \frac{e : pts \rightarrow V}{x := e : pts \rightarrow [pts \mid x : V']}(ass^p) \quad \frac{e : pts \rightarrow V}{x := [e] : pts \rightarrow [pts \mid x : \cup_{a \in V'} pts(a)]}(lok^p) \quad \frac{}{dispose(e) : pts \rightarrow pts}(dis^p)$$

$$\frac{v = \min\{t \mid \{a^t_{n,1}, \ldots, a^t_{n,n}\} \cap dom(pts) = \emptyset\} \quad \forall i. e_i : pts \rightarrow V_i}{x := cons(e_1, \ldots, e_n) : pts \rightarrow [pts \mid x \mid \{a^i_{n,1}\} \mid a^i_{n,1} : V'_1 \mid \cdots \mid a^i_{n,n} : V'_n]}(con^p) \quad \frac{\forall 1 \leq i \leq 2. e_i : pts \rightarrow V_i}{[e_1] := e_2 : pts \rightarrow \cup_{a \in V'_1} [pts \mid a : V'_2]}(mut^p)$$

$$\frac{S_1 : pts \rightarrow pts'' \quad S_2 : pts'' \rightarrow pts'}{S_1; S_2 : pts \rightarrow pts'}(seq^p) \quad \frac{S_t, S_f : pts \rightarrow pts'}{if\ b\ then\ S_t\ else\ S_f : pts \rightarrow pts'}(if^p) \quad \frac{S_t : pts \rightarrow pts}{while\ b\ do\ S_t : pts \rightarrow pts}(whl^p)$$

$$\frac{pts'_1 \leq pts_1 \quad S : pts_1 \rightarrow pts_2 \quad pts_2 \leq pts'_2}{S : pts'_1 \rightarrow pts'_2}(csq^p)$$

Fig.5. Inference rules of a type system for pointer analysis.

1. $V \subseteq Addrs$ or $\exists n \in$ Z. $V = \{n\}$,
2. $\forall n \in$ Z. $V = \{n\} \Rightarrow [\![e]\!]s = n$, and
3. $[\![e]\!]s \in Addrs \Rightarrow [\![e]\!]s \in V$.

Proof. By induction on the structure of e. We present the proof of the last item. If $e = n$ then $[\![e]\!]s = n \notin Addrs$. If $e = x$ then $[\![x]\!]s = s(x) \in Addrs$ implies $s(x) \in pts(x) = V$ because $(s, h) \models pts$. Now suppose $e = e_1 \oplus e_2$, $e_1 : pts \rightarrow V_1$, and $e_2 : pts \rightarrow V_2$. If $[\![e_1 \oplus e_2]\!]s \in Addrs$, then we have one of the following cases:
1. $[\![e_1]\!]s = a^m_{i,j}$, $[\![e_2]\!]s = n$, and $1 \leq j \oplus n \leq i$.
2. $[\![e_1]\!]s = n$, $[\![e_2]\!]s = a^m_{i,j}$, and $1 \leq j \oplus n \leq i$.
In the first case, by the induction hypothesis $a^m_{i,j} \in V_1$ and if $V_2 = \{t\}$, then, by (2), $n = t$ and $[\![e_1 \oplus e_2]\!]s = a^m_{i,j\oplus n} \in \{a^m_{i,j\oplus n} \mid a^m_{i,j} \in V_1 \wedge 1 \leq j \oplus n \leq i\} = V$. If $V_2 \subseteq Addrs$ then $[\![e_1 \oplus e_2]\!]s = a^m_{i,j\oplus n} \in V$. The second case is similar to the first case. ■

The following lemma is needed in the proof of the following soundness theorem and it is obvious because $(s, h) \models pts$ implies $dom(h) \subseteq dom(pts)$.

Lemma 2.
Suppose that $(s, h) \models pts$, $u = min\{t \mid \{a^t_{n,1}, \ldots, a^t_{n,n}\} \cap dom(h) = \emptyset\}$, and $v = min\{t \mid \{a^t_{n,1}, \ldots, a^t_{n,n}\} \cap dom(pts) = \emptyset\}$. Then $1 \leq u \leq v$.

The rules (ass_p) and (dis_p) are straightforward. For the rule (con_p) and by Lemma 2, executing the *cons* statement in a state of type *pts* results in allocating one of the arrays $\{a^j_{n,1}, \ldots, a^j_{n,n}\}$, $1 \leq j \leq v$. But it is not obvious which of these arrays will be allocated. Therefore the rule (con_p) takes into account all these possibilities by adding the addresses of these arrays to $pts(x)$ and adding V_i' to the image, under *pts*, of each location $a^j_{n,i}$.
In the rule (lok_p), V' contains any address that e evaluates to in a state of type *pts*. Therefore the set $\cup_{a\in V'} pts(a)$ captures any address that goes into x after executing the look-up statement in a state of type *pts*. For the rule (mut_p), there are two cases for V_1, namely either $|V_1| = 1$ or $|V_1| \neq 1$. In the first case, the rule (mut_p) cuts down to a form that is pretty much similar to the rule (ass_p). In the second case, it is not obvious to which address the assignment will happen. Hence the post-type is calculated from the pre-type by including the set V_2' in the image of every element of V_1'. The rules (seq_p), (if_p), and (csq_p) are clear.
As it is evident from the (whl_p) rule, an invariant type is necessary to type a while statement. The required invariant type is calculated as a fix-point of an order-preserving map over the complete lattice *pts* using a given pre-type. The consequence rule can be used to show that the fix-point is indeed the required invariant type.

Theorem 1. (*Soundness*)
1. $pts \leq pts'$ iff (For every state (s, h), $(s, h) \models pts \Rightarrow (s, h) \models pts'$).
2. Suppose that S: $pts \rightarrow pts'$ and S: $(s, h) \rightarrow (s', h')$. Then $(s, h) \models pts$ implies $(s', h') \models pts'$.
Proof. 1. The left-to-right direction is obvious. The other direction is proved as follows. Suppose $x \in Var$, a, $b \in Addrs$, and $a \in pts(x)$. Then the state $(s, h) = (\{(x, a), (y, 0) \mid x \neq y \in Var\}, \emptyset)$ is of type *pts* and therefore is of type *pts'*. So $a \in pts'(x)$ and hence $pts(x) \subseteq pts'(x)$. Similarly, we can show that $b \in dom(pts)$ implies $b \in dom(pts')$ and $pts(b) \subseteq pts'(b)$.
2. The proof is by induction on the structure of type derivation as follows:
(a) The type derivation has the form (ass_p). In this case, $pts' = [pts \mid x{:}V']$ and $(s', h') = ([s \mid x : [\![e]\!]s], h)$. If $[\![e]\!]s \in Addrs$, then $[\![e]\!]s \in V'$ by Lemma 1. Therefore $s'(x) \in Addrs$ implies $s'(x) \in pts'(x)$. We also have that $dom(h') = dom(h) \subseteq dom(pts) \subseteq dom(pts')$ because $(s, h) \models pts$. It is

obvious that for any $x \neq y \in Var$ and $a \in dom(h')$, $s'(y) \in Addrs$ implies $s'(y) \in pts'(y)$ and $h'(a) \in Addrs$ implies $h'(a) \in pts'(a)$. Hence $(s', h') \models pts'$.
(b)The type derivation has the form (lok_p). In this case, $pts' = [pts \mid x : \cup_{a\in V'} pts(a)]$ and $(s', h') = ([s \mid x : h([\![e]\!]s)], h)$. Also we have $[\![e]\!]s \in Addrs \cap dom(h)$ and hence $[\![e]\!]s \in V'$ by Lemma 1. If $h([\![e]\!]s) \in Addrs$, then $h([\![e]\!]s) \in pts([\![e]\!]s)$ because $(s, h) \models pts$. Therefore $s'(x) = h([\![e]\!]s) \in Addrs$ implies $s'(x) = h([\![e]\!]s) \in \cup_{a\in V'} pts(a) = pts'(x)$. We also have that $dom(h') = dom(h) \subseteq dom(pts) \subseteq dom(pts')$ because $(s, h) \models pts$. It is obvious that for any $x\neq y \in Var$ and $a \in dom(h')$, $s'(y) \in Addrs$ implies $s'(y) \in pts'(y)$ and $h;(a) \in Addrs$ implies $h'(a) \in pts'(a)$. Hence $(s', h') \models pts'$
(c) The type derivation has the form (con_p). In this case, $pts' = \cup_{1\leq i\leq v} [pts \mid x : \{a^i_{n,1}\} \mid a^i_{n,1}: V'_1 \mid . . . \mid a^i_{n,n}: V'_n]$ and $(s', h') = ([s \mid x : a^u_{n,1}], [h \mid a^u_{n,1}:[\![e_1]\!]s \mid . . . \mid a^u_{n,n}: [\![e_n]\!]s])$. By Lemma 2, $1 \leq u \leq v$. For every $1 \leq i \leq n$, if $[\![e_i]\!]s \in Addrs$ then $[\![e_i]\!]s \in V_i'$ by Lemma 1. We have $s'(x) = a^u_{1,n} \in\{a^1_{1,n}, . . . , a^v_{1,n} \} \subseteq pts'(x)$. We also have that $dom(h') \subseteq dom(pts')$ because $dom(h) \subseteq dom(pts)$ ($(s, h) \models pts$) and $1 \leq u \leq v$. It is obvious that for any $x \neq y \in Var$ and $a \in dom(h') \setminus \{a^u_{n,1}, . . . , a^u_{n,n} \}$, $s'(y) \in Addrs$ implies $s' (y) \in pts'(y)$ and $h'(a) \in Addrs$ implies $h'(a) = h(a) \in pts(a) \subseteq pts'(a)$. For every $1 \leq i \leq n$, if $h(a^u_{n,i}) \in Addrs$, then $h(a^u_{n,i}) = [\![e_i]\!]s \in Vi' \subseteq pts' (a^u_{n,i})$. Hence $(s', h') \models pts'$.
(d) The type derivation has the form (mut_p). In this case, $pts' = \cup_{a\in V1'} [pts \mid a:V'_2]$ and $(s',h') = (s, [h \mid [\![e_1]\!]s : [\![e_2]\!]s])$. We have $[\![e_1]\!]s \in dom(h) \cap V_1$ and if $[\![e_2]\!]s \in Addrs$ then $[\![e_2]\!]s \in V'_2$ by Lemma 1. If $h'([\![e_1]\!]s) \in Addrs$, then $h'([\![e_1]\!]s) = [\![e_2]\!]s \in V'_2 \subseteq pts' ([\![e_1]\!]s)$ because $[\![e_1]\!]s \in V'_1$. We also have that $dom(h') = dom(h) \subseteq dom(pts) \subseteq dom(pts')$ because $(s, h) \models pts$. It is obvious that for any $y \in Var$ and $a \in dom(h') \setminus V'_1$, $s'(y) \in Addrs$ implies $s' (y) = s(y) \in pts(y) = pts'(y)$ and $h'(a)$ □ $Addrs$ implies $h'(a) = h(a) \in pts(a)$ □ $pts' (a)$. Hence $(s', h') \models pts'$.
The remaining cases are straightforward to check. ■

4. Live stack-heap analysis

In this section, we show how the type system for pointer analysis, presented in the previous section, can be enriched to produce a type system for live stack-heap analysis. In other words, the type system presented in this section is a strict extension of the system presented in the previous section. This reflects the fact that pointer information obtained by previous system are used to improve the precision of the live stack-heap analysis.
The live stack-heap analysis associates with each program point the set of variables and memory locations live (according to Definition 3) at that point. The resulting type system is a generalized one of that presented in [20] for live variables analysis for the while language. Therefore the goal in this section is to utilize results of our type system for pointer analysis and to build o n it a type system that carries live stack-heap analysis. Towards this objective, we augment points-to types to get live stack-heap types defined below (Definition 4).

Definition 3.

- *A* variable (memory location) *is live at a program point if there is a computational path from that program point during which the variable (the memory location's content) gets usefully used before being modified.*
- *A variable (the content of a memory location) is* usefully used
  1- *if it is used in an assignment to a variable or a memory location that is live at the end of the assignment,*
  2- *the guard of an if-statement or a while-statement,*
  3- *an arithmetic expression of a dispose statement, or*
  4- *the left expression of a mutation statement.*

$$\frac{}{x : lsh \cup \{x\} \to lsh} \quad \frac{}{n : lsh \to lsh} \quad \frac{e_1 : lsh \to lsh'' \quad e_2 : lsh'' \to lsh'}{e_1 \oplus e_2 : lsh \to lsh'} \quad \frac{e_1 : lsh \to lsh'' \quad e_2 : lsh'' \to lsh'}{e_1 = e_2 : lsh \to lsh'} \quad \frac{}{skip : (pts, lsh) \to (pts, lsh)}$$

$$\frac{e : lsh \to lsh'}{dispose(e) : (pts, lsh) \to (pts, lsh')}(dis^l) \quad \frac{x \notin lsh \quad x := e : pts \to pts'}{x := e : (pts, lsh) \to (pts', lsh)}(ass^l_1) \quad \frac{\begin{array}{c} e : lsh \to lsh' \backslash \{x\} \\ x \in lsh' \quad x := e : pts \to pts' \end{array}}{x := e : (pts, lsh) \to (pts', lsh')}(ass^l_2)$$

$$\frac{I = \emptyset}{x := cons(e_1, \ldots, e_n) : (pts, lsh') \to (pts', lsh')}(con^l_1) \quad \frac{I = \{x\}}{x := cons(e_1, \ldots, e_n) : (pts, lsh' \backslash \{x\}) \to (pts', lsh')}(con^l_2)$$

$$\frac{\{x\} \neq I \neq \emptyset \qquad \forall 1 \leq i \leq n. e_i : lsh_i \to lsh' \{x\}}{x := cons(e_1, \ldots, e_n) : \left(pts, \cup_{a^j_{n,i} \in I} lsh_i\right) \to (pts', lsh')}(con^l_3) \quad \frac{x \in lsh' \quad e : lsh_i \to lsh' \backslash \{x\} \quad e : pts \to V}{x := [e] : (pts, lsh \cup V') \to (pts', lsh')}(lok^l_2)$$

$$\frac{x \notin lsh}{x := [e] : (pts, lsh) \to (pts', lsh)}(lok^l_1) \quad \frac{V \cap lsh' = \emptyset \quad e_1 : lsh \to lsh'}{[e_1] := e_2 : (pts, lsh) \to (pts', lsh')}(mut^l_1) \quad \frac{\begin{array}{c} e_1 : lsh \to lsh'' \\ V \cap lsh' \neq \emptyset \quad e_2 : lsh'' \to lsh' \end{array}}{[e_1] := e_2 : (pts, lsh) \to (pts', lsh')}(mut^l_2)$$

$$\frac{\begin{array}{c} S_1 : (pts, lsh) \to (pts'', lsh'') \\ S_2 : (pts'', lsh'') \to (pts', lsh') \end{array}}{S_1 ; S_2 : (pts, lsh) \to (pts', lsh')} \quad \frac{b : lsh \to lsh'' \quad S_t, S_f : (pts, lsh'') \to (pts', lsh')}{if\ b\ then\ S_t\ else\ S_f : (pts, lsh) \to (pts', lsh')} \quad \frac{b : lsh \to lsh' \quad S_t : (pts, lsh') \to (pts, lsh)}{while\ b\ do\ S_t : (pts, lsh) \to (pts, lsh')}$$

$$\frac{(pts'_1, lsh'_1) \leq (pts_1, lsh_1) \quad S : (pts_1, lsh_1) \to (pts_2, lsh_2) \quad (pts_2, lsh_2) \leq (pts'_2, lsh'_2)}{S : (pts'_1, lsh'_1) \to (pts'_2, lsh'_2)}$$

Fig.5. Inference rules of a type system for live stack-heap analysis.

Definition 4.
The set of live stack-heap types (live types in short) is denoted by lsh and equal to pts × $2^{Var \cup Addrs}$. *The second component of a live type is termed a live-component. The subtyping relation* ≤ *is defined as:*

$(pts, lsh) \leq (pts', lsh')$ *iff* $(pts \leq pts'$ *and* $lsh \supseteq lsh')$.

The judgment of an expression has the form $e : lsh \to lsh'$ and the intuition is that *lsh* is *lsh'* plus variables occurring free in *e*. The judgment of a statement *S* has the form $S : (pts, lsh) \to (pts', lsh')$ and it is meant to guarantee that if *lsh'* contains variables and memory locations live at the post-state of an execution of *S*, then *lsh* contains variables and memory locations live at the pre-state of this execution. This is formalized in Theorem 2 and consents with the fact that live stack-heap analysis is a backward analysis. This also gives an insight into the definition of $(s, h) \models lsh$ in Definition 6.
Suppose that we have a live-component *lsh'* and the result of a pointer analysis for a statement *S* (in the form $S : pts \to pts'$). The live stack-heap analysis is achieved for *S* via a pre-type derivation that calculates a set *lsh* such that $S: (pts, lsh) \to (pts', lsh')$. It is natural to let *lsh'* be the set of variables that we have interest in their values at the end of executing *S*.
The inference rules for our type system for live stack-heap analysis are presented in Figure 5. The inference rules for Boolean expressions other than $e_1 = e_2$ are similar to the inference rule for $e_1 = e_2$. In rules for allocation, we suppose that $x{:=}cons(e_1, \ldots, e_n) : pts \to pts'$, $v = min\{t \mid \{a^t_{n,1}, \ldots, a^t_{n,n}\} \cap dom(pts) = \emptyset\}$, and $I = \{x, a^i_{n,1}, \ldots, a^i_{n,n} \mid 1 \leq i \leq v\} \cap lsh'$. In rules for look-up, we assume that $x{:=} [e] : pts \to pts'$. For the mutation statement, we suppose that $[e_1] := e_2 : pts \to pts'$ and $e_1 : pts \to V$.
The set of variables and memory locations modified by the statement $x{:=} cons(e_1, \ldots, e_n)$ is contained in $\{x, a^1_{n,1}, \ldots, a^1_{n,n}, a^2_{n,1}, \ldots, a^2_{n,n}, \ldots, a^v_{n,1}, \ldots, a^v_{n,n}\}$. We have three cases concerning which elements of this set are in *lsh'* (possibly live after executing the statement); none, only *x*, or at least one address. For the first case when all modified elements are necessary dead after executing the statement, the rule ($cons^l_1$) equalizes live-components of the pre and post types. For the second case, the rule ($cons^l_2$) lets $lsh' \backslash \{x\}$ (as the assignment to *x* kills it) be the live-component of the pre-type. For the third case treated by the rule ($cons^l_3$) the live-component of the pre-type is constructed via augmenting $lsh' \backslash \{x\}$ with variables occurring free in every expression assigned to a location possibly live after execution.
For the look-up statement, the rule (lok^l_2) deals with the case that *x* is possibly live after executing the statement. In this case, the pointer information is used to calculate the set of addresses *V'*. Then, to form the live-component of the pre-type, the set $lsh' \backslash \{x\}$ is augmented to include *V'* and variables occurring free in *e*.
For the mutation statement, the pointer information is used to find the set *V'* containing any address that the expression e_1 evaluates to in a state of type *pts*. The type system has two rules dealing with the two possible cases; whether or not *V'* has an empty intersection with *lsh'*. The rule (mut^l_2) takes care of the case of nonempty intersection. In this rule the live-component of the pre-type is constructed by adding variables occurring free in e_1 and e_2 to *lsh'*. We note that in this case it is not obvious which location will be modified (and hence gets killed) but it is clear that this location is possibly live at the end of mutation. Therefore nothing is removed from *lsh'*; instead variables occurring

free in e_1 and e_2 are added.

Now we introduce necessary definitions and results towards proving the soundness of our type system for live stack-heap analysis.

Definition 5. *1.* $(s, h) \models_{lsh} pts \Leftrightarrow dom(h) \subseteq dom(pts), \forall x \in Var \cap lsh\ (s(x) \in Addrs \Rightarrow s(x) \in pts(x))$, *and* $\forall a \in dom(h) \cap lsh\ (h(a) \in Addrs \Rightarrow h(a) \in pts(a))$.
2. $(s, h) \sim_{lsh} (s', h') \Leftrightarrow \forall x \in lsh \cap Var.\ s(x) = s'(x)$, *and* $\forall a \in lsh \cap dom(h) \cap dom(h').\ h(a) = h'(a)$.
3. $(s, h) \sim_{(pts,lsh)} (s', h') \Leftrightarrow dom(h) = dom(h'), (s, h) \models_{lsh} pts, (s', h') \models_{lsh} pts$, *and* $(s, h) \sim_{lsh} (s', h')$.

Definition 6. *The expression* $(s, h) \models lsh$ *denotes the case when there is a variable or a memory location that is live at that state (computational point) and is not included in lsh. A state* (s,h) *has type* (pts,lsh)*, denoted by* $(s,h) \models (pts, lsh)$*, if* $(s, h) \models_{lsh} pts$ *and* $(s, h) \models lsh$.

The following lemma is proved by structure induction on *e* and *b*.

Lemma 3. *Suppose that* (s,h) *and* (s',h') *are states and lsh and* $lsh' \in 2^{Var \cup Addrs}$*. Then*
1. If $lsh \supseteq lsh'$ *and* $(s,h) \sim_{lsh} (s',h')$*, then* $(s,h) \sim_{lsh'} (s',h')$.
2. If $e : lsh \rightarrow lsh'$ *and* $(s, h) \sim_{lsh} (s',h')$*, then* $[\![e]\!]s = [\![e]\!]s'$ *and* $(s, h) \sim_{lsh'} (s', h')$.
3. If $b : lsh \rightarrow lsh'$ *and* $(s,h) \sim_{lsh} (s',h')$*, then* $[\![b]\!]s = [\![b]\!]s'$ *and* $(s,h) \sim_{lsh'} (s', h')$.

The following lemma follows from Lemma 1.

Lemma 4. *Suppose that* $(s, h) \models_{lsh} pts$, $FV(e) \subseteq lsh$, *and* $e : pts \rightarrow V$*. Then* $[\![e]\!]s \in Addrs \Rightarrow [\![e]\!]s \in V$.
Proof. Consider the state (s',h'), where $s' = \lambda x.$ if $x \in FV(e)$ then $s(x)$ else 0 and $h' = \emptyset$. It is not hard to see that $[\![e]\!]s = [\![e]\!]s'$ and $(s',h') \models pts$. Now by Lemma 1, $[\![e]\!]s' \in Addrs$ implies $[\![e]\!]s' \in V$ which completes the proof. ■

$$\frac{}{skip : (pts, lsh) \rightarrow (pts, lsh) \hookrightarrow skip} \qquad \frac{x \notin lsh \quad x := e : (pts, lsh) \rightarrow (pts', lsh)}{x := e : (pts, lsh) \rightarrow (pts', lsh) \hookrightarrow skip}(ass_1^d) \qquad \frac{x \in lsh' \quad x := e : (pts, lsh) \rightarrow (pts', lsh')}{x := e : (pts, lsh) \rightarrow (pts', lsh') \hookrightarrow x := e}(ass_2^d)$$

$$\frac{I = \emptyset}{\begin{array}{c} x := cons(e_1, \ldots, e_n) : (pts, lsh) \rightarrow (pts', lsh') \\ \hookrightarrow x := cons(0_1, \ldots, 0_n) \end{array}}(con_1^d) \qquad \frac{I \neq \emptyset}{\begin{array}{c} x := cons(e_1, \ldots, e_n) : (pts, lsh) \rightarrow (pts', lsh') \\ \hookrightarrow x := cons(e_1, \ldots, e_n) \end{array}}(con_2^d)$$

$$\frac{x \notin lsh \quad x := [e] : (pts, lsh) \rightarrow (pts', lsh)}{x := [e] : (pts, lsh) \rightarrow (pts', lsh) \hookrightarrow skip}(lok_1^d) \qquad \frac{x \in lsh' \quad x := [e] : (pts, lsh) \rightarrow (pts', lsh')}{x := [e] : (pts, lsh) \rightarrow (pts', lsh') \hookrightarrow x := [e]}(lok_2^d)$$

$$\frac{}{dispose(e) : (pts, lsh) \rightarrow (pts, lsh') \hookrightarrow dispose(e)}\ (dis^d) \qquad \frac{\begin{array}{c} S_1 : (pts, lsh) \rightarrow (pts'', lsh'') \hookrightarrow S'_1 \\ S_2 : (pts'', lsh'') \rightarrow (pts', lsh') \hookrightarrow S'_2 \end{array}}{S_1; S_2 : (pts, lsh) \rightarrow (pts', lsh') \hookrightarrow S'_1; S'_2}$$

$$\frac{V \cap lsh' = \emptyset}{[e_1] := e_2 : (pts, lsh) \rightarrow (pts', lsh') \hookrightarrow skip}(mut_1^l) \qquad \frac{V \cap lsh' \neq \emptyset}{[e_1] := e_2 : (pts, lsh) \rightarrow (pts', lsh') \hookrightarrow [e_1] := e_2}(mut_2^d)$$

$$\frac{\begin{array}{c} b : lsh \rightarrow lsh'' \quad S_t : (pts, lsh'') \rightarrow (pts', lsh') \hookrightarrow S'_t \\ S_f : (pts, lsh'') \rightarrow (pts', lsh') \hookrightarrow S'_f \end{array}}{\begin{array}{c} if\ b\ then\ S_t\ else\ S_f : (pts, lsh) \rightarrow (pts', lsh') \\ \hookrightarrow if\ b\ then\ S'_t\ else\ S'_f \end{array}} \qquad \frac{b : lsh \rightarrow lsh' \quad S_t : (pts, lsh') \rightarrow (pts, lsh) \hookrightarrow S'_t}{\begin{array}{c} while\ b\ do\ S_t : (pts, lsh) \rightarrow (pts, lsh') \\ \hookrightarrow while\ b\ do\ S'_t \end{array}}$$

$$\frac{(pts'_1, lsh'_1) \leq (pts_1, lsh_1) \quad S : (pts_1, lsh_1) \rightarrow (pts_2, lsh_2) \hookrightarrow S' \quad (pts_2, lsh_2) \leq (pts'_2, lsh'_2)}{S : (pts'_1, lsh'_1) \rightarrow (pts'_2, lsh'_2) \hookrightarrow S'}$$

Fig.6. Inference rules of a type system for dead-code elimination.

Theorem 2. *1.* $(pts, lsh) \leq (pts', lsh') \Rightarrow (\forall (s, h).\ (s,h) \models_{lsh} pts \Rightarrow (s, h) \models_{lsh'} pts')$.
2. Suppose that $S: (pts, lsh) \rightarrow (pts', lsh')$ *and* $S : (s, h) \rightarrow (s', h')$*. Then* $(s, h) \models_{lsh} pts$ *implies* $(s', h') \models_{lsh'} pts'$.
3. Suppose $S: (s,h) \rightarrow (s',h')$ *and* $S : (pts,lsh) \rightarrow (pts', lsh')$*. Then* $(s,h) \models lsh$ *implies* $(s',h') \models lsh'$ *. This guarantees that if the set of variables and memory locations live at the state* (s',h') *is included in lsh', then the set of variables and memory locations live at the state* (s, h) *is included in lsh.*
Proof. 1. Suppose $(s, h) \models_{lsh} pts$. This implies $(s,h) \models_{lsh'} pts$ because $lsh' \subseteq lsh$. The last fact implies $(s, h) \models_{lsh'} pts'$ because $pts \leq pts'$.
2. The proof is by induction on the structure of type derivation as follows:
(a) The type derivation has the form (dis^l). In this case, $lsh = lsh' \cup FV(e)$ and $(s', h') = (s, h](dom(h) \setminus \{[\![e]\!]s\}))$. Therefore $(s,h) \models_{lsh} pts$ implies $(s',h) \models_{lsh} pts'$ which implies $(s', h') \models_{lsh'} pts'$ because $h' \leq h$ and $lsh' \subseteq lsh$.
(b) The type derivation has the form (ass_1^l). In this case, $pts' = [pts \mid x : V']$ and $(s', h') = ([s \mid x : [\![e]\!]s], h)$. Therefore $(s, h) \models_{lsh} pts$ implies $(s', h) \models_{lsh} pts'$ because $x \notin lsh$. Clearly $(s', h) \models_{lsh} pts'$ implies $(s', h') \models_{lsh'} pts'$.

(c) The type derivation has the form (ass_2^l). In this case, $pts' = [pts \mid x{:}V']$, $(s', h') = ([s \mid x : [\![e]\!]s], h)$, and $lsh = (lsh' \setminus \{x\}) \cup FV(e)$. Therefore by Lemma 4 and similarly to Theorem 1 (2.a), we can conclude $(s', h') \models_{lsh'} pts'$.
(d) The type derivation has the form ($con^l{}_1$). In this case,
– $pts' = \cup_{1\leq i\leq v} [pts \mid x{:}\ \{a^i{}_{n,1}\} \mid a^i{}_{n,1} : V'_1 \mid \ldots \mid a^i{}_{n,n} : V'_n]$,
– $(s', h') = ([s \mid x{:}\ a^u{}_{n,1}], [h \mid a^u{}_{n,1}{:}\ [\![e_1]\!]s \mid \ldots \mid a^u{}_{n,n}{:}[\![e_n]\!]s])$,
– $v = min\{t \mid \{a^t{}_{n,1}, \ldots, a^t{}_{n,n}\} \cap dom(pts) = \emptyset\}$, and
– $u = min\{t \mid \{a^t{}_{n,1}, \ldots, a^t{}_{n,n}\} \cap dom(h) = \emptyset\}$.
By Lemma 2, $1 \leq u \leq v$. Because $I = \emptyset$, $(s, h) \models_{lsh'} pts$ implies $(s', h') \models_{lsh'} pts'$.
(e) The type derivation has the form (con_2^1). In this case, (s', h'), pts', u, and v have the same definitions as in the previous case (d). Moreover, $lsh = lsh' \setminus \{x\}$. Because $I = \{x\}$ and $1 \leq u \leq v$, $(s, h) \models_{lsh' \setminus \{x\}} pts$ implies $(s', h') \models_{lsh'} pts'$.
(f) The type derivation has the form (con_3^l). In this case, (s', h'), pts', u, and v have the same definitions as in the case (d). Moreover $lsh = \cup_{ajn,i\in I}\ lsh_i$. Therefore by Lemma 4 and similarly to Theorem 1 (2.c), we can conclude $(s', h') \models_{lsh'} pts'$.
(g) The type derivation has the form (lok_1^l). In this case, $pts' = [pts \mid x : \cup_{a\in V'} pts(a)]$, $(s', h') = ([s \mid x : h([\![e]\!]s)], h))$, and $lsh = lsh'$. Therefore $(s, h) \models_{lsh} pts$ implies $(s', h) \models_{lsh} pts'$ because $x \notin lsh$. Clearly $(s', h) \models_{lsh} pts'$ implies $(s', h') \models_{lsh'} pts'$.
(h) The type derivation has the form (lok_2^l). In this case, $pts' = [pts \mid x : \cup_{a\in V'} pts(a)]$, $(s', h') = ([s \mid x : h([\![e]\!]s)], h)$, and $lsh = (lsh' \setminus \{x\}) \cup FV(e) \cup V'$. Therefore by Lemma 4 and similarly to Theorem 1 (2.b), we can conclude $(s', h') \models_{lsh'} pts'$.
(i) The type derivation has the form (mut_1^l). In this case, $pts' = \cup_{a\in V'1}\ [pts \mid a{:}\ V_2']$, $(s', h') = (s, [h \mid [\![e_1]\!]s{:}\ [\![e_2]\!]s])$, and $lsh = lsh' \cup FV(e_1)$. Clearly, $(s, h) \models_{lsh} pts$ implies $(s, h) \models_{lsh'} pts$. Because $V \cap lsh' = \emptyset$, $(s, h) \models_{lsh'} pts$ implies $(s', h') \models_{lsh'} pts'$.
(j) The type derivation has the form (mut_2^l). In this case, $pts' = \cup_{a\in V'1'}\ [pts \mid a{:}\ V'_2]$, $(s', h') = (s, [h \mid [\![e_1]\!]s{:}\ [\![e_2]\!]s])$, and $lsh = lsh' \cup FV(e_1) \cup FV(e_2)$. Therefore by Lemma 4 and similarly to Theorem 1 (2.d), we can conclude $(s', h') \models_{lsh'} pts'$.
The remaining cases are straightforward to check.
3. The proof again is by induction on the structure of type derivation and it is straightforward. ■

Corollary 1. *Suppose* $S{:}\ (s, h) \rightarrow (s', h')$ *and* $S : (pts, lsh) \rightarrow (pts', lsh')$. *Then* $(s, h) \models (pts, lsh)$ *implies* $(s', h') \models (pts', lsh')$.
Proof. The proof follows from Theorem 2.

Theorem 3. *Suppose that* $S : (pts, lsh) \rightarrow (pts', lsh')$, $S{:}\ (s, h) \rightarrow (s', h')$, *and* $(s, h) \sim_{(pts,lsh)} (s_*, h_*)$. *Then there exists a state* (s'_*, h'_*) *such that* $S{:}\ (s_*, h_*) \rightarrow (s'_*, h'_*)$ *and* $(s', h') \sim_{(pts',lsh')} (s'_*, h'_*)$.
Proof. The proof is by induction on structure of type derivation as follows:

1. The type derivation has the form (dis^l). In this case, $lsh = lsh' \cup FV(e)$ and $(s', h') = (s, h](dom(h) \setminus \{[\![e]\!]s\}))$. Take $(s'_*, h'_*) = (s_*, h_*](dom(h_*) \setminus \{[\![e]\!]s_*\}))$.
2. The type derivation has one of the forms (ass_1^l) and (ass_2^l). In this case, $pts' = [pts \mid x : V']$ and $(s', h') = ([s \mid x : [\![e]\!]s], h)$. Take $(s'_*, h'_*) = ([s_* \mid x : [\![e]\!]s_*], h_*)$.
3. The type derivation has one of the forms (con_1^l), (con_2^l), and (con_3^l). In this case,
– $pts' = \cup_{1\leq i\leq v}[pts \mid x : \{a^i{}_{n,1}\} \mid a^i{}_{n,1} : V'_1 \mid \ldots \mid a^i{}_{n,n} : V'_n]$,
– $(s', h') = ([s \mid x : a^u{}_{n,1}], [h \mid a^u{}_{n,1}{:}\ [\![e_1]\!]s \mid \ldots \mid a^u{}_{n,n} : [\![e_n]\!]s])$,
– $v = min\{t \mid \{a^t{}_{n,1}, \ldots, a^t{}_{n,n}\} \cap dom(pts) = \emptyset\}$, and
– $u = min\{t \mid \{a^t{}_{n,1}, \ldots, a^t{}_{n,n}\} \cap dom(h) = \emptyset\}$.
By Lemma 2, $1 \leq u \leq v$. Take $(s'_*, h'_*) = ([s_* \mid x{:}\ a^u{}_{n,1}], [h_* \mid a^u{}_{n,1}{:}\ [\![e_1]\!]s_* \mid \ldots \mid a^u{}_{n,n} : [\![e_n]\!]s_*])$.
4. The type derivation has the form (lok_2^l) or (lok_1^l). In this case, $pts' = [pts \mid x{:}\ \cup_{a\in V'} pts(a)]$ and $(s', h') = ([s \mid x : h([\![e]\!]s)], h))$. Take $(s'_*, h'_*) = ([s_* \mid x{:}\ h_*([\![e]\!]s_*)], h))$.
5. The type derivation has one of the forms (mut_1^l) and (mut_2^l). In this case, $pts' = \cup_{a\in V'1'}\ [pts \mid a{:}\ V'_2]$ and $(s', h') = (s, [h \mid [\![e_1]\!]s{:}\ [\![e_2]\!]s])$. Take $(s'_*, h'_*) = (s_*, [h_* \mid [\![e_1]\!]s_*{:}\ [\![e_2]\!]s_*])$
The remaining cases are straightforward. ■

5. Dead-code elimination

A type system for dead-code elimination is presented in this section. In a program, statements that have no effect on the content of variables and memory locations live at the end of the program are considered to be dead code. It is the task of the type system presented here to optimize programs via eliminating dead code. If the dead code is faulty (causing the program to abort), then removing it may result in correcting the program.
A typical judgment of our type system takes the form $S{:}\ (pts, lsh) \rightarrow (pts', lsh') \hookrightarrow S'$. And it implies that S' optimizes S towards dead-code elimination (and may be program correction). The derivation of a judgment provides a justification for the optimization process. It is clear from the form of the judgment that the optimization process is built on the type information gathered by our type system for live stack-heap analysis.

Algorithm: *optimize*
- Input: a statement S of the language $while_p$ and a set of variables lsh' that we like to consider live (have interest in their values) at the end of executing S;
- Output: an optimized and may be corrected version S' of S such that the relation between S and S' is as stated in Theorem 4.
- Method:
1. Find pts such that $S{:}\ \perp \rightarrow pts$ in our type system for pointer analysis.
2. Find lsh such that $S{:}\ (\perp, lsh) \rightarrow (pts, lsh')$ in our type system for live stack-heap analysis.
3. Find S' such that $S{:}\ (\perp, lsh) \rightarrow (pts, lsh') \hookrightarrow S'$ in our type system for dead code elimination.

Fig.7. the algorithm *optimize*

The optimization process can be summarized in the algorithm *optimize* out-lined in Figure 7. The first step of the algorithm *optimize* annotates the program points of S with pointer information. This is done via a post type derivation of S for the bottom points-to-type $\perp$ as the pre type. The second step of the algorithm refines the information obtained in the first step via annotating the program points with information about live variables and memory locations. This is done via a pre type derivation of S for the set lsh', the set of variables whose values concerns us at the end of execution, as the post type. Finally the information calculated in the second step is used in the third step to find S' using our type system for dead code elimination.

The inference rules of our type system for dead-code elimination are presented in Figure 6. In the following rules for allocation, the set I has the same definition as in the previous type system and we suppose that $x{:=}cons(e_1, \ldots, e_n) : (pts, lsh) \rightarrow (pts', lsh')$. In rules for mutation, we suppose that $[e_1]{:=} e_2: (pts, lsh) \rightarrow (pts', lsh')$ and $e_1 : pts \rightarrow V$. In rules for dispose, we suppose that $dispose(e) : (pts, lsh) \rightarrow (pts, lsh')$ and $e : pts \rightarrow V$.

We note that the rule (con^d_1) transforms the allocation statement to $x{:=}cons(0, \ldots, 0)$, with n arguments, rather than to *skip*. This is so because the optimization to *skip* would led to the possibility that sequent allocations allocates different arrays in the original and optimized version of the program. And this in turn would complicate the definition of similarity between states (Definition 5) used in proving the equivalence of the original and optimized version of the program. This complication does not worth introducing the *skip* statement. However a simple extra forward traverse of the program can remove all such allocations ($cons(0, \ldots, 0)$) if necessary.

The following theorem guarantees that if the original and optimized programs are executed in similar states and the original one does not abort then:

1. the optimized program does not abort as well, and
2. the optimized program reaches a state similar to that reached by the original program.

Theorem 4. (*Soundness*)
Suppose that $S{:}(pts, lsh) \rightarrow (pts', lsh') \hookrightarrow S'$ *and* $(s, h) \sim_{(lsh,pts)} (s_*, h_*)$. *Then*
1. If S: $(s, h) \rightarrow (s', h')$, *then there exists a state* (s'_*, h'_*) *such that* S': $(s_*, h_*) \rightarrow (s'_*, h'_*)$ *and* $(s', h') \sim_{(pts',lsh')} (s'_*, h'_*)$.
2. If S': $(s_*, h_*) \rightarrow (s'_*, h'_*)$ *and S does not abort at* (s, h), *then there exists a state* (s', h') *such that* S: $(s, h) \rightarrow (s', h')$ *and* $(s', h') \sim_{(pts',lsh')} (s'_*, h'_*)$.

Proof. 1. The proof is by induction on structure of type derivation as follows:
(a) The type derivation has the form (ass_1^d). In this case, $S' = skip$. Take $(s'_*, h'_*) = (s_*, h_*)$.
(b) The type derivation has the form (ass_2^d). In this case, $S' = S$. This case follows from Theorem 3.
(c) The type derivation has the form (con_1^d). In this case, $S' = cons(0_1, \ldots, 0_n)$. Take $(s'_*, h'_*) = ([s_* \mid x{:}\ a^u_{n,1}], [h_* \mid a^u_{n,1} : 0 \mid \ldots \mid a^u_{n,n} : 0])$.
(d) The type derivation has the form (con_2^d). In this case, $S' = S$. This case follows from Theorem 3.
(e) The type derivation has the form (lok_1^d). In this case, $S' = skip$. Take $(s'_*, h'_*) = (s_*, h_*)$.
(f) The type derivation has the form (lok_2^d). In this case, $S' = S$. This case follows from Theorem 3.
(g) The type derivation has form (mut_1^d). In this case, $S' = skip$. Take $(s'_*, h'_*) = (s_*, h_*)$.
(h) The type derivation has the form (mut_2^d). In this case, $S' = S$. This case follows from Theorem 3.
The remaining cases are straightforward.
2. Similar to 1. ■

6. Related work

There are two fields of related work; the first is type systems for data-flow analysis and the second is pointer analysis for sequential languages.

In [9] it is shown that a good deal of static analysis can be done in the type-systems fashion. More precisely, for every analysis in a certain class of data-flow analyses, it is proved in [9] that there exists a type system such that a program checks with a type if and only if the type is a super-type for the set resulting from running the analysis on the program. Later on [20], based on [9], established compositional type systems to carry program and proof optimization based on dead-code and common sub-expression elimination for the while language. These type systems are equipped with a transformation component that does the actual optimization. Our paper builds on and extends results presented in [20] to pointer languages.

The type system in [11] and the flow-logic work in [15] (used in [12, 13] to study security of the coordinated systems) are very similar to [9]. [2] presents constant folding and dead-code elimination via type systems and also introduces relational Hoare logic used to prove correctness of optimizations. Type systems for bidirectional data-flow analyses and their program optimizations are presented in [6]. Earlier, related work (with type systems that are structurally-complex) is [16, 17]. However none of these papers consider pointer programs.

Pointer analysis for $C-$like programs has been actively studied for decades [10, 23, 22, 7, 1, 4, 5, 8, 21, 3]. However none of these papers utilize results of their pointer analyses in data-flow analyses resulting in program optimization and/or correction associated with a justification for the transformation. The objective of pointer analysis has been to obtain a sound analysis only for the sake of program transformation and/or understanding. In [21], the bi-similarity is used to find pointer equivalences in a technique optimizing the performance of inclusion-based pointer analysis. But this does not go farther by showing how the task of optimizing programs is affected. In [3], a conditionally-sound pointer analysis is presented. The results of this analysis are utilized towards checking memory safety, but again the result of the verification is not associated with a justification. [10] studies the existence of an equivalent to shape analysis for purely functional programs, and the"shapes" it discovers. The argument in [10] is that by treating binding environments as dynamically allocated structures, by treating bindings as addresses, and by treating value environments as heaps, the "shape" of higher-order functions can be analyzed.

Acknowledgments

This work was started during the author's sabbatical at Institute of Cybernetics, Estonia in the year 2009. The author is grateful to T. Uustalu for fruitful discussions. This work was partially supported by the EU FP6 IST project MOBIUS.

References

[1] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
[2] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, pages 14–25. ACM Press, 2004.
[3] C. Conway, D. Dams, K. Namjoshi, and C. Barrett. Pointer analysis, conditional soundness, and proving the absence of errors. In *SAS '08: Proceedings of the 15th international symposium on Static Analysis*, pages 62–77, Berlin, Heidelberg, 2008. Springer-Verlag.
[4] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI*, pages 35–46, 2000.
[5] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, pages 242–256, 1994.
[6] M. Frade, A. Saabas, and T. Uustalu. Bidirectional data-flow analyses, type-systematically. In G. Puebla and G. Vidal, editors, *PEPM*, pages 141–150. ACM, 2009.
[7] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In Z. Shao and B. Pierce, editors, *POPL*, pages 226–238. ACM, 2009.
[8] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *PLDI*, pages 24–34, 2001.
[9] P. Laud, T. Uustalu, and V. Vene. Type systems equivalent to data-flow analyses for imperative languages. *Theoretical Computer Science*, 364(3):292 – 310, 2006. Applied Semantics.
[10] M. Might. Shape analysis in the absence of pointers and structure. In G. Barthe and M. Hermenegildo, editors, *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 263–278. Springer, 2010.
[11] M. Naik and J. Palsberg. A type system equivalent to a model checker. *ACM Trans. Program. Lang. Syst.*, 30(5):1–24, 2008.
[12] R. De Nicola, D. Gorla, R. Hansen, F. Nielson, H. Nielson, C. Probst, and R. Pugliese. From flow logic to static type systems for coordination languages. *Sci. Comput. Program.*, 75(6):376 – 397, 2010.
[13] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999. Second printing, 2005.
[14] H. Riis Nielson and F. Nielson. Flow Logic: a multi-paradigmatic approach to static analysis. In T. Mogensen, D. Schmidt, and I. Sudburough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science*, pages 223–244. Springer, 2002.
[15] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. *ACM Trans. Program. Lang. Syst.*, 17(4):576–599, 1995.
[16] J. Palsberg and C. Pavlopoulou. From polyvariant flow information to intersection and union types. *J. Funct. Program.*, 11(3):263–317, 2001.
[17] J. Palsberg and Z. Su, editors. *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673 of *Lecture Notes in Computer Science*. Springer, 2009.
[18] J. Reynolds. Separation logic: A logic for shared mutable data structures. *Logic in Computer Science, Symposium on*, 0:55, 2002.
[19] A. Saabas and T. Uustalu. Program and proof optimizations with type systems. *Journal of Logic and Algebraic Programming*, 77(1-2):131 – 154, 2008. The 16th Nordic Workshop on the Prgramming Theory (NWPT 2006).
[20] L. Simon. Optimizing pointer analysis using bisimilarity. In Palsberg and Su [18], pages 222–237.
[21] M. Sridharan and S. Fink. The complexity of andersen's analysis in practice. In Palsberg and Su [18], pages 205–221.
[22] H. Yu, J. Xue, Wei Huo, X. Feng, and Z. Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In A. Moshovos, J. Steffan, K. Hazelwood, and D. Kaeli, editors, *CGO*, pages 218–229. ACM, 2010.

Mohamed A. El-Zawawy received: PhD in Computer Science from the University of Birmingham in 2007, M.Sc. in Computational Sciences in 2002 from Cairo University, and a BSc. in Computer Science in 1999 from Cairo University. Dr El-Zawawy is an assistant professor of Computer Science at Faculty of Science, Cairo University Since 2007. During the year 2009, Dr. El-Zawawy held the position of an extra-ordinary senior research at the Institute of Cybernetics, Tallinn University of Technology, Estonia. Dr. El-Zawawy worked as a teaching assistant at Cairo University from 1999 to 2003 and latter at Birmingham University from 2003 to 2007. Dr. El-Zawawy is interested in static analysis, shape analysis, type systems, and semantics of programming languages.