New Error-recovery Techniques for Faulty-Calls of Functions

Mohamed A. El-Zawawy1,2 & Nagwan M. Daoud2

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

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 mismatch 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 proceed 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 repairing 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 \( \mathbb{Z} \). We treat the Boolean values \( true \) and \( false \) as the numeric values \( 1 \) and \( 0 \).

\[
\begin{align*}
S & \::= a \mid S_1 \cdot S_2 \mid \text{if} a \text{ then } S_1 \text{ else } S_2 \mid \text{while } a \text{ do } S_1 \\
& \quad \mid \text{call } f(x_1; x_2; \ldots; x_n) \mid \text{call } f()
\end{align*}
\]

Figure 1. The programming language

Definition 1:
The semantics state can be defined as a pair \((\sigma, \delta)\) such that,

1. The store \( \sigma \) is the function that maps the set of variables \( \text{Var} \) to the integers \( \mathbb{Z} \), \( \sigma \in \text{Var} \rightarrow \mathbb{Z} \).
2. The status function \( \delta \) 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.

![Diagram](image-url)

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 \((fDef^1)\) and \((fDef^2)\)
The rule for sequence, \((\text{seq})\) has the familiar look that we know and it aborts if either of its statements get stuck in \((\text{seq}_1)\) or \((\text{seq}_2)\). Also in \((\text{if}_1)\) and \((\text{if}_2)\) every time we evaluate the guard expression, we execute either \(S_r\) or \(S_l\) according to the value of \(f\). The rules of if statement get stuck in if \(S_r\) or \(S_l\) get stuck in \((\text{if}_1)\) or \((\text{if}_2)\), respectively. The rule \((\text{while})_1\) evaluates \(S_r\) and the while statement if \(a\) evaluates to 1. The rule \((\text{while})_2\) deals with the case when \((a)\) evaluates to zero, meaning that the while statement gets stuck in if \(S_r\) gets stuck as in \((\text{while})_1\) or if the while statement gets stuck as in \((\text{while})_2\).

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 \(c : F \rightarrow \{a(n), na\}\), 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 \(S : c \rightarrow c'\). So if the type was \(c\) before executing \(S\) then it will be \(c'\) after the execution.

The rules \((\text{fDef}_1)\) and \((\text{fDef}_2)\) deal with the function definition. \((\text{fDef}_1)\) 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 \((\text{fDef}_2)\), it acts just the same but with mapping \(c\) to \(na\) meaning that \(f\) takes no parameters at all. The rule \((\text{call})_1\) 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 \((\text{call})_2\). The rules \((\text{get}), (\text{put}), (\text{skip}), \text{and (ass)}\) all act the same, they don’t change the type. The rules \((\text{whl}), (\text{if}), \text{and (seq)}\) are easy to follow.

**Definition 2:**

1. The set of types is defined as \(C = \{c : F \rightarrow \{a(n), na\}\} | F\) is the set of function names, \(n \in \mathbb{N}\}\). The bottom type is denoted by \(\bot\).
2. \(c \leq c'\) if and only if \(\text{dom}(c) \subseteq \text{dom}(c')\) and \(\forall f \in \text{dom}(c), c(f) = c'(f)\).
3. We say that a state \((\sigma, \delta)\) is of type \(c\) and write \((\sigma, \delta) \models c\) if and only if \(\forall f \in F, \delta(f) = c(f)\).

**Theorem 1:**

1. If \(c \leq c'\) then \(\forall (\sigma, \delta), ((\sigma, \delta) \models c)\) implies \((\sigma, \delta) \models c'\).
2. Suppose that \(S : c \rightarrow c'\) and \((\sigma, \delta) \models S \rightarrow (\sigma', \delta')\). Then \((\sigma, \delta) \models c\) implies \((\sigma', \delta') \models c'\).

**Proof:**

1. \(c \leq c' \Leftrightarrow \text{dom}(c) \leq \text{dom}(c')\) and let \((\sigma, \delta) \models c\) then:
   \(\forall f \in F, (\delta(f) = c(f))\) but \(c(f) = c'(f)\) therefore \(\forall f \in F, (\delta(f) = c'(f))\) then \((\sigma, \delta) \models c'\).
2. By structural induction on type derivation as follows:
   a. For the rule \((\text{fDef}_1)\), assume that \(f(x_1, x_2, ..., x_n) = \{S\} : c \rightarrow c'\) and \((\sigma, \delta) \rightarrow f(x_1, x_2, ..., x_n) = \{S\} \rightarrow (\sigma', \delta')\). Also assume \((\sigma, \delta) \models c\). We show that \((\sigma', \delta') \models c'\), i.e. \(\forall f \in F, \delta'(f) = c'(f)\). Since we have \(\sigma = \sigma'\) and \(\delta' = \delta[f \mapsto a(n)]\), then \(\delta'(f) = a(n)\). Also \(c' = c[f \mapsto a(n)]\) implies...
c'(f) = a(n). From the assumption we have ∀f ∈ F. δ(f) = c(f), and since c' = c'(f),
we can conclude ∀f ∈ F. δ'(f) = c'(f) as required.

b. For the rule (f De f'), assume that f() = {S}; c → c' and
(S, δ) → f() = {S} → (σ', δ'). Also assume (σ, δ) ≡ c. We show that (σ', δ') ≈ c', i.e. ∀f ∈
F. δ'(f) = c'(f). Since we have σ = σ' and δ' = δ[f → na], then δ'(f) = na. Also
c' = c[f → na] implies c'(f) = na. From the assumption we have ∀f ∈ F. δ(f) = c(f),
and since δ'(f) = c'(f), we can conclude ∀f ∈ F. δ'(f) = c'(f) as required.

c. For the rule (ass), assume that x := a : c → c' and (σ, δ) → x := a → (σ', δ'). Also
assume (σ, δ) ≡ c. We show that (σ', δ') ≈ c', i.e. ∀f ∈ F. δ'(f) = c'(f). We have σ' =
σ[x → [a][σ]] and δ = δ'. Also we have c = c'. Then
d. For the rule (whl), assume that while a do S : c → c' and (σ, δ) → while a do S → (σ', δ').
Also assume (σ, δ) ≡ c. We show that (σ', δ') ≈ c', i.e. ∀f ∈ F. δ'(f) = c'(f). We have two
cases:

Case 1: δ = δ'. Since c = c', then δ'(f) = c'(f) as required.

Case 2: δ ≠ δ'. Since all the semantics rules either keep δ without change or add new
functions to it's domain as in (f De f'), then dom(δ) ⊆ dom(δ'). Let dₜ = δ' - δ and c' =
c ∪ dₜ. Then c ≤ c' and while a do S : c → c'. From the assumption we have ∀f ∈
F. δ(f) = c(f) and ∀f ∈ dₜ, δ'(f) = c'(f). Then we can conclude ∀f ∈ F. δ'(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 → c' in the safety-type system. Then if (σ, δ) ≡ c then it is impossible to get
(S, δ) → S → |.

Proof:

By structural induction on type derivation as follows:

1. For the rule (f De f'), we assume f(x₁, x₂, ..., xₙ) = {S}; c → c' and (σ, δ) ≡ c. We show that it is
impossible to have (σ, δ) → f(x₁, x₂, ..., xₙ) = {S} → |, i.e. there exist (σ', δ') such that (σ, δ) →
f(x₁, x₂, ..., xₙ) = {S} → (σ', δ'). But the only case we have is (σ, δ) → f(x₁, x₂, ..., xₙ) = {S} →
(σ, δ)[f → a(n)] from (f De f').

2. For the rule (f De f'), we assume f() = {S}; c → c' and (σ, δ) ≡ c. We show that it is impossible to have
(σ, δ) → f() = {S} → |, i.e. there exist (σ', δ') such that (σ, δ) → f() = {S} → (σ', δ'). But the
only case we have is (σ, δ) → f() = {S} → (σ, δ)[f → na] from (f De f').

3. For the rule (ass), assume x := a : c → c' and (σ, δ) ≡ c. We show that it is impossible to have
(σ, δ) → x := a → |, i.e. there exist (σ', δ') such that (σ, δ) → x := a → (σ', δ'). But the
only case we have is (σ, δ) → x := a → (σ[x → [a][σ]], δ) from (ass).

4. For the rule (whl), we assume while a do S : c → c' and (σ, δ) ≡ c, and assume as a contradiction
that (σ, δ) → while a do S → |. The required results from the induction hypothesis on Sₜ.

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() =
\[ x = \text{get}; \\
\text{if } x < 10 \text{ then put}(x) \text{ else put}(x - 10); \\
\]

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, \( r() \) can be replaced by \( r() \) i.e. \( r(6) \mapsto r() \).

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

\[ f(x, y) = \\
\{
  u = x + y; \\
  w = x - y; \\
  \text{put}(u); \\
  \text{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. \( \text{put}(2) \)
2. \( x_1 = \text{get}; x_2 = \text{get}; \)
3. \( \text{call } f(x_1, x_2); \)

In line 1, \( \text{put}(2) \) 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 \( S: t \mapsto S' \). 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 \( (\text{call}^l) \) 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 \( \text{call } f(x_1, x_2, ..., x_m) \) with \( \text{put}(n) \) 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 \( x_1 = \text{get}; x_2 = \text{get}; ...; x_n = \text{get}; \). Finally, call the function again with the new \( n \) values. The rule \( (\text{call}^l) \) can be considered as a special case of \( (\text{call}^l) \). We separate them to deal with the problem in a simpler way. The rules \( (\text{get}^r), (\text{put}^r), (\text{ass}^r), \text{and } (\text{skip}^r) \) as well as the rules \( (\text{whl}^r), (if^r), \text{and } (\text{seq}^r) \) all are straightforward.

\[ \text{Figure 4. Repair type system} \]
**Definition 3:**
\((\sigma, \delta) \sim_c (\sigma', \delta') \Rightarrow \text{dom}(\sigma) \subseteq \text{dom}(\sigma'), \forall x \in \text{dom}(\sigma). \sigma(x) = \sigma'(x)\) and 
\(\text{dom}(\delta) = \text{dom}(\delta'), \forall f \in \text{dom}(\delta). \delta(f) = \delta'(f)\).

**Theorem 3:**
Suppose that \(S: c \rightarrow c' \rightarrow S'\) and \((\sigma, \delta) \sim_c (\sigma, \delta)\) then:

1. If \((\sigma, \delta) \Rightarrow S \Rightarrow (\sigma', \delta')\) then there exists a state \((\sigma', \delta')\) such that \((\sigma', \delta') \Rightarrow S' \Rightarrow (\sigma', \delta')\) and \((\sigma', \delta') \sim_c (\sigma', \delta')\).
2. If \((\sigma, \delta) \Rightarrow S \Rightarrow (\sigma', \delta')\) and \(S\) does not get stuck at \((\sigma, \delta)\), then there exists a state \((\sigma', \delta')\) such that: \((\sigma, \delta) \Rightarrow S \Rightarrow (\sigma', \delta')\) 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 \((\text{call}^1f)\), assume that
   \[
   \text{call} f(x_1, x_2, ..., x_m); c[f \rightarrow a(n)] \rightarrow c \leftarrow \begin{cases} x_1 = \text{get}; x_2 = \text{get}; ...; x_n = \text{get}; \\
   \text{call} f(x_1, x_2, ..., x_n) 
   \end{cases}
   \]
   and \((\sigma, \delta) \sim_c (\sigma, \delta)\). Also assume \((\sigma, \delta) \Rightarrow f(x_1, x_2, ..., x_n) \rightarrow (\sigma', \delta')\). We show that there exists a state \((\sigma', \delta')\) such that \((\sigma, \delta) \rightarrow \text{put}(n); x_1 = \text{get}; x_2 = \text{get}; ...; x_n = \text{get}; f(x_1, x_2, ..., x_n) \rightarrow (\sigma', \delta')\) and \((\sigma', \delta') \sim_c (\sigma', \delta')\).
   b. Also assume \((\sigma, \delta) \rightarrow \text{call} f(x_1, x_2, ..., x_n) \rightarrow (\sigma', \delta')\). So \(g \Rightarrow (\sigma, \delta) \Rightarrow \text{put}(n); x_1 = \text{get}; x_2 = \text{get}; ...; x_n = \text{get}; f(x_1, x_2, ..., x_n) \rightarrow (\sigma', \delta')\) . Since \(\text{dom}(\sigma) \subseteq \text{dom}(\sigma')\) and \(\text{dom}(\sigma) \subseteq \text{dom}(\sigma')\), \(\text{dom}(\sigma) \subseteq \text{dom}(\sigma')\). Since \((\sigma, \delta) \sim_c (\sigma, \delta), \forall x \in \text{dom}(\sigma). \sigma(x) = \sigma(x).\) Since all the variables added to \(\sigma\) are all fresh then \(\sigma, (x) = \sigma, (x)\). The by assumption \(\sigma'\) and \(\sigma\) satisfy the required conditions.
   c. For the rule \((\text{call}^1f)\), assume that \(\text{call} f(x_1, x_2, ..., x_m); c[f \rightarrow na] \rightarrow c \leftarrow \text{call} f()\), and \((\sigma, \delta) \sim_c (\sigma, \delta)\). Also assume \((\sigma, \delta) \Rightarrow f(x_1, x_2, ..., x_m) \rightarrow (\sigma', \delta')\). We show that there exists a state \((\sigma', \delta')\) such that \((\sigma, \delta) \Rightarrow \text{call} f() \rightarrow (\sigma', \delta')\), and \((\sigma', \delta') \sim_c (\sigma', \delta')\). Since \((\sigma, \delta) \Rightarrow \text{call} f() \rightarrow (\sigma, \delta)\) and \((\sigma', \delta') \sim (\sigma, \delta)\), then \((\sigma', \delta') \sim_c (\sigma', \delta')\).
   d. For the rule \((\text{ass}^*)\), assume that \(x := a: c \rightarrow c' \equiv x := a\), and \((\sigma, \delta) \sim_c (\sigma, \delta)\). Also assume \((\sigma, \delta) \rightarrow x := a \rightarrow (\sigma', \delta')\). We show that there exists a state \((\sigma', \delta')\) such that \((\sigma, \delta) \Rightarrow \text{call} a(x \rightarrow a); \sigma' \Rightarrow (\sigma', \delta')\). Since \((\sigma, \delta) \rightarrow x := a \rightarrow (\sigma, \delta)\), then \((\sigma', \delta') = (\sigma, \delta) \Rightarrow (\sigma', \delta')\). But \(\text{dom}(\sigma) \subseteq \text{dom}(\sigma)\) and \(\text{dom}(\sigma) \subseteq \text{dom}(\sigma)\). Then \(\text{dom}(\sigma) \subseteq \text{dom}(\sigma)\). So \((\sigma', \delta') \sim_c (\sigma', \delta')\).

The remaining rules are straightforward to check.
2. By structural induction on type derivation of Repair type system:
   a. For the rule(call1), assume that

   \[ \text{call}\ f(x_1, x_2, ..., x_m) : c[f \to a(n)] \rightarrow c \leftrightarrow \begin{cases} \text{put}(n); \\ x_1 = \text{get}; x_2 = \text{get}; ..., x_n = \text{get}; \end{cases} \]

   Also assume \((\sigma, \delta) \sim_c (\sigma', \delta')\) and \((\sigma, \delta) \Rightarrow \text{put}(n); x_1 = \text{get}; x_2 = \text{get}; ..., x_n = \text{get}; \text{call}\ f(x_1, x_2, ..., x_n) \rightarrow (\sigma', \delta').\)

   From Theorem 3.1, \(\sigma' = \sigma[x_1 \rightarrow n_1, x_2 \rightarrow n_2, ..., x_n \rightarrow n_n]\) and \(\delta = \delta'.\) From semantics rules we have \((\sigma, \delta) \Rightarrow \text{call}\ f(x_1, x_2, ..., x_m) \rightarrow (\sigma', \delta') = (\sigma, \delta)\). We show that there exists \((\sigma', \delta') \Rightarrow \text{put}(n); x_1 = \text{get}; x_2 = \text{get}; ..., x_n = \text{get}; \text{call}\ f(x_1, x_2, ..., x_n) \rightarrow (\sigma', \delta'),\) and \((\sigma', \delta') \sim_c (\sigma', \delta').\) Since \((\sigma', \delta') = (\sigma, \delta)\), and \((\sigma, \delta) \subseteq \text{dom}(\sigma),\) we get \((\sigma', \delta') \sim_c (\sigma', \delta').\)

   b. For the rule(call2), assume that \(\text{call}\ f(x_1, x_2, ..., x_m) : c[f \to na] \rightarrow c \leftrightarrow \text{call}\ f(),\)

   and \((\sigma, \delta) \sim_c (\sigma', \delta').\) Also assume \((\sigma, \delta) \Rightarrow \text{call}\ f().\) We show that there exists a state \((\sigma', \delta')\) such that \((\sigma, \delta) \Rightarrow \text{call}\ f(x_1, x_2, ..., x_m) \rightarrow (\sigma', \delta').\) We show that \((\sigma', \delta') \sim_c (\sigma', \delta').\) Since \((\sigma, \delta) \Rightarrow \text{call}\ f(x_1, x_2, ..., x_m) \rightarrow (\sigma', \delta') = (\sigma, \delta),\) and \((\sigma, \delta) \Rightarrow (\sigma', \delta').\) Then we can conclude that \((\sigma', \delta') = (\sigma, \delta)\sim_c (\sigma', \delta').\)

   c. For the rule(ass'), assume that \(x := a; c \rightarrow c' \leftarrow x := a,\) and \((\sigma, \delta) \sim_c (\sigma', \delta').\) Also we assume \((\sigma, \delta) \Rightarrow x := a \rightarrow (\sigma', \delta')\). We show that there exists a state \((\sigma', \delta')\) such that \((\sigma, \delta) \Rightarrow x := a \rightarrow (\sigma', \delta')\) and \((\sigma', \delta') \sim_c (\sigma', \delta').\) Let \((\sigma, \delta) \Rightarrow x := a \rightarrow (\sigma', \delta').\) Hence \((\sigma', \delta') = (\sigma[x := a], \delta')\) and \((\sigma', \delta') = (\sigma[x := a], \delta').\) To show that \((\sigma', \delta') \sim_c (\sigma', \delta'),\) it is enough to show that, \(\text{dom}(\sigma) \subseteq \text{dom}(\sigma'),\) and \(\forall x \in \text{dom}(\sigma'),\sigma(x) = \sigma'(x).\) Since \(\text{dom}(\sigma') = \text{dom}(\sigma) \cup \{x\}\) and \(\text{dom}(\sigma') = \text{dom}(\sigma) \cup \{x\}\) and \(\forall x \in \text{dom}(\sigma),\sigma(x) = \sigma(x)\) and \(\forall x \in \text{dom}(\sigma'),\sigma'(x) = \sigma'(x).\) So \(\forall x \in \text{dom}(\sigma'),\sigma'(x) = \sigma'(x).\) Then we get \((\sigma', \delta') \sim_c (\sigma', \delta').\) as required.

   d. For the rule(whl'), assume that \(\text{while}\ a\ do\ S_i; c \rightarrow c \leftarrow \text{while}\ a\ do\ S'_i\),\) and \((\sigma, \delta) \sim_c (\sigma', \delta').\) Also assume \((\sigma, \delta) \Rightarrow \text{while}\ a\ do\ S'_i \rightarrow (\sigma', \delta').\) Let \((\sigma, \delta) \Rightarrow \text{while}\ a\ do\ S_i \rightarrow (\sigma', \delta').\) We have two cases:

   Case 1: \([a] = 0\) In this case \((\sigma, \delta) = (\sigma', \delta')\) and \((\sigma, \delta) = (\sigma', \delta')\) by (whl2). Then we can conclude that \((\sigma', \delta') \sim_c (\sigma', \delta').\)

   Case 2: \([a] = 1\) In this case by (whl2') we have \((\sigma, \delta) \Rightarrow S'_i \rightarrow (\sigma', \delta')\) and \((\sigma', \delta') \Rightarrow \text{while}\ a\ do\ S'_i \rightarrow (\sigma', \delta').\) We also have \((\sigma, \delta) \Rightarrow S_i \rightarrow (\sigma', \delta')\) and \((\sigma', \delta') \Rightarrow \text{while}\ a\ do\ S_i \rightarrow (\sigma', \delta').\) 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 another 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, & Khursid, 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


