New Error-recovery Techniques for Faulty-Calls of Functions

Mohamed A. El-Zawawy[1,2] & Nagwan M. Daoud[2]

[1] College of Computer and Information Sciences, Al-Imam M. I.-S. I. University, Riyadh, Kingdom of Saudi Arabia

[2] Department of Mathematics, Faculty of Science, Cairo University, Giza, Egypt

Correspondence: Mohamed A. El-Zawawy, College of Computer and Information Sciences, Al-Imam M. I.-S. I. University, Riyadh, Kingdom of Saudi Arabia; Department of Mathematics, Faculty of Science, Cairo University, Giza, Egypt. Tel: 20-114-746-3448. E-mail: maelzawawy@cu.edu.eg

Received: February 2, 2012 Accepted: March 1, 2012 Online Published: May 1, 2012

doi:10.5539/cis.v5n3p67 URL: http://dx.doi.org/10.5539/cis.v5n3p67

Abstract

In this paper, we introduce type systems to detect faulty calls of functions in a program. The intended meaning of the faulty call is calling a function with a miss-match to the number of its arguments. We use *error-detecting semantics* that when detects the faulty calls, doesn't proceed to the next state. Type systems are used in the process of analysis and in repairing. The paper presents two type systems: the *safety type system* which checks the safety of a given program and the *repairing type system* which corrects errors. The repairing process is made by replacing the faulty call of function with a correct one. In the repairing process simple interactive input/output statements are used. The interaction (input/output) helps to get the lost parameters by interacting with the user; informing him about the number of lost parameters. The user can then input these parameters.

Keywords: type systems, semantics of programming languages, faulty function calls

1. Introduction

Programmers use functions in many cases as they make their work easier to follow. In most of programming languages, a function's definition consists of function's name, list of parameters and function's body. This is the basic structure of functions. Often the function's name with the list of parameters is called a declaration. The parameters are taken from the user to operate on by the function's body. The function's body is a group of statements that will be executed when the function is called. The value of the parameter is passed to the function during the call. When the programmer calls a function and passes to it a wrong number of parameters, an error occurs (El-Zawawy, 2012; El-Zawawy & Nayel, 2011).

This paper presents a semantic approach to repair this kind of problems, such that if a function is called with passing a wrong number of parameters, the semantics gets stuck and does not proceeds to the next state. This semantics is called *error-detecting semantics*. The analysis is made via type system as it easier to work with. It is also easy to be related to a mathematical proof. We introduced two type systems; the first is to check the safety of the program (*safety type system)* and the second is to repair the errors (*repair type system*) (El-Zawawy, 2011c; El-Zawawy, 2011a; El-Zawawy, 2012; El-Zawawy & Nayel, 2011; El-Zawawy, 2011b).

Example:

```
succ(x)
{x=x+1;}
```

The function succ calculates the successor of an integer x. To find the successor of 5 we need to run succ(5). But if we run succ( ) or succ(5,6), we get error messages. These two calls are called faulty calls. To solve this problem, we use the interaction between the program and the environment to get the missing parameter. If a faulty call exists, the *safety type system* detects it and doesn't continue to the next state. The *repair type system* repairs this error. The error is repaired by using a function *put* to inform the user about the right number of parameters needed. We use *get* to get the new values of the parameters. The new values will be assigned to fresh variables to save the new parameters values. After getting all the parameters, we call the function again with these new values. Note that: We ignore any values passed to the function in the first call "the faulty one" (El-Zawawy, 2011c; El-Zawawy, 2011a). We use type systems in the analysis as they provide useful features like optimization and documentation.

The rest of the paper is organized as follows. Section 2 introduces the syntax of the language and the *error detecting semantics* with the *safety type system*. In Section 3 the *repair type system* is presented with the proof of its soundness. Section 4 concludes the paper and Section 5 reviews some related work.

2. Error Detection

This section presents a small model for faulty function call (with respect to the number of arguments). When a function is called, it must be passed to the write number of parameters, according to its definition. The syntax of the *While* language appears in Figure 1. It is extended with statements for function call and interaction (getting or putting a value). The goal in this section is to capture the error when a faulty call is attempted (El-Zawawy, 2012; El-Zawawy & Nayel, 2011). In Figure 1, the set of statements **Stm**, and the set of the atomic statements **Atm**, are defined over the set of function names **F**. **Var** denotes the set of variables. *a* ranges over **Atm**, *S* over **Stm**, *x* over **Var,** and n over ℤ. We treat the Boolean values *true* and *false* as the numeric values **1** and **0**.

$$
\begin{aligned}
a &::= n \mid x \mid a_1 \text{ op } a_2 \mid x := a \mid skip \mid get \\
S &::= a \mid S_1; S_2 \mid if\ a\ then\ S_t\ else\ S_f \mid while\ a\ do\ S_t \mid f() = \{S\} \mid f(x_1;\ \mathrm{x}_2;\ \ldots;\ xn) = \{S\} \mid put(a) \\
&\quad \mid call\ f(x_1;\ x_2;\ \ldots;\ x_n) \mid call\ f()
\end{aligned}
$$

Figure 1. The programming language

Definition 1:

The semantics state can be defined as a pair (σ, δ) such that,

1- The *store* σ is the function that maps the set of variables **Var** to the integers ℤ , $\sigma \in \boldsymbol{Var} \to \mathbb{Z}$.

2- The *status function* δ is a map from the set of function names **F** to $\{a(n), na\}$, where $a(n)$ means that the function takes n arguments and na denotes taking no arguments.

$$
\frac{}{(\sigma,\delta)\rightarrowtail f(x_1,x_2,\ldots,x_n)=\{S\}\to(\sigma,\delta[f\to a(n)])}\ (\overrightarrow{fDef_1})
\qquad
\frac{}{(\sigma,\delta)\rightarrowtail f(\,)=\{S\}\to(\sigma,\delta[f\to na])}\ (\overrightarrow{fDef_2})
$$

$$
\frac{\delta(f)=a(n)}{(\sigma,\delta)\rightarrowtail call f(x_1,x_2,\ldots,x_n)\to(\sigma,\delta)}\ (\overrightarrow{call_1})
\quad
\frac{\delta(f)=na}{(\sigma,\delta)\rightarrowtail call f(\,)\to(\sigma,\delta)}\ (\overrightarrow{call_2})
\quad
\frac{}{(\sigma,\delta)\rightarrowtail x:=a\to(\sigma[x\to[\![a]\!]\sigma],\delta)}\ (\overrightarrow{ass})
$$

$$
\frac{}{(\sigma,\delta)\rightarrowtail skip\to(\sigma,\delta)}\ (\overrightarrow{skip})
\qquad
\frac{}{(\sigma,\delta)\rightarrowtail get\to(\sigma,\delta)}\ (\overrightarrow{get})
\quad
\frac{(\sigma,\delta)\rightarrowtail S_1\to(\sigma'',\delta'') \qquad (\sigma'',\delta'')\rightarrowtail S_2\to(\sigma',\delta')}{(\sigma,\delta)\rightarrowtail S_1;S_2\to(\sigma',\delta')}\ (\overrightarrow{seq})
$$

$$
\frac{[\![a]\!]\sigma=1 \qquad (\sigma,\delta)\rightarrowtail S_t\to(\sigma',\delta')}{(\sigma,\delta)\rightarrowtail if\ a\ then\ S_t else\ S_f\to(\sigma',\delta')}\ (\overrightarrow{if_1})
\qquad
\frac{[\![a]\!]\sigma=0 \qquad (\sigma,\delta)\rightarrowtail S_f\to(\sigma',\delta')}{(\sigma,\delta)\rightarrowtail if\ a\ then\ S_t else\ S_f\to(\sigma',\delta')}\ (\overrightarrow{if_2})
$$

$$
\frac{[\![a]\!]\sigma = 1 \qquad (\sigma,\delta) \rightarrowtail S_t \to (\sigma'',\delta'') \qquad (\sigma'',\delta'') \rightarrowtail while\ a\ do\ S_t \to (\sigma',\delta')}{(\sigma,\delta) \rightarrowtail while\ a\ do\ S_t \to (\sigma',\delta')}\ (\overrightarrow{whl_1})
$$

$$
\frac{[\![a]\!]\sigma=0}{(\sigma,\delta)\rightarrowtail while\ a\ do\ S_t\to(\sigma,\delta)}\ (\overrightarrow{whl_2})
\qquad
\frac{(\sigma,\delta)\rightarrowtail a\to(\sigma',\delta')}{(\sigma,\delta)\rightarrowtail put(a)\to(\sigma',\delta')}\ (\overrightarrow{put})
\quad
\frac{\delta(f)=a(n)\quad n\neq m}{(\sigma,\delta)\rightarrowtail call f(x_1,x_2,\ldots,x_m)\to|}\ (\overline{call_1})
$$

$$
\frac{\delta(f)=na}{(\sigma,\delta)\rightarrowtail call f(x_1,x_2,\ldots,x_n)\to|}\ (\overline{call_2})
\quad
\frac{(\sigma,\delta)\rightarrowtail S_1\to|}{(\sigma,\delta)\rightarrowtail S_1;S_2\to|}\ (\overline{seq_1})
\qquad
\frac{(\sigma,\delta)\rightarrowtail S_1\to(\sigma'',\delta'') \qquad (\sigma'',\delta'')\rightarrowtail S_2\to|}{(\sigma,\delta)\rightarrowtail S_1;S_2\to|}\ (\overline{seq_2})
$$

$$
\frac{[\![a]\!]\sigma=1 \qquad (\sigma,\delta)\rightarrowtail S_t\to|}{(\sigma,\delta)\rightarrowtail if\ a\ then\ S_t else\ S_f\to|}\ (\overline{if_1})
\quad
\frac{[\![a]\!]\sigma=0 \qquad (\sigma,\delta)\rightarrowtail S_f\to|}{(\,\sigma,\delta)\rightarrowtail if\ a\ then\ S_t else\ S_f\to|}\ (\overline{if_2})
\quad
\frac{[\![a]\!]\sigma=1 \qquad (\sigma,\delta)\rightarrowtail S_t\to|}{(\sigma,\delta)\rightarrowtail while\ a\ do\ S_t\to|}\ (\overline{whl_1})
$$

$$
\frac{[\![a]\!]\sigma = 1 \qquad (\sigma,\delta) \rightarrowtail S_t \to (\sigma'',\delta'') \qquad (\sigma'',\delta'') \rightarrowtail while\ a\ do\ S_t \to |}{(\sigma,\delta) \rightarrowtail while\ a\ do\ S_t \to |}\ (\overline{whl_2})
$$

Figure 2. Error detecting semantics

The rules of this transition system are presented in Figure 2. These rules allow no reach for a final state in case of faulty calls. So they get stuck when they found any faulty calls to the function. The Rules $(\overrightarrow{fDef_1})$ and

$(\overrightarrow{fDef_2})$ define a function *f* whose body is {*S*}. As we can see the function status is updated with *a*(*n*) or *na* according to the function definition. In $(\overrightarrow{fDef_1})$ the function is defined to take *n* parameter, so the status of *f* is *a*(*n*). But in $(\overrightarrow{fDef_2})$ the function is defined to take no parameter so *f* has the status *na*. The rules $(\overrightarrow{call_1})$ and $(\overrightarrow{call_2})$ treat function calls. But we have a condition that the function status must agree with the function being called so the rule could be applied. If that condition is not satisfied as in $(\overline{call_1})$ and $(\overline{call_2})$, the execution gets stocked. The rule $(\overrightarrow{ass})$ updates the store with the new value of *x*. The rules $(\overrightarrow{skip})$ *and* $(\overrightarrow{get})$ do not affect the store or the state. In the rule $(\overrightarrow{get})$, get only gets a numeric value so it doesn't affect the state. The rule for sequence,$(\overrightarrow{seq})$ has the familiar look that we know and it aborts if either of its statements get stuck in $(\overline{seq_1})$ or$(\overline{seq_2})$. Also in$(\overrightarrow{if_1})$ and $(\overrightarrow{if_2})$ every time we evaluate the guard expression, we execute either S_t or S_f according to the value of *a*. The rules of if statement get stuck in if S_t or S_f get stuck in $(\overline{if_1})$ *or* $(\overline{if_2})$, respectively. The rule $(\overrightarrow{whl_1})$ evaluates S_t and the *while* statement if *a* evaluates to 1. The rule $(\overrightarrow{whl_2})$ deals with the case when *(a)* evaluates to zero, meaning that the condition is not satisfied. The *while* statement gets stuck in if S_t gets stuck as in $(\overline{whl_1})$ or if the while statement gets stuck as in $(\overline{whl_2})$.

To check the safety of a program, a type system is used. This type system is defined in Figure 3 and is called *Safety type system*. The analysis here is a forward analysis and the types are the maps $c\colon \mathbf{F} \to \{a(n), na\}$, where *a*(*n*) stands for taking *n* arguments and *na* for taking no arguments. The type decides for every function whether it takes *n* parameters or none. In Figure 3, the type judgment takes the form $S\colon c \to c'$. So if the type was *c* before executing *S* then it will be *c'* after the execution.

The rules $(fDef_1)$ and $(fDef_2)$ deal with the function definition. $(fDef_1)$ is for a function named *f* that takes *n* parameters. After executing this statement, the type map *f* to *a*(*n*) otherwise act like *c*. As for $(fDef_2)$, it acts just the same but with mapping *f* to *na* meaning that *f* takes no parameters at all. The rule $(call_1)$ have a condition that the function *f* must have the type *a*(*m*) so we can detect any errors when passing the wrong number of parameter. It deals with both cases, less or more parameter than we need. But as we can see the *call* statement doesn't change the type, and the same is true for $(call_2)$. The rules $(get), (put), (skip),\ and (ass)$ all act the same, they don't change the type. The rules $(whl), (if), and\ (seq)$ are easy to follow.

Definition 2:

1. The set of types is defined as C=$\{c | c\colon F \to \{a(n), na\} | Fis\ the\ set\ of\ function\ names, n \in \mathbb{N}\}$. The bottom type is denoted by $\perp$.
2. $c \le c'$ if and only if $dom(c) \subseteq dom(c')$ and $\forall f \in dom(c), c(f) = c'(f)$.
3. We say that a state (σ, δ) is of type *c* and write $(\sigma, \delta) \vDash c$ if and only if $\forall f \in F\ \delta(f) = c(f)$.

$$\frac{}{f(x_1,x_2,\dots,x_n)=\{S\}\colon c\to c[f\to a(n)]}\ (fDef_1) \qquad \frac{}{f()=\{S\}\colon c\to c[f\to na]}\ (fDef_2) \qquad \frac{c(f)=a(m)}{call f(x_1,x_2,\dots,x_m)\colon c\to c}\ (call_1)$$

$$\frac{c(f)=na}{call f()\colon c\to c}\ (call_2) \qquad \frac{}{get\colon c\to c}\ (get) \qquad \frac{}{put(a)\colon c\to c}\ (put) \qquad \frac{}{x{:=}a\colon c\to c}\ (ass) \qquad \frac{}{skip\colon c\to c}\ (skip)$$

$$\frac{S_1\colon c\to c'' \quad S_2\colon c''\to c'}{S_1;S_2\colon c\to c'}\ (seq) \qquad \frac{S_t\colon c\to c' \quad S_f\colon c\to c'}{if\ a\ then\ S_t else\ S_f\colon c\to c'}\ (if) \qquad \frac{S_t\colon c\to c}{while\ a\ do\ S_t\colon c\to c'}\ (whl)$$

Figure 3. Safety type system

Theorem 1:

1. If $c \le c'$ then $\forall(\sigma,\delta).\,((\sigma,\delta) \vDash c$ implies $(\sigma,\delta) \vDash c')$.
2. Suppose that $S\colon c \to c'$ and $(\sigma,\delta) \rightarrowtail S \to (\sigma',\delta')$. Then $(\sigma,\delta) \vDash c$ implies $(\sigma',\delta') \vDash c'$.

Proof:

1. Let $c \le c' \Leftrightarrow dom(c) \le dom(c')$ and let$(\sigma,\delta) \vDash c$ then:

   $\forall f \in F.\,(\delta(f) = c(f))$ but $c(f) = c'(f)$ therefore $\forall f \in F.\,(\delta(f) = c'(f))$ then $(\sigma,\delta) \vDash c'$.

2. By structural induction on type derivation as follows:
   a. For the rule$(fDef_1)$, assume that $f(x_1,x_2,\dots,x_n) = \{S\}\colon c \to c'$ and $(\sigma,\delta) \rightarrowtail f(x_1,x_2,\dots,x_n) = \{S\} \to (\sigma',\delta')$. Also assume$(\sigma,\delta) \vDash c$. We show that$(\sigma',\delta') \vDash c'$, i.e.$\forall f \in F.\,\delta'(f) = c'(f)$. Since we have $\sigma = \sigma'$ and $\delta' = \delta[f \to a(n)]$, then $\delta'(f) = a(n)$. Also $c' = c[f \to a(n)]$ implies

$c'(f) = a(n)$. From the assumption we have $\forall f \in F. \delta(f) = c(f)$, and since $\delta'(f) = c'(f)$, we can conclude $\forall f \in F. \delta'(f) = c'(f)$ as required.

b. For the rule $(fDef_2)$, assume that $f(\,) = \{S\}: c \to c'$ and $(\sigma,\delta) \rightarrowtail f(\,) = \{S\} \to (\sigma',\delta')$. Also assume $(\sigma,\delta) \vDash c$. We show that $(\sigma',\delta') \vDash c'$, i.e. $\forall f \in F. \delta'(f) = c'(f)$. Since we have $\sigma = \sigma'$ and $\delta' = \delta[f \to na]$, then $\delta'(f) = na$. Also $c' = c[f \to na]$ implies $c'(f) = na$. From the assumption we have $\forall f \in F. \delta(f) = c(f)$, and since $\delta'(f) = c'(f)$, we can conclude. $\forall f \in F. \delta'(f) = c'(f)$ as required.

c. For the rule (ass), assume that $x \coloneqq a : c \to c'$ and $(\sigma,\delta) \rightarrowtail x \coloneqq a \to (\sigma',\delta')$. Also assume $(\sigma,\delta) \vDash c$. We show that $(\sigma',\delta') \vDash c'$, i.e. $\forall f \in F. \delta'(f) = c'(f)$. We have $\sigma' = \sigma[x \to [\![a]\!]\sigma]$ and $\delta = \delta'$. Also we have $c=c'$. Then $\delta'(f) = \delta(f) = c(f) = c'(f)$. So we can conclude $(\sigma',\delta') \vDash c'$.

d. For the rule (whl), assume that $while\ a\ do\ S_t : c \to c'$ and $(\sigma,\delta) \rightarrowtail while\ a\ do\ S_t \to (\sigma',\delta')$. Also assume $(\sigma,\delta) \vDash c$. We show that $(\sigma',\delta') \vDash c'$, i.e. $\forall f \in F. \delta'(f) = c'(f)$. We have two cases:

Case 1: $\delta = \delta'$ Since $c=c'$, then $\delta'(f) = c'(f)$ as required.

Case 2: $\delta \neq \delta'$ Since all the semantics rules either keep δ without change or add new functions to it's domain as in $(\overrightarrow{fDef_1})$, then $dom(\delta) \subseteq dom(\delta')$. Let $d_f = \delta' - \delta$ and $c' = c \cup d_f$. Then $c \le c'$ and $while\ a\ do\ S_t : c \to c'$. From the assumption we have $\forall f \in F. \delta(f) = c(f)$ and $\forall f \in d_f. \delta'(f) = c'(f)$. Then we can conclude $\forall f \in F. \delta'(f) = c'(f)$ as required.

The remaining rules are straightforward to check.

Now we can say that the type system is sound so there are no type errors according to:

Theorem 2:

Suppose that $S : c \to c'$ in the *safety-type system*. Then if $(\sigma,\delta) \vDash c$ then it is impossible to get $(\sigma,\delta) \rightarrowtail S \to |$.

Proof:

By structural induction on type derivation as follows:

1. For the rule $(fDef_1)$, we assume $f(x_1, x_2, \dots, x_n) = \{S\}: c \to c'$ and $(\sigma,\delta) \vDash c$. We show that it is impossible to have $(\sigma,\delta) \rightarrowtail f(x_1, x_2, \dots, x_n) = \{S\} \to |$, i.e. there exist (σ',δ') such that $(\sigma,\delta) \rightarrowtail f(x_1, x_2, \dots, x_n) = \{S\} \to \underline{(\sigma',\delta')}$. But the only case we have is $(\sigma,\delta) \rightarrowtail f(x_1, x_2, \dots, x_n) = \{S\} \to (\sigma, \delta[f \to a(n)])$ from $(\overrightarrow{fDef_1})$.
2. For the rule $(fDef_2)$, we assume $f(\,) = \{S\}: c \to c'$ and $(\sigma,\delta) \vDash c$. We show that it is impossible to have $(\sigma,\delta) \rightarrowtail f(\,) = \{S\} \to |$, i.e. there exist (σ',δ') such that $(\sigma,\delta) \rightarrowtail f(\,) = \{S\} \to (\sigma',\delta')$. But the only case we have is $(\sigma,\delta) \rightarrowtail f(\,) = \{S\} \to (\sigma, \delta[f \to na])$ from $(\overrightarrow{fDef_2})$.
3. For the rule (ass), we assume $x \coloneqq a: c \to c'$ and $(\sigma,\delta) \vDash c$. We show that it is impossible to have $(\sigma,\delta) \rightarrowtail x \coloneqq a \to |$, i.e. there exist (σ',δ') such that $(\sigma,\delta) \rightarrowtail x \coloneqq a \to (\sigma',\delta')$. But the only case we have is $(\sigma,\delta) \rightarrowtail x \coloneqq a \to (\sigma[x \to [\![a]\!]\sigma], \delta)$ from $(\overrightarrow{ass})$.
4. For the rule (whl), we assume $while\ a\ do\ S_t: c \to c'$ and $(\sigma,\delta) \vDash c$, and assume as a contradiction that $(\sigma,\delta) \rightarrowtail while\ a\ do\ S_t \to |$. The required results from the induction hypothesis on S_t.

The remaining rules are straightforward to check.

3. Repairing Faulty Calls

Now, after we have checked the safety of the program, we may proceed to the repairing process. We are interested in only the errors that result from faulty calls. We have two kinds of functions; one that takes no arguments; and the other takes *n* arguments. The easy part is when we deal with the one that takes no argument. In this case, if by mistake any argument is passed to the function, we can easily ignore these values. The tricky part occurs when we deal with a function that takes parameters. We have two cases to deal with; the first case is when the number of parameters passed is less than the number in the function definition. The second case is when this number is more than the function needed. In both cases, we ignore all the values passed to the function in the faulty call. Then, we use *put* to output the right number of parameters needed, and then use get to get a new values for the parameters from the user. For example, if we define a function named *r* by:

$r(\,) =$

{

```
    $x = get;$
    $if\ x < 10\ then\ put(x)\ else\ put(x - 10);$
}
```

To call this function in the right way, we have to run $r(\,)$. But running for example $r(6)$ is wrong. This can be repaired by ignoring the value passed here which is 6. So, $r(6)$ can be replaced by $r(\,)$ i.e. $r(6) \hookrightarrow r(\,)$.

Now, we have a function say $f(x, y)$ defined by:

```
$f(x,y) =$
{
    $u = x + y;$
    $w = x - y;$
    $put(u);$
    $put(w);$
}
```

To call this function, we may run $f(2,3)$. But if we run $f(\,)$, $f(2)$, or $f(2,4,6)$, we get error messages since the function only needs two parameters. To solve this problem, we do the following:

1. $put(2)$
2. $x_1 = get;\ x_2 = get;$
3. $call\ f(x_1, x_2);$

In line 1, $put(2)$ informs the user that we need only two parameters to the function. In line 2, the user enters the new values and they are stored in x_1 and x_2, respectively. In line 3, the function call is made with the new values and the right number of parameters. For the repair type system, the types are the same as that of the safety type system. But the form of the type judgment will have a transformation component. The judgment will take the form $S: c \to c' \hookrightarrow S'$. This means that *S* can be replaced by *S'*. The rules of this type system appear in Figure 4. This type system is called the *Repair type system* as it replaces the wrong statement whenever it is possible. In Figure 4, the rule $(call_1^r)$ makes the replacement if the number of parameters is not the same as the one that the function takes. The type system detects it, since it is a forward type system. The repairing is done by replacing $call\ f(x_1, x_2, \dots, x_m)$ with $put(n)$ where *n* is the number of parameters needed. After informing the user that we need *n* new values, it proceeds with getting those values via the commands $x_1 = get;\ x_2 = get; \dots; x_n = get;$. Finally, call the function again with the new *n* values. The rule $(call_2^r)$ can be considered as a special case of $(call_1^r)$. We separate them to deal with the problem in a simpler way. The rules (get^r), (put^r), (ass^r), and $(skip^r)$ as well as the rules (whl^r), (if^r), and (seq^r) all are straightforward.

$$\frac{m \neq n}{call\ f(x_1, x_2, \dots, x_m): c[f \to a(n)] \to c \hookrightarrow \begin{cases} put(n); \\ x_1 = get; x_2 = get; \dots, x_n = get; \\ call\ f(x_1, x_2, \dots, x_n) \end{cases}} \quad (call_1^r)$$

$$\frac{}{call\ f(x_1, x_2, \dots, x_m): c[f \to na] \to c \hookrightarrow call\ f(\,)} \quad (call_2^r) \qquad \frac{}{x \coloneqq a: c \to c \hookrightarrow x \coloneqq a} \quad (ass^r)$$

$$\frac{}{get: c \to c \hookrightarrow get} \quad (get^r) \qquad \frac{}{put(a): c \to c \hookrightarrow put(a)} \quad (put^r) \qquad \frac{}{skip: c \to c \hookrightarrow skip} \quad (skip^r)$$

$$\frac{S_1: c \to c'' \hookrightarrow S_1' \qquad S_2: c'' \to c' \hookrightarrow S_2'}{S_1; S_2: c \to c' \hookrightarrow S_1'; S_2'} \quad (seq^r) \qquad \frac{S_t: c \to c' \hookrightarrow S_t' \qquad S_f: c \to c' \hookrightarrow S_f'}{if\ a\ then\ S_t\ else\ S_f: c \to c' \hookrightarrow if\ a\ then\ S_t'\ else\ S_f'} \quad (if^r)$$

$$\frac{S_t: c \to c \hookrightarrow S_t'}{while\ a\ do\ S_t: c \to c \hookrightarrow while\ a\ do\ S_t'} \quad (whl^r)$$

Figure 4. Repair type system

Definition 3:

$(\sigma,\delta) \sim_c (\sigma',\delta') \Leftrightarrow dom(\sigma) \subseteq dom(\sigma'), \forall x \in dom(\sigma).\sigma(x) = \sigma'(x)\ and$

$dom(\delta) = dom(\delta'), \forall f \in dom(\delta).\delta(f) = \delta'(f).$

Theorem 3:

Suppose that $S: c \to c' \hookrightarrow S'$ and $(\sigma,\delta) \sim_c (\sigma_*,\delta_*)$ then:

1. If $(\sigma,\delta) \rightarrowtail S \to (\sigma',\delta')$ then there exists a state (σ'_*,δ'_*) such that $(\sigma_*,\delta_*) \rightarrowtail S' \to (\sigma'_*,\delta'_*)$ and $(\sigma',\delta') \sim_c (\sigma'_*,\delta'_*)$.
2. If $(\sigma_*,\delta_*) \rightarrowtail S' \to (\sigma'_*,\delta'_*)$ and S does not get stuck at (σ,δ), then there exist a state (σ',δ') such that: $(\sigma,\delta) \rightarrowtail S \to (\sigma',\delta')$ and $(\sigma',\delta') \sim_c (\sigma'_*,\delta'_*)$.

Proof:

1. By structural induction on type derivation of repair type system as follows:
   a. For the rule$(call_1^r)$, assume that

$$call\ f(x_1,x_2,\dots,x_m): c[f \to a(n)] \to c \hookrightarrow \begin{cases} put(n); \\ x_1 = get; x_2 = get; \dots, x_n = get; \\ call\ f(x_1,x_2,\dots,x_n) \end{cases}$$

, and$(\sigma,\delta) \sim_c (\sigma_*,\delta_*)$. Also assume $(\sigma,\delta) \rightarrowtail f(x_1,x_2,\dots,x_m) \to (\sigma',\delta')$. We show that there exists a state (σ'_*,δ'_*) such that
$(\sigma_*,\delta_*) \rightarrowtail put(n); x_1 = get; x_2 = get; \dots, x_n = get; call\ f(x_1,x_2,\dots,x_n) \to (\sigma'_*,\delta'_*)$, and$(\sigma',\delta') \sim_c (\sigma'_*,\delta'_*)$. Since $(\sigma_*,\delta_*) \rightarrowtail put(n) \to (\sigma_*,\delta_*)$,
$(\sigma_*,\delta_*) \rightarrowtail x_1 = get \to (\sigma_*[x_1 \to n_1],\delta_*)$, where n_1 is the value we have by get. Also $(\sigma_*[x_1 \to n_1],\delta_*) \rightarrowtail x_2 = get \to (\sigma_*[x_1 \to n_1, x_2 \to n_2],\delta_*)$, … ,
$(\sigma_*[x_1 \to n_1, x_2 \to n_2, \dots, x_{n-1} \to n_{n-1}],\delta_*) \rightarrowtail x_n = get \to$
$(\sigma_*[x_1 \to n_1, x_2 \to n_2, \dots, x_n \to n_n],\delta_*)$.
Let $[x_1 \to n_1, x_2 \to n_2, \dots, x_n \to n_n]$. Then $(\sigma'_*,\delta_*) \rightarrowtail call\ f(x_1,x_2,\dots,x_n) \to (\sigma'_*,\delta_*)$. So g $(\sigma_*,\delta_*) \rightarrowtail put(n); x_1 = get; x_2 = get; \dots, x_n = get; call\ f(x_1,x_2,\dots,x_n) \to (\sigma'_*,\delta_*)$. Since $dom(\sigma) \subseteq dom(\sigma_*)$ and $dom(\sigma_*) \subseteq dom(\sigma'_*)$, $dom(\sigma) \subseteq dom(\sigma'_*)$. Since $(\sigma,\delta) \sim_c (\sigma_*,\delta_*)$, $\forall x \in dom(\sigma).\sigma(x) = \sigma_*(x)$. Since all the variables added to σ_* are all fresh then $\sigma_*(x) = \sigma'_*(x), \forall x \in dom(\sigma)$. The by assumption σ' and σ_* satisfy the required conditions.

   b. For the rule $(call_2^r)$, assume that $call\ f(x_1,x_2,\dots,x_m): c[f \to na] \to c \hookrightarrow call\ f(\,)$, and$(\sigma,\delta) \sim_c (\sigma_*,\delta_*)$. Also assume $(\sigma,\delta) \rightarrowtail f(x_1,x_2,\dots,x_m) \to (\sigma',\delta')$. We show that there exists a state (σ'_*,δ'_*) such that $(\sigma_*,\delta_*) \rightarrowtail call\ f(\,) \to (\sigma'_*,\delta'_*)$, and$(\sigma',\delta') \sim_c (\sigma'_*,\delta'_*)$. Since $(\sigma_*,\delta_*) \rightarrowtail call\ f(\,) \to (\sigma_*,\delta_*)$, $(\sigma,\delta) \sim_c (\sigma_*,\delta_*)$ and $(\sigma'_*,\delta'_*) = (\sigma_*,\delta_*)$, then $(\sigma',\delta') \sim_c (\sigma'_*,\delta'_*)$.

   c. For the rule (ass^r), assume that $x \coloneqq a: c \to c' \hookrightarrow x \coloneqq a$, and$(\sigma,\delta) \sim_c (\sigma_*,\delta_*)$. Also assume $(\sigma,\delta) \rightarrowtail x \coloneqq a \to (\sigma',\delta')$. We show that there exists a state (σ'_*,δ'_*) such that $(\sigma_*,\delta_*) \rightarrowtail x \coloneqq a \to (\sigma'_*,\delta'_*)$, and $(\sigma',\delta') \sim_c (\sigma'_*,\delta'_*)$. Since $(\sigma_*,\delta_*) \rightarrowtail x \coloneqq a \to (\sigma_*[x \to a],\delta_*)$, then $(\sigma'_*,\delta'_*) = (\sigma_*[x \to a],\delta_*)$. But $dom(\sigma) \subseteq dom(\sigma_*)$ and $dom(\sigma_*) \subseteq dom(\sigma'_*)$. Then $dom(\sigma) \subseteq dom(\sigma'_*)$. So $(\sigma',\delta') \sim_c (\sigma'_*,\delta'_*)$.

   d. For the rule (whl^r) , assume that $while\ a\ do\ S_t: c \to c \hookrightarrow while\ a\ do\ S'_t$, and $(\sigma,\delta) \sim_c (\sigma_*,\delta_*)$. Also assume $(\sigma,\delta) \rightarrowtail while\ a\ do\ S_t \to (\sigma',\delta')$. We show that there exists a state (σ'_*,δ'_*) such that$(\sigma_*,\delta_*) \rightarrowtail while\ a\ do\ S'_t \to (\sigma'_*,\delta'_*)$, and $(\sigma',\delta') \sim_c (\sigma'_*,\delta'_*)$. We have two cases:
   Case 1: $[\![a]\!]\sigma = 0$ In this case $(\sigma,\delta) = (\sigma',\delta')$ and $(\sigma_*,\delta_*) = (\sigma'_*,\delta'_*)$ by $(\overrightarrow{whl_2})$. So we can conclude that $(\sigma',\delta') \sim_c (\sigma'_*,\delta'_*)$.
   Case 2: $[\![a]\!]\sigma = 1$ In this case by $(\overrightarrow{whl_1})$ we have $(\sigma,\delta) \rightarrowtail S_t \to (\sigma'',\delta'')$ and$(\sigma'',\delta'') \rightarrowtail while\ a\ do\ S_t \to (\sigma',\delta')$. We also have $(\sigma_*,\delta_*) \rightarrowtail S_t \to (\sigma''_*,\delta''_*)$ and $(\sigma''_*,\delta''_*) \rightarrowtail while\ a\ do\ S_t \to (\sigma'_*,\delta'_*)$. By induction hypothesis we may assume he required for the sub statements. So$(\sigma'',\delta'') \sim_c (\sigma''_*,\delta''_*)$ and hence $(\sigma',\delta') \sim_c (\sigma'_*,\delta'_*)$.

The remaining rules are straightforward to check.

2. By structural induction on type derivation of Repair type system:
   a. For the rule$(call_1^r)$, assume that

$$call\, f(x_1, x_2, \dots, x_m): c[f \to a(n)] \to c \hookrightarrow \begin{cases} put(n); \\ x_1 = get; x_2 = get; \dots, x_n = get; \\ call\, f(x_1, x_2, \dots, x_n) \end{cases}$$

   Also assume $(\sigma, \delta) \sim_c (\sigma_*, \delta_*)$ and $(\sigma_*, \delta_*) \rightarrowtail put(n); x_1 = get; x_2 = get; \dots, x_n = get; call\, f(x_1, x_2, \dots, x_n) \to (\sigma'_*, \delta'_*)$. From Theorem 3.1, $\sigma'_* = \sigma_*[x_1 \to n_1, x_2 \to n_2, \dots, x_n \to n_n]$ and $\delta_* = \delta'_*$. From semantics rules we have $(\sigma, \delta) \rightarrowtail call\, f(x_1, x_2, \dots, x_m) \to (\sigma', \delta') = (\sigma, \delta)$. We show that there exists (σ'_*, δ'_*) $(\sigma_*, \delta_*) \rightarrowtail put(n); x_1 = get; x_2 = get; \dots, x_n = get; call\, f(x_1, x_2, \dots, x_n) \to (\sigma'_*, \delta'_*)$, and$(\sigma', \delta') \sim_c (\sigma'_*, \delta'_*)$. Since$(\sigma', \delta') = (\sigma, \delta) \sim_c (\sigma_*, \delta_*)$, and $(\sigma_*) \sqsubseteq dom(\sigma'_*)$, we get $(\sigma', \delta') \sim_c (\sigma'_*, \delta'_*)$.
   b. For the rule $(call_2^r)$, we assume that $call\, f(x_1, x_2, \dots, x_m): c[f \to na] \to c \hookrightarrow call\, f(\,)$, and $(\sigma, \delta) \sim_c (\sigma_*, \delta_*)$. Also assume$(\sigma_*, \delta_*) \rightarrowtail call\, f() \to (\sigma'_*, \delta'_*)$. We show that there exists a state (σ', δ') such that $(\sigma, \delta) \rightarrowtail call\, f(x_1, x_2, \dots, x_m) \to (\sigma', \delta')$, and $(\sigma', \delta') \sim_c (\sigma'_*, \delta'_*)$. Since $(\sigma, \delta) \rightarrowtail call\, f(x_1, x_2, \dots, x_m) \to (\sigma', \delta') = (\sigma, \delta)$, and (σ_*, δ_*)=(σ'_*, δ'_*). Then we can conclude that $(\sigma', \delta') = (\sigma, \delta) \sim_c (\sigma_*, \delta_*) = (\sigma'_*, \delta'_*)$.
   c. For the rule(ass^r), we assume that $x := a: c \to c' \hookrightarrow x := a$, and $(\sigma, \delta) \sim_c (\sigma_*, \delta_*)$. Also we assume$(\sigma_*, \delta_*) \rightarrowtail x := a \to (\sigma'_*, \delta'_*)$. We show that there exists a state (σ', δ') such that $(\sigma, \delta) \rightarrowtail x := a \to (\sigma', \delta')$ and $(\sigma', \delta') \sim_c (\sigma'_*, \delta'_*)$. Let $(\sigma, \delta) \rightarrowtail x := a \to (\sigma', \delta')$. Hence $(\sigma', \delta') = (\sigma'[x \to a], \delta')$ and $(\sigma'_*, \delta'_*) = (\sigma'_*[x \to a], \delta'_*)$. To show that $(\sigma', \delta') \sim_c (\sigma'_*, \delta'_*)$, it is enough to show that, $dom(\sigma') \subseteq dom(\sigma'_*)$ and $\forall x \in dom(\sigma'). \sigma'(x) = \sigma'_*(x)$. Since $dom(\sigma') = dom(\sigma) \cup \{x\}$ and $dom(\sigma'_*) = dom(\sigma_*) \cup \{x\}$. But$dom(\sigma) \subseteq dom(\sigma_*)$, then $dom(\sigma) \cup \{x\} \subseteq dom(\sigma_*) \cup \{x\}$. Also we have $\forall x \in dom(\sigma). \sigma\,(x) = \sigma_*(x)$ and $\sigma\,'(x) = \sigma'_*(x)$. So$\forall x \in dom(\sigma'). \sigma'\,(x) = \sigma'_*(x)$. Then we get $(\sigma', \delta') \sim_c (\sigma'_*, \delta'_*)$ as required.
   d. For the rule (whl^r), assume that $while\ a\ do\ S_t: c \to c \hookrightarrow while\ a\ do\ S'_t$, and $(\sigma, \delta) \sim_c (\sigma_*, \delta_*)$. Also assume $(\sigma_*, \delta_*) \rightarrowtail while\ a\ do\ S'_t \to (\sigma'_*, \delta'_*)$. Let $(\sigma, \delta) \rightarrowtail while\ a\ do\ S_t \to (\sigma', \delta')$. We have two cases:
   Case 1: $[\![a]\!]\sigma = 0$ In this case $(\sigma_*, \delta_*) = (\sigma'_*, \delta'_*)$ and $(\sigma, \delta) = (\sigma', \delta')$ by $(\overrightarrow{whl_2})$. Then we can conclude that $(\sigma', \delta') \sim_c (\sigma'_*, \delta'_*)$.
   Case 2: $[\![a]\!]\sigma = 1$ In this case by $(\overrightarrow{whl_1})$ we have $(\sigma_*, \delta_*) \rightarrowtail S'_t \to (\sigma''_*, \delta''_*)$ and $(\sigma''_*, \delta''_*) \rightarrowtail while\ a\ do\ S'_t \to (\sigma'_*, \delta'_*)$. We also have $(\sigma, \delta) \rightarrowtail S_t \to (\sigma'', \delta'')$ and $(\sigma'', \delta'') \rightarrowtail while\ a\ do\ S_t \to (\sigma', \delta')$. As induction hypothesis we may assume the required for the sub statements. So $(\sigma''_*, \delta''_*) \sim_c (\sigma'', \delta'')$, then $(\sigma', \delta') \sim_c (\sigma'_*, \delta'_*)$.

The remaining rules are straightforward to check.

4. Conclusion

This paper presents a type system for detecting faulty calls of functions and a type system for repairing these errors by interacting with the user through input/output statements. The errors that we treat are the function calls that have wrong number of arguments (more or less) than the function needs. To repair these errors, we ignore all the parameter values and ask the user to re-input them and when we get these values we call the function again with them. The soundness of both type systems is proved via induction on the type rules.

5. Related and Future Work

In (Fischer, Saabas, & Uustalu, 2009) two problems related to file access errors and queues are approached. The file problem happens when opening a file that is already opened or closing a file that is already closed. Also reading from a closed file is another source of abortion. The work in (Fischer, Saabas, & Uustalu, 2009) ignores end of file errors for simplicity. For the concept of queue, this paper treats situations of adding values to full queues (over/under flow).

Type systems are used intensively (El-Zawawy, 2011c; El-Zawawy, 2011a; El-Zawawy, 2012; El-Zawawy & Nayel, 2011; El-Zawawy, 2011b) in program analysis. In (El-Zawawy, 2011c), the problem of dead-code elimination was approached with type systems. This optimization is based on flow-sensitive pointer analysis. The final type system is an enrichment of that pointer analysis. The work in (El-Zawawy, 2011c) deals with the memory safety of multi-threaded programs. This paper presents a type system for pointer analysis of multi-threaded programs. The memory-safety type system is a flow sensitive which invokes anther flow insensitive type system (for pointer analysis). Basic constructs treated by these flow-insensitive type systems (for pointer analysis) are parallel programming constructs.

In Benton's work (Benton, 2004), static analysis is presented via elementary logic; type systems are used in the analysis to generalize it with Hoare logic which is used in the optimization of while programs. A data-structure repair-system is presented in (Demsky & Rinard, 2006). This system generates a repair algorithm for the input data structures that have the form of relational model. This algorithm detects and repairs the errors during the program execution. In (Denney & Fischer, 2003), the static safety of the program is proved to guarantee a dynamic safety. This is done using Hoare reference rules to check the safety policies. In this work, the soundness and completeness of safety policies on memory access and memory read and write are proved. The assertion violation is treated in (Elkarablieh, Garcia, Suen, & Khurshid, 2007) with a repairing algorithm that finds the errors and repairs them without terminating the program. This algorithm repairs complex structures with the ability to recover from future errors.

In (Frade, Saabas, & Uustalu, 2007) live variables analysis is treated as a classical data flow analysis, and shown to be certified on a variety of levels; completely analogous to certification of program safety or functional correctness. This paper shows that the type systems should be seen as foundational Hoare logic to study the same abstract semantics. The programs studied in (Frade, Saabas, & Uustalu, 2007) contain a provision for error-detection and error-recovery presented in (Horning, Lauer, Melliar-Smith, & Randell, 1974). The work in (Knoop, Rüthing, & Stefen, 1994) presents a version of the algorithm of lazy code motion that works on a flow graph. The algorithm is a life time optimal with a unidirectional analysis.

The work (Paleri, Srikant, & Shankar, 2003) presents a simple for partial redundancy elimination which is built up on the two concepts of partial availability and safe partial anticipability. This algorithm works on flow graph with four unidirectional analyses. This algorithm also integrates the notations of safety from the definition of partial availability and from the definition of safe partial anticipability. A program optimization approach is presented in (Saabas & Uustalu, 2008a). This work presents compositional type systems with a transformation component. Dead-code elimination and common sub-expression elimination are studied in (Saabas & Uustalu, 2008a).

The work in (Saabas & Uustalu, 2007) presents a type system for optimizing stack-based code. In this work, dead store instructions and load-pop pairs are treated with no need for assumption about input code. An algorithm for soundness proofs and strongest analysis is presented in a simple way in (Saabas & Uustalu, 2007). Optimizations of partial redundancy elimination are studied in (Saabas & Uustalu, 2008b). More precisely (Saabas & Uustalu, 2008b) optimizes the Hoare logic proofs of the given program with the help of a type derivation representation of the result of the underlying data flow analyses.

Mathematical domains and maps between domains can be used to mathematically represent programs and data structures. This representation is called denotation semantics of programs (Cazorla, Cuartero, Ruiz, & Pelayo, 2000; Guo, 2001; Schwartz, 1979). One of our directions for future research is to translate concepts of function repair to the side of denotation semantics (El-Zawawy & Jung, 2006; El-Zawawy, 2007). Doing so provides a good tool to mathematically study in deep function repair. Then obtained results can be translated back to the side of programs and data structures.

References

Benton, N. (2004). Simple relational correctness proofs for static analyses and program transformations. In *Jones N., Leroy X., editors, POPL, ACM*, 14-25.

Cazorla, D., Cuartero, F., Ruiz, V., & Pelayo, F. (2000). A denotational model for probabilistic and nondeterministic processes. In *ICDCS Workshop on Distributed System Validation and Verification*, 41-48.

Demsky, B., & Rinard, M. (2006). Goal-directed reasoning for specification-based data structure repair. *IEEE Trans. Software Eng., 32*(12), 931-951. http://dx.doi.org/10.1109/TSE.2006.122

Denney, E., & Fischer, B. (2003). Correctness of source-level safety policies. *In Araki, K., Gnesi, S., & Mandrioli, D., editors, FME, volume 2805 of Lecture Notes in Computer Science, Springer*, 894-913.

El-Zawawy, M. A. (2007). *Semantic spaces in Priestley form*. PhD thesis, University of Birmingham, UK.

El-Zawawy, M. A. (2011a). Flow sensitive-insensitive pointer analysis based memory safety for multithreaded programs. *In Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., & Apduhan, B., editors, ICCSA (5), volume 6786 of Lecture Notes in Computer Science, Springer,* 355-369.

El-Zawawy, M. A. (2011b). Probabilistic pointer analysis for multithreaded programs. *ScienceAsia*, *37*(4).

El-Zawawy, M. A. (2011c). Program optimization based pointer analysis and live stack-heap analysis. *International Journal of Computer Science Issues*, *8*(2).

El-Zawawy, M. A. (2012). Dead code elimination based pointer analysis for multithreaded programs. *Journal of the Egyptian Mathematical Society*. http://dx.doi.org/10.1016/j.joems.2011.12.011

El-Zawawy, M. A., & Jung, A. (2006). Priestley duality for strong proximity lattices. *Electr. Notes Theor. Comput. Sci*., 158, 199-217. http://dx.doi.org/10.1016/j.entcs.2006.04.011

El-Zawawy, M. A., & Nayel, H. (2011). Partial redundancy elimination for multi-threaded programs. *IJCSNS International Journal of Computer Science and Network Security, 11*(10).

Elkarablieh, B., Garcia, I., Suen, Y., & Khurshid, S. (2007). Assertion-based repair of complex data structures. *In Stirewalt, R., Egyed, A., & Fischer, B., editors, ASE, ACM*, 64-73.

Fischer, B., Saabas, A., & Uustalu, T. (2009). Program repair as sound optimization of broken programs. *In Chin, W., & Qin, S., editors, TASE, IEEE Computer Society,* 165-173.

Frade, M., Saabas, A., & Uustalu, T. (2007). Foundational certification of data-flow analyses. *In TASE, IEEE Computer Society*, 107-116.

Guo, M. (2001). Denotational semantics of an hpf-like data-parallel language model. *Parallel Processing Letters*, *11*(2/3), 363-374. http://dx.doi.org/10.1016/S0129-6264(01)00065-8

Horning, J., Lauer, H., Melliar-Smith, P., & Randell, B. (1974). A program structure for error detection and recovery. *In Gelenbe, E., & Kaisr, C., editors, Symposium on Operating Systems, volume 16 of Lecture Notes in Computer Science, Springer*, 171-187. http://dx.doi.org/10.1007/BFb0029359

Knoop, J., Rüthing, O., & Stefen, B. (1994). Optimal code motion: Theory and practice. *ACM Trans. Program. Lang. Syst., 16*(4), 1117-1155. http://dx.doi.org/10.1145/183432.183443

Paleri, V., Srikant, Y., & Shankar, P. (2003). Partial redundancy elimination: a simple, pragmatic, and provably correct algorithm. *Sci. Comput. Program., 48*(1), 1-20. http://dx.doi.org/10.1016/S0167-6423(02)00083-7

Saabas, A., & Uustalu, T. (2007). Type systems for optimizing stack-based code. *Electr. Notes Theor. Comput. Sci.*, *190*(1), *103-119*. http://dx.doi.org/10.1016/j.entcs.2007.02.063

Saabas, A., & Uustalu, T. (2008a). Program and proof optimizations with type systems. *J. Log. Algebr. Program., 77*(1-2), 131-154. http://dx.doi.org/10.1016/j.jlap.2008.05.007

Saabas, A., & Uustalu, T. (2008b). Proof optimization for partial redundancy elimination. *In PEPM*, 91-101.

Schwartz, J. (1979). Denotational semantics of parallelism. *In Kahn G., editor, Semantics of Concurrent Computation, volume 70 of Lecture Notes in Computer Science, Springer,* 191-202.