Milestone 2 - Type Analysis

1 Introduction

1.1 Abstract

This project is aimed to do type checking for the *MiniC* language. Our SDD introduces an attribute e as the environment for identifiers, including variables and function names. Another attribute, *ok*, performs the type check and indicates whether the type check succeeds. When the program does not type check, this attribute will be `false`. The "deepest" `false` in the abstract syntax tree will indicate the precise location of type check error. In this project, the hard part is to deal with function calls. We have to check the number of the parameters, the type of the corresponding parameters, the function name as well as the return type. To deal with this problem, we used a function $GetGlobals()$ to read the top level function definitions and then use attribute e store the function identifiers with its parameter type, number and return type.

Another new feature is the pointer type. However, this is relatively easy to handle. $IsPointer(T\ t)$ was introduced to decide whether the type is pointer or not. In our SDD denotation, *T simply means a pointer of type T. The other parts stay pretty much the same with the dragon book, just normal expressions as well as necessary type checking. Our detailed design will be written in the following sections.

1.2 Attributes

- e attribute stands for the environment for identifiers to types, in which a map is used with **id**s as keys and *types* as values. Since *parameters* → *types* can also be considered as *types*, e can be compatible with d and ds. We can use $e1 \,||\, e2$, $e1 \,||\, d2$ and $e1 \,||\, ds2$ to represent concatenation of maps. This attribute is inherited.

- t attribute stands for type. This attribute is synthesized.

- *ok* attribute shows whether the type check passes. It will be set to `true` when the type check passes, and `false` otherwise. This attribute is synthesized.

- d attribute stands for a single declaration, in the format of a tuple, in which **id** is the key and *parameters* → *types* is the value. This attribute is synthesized.

- ds attribute stands for the environment for function declarations to types, in which a map is used with **ids** as keys and *parameters* → *types* as values. This attribute is synthesized.

- ps attribute stands for the parameter types for functions, in the format of a n-tuple $(t1, ..., tn)$. This attribute is synthesized.

- *sym* attribute stands for the identifier symbol (name). This attribute is synthesized.

- *ids* attribute stands for the identifier names for function parameters, in the format of n-tuple $(\mathbf{id}1, ..., \mathbf{id}n)$. This attribute is synthesized.

1.3 Environment Operations in SDDs

- $Defined(e, \mathbf{id})$ tests if **id** exists in e.
- $Lookup(e, \mathbf{id})$ denotes what **id** is bound to in e.
- $Extend(e, \mathbf{id}, T)$ denotes a new environment extending e with binding **id** to T. It can cause shadowing based on scopes.

1.4 Logical Operators

In the SDDs we use the following logical operators to compute the attribute ok in several cases. There are only two values available for this attribute: `true` and `false`.

- $\neg$ stands for negation. $\neg A$ is `false` if and only if A is `true`; $\neg A$ is `true` if and only if A is `false`.
- $\wedge$ stands for logical conjunction. $A \wedge B$ is `true` if and only if A and B are both `true`; otherwise it is `false`.
- $\vee$ stands for logical (inclusive) disjunction. $A \vee B$ is `false` if and only if A and B are both `false`; otherwise it is `true`.
- $\oplus$ stands for exclusive disjunction. $A \oplus B$ is `true` if and only if either A or B, but not both, is `true`; otherwise it is `false`.

1.5 Additional Helper Functions

- $GetGlobals()$ will read and store the globally defined functions in ds, following the format of (**id**, $parameters \rightarrow types$). The result should be compatible with attribute e and ds. In this project, the functions that should be loaded by this function should be: `malloc, puts, puti, atoi, div, mod`.
- $IsEqualType(T\ t1, T\ t2)$ checks whether $t1$ and $t2$ are of the same type. It can be done by simply checking the names of types. It will return `true` or `false`.
- $IsPointer(T\ t)$ checks whether t is of a pointer type. It can be done by simply checking the prefix of the type name. It will return `true` or `false`.
- $CheckIdDistinct(ids)$ checks whether the identifiers in tuple ids are distinct. It will return `true` or `false`.
- $CheckMain(\mathbf{id}.sym)$ check whether the $\mathbf{id}.sym$ is equal to main, in other words, whether this is a main function. It will return `true` or `false`.
- $CheckParams(ps1, ps2)$ checks whether the parameters in $ps1$ is of the same format as $ps2$, which means, they have the same number of elements and corresponding types. It will return `true` or `false`.
- $GetDerefType(T\ t)$ returns the dereferenced type, which means, when given type *T, this function will return T. If t is not a pointer, this function will raise an error.
- $GetReturnType(e, E)$ finds the function identifier associated with E and find the function return type in the current environment. Attribute e stores every declared function identifier and their types as $ps \rightarrow T$. This function will look through the environment map, finds the corresponding $ps \rightarrow T$ and returns T. If the identifier is not found, this function will raise an error.

- $GetParams(e, E)$ finds the function identifier associated with E and find the function parameter types ps in the current environment. Attribute e stores every declared function identifier and their types as $ps \rightarrow T$. This function will look through the environment map, finds the corresponding $ps \rightarrow T$ and returns ps. If the identifier is not found, this function will raise an error.

1.6 Additional Operators

- $||$ stands for concatenation of maps. It can be used to concatenate e, d or ds to form a new map.
- $A?...B... : ...C...$ stands for `if (A) then B else C`, which means, if A is true then choose B, otherwise choose C.

2 SDD

Below is the detailed SDD based on the abstract syntax of *MiniC*. It uses the abbreviations P for *Program*, D for *Declaration*, Ss for *Statements*, S for *Statement*, E for *Expression*, and T for *Type*.

Production	Semantic Rules
P → D1 ... Dn	$P.ds = D1.d \mid\mid ... \mid\mid Dn.d$ $D1.e = D2.e = ... = Dn.e = P.e = GetGlobals() \mid\mid P.ds$ $P.ids = (D1.sym, ..., Dn.sym)$ $P.ok = D1.ok \wedge ... \wedge Dn.ok \wedge CheckIdDistinct(P.ids)$
D → `function` T **id** (T1 **id**1, ..., Tn **id**n) {Ss}	$D.ps = (T1, ..., Tn);\ D.ids = (\mathbf{id}1.sym, ..., \mathbf{id}n.sym);$ $D.d = (\mathbf{id}.sym, ps \rightarrow T);\ D.sym = \mathbf{id}.sym$ $Ss.e = Extend(D.e, \mathbf{id}.sym, ps \rightarrow T)$ $D.ok = IsEqualType(T, Ss.t) \wedge Ss.ok \wedge CheckIdDistinct(D.ids) \wedge$ $(CheckMain(\mathbf{id}.sym)\ ?$ $(IsEqualType(T, \texttt{int}) \wedge$ $CheckParams(D.ps, (\texttt{*char, ..., *char})) : \texttt{true})$
Ss → `var` T1 **id**2; Ss3	$Ss3.e = Extend(Ss.e, \mathbf{id}2.sym, T1)$ $Ss.t = Ss3.t$ $Ss.ok = Ss3.ok$
Ss → E1 = E2; Ss3	$E1.e = E2.e = Ss3.e = Ss.e$ $Ss.t = Ss3.t$ $Ss.ok = IsEqualType(E1.t, E2.t) \wedge Ss3.ok \wedge E1.ok \wedge E2.ok$
Ss → `if` (E1) S2; Ss3	$E1.e = S2.e = Ss3.e = Ss.e$ $Ss.t = Ss3.t$ $Ss.ok = (IsEqualType(E1.t, \texttt{int}) \vee IsPointer(E1.t)) \wedge$ $E1.ok \wedge S2.ok \wedge Ss3.ok$
Ss → `if` (E1) S2 `else` S3; Ss4	$E1.e = S2.e = S3.e = Ss4.e = Ss.e$ $Ss.t = Ss4.t$ $Ss.ok = (IsEqualType(E1.t, \texttt{int}) \vee IsPointer(E1.t)) \wedge$ $E1.ok \wedge S2.ok \wedge S3.ok \wedge Ss4.ok$
Ss → `while` (E1) S2; Ss3	$E1.e = S2.e = Ss3.e = Ss.e$ $Ss.t = Ss3.t$ $Ss.ok = (IsEqualType(E1.t, \texttt{int}) \vee IsPointer(E1.t)) \wedge$ $E1.ok \wedge S2.ok \wedge Ss3.ok$

Ss → `return` E1; Ss2	$E1.e = Ss2.e = Ss.e$ $Ss.t = E1.t$ $Ss.ok = E1.ok \wedge Ss2.ok$
Ss → {Ss1} Ss2	$Ss1.e = Ss2.e = Ss.e$ $Ss.t = IsEqualType(Ss1.t, \epsilon)\ ?\ Ss2.t\ :\ Ss1.t$ $Ss.ok = Ss1.ok \wedge Ss2.ok$
Ss → ϵ	$Ss.t = \epsilon$ $Ss.ok = \texttt{true}$
E → **id**1	$E.t = Lookup(E.e, \mathbf{id}1.sym)$ $E.ok = Defined(E.e, \mathbf{id}1.sym)$
E → **str**1	$E.t = \texttt{*char}$ $E.ok = \texttt{true}$
E → **int**1	$E.t = \texttt{int}$ $E.ok = \texttt{true}$
E → E0(E1, ..., En)	$E0.e = E1.e = ... = En.e = E.e$ $E.t = GetReturnType(E.e, E0)$ $E.ok = CheckParams(GetParams(E.e, E0), (E1.t, ..., En.t)) \wedge$ $E0.ok \wedge E1.ok \wedge ... \wedge En.ok$
E → `null`(T1)	$E.t = T1$ $E.ok = \texttt{true}$
E → `sizeof`(T1)	$E.t = \texttt{int}$ $E.ok = \texttt{true}$
E → ! E1	$E1.e = E.e$ $E.t = \texttt{int}$ $E.ok = IsEqualType(E.t, E1.t) \wedge E1.ok$
E → - E1	$E1.e = E.e$ $E.t = \texttt{int}$ $E.ok = IsEqualType(E.t, E1.t) \wedge E1.ok$
E → + E1	$E1.e = E.e$ $E.t = \texttt{int}$ $E.ok = IsEqualType(E.t, E1.t) \wedge E1.ok$
E → * E1	$E1.e = E.e$ $E.t = GetDerefType(E1.t)$ $E.ok = E1.ok \wedge IsPointer(E1.t)$
E → & E1	$E1.e = E.e$ $E.t = *E1.t$ $E.ok = E1.ok$ $E.ok = IsPointer(E1.t) \wedge E1.ok$
E → E1 * E2	$E1.e = E2.e = E.e$ $E.t = \texttt{int}$ $E.ok = IsEqualType(E.t, E1.t) \wedge$ $IsEqualType(E.t, E2.t) \wedge E1.ok \wedge E2.ok$
E → E1 / E2	$E1.e = E2.e = E.e$ $E.t = \texttt{int}$ $E.ok = IsEqualType(E.t, E1.t) \wedge$ $IsEqualType(E.t, E2.t) \wedge E1.ok \wedge E2.ok$

E → E1 % E2	$E1.e = E2.e = E.e$ $E.t = \texttt{int}$ $E.ok = IsEqualType(E.t, E1.t) \wedge$ $\quad IsEqualType(E.t, E2.t) \wedge E1.ok \wedge E2.ok$
E → E1 + E2	$E1.e = E2.e = E.e$ $E.t = IsPointer(E1.t)\ ?\ E1.t\ :\ \texttt{int}$ $E.ok = IsEqualType(E.t, E1.t) \wedge$ $\quad IsEqualType(\texttt{int}, E2.t) \wedge E1.ok \wedge E2.ok$
E → E1 - E2	$E1.e = E2.e = E.e$ $E.t = IsPointer(E1.t)\ ?\ E1.t\ :\ \texttt{int}$ $E.ok = IsEqualType(E.t, E1.t) \wedge$ $\quad IsEqualType(\texttt{int}, E2.t) \wedge E1.ok \wedge E2.ok$
E → E1 < E2	$E1.e = E2.e = E.e$ $E.t = \texttt{int}$ $E.ok = IsEqualType(E.t, E1.t) \wedge$ $\quad IsEqualType(E.t, E2.t) \wedge E1.ok \wedge E2.ok$
E → E1 > E2	$E1.e = E2.e = E.e$ $E.t = \texttt{int}$ $E.ok = IsEqualType(E.t, E1.t) \wedge$ $\quad IsEqualType(E.t, E2.t) \wedge E1.ok \wedge E2.ok$
E → E1 <= E2	$E1.e = E2.e = E.e$ $E.t = \texttt{int}$ $E.ok = IsEqualType(E.t, E1.t) \wedge$ $\quad IsEqualType(E.t, E2.t) \wedge E1.ok \wedge E2.ok$
E → E1 >= E2	$E1.e = E2.e = E.e$ $E.t = \texttt{int}$ $E.ok = IsEqualType(E.t, E1.t) \wedge$ $\quad IsEqualType(E.t, E2.t) \wedge E1.ok \wedge E2.ok$
E → E1 == E2	$E1.e = E2.e = E.e$ $E.t = \texttt{int}$ $E.ok = ((IsEqualType(E.t, E1.t) \wedge IsEqualType(E.t, E2.t)) \vee$ $\quad (IsPointer(E1.t) \wedge IsEqualType(E1.t, E2.t))) \wedge$ $\quad E1.ok \wedge E2.ok$
E → E1 != E2	$E1.e = E2.e = E.e$ $E.t = \texttt{int}$ $E.ok = ((IsEqualType(E.t, E1.t) \wedge IsEqualType(E.t, E2.t)) \vee$ $\quad (IsPointer(E1.t) \wedge IsEqualType(E1.t, E2.t))) \wedge$ $\quad E1.ok \wedge E2.ok$
E → E1 && E2	$E1.e = E2.e = E.e$ $E.t = \texttt{int}$ $E.ok = (IsEqualType(E.t, E1.t) \vee IsPointer(E1.t)) \wedge$ $\quad (IsEqualType(E.t, E2.t) \vee IsPointer(E2.t)) \wedge$ $\quad E1.ok \wedge E2.ok$

E → E1 || E2	$E1.e = E2.e = E.e$ $E.t = \texttt{int}$ $E.ok = (IsEqualType(E.t, E1.t) \vee IsPointer(E1.t)) \wedge$ $(IsEqualType(E.t, E2.t) \vee IsPointer(E2.t)) \wedge$ $E1.ok \wedge E2.ok$

3 Appendix

3.1 Non-Terminals and Associated Attributes

Below is a detailed list of the non-terminals in the production and their attributes. A $\surd$ mark indicates that this non-terminal is associated with the current attribute.

	e	t	ok	d	ds	ps	ids	sym
P	$\surd$		$\surd$		$\surd$		$\surd$	
D	$\surd$		$\surd$	$\surd$		$\surd$	$\surd$	$\surd$
S	$\surd$		$\surd$					
Ss	$\surd$	$\surd$	$\surd$					
E	$\surd$	$\surd$	$\surd$					

3.2 Additional Design Choices

There are several additional design choices made when designing this SDD. These choices are made where the assignment does not give instruction towards the choices.

- In function declaration D, we restricted that the parameter list should not have the same **id**.sym twice.
- In function declaration D, (`*char, ..., *char`) means this tuple has the same number of elements as its comparing *D.ps* but the types are all `*char`.
- For statements, when facing `return`, our design will directly set the type of Ss to the type of the return expression, no matter what other Ss follows behind it. This is reasonable because the codes after `return` within the current scope cannot be reached.