UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCES
Institute of Computer Science
Specialty of Computer Science

Jaak Ristioja

An analysis framework for an imperative privacy-preserving programming language

Master's thesis (30 ECTS)

Supervisor: Dan Bogdanov, MSc

Author: .... 15. June 2010
Supervisor: .... 15. June 2010

Allowed for defense
Professor .... "......" June 2010

Tartu 2010

Contents

1 Introduction

Program analysis has a major role in modern-day software development. It is used in many programming environments, most importantly in optimizing compilers, debugging tools and integrated development environments to provide syntax-directed editing of source code, context sensitive auto-completion etc. The primary advantage provided by static analyses is the information given about the dynamic behavior of the program without actually running it.

Sharemind [?] is a secure multi-party general-purpose computation system, which is able to process data without disclosing it. It is foremost meant to be used for data mining applications [?, ?]. Sharemind consists of a hybrid virtual machine which is run on n participating computers in a synchronous fashion [?][1]. The participants are called *miners*. The Sharemind hybrid virtual machine currently interprets an assembly language [?] which makes a clear distinction between public and secret (or private) data registers and values, which can be viewed as different domains for data. Each assembly program executed on the hybrid machine is run on all miners in parallel. The Sharemind assembly language has different instructions for operating on public and secret data. It also provides means for input of public and secret data, output of public data, and transforming public data to secret data and vice-versa.

Public data is entered by sending the input data to each miner in the hybrid machine. All the miners have identical copies of all public data. On the other hand, all secret data in Sharemind hybrid virtual machine is distributed among the n miners by using a secret sharing scheme. The secret sharing scheme uses a randomized algorithm to split the sensitive data values into n shares. All n shares are distributed among the miners, so that each miner gets one share. The secret sharing scheme guarantees that the secret data can only be deduced from no less than n shares. All secret data is input to the hybrid machine as shares. To mitigate a side-channel attack, only public data is allowed to change the control flow of the hybrid machine during execution.

To perform computations on the sensitive data, secure computation protocols are executed between the parties. Different share computation protocols are used for each atomic operation in Sharemind. These include protocols for addition, multiplication, comparisons of 32-bit unsigned integers. Several protocols for other simple operations are currently being developed. All intermediate and final results of sensitive computations done on the shares are also distributed as n shares among the parties. The share computation protocols are also formulated in ways to ensure that at no point in time can any $n - 1$ parties deduce correct information about the sensitive input or result data. To read the final result of any computation, a trusted party collects all the n shares of the result and calculates its real value.

Since writing programs in an assembly language is an error-prone and tedious task, a higher-level imperative programming language for the Sharemind hybrid machine was developed [?]. Its syntax is inspired by the syntax of the C programming language, hence the programming language was named SecreC. The most important feature to distinguish SecreC from similar simple programming languages is its separation of public and private data, which correspond to public and secret data in the Sharemind hybrid machine. The handling of public and secret data is somewhat similar to [?, ?], but an additional *declassification* operator (**`declassify`**) is provided to publish private data values and conditional branching on private values is not allowed.

Although the Sharemind hybrid virtual machine provides adequate security guarantees for individual

[1]Currently the implementation of the Sharemind hybrid virtual machine is limited to three participants.

operations, it is still left to the programmer to decide exactly what secret data to publish. Before making any secret data public, it is advisable to process the data to an extent which makes the task of deducing any of the original secret input values from it infeasible. Publishing secret inputs directly is clearly something the programmer should not do.

SecreC is aimed to give guarantees for the security of private data processed by programs written in the SecreC programming language. We have decided to create a program analysis framework to aid the programmer in minimizing the amount of information leaked by programs. Currently, our framework is a work-in-progress to provide tools to help the programmer measure data leaks in SecreC programs. To write programs that preserve the privacy of the their secret inputs, the programmer needs to know exactly in which conditions data is published, and what kind of processing the secret data has gone through before being published. A strictly formal specification of the grammar and semantics of SecreC is needed before any correct analysis with safety guarantees can be formulated on top of the language.

Hence, mostly for future purposes we have formalized a major subset of the SecreC language with the means of a context free grammar, formulation of its type system together with static checking rules, and operational semantics. Finally, we have created an intermediate representation which, if used together with a symbol table, is capable of expressing all SecreC programs described by the rules in this paper. This intermediate representation is used for analysis purposes by our analyzer. SecreC Analyzer provides a simple framework for running forward and backward data-flow analyses. Currently, a few simple data-flow analysis algorithms are implemented.

One of the principles formulated for the design of SecreC Analyzer dictates that the framework must include a library which can be used in integrated development environments (IDEs), such as SecreCIDE [**?**]. In the future, our analysis framework is also meant to serve as a front-end for optimizing SecreC compilers.

1.1 Outline of this thesis

This master's thesis presents a formalization of the core subset of the SecreC programming language – its grammar, static checking rules and semantics – and describes the analysis framework and analyses currently being developed.

- Chapter 2 presents the formal grammar for a subset of SecreC, which is to be used for the basis of our analyses and as a starting point for any future improvements to the language. The rules given also serve as an initial reference point to anyone interested in the language, since they cover its basics.

- Chapter 3 is dedicated to formulate static checking rules to which all SecreC programs must comply. We have developed a type system which reflects the allowed data flow between private and public data domains. These rules also seek to enforce good coding practices, e.g. by disallowing some unreachable code.

- Chapter 4 gives formal semantics for the SecreC programming language, which is of most use in SecreC compilers and interpreters. These semantics are also used by the SecreC Analyzer presented by in following chapters.

- Chapter 5 presents an intermediate representation used for reasoning about SecreC programs.

SecreC Analyzer uses the intermediate representation to perform data-flow analyses. In principle, this intermediate representation can be translated into assembly code format.

- Chapter 6 gives an overview of data-flow analysis in the SecreC Analyzer. We describe the format of the control flow graph used in our analyzer, and the implemented analysis.
- Chapter 7 describes three different data-flow analyses we have implemented using the framework provided by the SecreC Analyzer. These serve to demonstrate some of the capabilities of the analyzer and also form a minimum basis for future analyses.

1.2 Author's contribution

The main contribution to this thesis is the formal specification for the core subset of the SecreC programming language. It is provided as a context-free grammar, static checking rules and semantic rules given in Chapters 2, 3 and 4 respectively. Before writing this specification, SecreC was only defined by its current compiler to Sharemind assembly code [?]. The author developed this formalization based on the grammar file used in the SecreC compiler, on the explanations of language features provided by Roman Jagomägis and on a future vision for the language by Dan Bogdanov. Parallel to writing this specification, the author developed the SecreC Analyzer for analysis of SecreC programs. The intermediate representation for SecreC programs used by the analyzer is given in Chapter 5 and the principles it uses to perform data-flow analysis are given in Chapter 6. The author also implemented three simple data-flow analyses for the analyzer as described in Chapter 7.

2 Grammar

SecreC is a domain-specific programming language strongly influenced by the syntax of C. Since it is aimed at providing a higher-level language for writing computer programs for the Sharemind hybrid virtual machine, SecreC needs to be fully formalized to provide strong security guarantees in the future.

The grammar given in this chapter does not exactly describe the language accepted by the current SecreC compiler described in [?]. Major omissions in this paper include vectors, matrices and type casting which are still being worked on. Compared to [?], we allow global variable definitions and procedure definitions to occur in any order. Also, specifying the SecreC standard library is not a part of this thesis.

We specify the syntax for the subset of SecreC using a context free grammar. The grammar rules are described using a variant of the Backus-Naur Form (BNF) extended with some regular expression constructs, such as regular braces for grouping subexpressions, the repetition operator $*$ as superscript, and the ? suffix for denoting optionality of the preceding subexpression.

In the specification of the grammar, we use angle brackets to denote ⟨`nonterminals`⟩, apostrophes to denote terminals, i.e. `'exact strings'` as returned by the scanner, and all capitals for certain sets of strings such as the set of `IDENTIFIER` names.

2.1 Character set and whitespace

Tokens returned by the SecreC scanner are either keywords, operators, string literals, integer literals, unsigned integer literals or identifiers. When tokenizing the input, whitespace is allowed to appear between tokens. There are also two kinds of comments which are treated as whitespace between tokens. Any such whitespace is ignored, except inside string literals. Inside string literals, whitespace and characters denoting comment constructs are returned as part of the string literal token.

In SecreC, there are 18 keywords: **bool**, **break**, **continue**, **declassify**, **do**, **else**, **false**, **for**, **if**, **int**, **private**, **public**, **return**, **string**, **true**, **unsigned**, **void** and **while**; and 28 other tokens with fixed contents:

`+=`	`-=`	`*=`	`/=`	`%=`	`&&`	`||`
`=`	`<=`	`>=`	`<`	`>`	`==`	`!=`
`{`	`}`	`(`	`)`	`?`	`:`	`,`
`+`	`*`	`/`	`%`	`-`	`!`	`;`

String literals (strings in the set `STRING_LITERAL`) in SecreC are tokens which start with a double quote character (") and continue up to (and including) another double quote character not preceded by a backslash character (\). The string literal therefore always ends and starts with double quote characters and may contain *escape sequences*, i.e. \" to denote a single double quote character in the string and \\ denoting a single backslash in the string. The string corresponding to the string literal is stripped of the two enclosing double quotes, and has the escape sequences replaced by their single-character counterparts. Implementation may also define additional escape sequences.

Integer literals (strings in the set `INT_LITERAL`) in SecreC are tokens which are used to represent signed integer values in the source code of SecreC programs. These are required to match the following regular expression:

$$0 \mid (1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)\,(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^*$$

Unsigned integer literals (strings in the set `UINT_LITERAL`) are used to represent unsigned integer values and are required to match the following regular expression:

$$0\left(0 \mid (1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)\,(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^*\right)$$

Identifiers (string in the set `IDENTIFIER`) are tokens which represent variable and procedure names in SecreC. These strings consist only of lowercase and uppercase Latin characters, underscores and decimal digits, the first character of any identifier must not be a decimal digit. Keywords are not considered to be identifiers.

There are two types of comments in SecreC, both of which should not be included in the output list of tokens. The single line comments start with two consecutive forward slashes (//) and continue until a new line character (e.g. bytes with hexadecimal values 0x0A and 0x0D in ASCII). Multiline comments start with a consecutive forward slash and asterisk (/*) and continue up to (and including) a consecutive asterisk and forward slash (*/). If the character sequences that start comments appear inside string literals or other comments, they are not considered to start comments. This also means that comments can not be nested inside other comments.

2.2 Program

The top-level of any SecreC program consists of one or more global variable or procedure definitions:

```
⟨program⟩ ::= ⟨variable_definition⟩ ⟨program⟩
            | ⟨procedure_definition⟩ ⟨program⟩
            | ⟨variable_definition⟩
            | ⟨procedure_definition⟩
```

Later static checking rules require that there be at least a procedure called `main` present in the program. Note that while variable definitions can also appear in local scopes (see Section 2.6), procedures can only be defined globally, under the ⟨`program`⟩ nonterminal rule. According to the semantics of SecreC, all definitions after the definition of procedure **`void`** `main()` are unreachable. The static checking rules and semantics for the top-level program constructs are given in detail in Subsection 3.2.5 and Section 4.6, respectively.

2.3 Variable definitions

Variables are defined by specifying the type of the variable, its name and an optional initializer expression:

```
⟨variable_definition⟩ ::= ⟨type_specifier⟩ IDENTIFIER '=' ⟨expression⟩ ';'
                          ⟨type_specifier⟩ IDENTIFIER ';'
```

2.4 Types

Type annotations in SecreC consist of the security type annotation followed by the data type annotation:

```
      ⟨type_specifier⟩ ::= ⟨sectype_specifier⟩ ⟨datatype_specifier⟩
   ⟨sectype_specifier⟩ ::= 'public' | 'private'
  ⟨datatype_specifier⟩ ::= 'string' | 'int' | 'unsigned' 'int' | 'bool'
```

2.5 Procedures

Procedure definitions consist of a return type annotation, the name of the procedure, its list of type-annotated parameters and the function body:

```
     ⟨procedure_definition⟩ ::= ('void' | ⟨type_specifier⟩) IDENTIFIER
                                 '(' ⟨procedure_parameter_list⟩? ')'
                                 ⟨compound_statement⟩
 ⟨procedure_parameter_list⟩ ::= ⟨procedure_parameter⟩ (',' ⟨procedure_parameter⟩)*
      ⟨procedure_parameter⟩ ::= ⟨type_specifier⟩ IDENTIFIER
```

Note, that if the procedure does not return anything, its return type should be annotated as `'void'`. The body of the procedure is always enclosed in curly brackets (i.e. `'{'` and `'}'`) because we use ⟨`compound_statement`⟩ as described in Section 2.6.

2.6 Statements

The set of statements in SecreC is also similar to those in C, except that there are no **`switch`** statements, **`goto`** statements and labels. Statements only appear inside procedure bodies.

```
⟨compound_statement⟩ ::= '{' ⟨statement_list⟩? '}'
    ⟨statement_list⟩ ::= ⟨variable_definition⟩ ⟨statement_list⟩
                        | ⟨statement⟩ ⟨statement_list⟩
                        | ⟨statement⟩
         ⟨statement⟩ ::= ⟨compound_statement⟩ | ⟨if_statement⟩ | ⟨for_statement⟩
                        | ⟨while_statement⟩ | ⟨dowhile_statement⟩ | 'return' ⟨expression⟩ ';'
                        | 'return' ';' | 'continue' ';' | 'break' ';' | ';' | ⟨expression⟩ ';'

      ⟨if_statement⟩ ::= 'if' '(' ⟨expression⟩ ')' ⟨statement⟩ ('else' ⟨statement⟩)?
     ⟨for_statement⟩ ::= 'for' '(' ⟨expression⟩? ';' ⟨expression⟩? ';' ⟨expression⟩? ')'
                            ⟨statement⟩
   ⟨while_statement⟩ ::= 'while' '(' ⟨expression⟩ ')' ⟨statement⟩
 ⟨dowhile_statement⟩ ::= 'do' ⟨statement⟩ 'while' '(' ⟨expression⟩ ')'
```

In contrast to these grammar rules, static checking rules in Chapter 3 allow `'break'` and `'continue'` statements only to appear inside loop bodies. Similarly, the two different kinds of `'return'` statements are checked for validity with respect to the return type of the procedure the statements are contained in.

2.7 Expressions

The following grammar rules are used for expressions:

```
               ⟨expression⟩ ::= ⟨assignment_expression⟩
    ⟨assignment_expression⟩ ::= IDENTIFIER ⟨assignment_operator⟩ ⟨assignment_expression⟩
                              | ⟨conditional_expression⟩
      ⟨assignment_operator⟩ ::= '=' | '*=' | '/=' | '%=' | '+=' | '-='
   ⟨conditional_expression⟩ ::= ⟨logical_or_expression⟩ ('?' ⟨expression⟩ ':' ⟨expression⟩)?
    ⟨logical_or_expression⟩ ::= (⟨logical_or_expression⟩ '||')? ⟨logical_and_expression⟩
   ⟨logical_and_expression⟩ ::= (⟨logical_and_expression⟩ '&&')? ⟨equality_expression⟩
      ⟨equality_expression⟩ ::= (⟨equality_expression⟩ ('==' | '!='))? ⟨relational_expression⟩
    ⟨relational_expression⟩ ::= (⟨relational_expression⟩ ('<' | '>' | '<=' | '>='))?
                                  ⟨additive_expression⟩
      ⟨additive_expression⟩ ::= (⟨additive_expression⟩ ('+' | '-'))? ⟨multiplicative_expression⟩
⟨multiplicative_expression⟩ ::= (⟨multiplicative_expression⟩ ('*' | '/' | '%'))?
                                  ⟨unary_expression⟩
         ⟨unary_expression⟩ ::= ('-' | '!') ⟨postfix_expression⟩
                              | ⟨postfix_expression⟩
       ⟨postfix_expression⟩ ::= 'declassify' '(' ⟨expression⟩ ')'
                              | IDENTIFIER '(' (⟨expression⟩ (',' ⟨expression⟩)*)? ')'
                              | ⟨primary_expression⟩
       ⟨primary_expression⟩ ::= '(' ⟨expression⟩ ')' | IDENTIFIER | ⟨constant⟩
                 ⟨constant⟩ ::= INT_LITERAL | UINT_LITERAL | STRING_LITERAL | 'true' | 'false'
```

The keywords **true** and **false** in the nonterminal ⟨constant⟩ are used to denote boolean constants.

We have summarized the associativity and precedence of relevant SecreC operators in the table below. The operators in the table are given in order of precedence from the highest to the lowest.

Level	**Operator**	**Description**	**Associativity**
1.	`()`	Procedure call	Left-associative
2.	`-`	Unary arithmetic negation	Right-associative
	`!`	Unary logical negation	
3.	`*`	Multiplication	Left-associative
	`/`	Division	
	`%`	Modulus	
4.	`+`	Addition	Left-associative
	`-`	Subtraction	
5.	`<`	Relational “less than”	Left-associative
	`<=`	Relational “less than or equal to”	
	`>`	Relational “greater than”	
	`>=`	Relational “greater than or equal to”	

Level	Operator	Description	Associativity
6.	`==`	Relational “is equal to”	Left-associative
	`!=`	Relational “is not equal to”	
7.	`&&`	Logical AND	Left-associative
8.	`||`	Logical OR	Left-associative
9.	`?:`	Ternary conditional	Right-associative
10.	`=`	Assignment	Right-associative
	`+=`	Arithmetic addition assignment	
	`-=`	Arithmetic subtraction assignment	
	`*=`	Arithmetic multiplication assignment	
	`/=`	Arithmetic division assignment	
	`%=`	Arithmetic modulus assignment	

3 Static checking

SecreC is a statically typed language, meaning that type checking can and should be performed at compile-time only to ensure run-time type-safety. Type-annotated definitions of variables and procedures as described in Chapter 2 give a sound foundation for this. In this chapter we present a system for static checking (and type-checking) of SecreC programs which also defines the set of all grammatically correct SecreC programs that are also valid semantically.

For static checking of SecreC programs, we stratify the type system and typing rules into three separate layers, each of which corresponds to a certain layer of rules in the SecreC grammar (or an abstract syntax tree). The lower layer of the type system deals with notions like values, variables, expressions. The middle layer corresponds to typing statements. The higher layer types and rules correspond to whole SecreC programs. Although all three layers are used jointly in static checking, they are distinct in their construction.

In this chapter we first formulate a type system for SecreC, used to statically check SecreC programs, statements and expressions. After this we present formal inference rules for static checking.

3.1 Type system

The type system used for static checking of SecreC programs provides means to denote certain properties of interest for SecreC programs and parts thereof. We consider the aforementioned three strata separately in the following sections.

3.1.1 Regular types

The lower layer of the type system for SecreC defines properties for values and variables in the language itself. Namely, each constant and expression, as well as each defined variable and procedure in any valid SecreC program has a type from the lower layer of this type system used for static checking.

Let's have a meta-variable ρ to denote these types. Each such type either consists of an abstract security type ρ_τ and an abstract data type ρ_δ, or is **`void`**:

$$\rho ::= \left(\rho^\tau, \rho^\delta\right) \ \mid \mathtt{void}$$

The type **`void`** (also know as the *unit type*) denotes data holding no information, i.e. allows only one possible value. It is not possible in SecreC to define variables of type **`void`**, or to work on values of type **`void`**. However, the return type of procedures may be defined to be **`void`**. In the latter case the procedure does not return a value, and such a procedure call can not be used as a subexpression, but only as a separate expression statement.

3.1.1.1 Data types

Data types are further stratified into two layers. The lower layer data types are called fundamental data types and the higher level data types are referred to as abstract data types.

The SecreC programming language has four fundamental data types, denoted by the meta-variable δ: booleans (**bool**), 32-bit signed integers (**int**), 32-bit unsigned integers (**unsigned int**) and strings (**string**). Formally:

$$\delta ::= \texttt{bool} \mid \texttt{int} \mid \texttt{unsigned int} \mid \texttt{string}$$

An abstract data type, denoted by the meta-variable ρ^δ, is either

- a fundamental data type δ denoting the types of values,
- a variable type δ *var* denoting the type of variables holding values of fundamental data type δ, i.e. the abstract data type of the *l-value* for such variables[2],
- a procedure type $(\delta_1 \times \ldots \times \delta_n) \rightarrow \delta$ denoting procedures taking n parameters (values) of fundamental data types $\delta_1, \ldots, \delta_n$ and returning a value of fundamental data type δ.
- a procedure type $(\delta_1 \times \ldots \times \delta_n) \rightarrow ()$ denoting procedures taking n parameters (values) of fundamental data types $\delta_1, \ldots, \delta_n$ and returning no value.

More formally:

$$\rho^\delta ::= \delta \mid \delta\ var \mid (\delta_1 \times \ldots \times \delta_n) \rightarrow \delta \mid (\delta_1 \times \ldots \times \delta_n) \rightarrow ()$$

3.1.1.2 Security types

Data in SecreC is classified into the public and private domains. Hence, we use **public** and **private** security classes to denote data accordingly. These security classes can also be modeled after [?] to form a lattice. We denote the security classes using the meta-variable τ:

$$\tau ::= \texttt{public} \mid \texttt{private}$$

An abstract security type, denoted by the meta-variable ρ^τ, is (similarly to abstract data types) one of the following:

- a security class τ denoting the security class of data held as values and data held by variables
- a procedure type $(\tau_1 \times \ldots \times \tau_n) \rightarrow \tau$ denoting procedures taking n parameters (values) of security classes $\tau_1, \ldots, \tau_n$ and returning a value of security class τ.
- a procedure type $(\tau_1 \times \ldots \times \tau_n) \rightarrow ()$ denoting procedures taking n parameters (values) of security classes $\tau_1, \ldots, \tau_n$ and returning no value.

More formally:

$$\rho^\tau ::= \tau \mid (\tau_1 \times \ldots \times \tau_n) \rightarrow \tau \mid (\tau_1 \times \ldots \times \tau_n) \rightarrow ()$$

In addition, we define the binary $\oplus$ operator which is similar to the $\oplus$ operator in [?]. It is used in the static checking rules in Subsection 3.2.3 to infer security types for results of certain kinds of expressions. Although both return identical results for security types, the latter is a total binary operation on the set of security classes τ while on the other hand our $\oplus$ operator is a partial binary

[2]The primary reason for specifying a separate variable data type is for future purposes when support for references will be added to SecreC.

operation on the set of abstract security types. Currently, we define the exact domain of $\oplus$ only to include security classes:

$$\oplus : \rho^\tau \times \rho^\tau \to \rho^\tau$$

$$\rho_1^\tau \oplus \rho_2^\tau = \begin{cases} \texttt{public} & \text{if } \rho_1^\tau = \texttt{public} \text{ and } \rho_2^\tau = \texttt{public} \\ \texttt{private} & \text{if } \rho_1^\tau = \texttt{private} \text{ and } \rho_2^\tau = \texttt{public} \\ & \quad \text{or } \rho_1^\tau = \texttt{public} \text{ and } \rho_2^\tau = \texttt{private} \\ & \quad \text{or } \rho_1^\tau = \texttt{private} \text{ and } \rho_2^\tau = \texttt{private} \end{cases}$$

The relation $\rightarrow$ between security types defines the allowed direction of data flow. For our security types, only the following hold:

$$\texttt{private} \rightarrow \texttt{private}$$
$$\texttt{public} \rightarrow \texttt{public}$$
$$\texttt{public} \rightarrow \texttt{private}$$

meaning that data can flow in its own domain, and public data can flow into the private domain.

3.1.2 Statement types

The middle layer of the type system for static checking deals with statements. All statements in SecreC programs are contained in procedure bodies. Some statements, like **if**-statements, loop constructs and compound statements (statement blocks) may also contain other statements. For this reason, we first define four properties for statements:

- $fallthru$ - the execution of the statement may end without reaching a **return**, **break** or **continue** statement.
- *return* - the execution of the statement may end because of reaching a **return** statement.
- *break* - the execution of the statement may end because of reaching a **break** statement.
- *continue* - the execution of the statement may end because of reaching a **continue** statement.

The type of a specific statement is determined by the set of properties that hold for that statement. We denote the set of statement types with the meta-variable Γ_S:

$$\Gamma_S \in \mathcal{P}(\{fallthru, return, break, continue\})$$

3.1.3 Program types

The higher layer of the type system for static checking deals with whole programs. In this higher layer of the type system we denote the types of programs with the meta-variable Γ_P. Currently, we are only interested in whether a program is well-typed or not. Therefore, we only allow valid programs to by typeable:

$$\Gamma_P \in \{prog\}$$

where *prog* is the type of all well-typed programs.

3.2 Typing rules

Before presenting individual typing rules, we start by defining mangled identifiers and a type environment, and describe how procedure overloading is handled. The typing rules used for static checking are written as inference rules with a syntax similar to [?] which has served as a basis for inspiration.

3.2.1 Identifiers and procedure overloading

In addition to identifiers id_S returned by the scanner (i.e. `IDENTIFIER` tokens in Chapter 2), we define the entire set of identifiers id also to include *mangled* identifiers from a set id_M and a special identifier `thisproc`. A mangled identifier for a procedure contains the name of the procedure and information about the data types of its parameters. Mangled identifiers are used to emulate procedure overloading. The special identifier `thisproc` is used to denote the type of the procedure being processed[3]. The concrete set of elements in id_M as well as the special identifier `thisproc` are implementation-specific, but they are both required to be distinct from all possible identifiers returned by the scanner, and distinct from each other. Hence, the whole set of identifiers id can be formalized as

$$\mathrm{id} = \mathrm{id}_S \uplus \mathrm{id}_M \uplus \{\texttt{thisproc}\}$$

For procedure overloading, we define a bijective function $\mathcal{M}$ for procedure name mangling, which takes an identifier as returned by the scanner and the list of data types of the procedure parameters, and returns a mangled identifier:

$$\mathcal{M} : \mathrm{id}_S \times (\delta_1 \times \ldots \times \delta_n) \rightarrow \mathrm{id}_M$$

The procedure overloading mechanism in this paper allows several procedures with the same name to be present in SecreC programs under some conditions. Namely, for each two procedures with the same name, they must either take a different number of parameters, or if they both take n parameters then the data type of the i^{th} parameter must differ, where $n > 0$ and $i \in \{1 \ldots n\}$. This is a reasonable way to solve the ambiguity introduced by [?] which also allows overloading on security types in the context where the security types of the arguments can implicitly change.

3.2.2 Type environment

According to the SecreC grammar in Chapter 2, there are only three distinct places where lookup or checking for the type for identifiers is necessary: when assigning to variables in ⟨`assignment_expression`⟩, when reading the variables in ⟨`primary_expression`⟩ and when calling procedures ⟨`postfix_expression`⟩. Every identifier typed by the programmer, should only either refer to a variable, or to one or more procedures.

According to the scoping rules formally defined later, for all identifiers i, a definition of procedure i irreversibly hides any previous definitions of global variables i. A definition of global variable

[3]Note that additional special identifiers may also be added to the set of identifiers for the purpose of providing context-sensitive information for inference rules defined in this chapter and also in the semantics.

i shadows all previous definitions of i, including the definitions of procedures i. However, when defining another procedure i, any previous procedure definitions are again *un-shadowed* while the global variable definition of i is hidden. Similar rules are set for procedure parameters and local variable definitions, which also hide all previous definitions for their scope.

Let γ be a type environment, a partial function from the set of all identifiers to the set of types:

$$\gamma : \mathrm{id} \hookrightarrow \rho \uplus \{\texttt{proc}\}$$

where `proc` denotes that the identifier given to γ points to some procedure. The rationale behind this is that when in some scope the last definition of the identifier $i \in \mathrm{id}_S$ is a procedure definition, i.e. when $\gamma(i) = \texttt{proc}$ for the type environment γ of that scope, then γ also returns the full type of the procedure (including the security types of the parameters and the return type) when given some mangled identifier of the procedure.

Updating a type environment γ with a definition of x having type t, where $t \in \rho \uplus \{\texttt{proc}\}$, is denoted by $\gamma[x \mapsto t]$ and defined by the following equation:

$$\gamma[x \mapsto t]\,(x') = \begin{cases} t & \text{if } x' = x \\ \gamma(x') & \text{if } x' \neq x \end{cases}$$

Updating the type environment is what hides or shadows (or un-shadows) any previous definitions of the given identifier. When defining a new procedure denoted by the identifier i, one needs to update the type environment twice: first to point the identifier i to `proc`, and secondly to point the mangled i to the actual type of the procedure.

3.2.3 Checking expressions

In this section, we denote expressions with e, variables with x, procedure names with f and values with the meta-variable v.

The ηEConst rule defines the type for explicit values typed by the programmer such as decimal or string literals and keywords **true** and **false** which correspond to boolean constants:

$$\eta\text{EConst}\;\frac{\mathrm{validLiteral}(v)}{\gamma \vdash v : (\textbf{public}, \mathrm{datatype}(v))}$$

where function datatype(v) is the data type of value or literal v, and the predicate validLiteral checks whether the given literal is a valid string, integer, unsigned integer or boolean literal. For string literals the predicate validLiteral checks that there are no invalid or incomplete escape sequences in the string. For integer and unsigned integer literals it checks whether the corresponding decimal value fits into the corresponding 32-bit signed or unsigned integer data type. The function datatype is defined as follows:

$$\mathrm{datatype}(v) = \begin{cases} \textbf{string} & \text{if } v \text{ is a string literal} \\ \textbf{bool} & \text{if } v \text{ is } \textbf{true} \text{ or } \textbf{false} \\ \textbf{unsigned int} & \text{if } v \text{ is a integer literal} \\ \textbf{int} & \text{if } v \text{ is an unsigned integer literal} \end{cases}$$

For variables appearing as *r-values*, the ηERValue can be used, which infers the type of the value held by some variable.

$$\eta\text{ERValue}\,\frac{\gamma(x) = (\tau, \delta\ var)}{\gamma \vdash x : (\tau, \delta)}$$

For ternary expressions, the rule ηETernary requires the conditional expression to be of type (**public**, **bool**) to do the branching. The data types of both branches of the ternary expression are required to be fundamental data types and equal, and the security type of the ternary expression depends on the security types of the branches.

$$\eta\text{ETernary}\,\frac{\gamma \vdash e : (\textbf{public}, \textbf{bool}) \quad \gamma \vdash e_1 : (\tau_1, \delta) \quad \gamma \vdash e_2 : (\tau_2, \delta) \quad \tau = \tau_1 \oplus \tau_2}{\gamma \vdash e \texttt{ ? } e_1 \texttt{ : } e_2 : (\tau, \delta)}$$

For a ternary expression a simple optimization is to calculate one branch, depending on the value of the conditional. Since directing the control flow on **private** conditions causes side-channel information leaks in Sharemind, we require the ternary conditional always to be **public** to avoid ambiguity which would arise from different semantic rules depending on the security type of the conditional. Namely, it is in some cases possible to have safe ternary expressions with **private** conditional expressions, but its semantics would require both branches to be calculated regardless of the value of the conditional.

According to ηEBinary, the data type of binary expressions depends on the types of the operands. As in ηETernary, the subexpressions are required to have fundamental data types, and the security type of the binary expression is the safest security type of the subexpressions. The partial function $\text{BOT}_{\circledast}$ returns the data type for all binary operations $\circledast \in \{$`||`, `&&`, `==`, `!=`, `<`, `<=`, `>=`, `>`, `+`, `-`, `*`, `/`, `%`$\}$ and is tabulated in Appendix A. Note, that for some operators both the allowed security types and data types differ from [?].

$$\eta\text{EBinary}\,\frac{\gamma \vdash e_1 : (\tau_1, \delta_1) \quad \gamma \vdash e_2 : (\tau_2, \delta_2) \quad \tau = \tau_1 \oplus \tau_2 \quad \delta = \text{BOT}_{\circledast}(\delta_1, \delta_2)}{\gamma \vdash e_1 \circledast e_2 : (\tau, \delta)}$$

In rule ηEAssignOp for regular assignment expressions, we just require the variable we assign to (the l-value) to hold values of the same type as the expression (r-value) we assign to it.

$$\eta\text{EAssignOp}\,\frac{\gamma(x) = (\tau, \delta\ var) \quad \gamma \vdash e : (\tau', \delta) \quad \tau' \to \tau}{\gamma \vdash x \ = e : (\tau, \delta)}$$

For arithmetic assignment expressions that perform calculations based on the current value of the variable, ηEAssignOp2 requires that the corresponding binary arithmetic operation between the l-value and the r-value is typeable, and that the variable being assigned to can hold the result of this operation.

$$\eta\text{EAssignOp2}\,\frac{\gamma(x) = (\tau, \delta\ var) \quad \gamma \vdash x \circledast e : (\tau', \delta) \quad \tau' \to \tau}{\gamma \vdash x \circledast\!= e : (\tau, \delta)}$$

where $\circledast \in \{$`+`, `-`, `*`, `/`, `%`$\}$.

Typing rules for unary operators only require certain data types for its operands, namely the ηENot rule for logical negation and the ηENeg for numerical negation require their operands to be of data types **bool** or **int** respectively.

$$\eta\text{ENot}\frac{\gamma \vdash e : \left(\tau, \texttt{bool}\right)}{\gamma \vdash \texttt{!}e : \left(\tau, \texttt{bool}\right)} \qquad \eta\text{ENeg}\frac{\gamma \vdash e : \left(\tau, \texttt{int}\right)}{\gamma \vdash \texttt{-}e : \left(\tau, \texttt{int}\right)}$$

The ηEDeclassify rule is used to convert **private** data into **public** data. Without this rule it is impossible to transfer any information from **private** values to **public** variables or output. By requiring the argument to be of **private** security type, expressions like **declassify(declassify(e))** are not possible.

$$\eta\text{EDeclassify}\frac{\gamma \vdash e : (\texttt{private}, \delta)}{\gamma \vdash \texttt{declassify(}e\texttt{)} : (\texttt{public}, \delta)}$$

For procedure calls, the ηEProcCall rule first checks whether the procedure identifier f is of type proc according to the type environment, that all the given arguments type-check properly, and have value types. After this, it must be checked whether the mangled identifier f is in the type environment and whether the security types for the arguments can satisfy the security types for the procedure parameters. These checks are part of the procedure overloading mechanism. The result type for the procedure call can also be calculated from the procedure type returned from the type environment by its mangled identifier.

$$\eta\text{EProcCall}\frac{\begin{array}{c}\gamma \vdash f : \texttt{proc} \\ \gamma \vdash e_1 : (\tau_1, \delta_1) \\ \vdots \\ \gamma \vdash e_n : (\tau_n, \delta_n) \\ \gamma \vdash \mathcal{M}(f, \delta_1, \ldots, \delta_n) : \left(\left(\underline{\tau_1} \times \ldots \times \underline{\tau_n}\right) \to \tau, (\delta_1 \times \ldots \times \delta_n) \to \delta\right) \\ \tau_1 \to \underline{\tau_1} \\ \vdots \\ \tau_n \to \underline{\tau_n}\end{array}}{\gamma \vdash f\texttt{(}e_1\texttt{, }\ldots\texttt{,}e_n\texttt{)} : \rho}$$

where

$$\rho = \begin{cases} (\rho^\tau, \rho^\delta) & \text{if} \quad \rho^\tau \neq () \quad \text{and} \quad \rho^\delta \neq () \\ \texttt{void} & \text{if} \quad \rho^\tau = () \quad \text{and} \quad \rho^\delta = () \end{cases}$$

In the following sections, let the predicate goodexpr(γ, e) check whether the expression e properly type-checks in the type environment γ, i.e. that there exists some ρ so that $\gamma \vdash e : \rho$.

3.2.4 Checking statements

Statements are checked in a sequential manner – one after the other – using the meta-variable S to denote any following statements. For some rules, however, no following statements are allowed. The type for any set of sequential statements denotes the set of possible results for executing these statements. For example, for an **if** statement where one branch always ends with a **return** and the other branch always ends with a **break**, then the type of that **if** statement would be $\{break, return\}$, and static checking does not allow any statements to directly follow that **if** statement.

The static checking rules ηSBreak for **break** statements, ηSContinue for **continue** statements, ηSReturn and ηSReturnVoid for **return** statements don't allow any statements to follow. This forces the programmer not to write any unreachable code directly after these statements. For **return** statements, an additional check for the return type is done. The type of the empty statement according to ηSEmpty is $\emptyset$.

$$\eta\text{SBreak}\ \frac{}{\gamma \vdash \texttt{break;} : \{break\}} \qquad \eta\text{SContinue}\ \frac{}{\gamma \vdash \texttt{continue;} : \{continue\}}$$

$$\eta\text{SReturn}\ \frac{\begin{array}{c}\gamma \vdash e : (\tau', \delta) \\ \gamma(\texttt{thisproc}) = ((\tau_1, \ldots, \tau_n) \to \tau, (\delta_1, \ldots, \delta_n) \to \delta) \\ \tau' \to \tau\end{array}}{\gamma \vdash \texttt{return}\ e\texttt{;} : \{return\}}$$

$$\eta\text{SReturnVoid}\ \frac{\gamma(\texttt{thisproc}) = ((\tau_1, \ldots, \tau_n) \to (), (\delta_1, \ldots, \delta_n) \to ())}{\gamma \vdash \texttt{return;} : \{return\}}$$

$$\eta\text{SEmpty}\ \frac{}{\gamma \vdash \varepsilon\texttt{;} : \emptyset}$$

For compound statements, three different cases have to be considered. The first two cases deal with local variable definitions followed by statements in the type environment updated with the new variable type. Each local variable definition can be viewed as a definition of a new scope for variables. Therefore, the rules ηSCompoundVarDef and ηSCompoundVarDefInit only allow variable definitions to be followed by at least one non-empty statement. The opposite case – to define the variable and then just discard it together with the scope – would make no sense, since one could instead just use an expression statement. In the ηSCompoundVarDefInit rule, the type of the initializer expression is also checked to correspond to the type of the variable being defined.

$$\eta\text{SCompoundVarDef}\ \frac{\gamma[x \mapsto (\tau, \delta\ var)] \vdash S : \Gamma_S \qquad \Gamma_S \neq \emptyset}{\gamma \vdash \tau\ \delta\ x\texttt{;}\ S : \Gamma_S}$$

$$\eta\text{SCompoundVarDefInit}\ \frac{\gamma \vdash e : (\tau, \delta) \qquad \gamma[x \mapsto (\tau, \delta\ var)] \vdash S : \Gamma_S \qquad \Gamma_S \neq \emptyset}{\gamma \vdash \tau\ \delta\ x = e\texttt{;}\ S : \Gamma_S}$$

The third case considers some regular statement following the rest of the compound statement. The type of the regular statement is expected to include $fallthru$, otherwise the rest of the statements would be unreachable code. The ηSCompound rule also does not allow empty statements (with statement type $\emptyset$) to precede any other statements.

$$\eta\text{SCompound}\ \frac{\gamma \vdash S : \Gamma_S \qquad fallthru \in \Gamma_S \qquad \gamma \vdash S' : \Gamma'_S}{\gamma \vdash S\ S' : (\Gamma_S \setminus \{fallthru\}) \cup \Gamma'_S}$$

For expression statements, it is only checked whether the expression is valid. The type for expression statements is defined just to be $\{fallthru\}$:

$$\eta\text{SExpr}\ \frac{\text{goodexpr}(\gamma, e)}{\gamma \vdash e : \{fallthru\}}$$

All guards (conditional expressions) in statements are checked to be of type (**public**, **bool**). The security type is enforced to be **public** to prevent information leakage from control flow in the underlying virtual machine. Let the predicate goodguard(γ, e) check whether the type of the given expression e is (**public**, **bool**) in the given type environment γ, i.e. that $\gamma \vdash e : (\textbf{public}, \textbf{bool})$ holds.

The type of **`if`** statements is determined by the types of its branches. First, it is required that the branches are not empty. If one or both branches are found of type $\emptyset$, the statement should be refactored by the programmer for code clarity. For **`if`** statements without an **`else`** branch, it is still possible that the statements in the first branch are not executed, therefore the type of the **`if`** statement must also contain $fallthru$. For **`if`** statements with both branches, the type of the whole **`if`** statement is just the set union of the types of its two branches.

$$\eta\text{SIf}\frac{\text{goodguard}(\gamma, e) \qquad \gamma \vdash S : \Gamma_S \qquad \Gamma_S \neq \emptyset}{\gamma \vdash \texttt{if (}e\texttt{)}\ S : \Gamma_S \cup \{fallthru\}}$$

$$\eta\text{SIfElse}\frac{\text{goodguard}(\gamma, e) \qquad \gamma \vdash S : \Gamma_S \qquad \Gamma_S \neq \emptyset \qquad \gamma \vdash S' : \Gamma'_S \qquad \Gamma'_S \neq \emptyset}{\gamma \vdash \texttt{if (}e\texttt{)}\ S\ \texttt{else}\ S' : \Gamma_S \cup \Gamma'_S}$$

To further enforce a good programming style, the types of the bodies of all loops in SecreC are required either to contain $fallthru$ or *continue* or be $\emptyset$, meaning that it might be possible for the loop to do more than one iteration. Otherwise, the loop can be substituted with acyclic code. The bodies of the loops also filter out *break* and *continue* from the type of the loop body. Hence, in principle, the type of a loop is $\{fallthru, return\}$ if the type of the body of the loop contains *return*, and $\{fallthru\}$ otherwise.

Let Γ_B denote the type of the loop body for all static checking rules for loops, and Γ_L denote the set $(\Gamma_B \setminus \{break, continue\}) \cup \{fallthru\}$, which is the type of the entire loop. Let the predicate goodbody(Γ_B) check whether $(\Gamma_B = \emptyset) \vee (\Gamma_B \cap \{continue, fallthru\} \neq \emptyset)$ holds for the given Γ_B.

The static checking rules for **`while`** and **`do-while`** loops are identical:

$$\eta\text{SWhile}\frac{\text{goodguard}(\gamma, e) \qquad \gamma \vdash S : \Gamma_B \qquad \text{goodbody}(\Gamma_B)}{\gamma \vdash \texttt{while (}e\texttt{)}\ S : \Gamma_L}$$

$$\eta\text{SDoWhile}\frac{\text{goodguard}(\gamma, e) \qquad \gamma \vdash S : \Gamma_B \qquad \text{goodbody}(\Gamma_B)}{\gamma \vdash \texttt{do}\ S\ \texttt{while (}e\texttt{)} : \Gamma_L}$$

The most complex loop in SecreC is the **`for`** loop. Since all the three expressions are optional, we have formulated a separate rule for each such case, and have a total of eight rules. If present, the first and last expressions are only checked for validity. The second expression is the loop guard, and is checked just like guards in other statements. When the guard is omitted, the respective rules ensure, that the type of the body of the loop contains either *break* or *return* to catch trivial cases of non-terminating loops:

$$\eta\text{SFor}\frac{\begin{array}{c}\text{goodexpr}(\gamma, e)\\ \text{goodguard}(\gamma, e')\\ \text{goodexpr}(\gamma, e'')\\ \gamma \vdash S : \Gamma_B\\ \text{goodbody}(\Gamma_B)\end{array}}{\gamma \vdash \texttt{for (}e\texttt{;}\ e'\texttt{;}\ e''\texttt{)}\ S : \Gamma_L} \qquad \eta\text{SFor2}\frac{\begin{array}{c}\text{goodguard}(\gamma, e')\\ \text{goodexpr}(\gamma, e'')\\ \gamma \vdash S : \Gamma_B\\ \text{goodbody}(\Gamma_B)\end{array}}{\gamma \vdash \texttt{for (;}\ e'\texttt{;}\ e''\texttt{)}\ S : \Gamma_L}$$

$$\eta\text{SFor3}\;\frac{\begin{array}{c}\text{goodexpr}(\gamma, e)\\ \text{goodexpr}(\gamma, e'')\\ \gamma \vdash S : \Gamma_B\\ \text{goodbody}(\Gamma_B)\\ \Gamma_B \cap \{\mathit{break}, \mathit{return}\} \neq \emptyset\end{array}}{\gamma \vdash \texttt{for (}e\texttt{;; }e''\texttt{)}\ S : \Gamma_L}$$

$$\eta\text{SFor4}\;\frac{\begin{array}{c}\text{goodexpr}(\gamma, e'')\\ \gamma \vdash S : \Gamma_B\\ \text{goodbody}(\Gamma_B)\\ \Gamma_B \cap \{\mathit{break}, \mathit{return}\} \neq \emptyset\end{array}}{\gamma \vdash \texttt{for (;; }e''\texttt{)}\ S : \Gamma_L}$$

$$\eta\text{SFor5}\;\frac{\begin{array}{c}\text{goodexpr}(\gamma, e)\\ \gamma \vdash S : \Gamma_B\\ \text{goodbody}(\Gamma_B)\\ \Gamma_B \cap \{\mathit{break}, \mathit{return}\} \neq \emptyset\end{array}}{\gamma \vdash \texttt{for (}e\texttt{;;)}\ S : \Gamma_L}$$

$$\eta\text{SFor6}\;\frac{\begin{array}{c}\gamma \vdash S : \Gamma_B\\ \text{goodbody}(\Gamma_B)\\ \Gamma_B \cap \{\mathit{break}, \mathit{return}\} \neq \emptyset\end{array}}{\gamma \vdash \texttt{for (;;)}\ S : \Gamma_L}$$

$$\eta\text{SFor7}\;\frac{\begin{array}{c}\text{goodexpr}(\gamma, e)\\ \text{goodguard}(\gamma, e')\\ \gamma \vdash S : \Gamma_B\\ \text{goodbody}(\Gamma_B)\end{array}}{\gamma \vdash \texttt{for (}e\texttt{; }e'\texttt{;)}\ S : \Gamma_L}$$

$$\eta\text{SFor8}\;\frac{\begin{array}{c}\text{goodguard}(\gamma, e')\\ \gamma \vdash S : \Gamma_B\\ \text{goodbody}(\Gamma_B)\end{array}}{\gamma \vdash \texttt{for (; }e'\texttt{;)}\ S : \Gamma_L}$$

3.2.5 Checking programs

Similarly to the techniques described in Subsection 3.2.4, the static checking of programs is also performed in a sequential manner – on a definition-by-definition basis. This is done by checking each global variable or procedure definition by itself and then checking the rest of the program in a type environment updated with all previous definitions. Checking starts with a predefined type environment γ_0 which may also be empty, i.e. with an empty domain, meaning that there are no predefined variables or procedures.

We denote the domain of a (partial) function f by $\text{dom}(f)$, and use the meta-variable P to denote program fragments.

For global variable definitions without initializer expressions, only the type environment γ is updated with the variable being defined, and the rest of the program is checked in the updated type environment:

$$\eta\text{VarDef}\;\frac{\gamma[x \mapsto (\tau, \delta\ \mathit{var})] \vdash P : \Gamma_P}{\gamma \vdash \tau\ \delta\ x\texttt{;}\ P : \Gamma_P}$$

Static checking of global variable definitions with initializer expressions is performed as without initializer expressions, with the only exception, that the initializer expression is first checked in the type environment before updating it.

$$\eta\text{VarDefInit}\;\frac{\gamma \vdash e : (\tau, \delta) \qquad \gamma[x \mapsto (\tau, \delta\ \mathit{var})] \vdash P : \Gamma_P}{\gamma \vdash \tau\ \delta\ x = e\texttt{;}\ P : \Gamma_P}$$

For procedure definitions, it is first verified that the type environment does not already contain a procedure with the same signature (its mangled identifier). Secondly, it is verified that no two names for the procedure parameters are equal, ensuring that the parameters do not hide each other. Thirdly, the body of the function is type-checked to verify its internal structure and that it always returns a value of the correct type (or falls through without returning in case the return type is **`void`**). Finally, the program following this definition is checked in the type environment updated with this procedure.

An updated type environment is used for type-checking the procedure body. This type environment is not only updated with the definition of the current procedure (to allow recursion), but also with the procedure parameters. To type-check all the return statements in the function body we also extend the type environment using `thisproc` to point to the type of the enclosing procedure. For procedures returning a value in rule ηProcdef, the body of the procedure must be of statement type *return*, while for **`void`** procedures in rule ηProcdefVoid, the body of the procedure might also be of type *fallthru*. An additional check ensures that a procedure named `main` taking no parameters can only be of **`void`** return type.

$$\eta\text{Procdef}\;\frac{\begin{array}{c}\mathcal{M}^f \notin \mathrm{dom}(\gamma)\\ |\{p_1,\ldots,p_n\}| = n\\ (f \neq \texttt{main}) \vee (n > 0)\\ \gamma_c \vdash S : \{\mathit{return}\}\\ \gamma' \vdash P : \Gamma_P\end{array}}{\gamma \vdash \tau\ \delta\ f\texttt{(}\tau_1\ \delta_1\ p_1\texttt{,}\ \ldots\texttt{,}\ \tau_n\ \delta_n\ p_n\texttt{)}\ \texttt{\{}S\texttt{\}}\ P : \Gamma_P}$$

$$\eta\text{ProcdefVoid}\;\frac{\begin{array}{c}\mathcal{M}^f \notin \mathrm{dom}(\gamma)\\ |\{p_1,\ldots,p_n\}| = n\\ \gamma_c \vdash S : \Gamma_S\\ \Gamma_S \subseteq \{\mathit{fallthru}, \mathit{return}\}\\ \gamma' \vdash P : \Gamma_P\end{array}}{\gamma \vdash \textbf{\texttt{void}}\ f\texttt{(}\tau_1\ \delta_1\ p_1\texttt{,}\ \ldots\texttt{,}\ \tau_n\ \delta_n\ p_n\texttt{)}\ \texttt{\{}S\texttt{\}}\ P : \Gamma_P}$$

where

$$\begin{aligned}\mathcal{M}^f &= \mathcal{M}(f, \delta_1, \ldots, \delta_n)\\ \gamma' &= \gamma[f \mapsto \texttt{proc}]\left[\mathcal{M}^f \mapsto \rho\right]\\ \gamma_c &= \gamma'[\texttt{thisproc} \mapsto \rho][p_1 \mapsto (\tau_1, \delta_1)]\cdots[p_n \mapsto (\tau_n, \delta_n)]\\ \rho &= \begin{cases}((\tau_1 \times \ldots \times \tau_n) \to \tau, (\delta_1 \times \ldots \times \delta_n) \to \delta) & \text{for Procdef}\\ ((\tau_1 \times \ldots \times \tau_n) \to (), (\delta_1 \times \ldots \times \delta_n) \to ()) & \text{for ProcdefVoid}\end{cases}\end{aligned}$$

After checking whether all the top-level definitions in the SecreC program are correct, static checking verifies that a function called "main" taking no parameters and returning no value exists:

$$\eta\text{ProgEnd}\;\frac{\gamma(\mathcal{M}(\texttt{main}, ())) = (() \to (), () \to ())}{\gamma \vdash \varepsilon : \mathit{prog}}$$

4 Natural semantics

For the natural semantics (or big-step semantics), we use $\rightarrow$ to denote different transitions of state, and $\rightarrow^*$ to denote these transitions being applied one or multiple times. Transitions for evaluating expressions are denoted as $\rightarrow_E$, transitions for evaluating defining variables and procedures are denoted as $\rightarrow_{DV}$ and $\rightarrow_{DP}$ respectively, transitions for evaluating statements are denoted as $\rightarrow_S$, and the transitions for evaluating the top level of the program are denoted by $\rightarrow$ only.

In this section, we first describe a more flexible way to handle variables, and then continue to present evaluation rules for SecreC expressions, variable definitions, statements, procedure definitions and the program as a whole.

4.1 Locations, stores and environments

In SecreC, variables hold values of certain types according to the type of the variable itself. In this section we extend this notion with *locations* and *stores* in the fashion described in [**?**, **?**]. First of all, let us consider *locations* which can be thought of as memory addresses, each of them pointing to a single memory cell capable of holding exactly one value of any type. Depending on the security type of the data held in the cell, we can partition the set of locations (loc) into private locations loc_{pri} and public locations loc_{pub}. A special **null** location is used for the location returned by **void** procedures, meaning "points to nothing":

$$\mathrm{loc} = \mathrm{loc}_{pub} \cup \mathrm{loc}_{pri} \cup \{\mathbf{null}\}$$

We also define a function getloc(l) which returns the type of the location l given to it:

$$\mathrm{getloc}(l) = \begin{cases} \mathrm{loc}_{pri} & \text{if } l \in \mathrm{loc}_{pri} \\ \mathrm{loc}_{pub} & \text{if } l \in \mathrm{loc}_{pub} \\ \{\mathbf{null}\} & \text{if } l = \mathbf{null} \end{cases}$$

Let an abstract *store* denoted by μ be a partial function from the set of *locations* to the set of values in each of these locations:

$$\mu : \mathrm{loc} \hookrightarrow \mathrm{val}$$

Let the function $\mathrm{newloc}_{pub}\,(\mu)$ return a new public location, that is not yet in the domain of μ and let the function $\mathrm{newloc}_{pri}\,(\mu)$ return a new private location, that is not in the domain of μ. We also define a helper function named newloc:

$$\mathrm{newloc}(\tau, \mu) = \begin{cases} \mathrm{newloc}_{pub}(\mu) & \text{if } \tau = \mathbf{public} \\ \mathrm{newloc}_{pri}(\mu) & \text{if } \tau = \mathbf{private} \end{cases}$$

We define another partial function, the variable environment named env_V that, for each scanner identifier denoting a variable, stores the location of the value for that variable:

$$\mathrm{env}_V : \mathrm{id}_S \hookrightarrow \mathrm{loc}$$

Given the store μ and variable environment env_V, variables in SecreC can now be considered to denote locations in store which hold the actual value of the variable. Formally, the value of variable

x is $\mu(\mathrm{env}_V(x))$.

Besides providing means for specifying rules for declassification and classification in subsection 4.2.1, the notion of locations and stores is currently not of much use in the semantics. However, it will yield a more seamless transition in the future, if formal support for arrays and references is added to SecreC.

We also define a procedure environment env_P to be a partial function from the set of all mangled identifiers to the set of procedure definitions:

$$\mathrm{env}_P : \mathrm{id}_M \hookrightarrow \left((p_1, \dots, p_n), S, \gamma', \mathrm{env}'_V, \mathrm{env}'_P\right)$$

Procedure definitions are in the form of 5-tuples $\left((p_1, \dots, p_n), S, \gamma', \mathrm{env}'_V, \mathrm{env}'_P\right)$, where $p_1, \dots, p_n$ are the identifiers of the procedure parameters, S is the body of the procedure. γ', env'_V and env'_P are respectively the type environment, the variable environment and the procedure environment for the procedure body at the place of the procedure definition.

In Subsection 3.2.2 we have defined the type environment γ and have shown how to update it. We now define updating the environments env_V and env_P, and the store μ analogously, using the following three equations:

$$\mathrm{env}_V[x \mapsto c](x') = \begin{cases} c & \text{if } x' = x \\ \mathrm{env}_V(x') & \text{if } x' \neq x \end{cases} \qquad \mathrm{env}_P[f \mapsto d](f') = \begin{cases} d & \text{if } f' = f \\ \mathrm{env}_P(f') & \text{if } f' \neq f \end{cases}$$

$$\mu[l \mapsto v](l') = \begin{cases} v & \text{if } l' = l \\ \mu(l') & \text{if } l' \neq l \end{cases}$$

4.2 Evaluation of expressions

In SecreC, no new variables and procedures can be defined or redefined by expressions alone. Therefore, the environments γ, env_V and env_P are constant during the evaluation of expressions. Of course, expressions may be viewed as implicitly creating some temporary variables, but since those are not needed to be accessible from outside the expressions, it is not necessary to add them to the these environments at this point.

However, the values of variables can be changed, meaning that store μ is mutable, due to procedure calls and assignment expressions. Hence, state transitions for expressions given in the form of

$$\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu \rangle \to_E \langle l, \mu' \rangle$$

denote that in the environments γ, env_V and env_P, and for the initial state of the store μ, evaluation of expression e results in the changed store μ' where the value corresponding to the expression is stored at location l. If the evaluation of expression e results in no value (e.g. when e is a procedure call to a procedure with the return type of **void**) then l is **null**.

For unary, arithmetic and assignment expressions, we use a helper function $\mathcal{E}$ which, given an expression with values to operate on, returns the resulting value of the expression. For operands of type **bool**, $\mathcal{E}$ works as in the C++ programming language[?] for all the defined operations (`!`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `&&` and `||`). When both operands v_1 and v_2 are of type **string** then $\mathcal{E}(v_1 + v_2)$ returns a concatenated string $v_1 v_2$. For binary `==` and `!=` on strings, function $\mathcal{E}$ returns whether given strings are equal or unequal, for <, <=, >= and > on strings, function $\mathcal{E}$ returns whether the strings are respectively ordered in a lexicographical manner. For all other operations allowed by the SecreC type system in Subsection 3.2.3, the result of $\mathcal{E}$ of some operation is identical to the result of the corresponding expression in C++, given that the SecreC types **unsigned** and **int** correspond to 32-bit **unsigned** and **int** types in C++.

4.2.1 Declassification and classification

Since all data is either stored in the private data space or the public data space in the Sharemind virtual machine, we need mechanisms to move data from one data space to the other. The declassification operator takes as input a private value, moves the value to public data space and returns it. To do so, a new public location $l' \in \text{loc}_{pub}$ must be utilized:

$$\text{EDeclassify}\ \frac{\gamma, \text{env}_V, \text{env}_P \vdash \langle e, \mu \rangle \to_E \langle l, \mu' \rangle \qquad l' = \text{newloc}_{pub}(\mu') \qquad \mu'' = \mu'[l' \mapsto \mu'(l)]}{\gamma, \text{env}_V, \text{env}_P \vdash \langle \mathtt{\textbf{declassify}(}e\mathtt{)}, \mu \rangle \to_E \langle l', \mu'' \rangle}$$

Contrary to declassification, classification of data (or moving data from public data space to private data space) is an implicit operation according to the grammar of expressions given in Section 2.7. Therefore, we cannot present syntax-directed rules for classification. In some rules in this section we have explicitly noted that depending on the security types of subexpressions classification might occur. For other rules in this section that perform implicit classification, we provide a new type of state transition and a number of transition rules. These state transitions are in the form of

$$\gamma \vdash \langle \tau, e, l, l', \mu, F \rangle \to_C \langle l'', \mu' \rangle$$

where τ is a security class, e is an expression, l, l' and l'' are locations, μ and μ' are stores and $F \in \{force, noforce\}$. It is presumed, that l holds the value of the expression e at some point during the evaluation, so that if if e is of a **public** security type, then $l \in \text{loc}_{pub}$, otherwise $l \in \text{loc}_{pri}$. If $F = noforce$, then location l' is the location where the classified value must be written to, should classification be needed. If $F = force$ then the value at l should be written to l' regardless of whether it was classified or not. The resulting state consists of a potentially changed store μ', and a location l'' which holds the classified value. Depending on F and whether classification was performed, l'' is either l or l'.

The EClassify rule denotes that if τ is **private**, but the expression e is of a **public** security type, then the value $\mu(l)$ should be classified. The classification occurs by copying the value to the location $l' \in \text{loc}_{pri}$ in store μ. The resulting state consists of l' and the changed store:

$$\text{EClassify}\ \frac{\gamma \vdash e : (\mathtt{\textbf{public}}, \delta) \qquad \mu' = \mu[l' \mapsto \mu(l)]}{\gamma \vdash \langle \mathtt{\textbf{private}}, e, l, l', \mu, F \rangle \to_C \langle l', \mu' \rangle}$$

In other cases, when the security type of expression e is equal to τ, classification is not needed. However, depending on the parameter F, we might still need to copy the value to l'. The EDontClassify rule deals with all these cases:

$$\text{EDontClassify}\ \frac{\gamma \vdash e : (\tau, \delta)}{\gamma \vdash \langle \tau, e, l, l', \mu, F \rangle \to_C \langle l'', \mu' \rangle}$$

where

$$l'' = \begin{cases} l & \text{if } F = noforce \\ l' & \text{if } F = force \end{cases}$$

$$\mu' = \begin{cases} \mu & \text{if } F = noforce \\ \mu[l' \mapsto \mu(l)] & \text{if } F = force \end{cases}$$

4.2.2 Variables and constants

For identifiers x in expressions where x is not used in a procedure call as the procedure identifier, x must appear in the type environment and denote a variable. This is statically checked at compilation time by the ηERValue rule in Subsection 3.2.3. The ERValue rule only queries the location of the value of x is from env_V:

$$\text{ERValue}\frac{l = \text{env}_V(x)}{\gamma, \text{env}_V, \text{env}_P \vdash \langle x, \mu\rangle \rightarrow_E \langle l, \mu\rangle}$$

For literal values v, we currently have a rule only for completeness. Namely, because all other expression rules we give in this chapter operate on locations rather than values, it is simplest for us to specify the following rule:

$$\text{EConst}\frac{l = \text{newloc}_{pub}(\mu) \qquad \mu' = \mu[l \mapsto v]}{\gamma, \text{env}_V, \text{env}_P \vdash \langle v, \mu\rangle \rightarrow_E \langle l, \mu'\rangle}$$

It must, of course, be noted, that in the output of decent compilers this change of store (or memory) is optimized out, because they usually output instructions, that take constants as immediate operands and do not need constant values to appear at certain locations in store.

4.2.3 Simple expressions

In unary expressions, first the underlying subexpression is evaluated, and then the corresponding unary operation $\circledast \in \{\texttt{-}, \texttt{!}\}$ is done on the value of the evaluated subexpression using the helper function $\mathcal{E}$ and the result is stored in a new memory location $l' \in \text{getloc}(l)$:

$$\text{EUnary}\frac{\begin{array}{c}\gamma \vdash e : (\tau, \delta) \\ \gamma, \text{env}_V, \text{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle \\ l' = \text{newloc}(\tau, \mu') \\ \mu'' = \mu'[l' \mapsto \mathcal{E}(\circledast(\mu'(l)))]\end{array}}{\gamma, \text{env}_V, \text{env}_P \vdash \langle \circledast e, \mu\rangle \rightarrow_E \langle l', \mu''\rangle}$$

For all arithmetic binary operations $\circledast \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}$ the operands are first evaluated in order from left to right. Secondly, if needed, classification is performed on one of them. Finally, the corresponding binary operation is executed on the (potentially classified) values and the result is stored in a new memory location $l'' \in L$ where $L = \text{getloc}(l_c) = \text{getloc}(l'_c)$:

$$\text{EArithBinary}\ \dfrac{\begin{array}{c}\gamma \vdash e : (\tau, \delta) \\ \gamma \vdash e' : (\tau', \delta') \\ \gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu \rangle \rightarrow_E \left\langle l, \mu^{(1)} \right\rangle \\ \gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \left\langle e', \mu^{(1)} \right\rangle \rightarrow_E \left\langle l', \mu^{(2)} \right\rangle \\ \gamma \vdash \left\langle \tau', e, l, \mathrm{newloc}_{pri}\left(\mu^{(2)}\right), \mu^{(2)}, noforce \right\rangle \rightarrow_C \left\langle l_c, \mu^{(3)} \right\rangle \\ \gamma \vdash \left\langle \tau, e', l', \mathrm{newloc}_{pri}\left(\mu^{(3)}\right), \mu^{(3)}, noforce \right\rangle \rightarrow_C \left\langle l'_c, \mu^{(4)} \right\rangle \\ l'' = \mathrm{newloc}\left(\tau \oplus \tau', \mu^{(4)}\right) \\ \mu' = \mu^{(4)}\left[l'' \mapsto \mathcal{E}\left(\left(\mu^{(4)}(l_c)\right) \circledast \left(\mu^{(4)}(l'_c)\right)\right)\right]\end{array}}{\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e \circledast e', \mu \rangle \rightarrow_E \langle l'', \mu' \rangle}$$

4.2.4 Logical expressions

For binary logical operators `&&` and `||` two types of evaluation rules exist depending on the security type of the first subexpression. Namely, when the first subexpression is of public type, we can in some cases *shorthand* the logical expression. For example, in e_1 `&&` e_2 if the value of e_1 is found to be **`false`**, or if in e_1 `||` e_2 if the value of e_1 is found to be **`true`** there is no need to evaluate the e_2-s, because the result of the logical expression is already determined by the value of the first expression. However, at execution time this requires a control flow change depending on the value of the first subexpression. Because conditional branching on private conditions is prohibited in Sharemind to prevent side-channel information leakage, we only allow this kind of shorthand semantics if the security type for the first subexpression is **`public`**. If the security type for the first subexpression is **`private`**, we always need to evaluate the second subexpression. This issue is even more clear when expressions with side-effects (i.e. procedure calls and assignments) are taken into account. For example, the expression statement e_1 `&&` f`();` is equivalent to the statement **`if`** `(`e_1`)` f`();` in case of shorthand semantics.

Hence, for logical conjunction and logical disjunction where the first subexpression is of **`private`** security type, we evaluate both subexpressions from left to right and calculate the result as for arithmetic operations. The result is analogously stored in a new memory location $l'' \in \mathrm{loc}_{pri}$:

$$\text{EAndPrivate}\ \dfrac{\begin{array}{c}\gamma \vdash e : (\texttt{\textbf{private}}, \texttt{\textbf{bool}}) \\ \gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu \rangle \rightarrow_E \left\langle l, \mu^{(1)} \right\rangle \\ \gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \left\langle e', \mu^{(1)} \right\rangle \rightarrow_E \left\langle l', \mu^{(2)} \right\rangle \\ \gamma \vdash \left\langle \texttt{\textbf{private}}, e', l', \mathrm{newloc}_{pri}\left(\mu^{(2)}\right), \mu^{(2)}, noforce \right\rangle \rightarrow_C \left\langle l'', \mu^{(3)} \right\rangle \\ l''' = \mathrm{newloc}_{pri}\left(\mu^{(3)}\right) \\ \mu' = \mu^{(3)}\left[l''' \mapsto \mathcal{E}\left(\left(\mu^{(3)}(l)\right)\ \texttt{\&\&}\ \left(\mu^{(3)}(l'')\right)\right)\right]\end{array}}{\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e\ \texttt{\&\&}\ e', \mu \rangle \rightarrow_E \langle l''', \mu' \rangle}$$

$$\text{EOrPrivate}\ \dfrac{\begin{array}{c}\gamma \vdash e : (\texttt{\textbf{private}}, \texttt{\textbf{bool}}) \\ \gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu \rangle \rightarrow_E \left\langle l, \mu^{(1)} \right\rangle \\ \gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \left\langle e', \mu^{(1)} \right\rangle \rightarrow_E \left\langle l', \mu^{(2)} \right\rangle \\ \gamma \vdash \left\langle \texttt{\textbf{private}}, e', l', \mathrm{newloc}_{pri}\left(\mu^{(2)}\right), \mu^{(2)}, noforce \right\rangle \rightarrow_C \left\langle l'', \mu^{(3)} \right\rangle \\ l''' = \mathrm{newloc}_{pri}\left(\mu^{(3)}\right) \\ \mu' = \mu^{(3)}\left[l''' \mapsto \mathcal{E}\left(\left(\mu^{(3)}(l)\right)\ \texttt{||}\ \left(\mu^{(3)}(l'')\right)\right)\right]\end{array}}{\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e\ \texttt{||}\ e', \mu \rangle \rightarrow_E \langle l''', \mu' \rangle}$$

When the first subexpression has a **`public`** security type, we use the shorthand version. We first evaluate the first expression and if its resulting value is **`false`** for conjunction or **`true`** for disjunction we return the result without evaluating the second subexpression:

$$\text{EAnd}\;\frac{\begin{array}{c}\gamma \vdash e : (\texttt{public}, \texttt{bool})\\ \gamma \vdash e' : (\tau, \texttt{bool})\\ \gamma, \text{env}_V, \text{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle\\ \mu'(l) = \texttt{false}\\ \gamma \vdash \langle \tau, e, l, \text{newloc}_{pri}(\mu'), \mu', noforce\rangle \rightarrow_C \langle l_c, \mu''\rangle\end{array}}{\gamma, \text{env}_V, \text{env}_P \vdash \langle e \;\texttt{\&\&}\; e', \mu\rangle \rightarrow_E \langle l_c, \mu''\rangle}$$

$$\text{EOr}\;\frac{\begin{array}{c}\gamma \vdash e : (\texttt{public}, \texttt{bool})\\ \gamma \vdash e' : (\tau, \texttt{bool})\\ \gamma, \text{env}_V, \text{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle\\ \mu'(l) = \texttt{true}\\ \gamma \vdash \langle \tau, e, l, \text{newloc}_{pri}(\mu'), \mu', noforce\rangle \rightarrow_C \langle l_c, \mu''\rangle\end{array}}{\gamma, \text{env}_V, \text{env}_P \vdash \langle e \;\texttt{||}\; e', \mu\rangle \rightarrow_E \langle l_c, \mu''\rangle}$$

When the first subexpression is of **public** security type, but the result of evaluating the first expression is **true** for conjunction or **false** for disjunction, we also have to evaluate the second subexpression, and return its result as the result for the whole logical expression:

$$\text{EAnd2}\;\frac{\begin{array}{c}\gamma \vdash e : (\texttt{public}, \texttt{bool})\\ \gamma, \text{env}_V, \text{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle\\ \mu'(l) = \texttt{true}\\ \gamma, \text{env}_V, \text{env}_P \vdash \langle e', \mu'\rangle \rightarrow_E \langle l, \mu''\rangle\end{array}}{\gamma, \text{env}_V, \text{env}_P \vdash \langle e \;\texttt{\&\&}\; e', \mu\rangle \rightarrow_E \langle l, \mu''\rangle}$$

$$\text{EOr2}\;\frac{\begin{array}{c}\gamma \vdash e : (\texttt{public}, \texttt{bool})\\ \gamma, \text{env}_V, \text{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle\\ \mu'(l) = \texttt{false}\\ \gamma, \text{env}_V, \text{env}_P \vdash \langle e', \mu'\rangle \rightarrow_E \langle l, \mu''\rangle\end{array}}{\gamma, \text{env}_V, \text{env}_P \vdash \langle e \;\texttt{||}\; e', \mu\rangle \rightarrow_E \langle l, \mu''\rangle}$$

It might not be intuitively clear whether or not the shorthand version is used in logical expressions. Therefore, programmers writing SecreC programs must exercise caution regarding this.

4.2.5 Ternary operation

The ternary operation in SecreC is an expression taking a conditional expression of type (**public**, **bool**) and two other subexpressions with the same data type (as described in Subsection 3.2.3). First, the conditional expression is evaluated, and if its value was **true**, then the second subexpression is evaluated and its value is returned:

$$\text{ETernaryTrue}\;\frac{\begin{array}{c}\gamma, \text{env}_V, \text{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle\\ \mu'(l) = \texttt{true}\\ \gamma, \text{env}_V, \text{env}_P \vdash \langle e', \mu'\rangle \rightarrow_E \langle l', \mu''\rangle\\ \gamma \vdash e'' : (\tau, \delta)\\ \gamma \vdash \langle \tau, e', l', \text{newloc}_{pri}(\mu''), \mu'', noforce\rangle \rightarrow_C \langle l_c, \mu'''\rangle\end{array}}{\gamma, \text{env}_V, \text{env}_P \vdash \langle e \;\texttt{?}\; e' \;\texttt{:}\; e'', \mu\rangle \rightarrow_E \langle l_c, \mu'''\rangle}$$

Otherwise, if the value of the conditional expression evaluated to **false**, then the third subexpression is evaluated and its value is returned:

$$\text{ETernaryFalse}\,\frac{\begin{array}{c}\gamma, \text{env}_V, \text{env}_P \vdash \langle e, \mu \rangle \rightarrow_E \langle l, \mu' \rangle \\ \mu'(l) = \texttt{false} \\ \gamma, \text{env}_V, \text{env}_P \vdash \langle e'', \mu' \rangle \rightarrow_E \langle l'', \mu'' \rangle \\ \gamma \vdash e' : (\tau, \delta) \\ \gamma \vdash \langle \tau, e'', l'', \text{newloc}_{pri}(\mu''), \mu'', \mathit{noforce} \rangle \rightarrow_C \langle l_c, \mu''' \rangle\end{array}}{\gamma, \text{env}_V, \text{env}_P \vdash \langle e \text{ ? } e' \text{ : } e'', \mu \rangle \rightarrow_E \langle l_c, \mu''' \rangle}$$

4.2.6 Assignment expressions

For regular assignment expressions, we first evaluate the expression being assigned, then assign its value to the variable being assigned to. To do this, we change the value of the variable in the store at the location we get from the variable environment env_V. Depending on the security type of the variable and the security type of the expression, we may also have to classify the value before assignment.

$$\text{EAssign}\,\frac{\begin{array}{c}\gamma(x) = (\tau, \delta\ \mathit{var}) \\ \gamma, \text{env}_V, \text{env}_P \vdash \langle e, \mu \rangle \rightarrow_E \langle l, \mu' \rangle \\ \gamma \vdash \langle \tau, e, l, \text{env}_V(x), \mu', \mathit{force} \rangle \rightarrow_C \langle l', \mu'' \rangle\end{array}}{\gamma, \text{env}_V, \text{env}_P \vdash \langle x \texttt{ = } e, \mu \rangle \rightarrow_E \langle l', \mu'' \rangle}$$

For arithmetic assignment expressions, before doing the actual assignment, we exploit the EArithBinary rule to perform the arithmetic operation and any needed classification:

$$\text{EAssignArith}\,\frac{\gamma, \text{env}_V, \text{env}_P \vdash \langle x \circledast e, \mu \rangle \rightarrow_E \langle l, \mu' \rangle}{\gamma, \text{env}_V, \text{env}_P \vdash \langle x \ \circledast\!\texttt{=}\ e, \mu \rangle \rightarrow_E \langle l, \mu'[\text{env}_V(x) \mapsto \mu'(l)] \rangle}$$

4.2.7 Procedure calls

The inference rule for the semantics of procedure calls in SecreC is notationally the most complex rule in these semantics. For readability, we have boxed and numbered the premises of the inference rule into five distinct parts:

$$\text{EProc}\ \dfrac{\begin{array}{c}
1.\ \boxed{\begin{array}{c} \gamma, \text{env}_V, \text{env}_P \vdash \langle e_1, \mu \rangle \to_E \left\langle l^{(1)}, \mu^{(1)} \right\rangle \\ \vdots \\ \gamma, \text{env}_V, \text{env}_P \vdash \left\langle e_n, \mu^{(n-1)} \right\rangle \to_E \left\langle l^{(n)}, \mu^{(n)} \right\rangle \end{array}} \\
2.\ \boxed{\begin{array}{c} \gamma \vdash e_1 : (\tau_a^{(1)}, \delta_a^{(1)}) \\ \vdots \\ \gamma \vdash e_n : (\tau_a^{(n)}, \delta_a^{(n)}) \end{array}} \\
3.\ \boxed{\gamma\Big(\mathcal{M}\Big(f, \delta_a^{(1)}, \dots, \delta_a^{(n)}\Big)\Big) = \begin{cases} \Big(\Big(\tau_p^{(1)}, \dots, \tau_p^{(n)}\Big) \to \tau_p, \Big(\delta_a^{(1)}, \dots, \delta_a^{(n)}\Big) \to \delta_a\Big) \\ \text{or} \\ \Big(\Big(\tau_p^{(1)}, \dots, \tau_p^{(n)}\Big) \to (), \Big(\delta_a^{(1)}, \dots, \delta_a^{(n)}\Big) \to ()\Big) \end{cases}} \\
4.\ \boxed{\begin{array}{c} \langle \tau_p^{(1)}, e_1, l^{(1)}, \text{newloc}\Big(\tau_p^{(1)}, \mu^{(n)}\Big), \mu^{(n)}, \mathit{force} \rangle \to_C \langle l^{(1)'}, \mu^{(n+1)} \rangle \\ \vdots \\ \langle \tau_p^{(n)}, e_n, l^{(n)}, \text{newloc}\Big(\tau_p^{(n)}, \mu^{(2n-1)}\Big), \mu^{(2n-1)}, \mathit{force} \rangle \to_C \langle l^{(n)'}, \mu^{(2n)} \rangle \end{array}} \\
5.\ \boxed{\begin{array}{c} \text{env}_P\Big(\mathcal{M}\Big(f, \delta_a^{(1)}, \dots, \delta_a^{(n)}\Big)\Big) = \left(\left(p_1, \dots, p_n\right), S, \gamma', \text{env}'_V, \text{env}'_P\right) \\ \text{env}''_V = \text{env}'_V\left[p_1 \mapsto l^{(1)'}\right] \cdots \left[p_n \mapsto l^{(n)'}\right] \\ \left\langle S, \gamma', \text{env}''_V, \text{env}'_P, \mu^{(2n)} \right\rangle \to_S^* \langle l, \mu' \rangle^{\texttt{return}} \end{array}}
\end{array}}{\gamma, \text{env}_V, \text{env}_P \vdash \langle f\texttt{(}e_1\texttt{, ... , }e_n\texttt{)}, \mu \rangle \to_E \langle l, \mu' \rangle}$$

For procedure calls, first, all the arguments are evaluated in order from left to right as shown in Box 1 in the EProc rule. Next, depending on the types of the arguments and the type of the procedure as inferred in Boxes 2 and 3, for each parameter a new location in store is allocated and the value of the respective argument is copied to that location in Box 4. The locations allocated are either private or public, depending on the security types of the parameters. Recall from Subsection 3.2.3 that the security types of the parameters and the arguments may differ and hence, we allow classification of values. Also, the order in which the new locations are initialized and the order of classification do not actually matter.

In Box 5, first the identifiers $p_1, \dots, p_n$ of the procedure parameters, the procedure body S and the respective environments are inferred from env_P. Next, based on the variable environment of the procedure, a new variable environment env''_V is created in which the parameters are bound to the newly allocated locations corresponding to the values of the arguments. Finally, the procedure body is executed. If the procedure executed returns, the return value is taken to be the value of the expression. After executing the procedure, the locations allocated for the arguments and the information they point to can safely be discarded, because they will not be available elsewhere.

The actual procedure to execute can always be determined statically, similarly to what is described in Subsection 3.2.3. Here, the procedure environment env_P can be viewed as carrying with it only static information about the procedure (i.e. when we can consider the locations of global variables and procedure parameters to be fixed values). The semantics for procedure definitions are given in detail in Subsection 4.5.

We have no separate rule for calling procedures of type **`void`**, since the latter always return a **`null`** location as defined later by the SReturnVoid rule in Section 4.4.4. Type checking rules in Section

3.2 already ensure that the returned **null** location is not used anywhere to read a value. Hence, we use the **null** location only to reason about such procedures and it is generally not advised for implementations to factually return a **null** value.

4.3 Evaluation of variable definitions

For variable definitions, two semantic state transition rules exist – one for variable definition and one for variable definition and initialization. Both rules update the type environment and the variable environment env_V with the variable being defined, initializing the variable at a new (or unused) location l. The state transitions for variable definitions are given in the form of

$$\mathrm{env}_P \vdash \langle D_V\,\texttt{;}, \gamma, \mathrm{env}_V, \mu\rangle \rightarrow_{DV} \langle \gamma'\ \mathrm{env}'_V, \mu'\rangle$$

where D_V **;** is the variable definition with the semicolon following it, γ, env_V and μ are respectively the type environment, variable environment and store before handling the definition, their primed versions γ', env'_V and μ' correspond to the environments and store after handling the variable definition.

The first rule initializes the value for the variable to its default value in the store:

$$\text{VarDef}\,\frac{l = \mathrm{newloc}(\tau, \mu)}{\mathrm{env}_P \vdash \langle \tau\ \delta\ x, \gamma, \mathrm{env}_V, \mu\rangle \rightarrow_{DV} \langle \gamma[x \mapsto (\tau, \delta\ var)]\,, \mathrm{env}_V[x \mapsto l]\,, \mu[l \mapsto \mathrm{DEF}_\delta]\rangle}$$

where DEF_δ is the default value for a variable of data type δ. DEF_δ is currently defined as follows:

$$\mathrm{DEF}_{\texttt{bool}} \equiv \texttt{false} \qquad \mathrm{DEF}_{\texttt{int}} \equiv \texttt{0}$$
$$\mathrm{DEF}_{\texttt{unsigned int}} \equiv \texttt{0} \qquad \mathrm{DEF}_{\texttt{string}} \equiv \texttt{""}$$

The second rule for variable definition and initialization is almost identical to the previous rule, with the exception that before updating the type and variable environments, the initializer expression is evaluated to some value which is – instead of the default value – assigned to the newly-defined variable. If needed, the value is classified:

$$\text{VarDefInit}\,\frac{\begin{array}{c}\gamma \vdash e : (\tau', \delta)\\ \gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle\\ \langle \tau, e, l, \mathrm{newloc}(\tau, \mu')\,, \mu', force\rangle \rightarrow_C \langle l', \mu''\rangle\end{array}}{\mathrm{env}_P \vdash \langle \tau\ \delta\ x\ \texttt{=}\ e, \gamma, \mathrm{env}_V, \mu\rangle \rightarrow_{DV} \langle \gamma[x \mapsto (\tau, \delta\ var)]\,, \mathrm{env}_V[x \mapsto l']\,, \mu''\rangle}$$

4.4 Evaluation of statements

State transitions for statements have five different forms, all of which we can denote as

$$\mathrm{env}_P \vdash \langle S, \gamma, \mathrm{env}_V, \mu\rangle \rightarrow_S\ ?$$

where S is the statement (or consecutive statements) left to execute and ? corresponds to the resulting state. Since procedure definitions are only allowed in the top level of the program, env_P stays constant while evaluating statements. The resulting state ? has five different forms:

1. The *next* form is denoted by $\langle S', \gamma', \mathrm{env}'_V, \mu'\rangle$, where S' is the statement (or consecutive statements) still left to execute, γ' and env'_V are the resulting type and variable environment,

and μ' is the resulting store.

2. The *empty* form is denoted by $\langle \gamma', \mathrm{env}'_V, \mu' \rangle$, which is identical to the first form, with the only exception that there are no statements left to execute.

3. The *return* form, written as $\langle l, \mu' \rangle^{\texttt{return}}$ denotes that the evaluation of the statement S resulted in store μ' and some **`return`** statement being called and the location l being returned which holds the return value in μ'. For **`void`** procedures, l is **`null`**.

4. The *break* form, written as $\langle \mu' \rangle^{\texttt{break}}$ denotes that the evaluation of the statement S resulted in store μ' and some **`break`** statement being called.

5. The *continue* form, written as $\langle \mu' \rangle^{\texttt{continue}}$ denotes that the evaluation of the statement S resulted in store μ' and some **`continue`** statement being called.

Of these five forms, only the *next* form can also appear on the left side in state transitions, hence the other forms are not subject to further rewriting. However they can still appear as conditions in the semantic rules for statements.

4.4.1 Compound statements

First, let us consider compound statements, which correspond to the right-hand sides of the ⟨`statement_list`⟩ nonterminal rule in Section 2.6 with two distinct parts. More specifically, this means statements followed by statements, statements followed by compound statements, variable definitions followed by statements and variable definitions followed by compound statements.

If the first part of the compound statement evaluates to an empty form, the compound statement evaluates to the second part of itself. If the first part of the compound statement is a variable definition, the type environment γ will be updated to γ' and the variable environment env_V will be updated to env'_V. Except where the first part is an empty statement, the store μ will also be transformed into μ':

$$\text{SCompound}\,\frac{\mathrm{env}_P \vdash \langle S, \gamma, \mathrm{env}_V, \mu \rangle \to_S^* \langle \gamma', \mathrm{env}'_V, \mu' \rangle}{\mathrm{env}_P \vdash \langle S\texttt{;}\ S', \gamma, \mathrm{env}_V, \mu \rangle \to_S \langle S', \gamma', \mathrm{env}'_V, \mu' \rangle}$$

If the first part of the compound statement evaluates to a return, break or continue form, that evaluation result is also taken to be the result for the whole compound statement. This is described by the following three rules:

$$\text{SCompoundReturn}\,\frac{\mathrm{env}_P \vdash \langle S, \gamma, \mathrm{env}_V, \mu \rangle \to_S^* \langle l, \mu' \rangle^{\texttt{return}}}{\mathrm{env}_P \vdash \langle S\texttt{;}\ S', \gamma, \mathrm{env}_V, \mu \rangle \to_S \langle l, \mu' \rangle^{\texttt{return}}}$$

$$\text{SCompoundContinue}\,\frac{\mathrm{env}_P \vdash \langle S, \gamma, \mathrm{env}_V, \mu \rangle \to_S^* \langle \mu' \rangle^{\texttt{continue}}}{\mathrm{env}_P \vdash \langle S\texttt{;}\ S', \gamma, \mathrm{env}_V, \mu \rangle \to_S \langle \mu' \rangle^{\texttt{continue}}}$$

$$\text{SCompoundBreak}\,\frac{\mathrm{env}_P \vdash \langle S, \gamma, \mathrm{env}_V, \mu \rangle \to_S^* \langle \mu' \rangle^{\texttt{break}}}{\mathrm{env}_P \vdash \langle S\texttt{;}\ S', \gamma, \mathrm{env}_V, \mu \rangle \to_S \langle \mu' \rangle^{\texttt{break}}}$$

4.4.2 Statement blocks

Rules for blocks of statements are similar to rules for compound statements. A major exception to this is the case where the result of evaluating the body of the block is in the empty form. In this case, any modifications to the environments are discarded. Only changes to the store are kept, but since all local variables defined inside the block are unreachable from the outside, the locations in store for those variables may safely be reused for new variables later. What this describes is variables going *out of scope.*

$$\text{SBlock}\,\frac{\text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu\rangle \to_S^* \langle \gamma', \text{env}'_V, \mu'\rangle}{\text{env}_P \vdash \langle \texttt{\{}S\texttt{\}}, \gamma, \text{env}_V, \mu\rangle \to_S \langle \gamma, \text{env}_V, \mu'\rangle}$$

If the result of evaluating the body of the block is in either return, break or continue form, it is propagated by the respective SBlockReturn, SBlockBreak and SBlockContinue rules:

$$\text{SBlockReturn}\,\frac{\text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu\rangle \to_S^* \langle l, \mu'\rangle^{\texttt{return}}}{\text{env}_P \vdash \langle \texttt{\{}S\texttt{\}}, \gamma, \text{env}_V, \mu\rangle \to_S \langle l, \mu'\rangle^{\texttt{return}}}$$

$$\text{SBlockContinue}\,\frac{\text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu\rangle \to_S^* \langle \mu'\rangle^{\texttt{continue}}}{\text{env}_P \vdash \langle \texttt{\{}S\texttt{\}}, \gamma, \text{env}_V, \mu\rangle \to_S \langle \mu'\rangle^{\texttt{continue}}}$$

$$\text{SBlockBreak}\,\frac{\text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu\rangle \to_S^* \langle \mu'\rangle^{\texttt{break}}}{\text{env}_P \vdash \langle \texttt{\{}S\texttt{\}}, \gamma, \text{env}_V, \mu\rangle \to_S \langle \mu'\rangle^{\texttt{break}}}$$

4.4.3 Local variable definitions

For local variable definitions, the rules in Subsection 4.3 are used just like in the global scope (see Section 4.6):

$$\text{SVarDef}\,\frac{\text{env}_P \vdash \langle D_V, \gamma, \text{env}_V, \mu\rangle \to_{DV} \langle \gamma', \text{env}'_V, \mu'\rangle}{\text{env}_P \vdash \langle D_V\,\texttt{;}, \gamma, \text{env}_V, \mu\rangle \to_S \langle \gamma', \text{env}'_V, \mu'\rangle}$$

4.4.4 Return statements

Return statements transform the state to a final return form. Inside **`void`** procedures, the store is not changed and a **`null`** location is returned:

$$\text{SReturnVoid}\,\frac{}{\text{env}_P \vdash \langle \texttt{return;}\ S, \gamma, \text{env}_V, \mu\rangle \to_S \langle \texttt{null}, \mu\rangle^{\texttt{return}}}$$

Inside non-**`void`** procedures, first the given expression is evaluated and then the location of its value is returned together with the resulting state of the store, which might be have changed as a result of evaluating the expression in the statement. Additionally, the value of the expression may need to be moved to the private data domain:

$$\text{SReturn}\;\frac{\begin{array}{c}\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle \\ \gamma(\texttt{thisproc}) = ((\tau_1, \ldots, \tau_n) \rightarrow \tau, (\delta_1, \ldots, \delta_n) \rightarrow \delta) \\ \langle \tau, e, l, \mathrm{newloc}(\tau, \mu'), \mu', \mathit{noforce}\rangle \rightarrow_C \langle l', \mu''\rangle\end{array}}{\mathrm{env}_P \vdash \langle \texttt{\textbf{return}}\ e\texttt{;}, \gamma, \mathrm{env}_V, \mu\rangle \rightarrow_S \langle l', \mu''\rangle^{\texttt{\textbf{return}}}}$$

4.4.5 Break and continue statements

Break and return statements transform the state to the final break form or return form respectively:

$$\text{SBreak}\;\frac{}{\mathrm{env}_P \vdash \langle \texttt{\textbf{break;}}, \gamma, \mathrm{env}_V, \mu\rangle \rightarrow_S \langle \mu\rangle^{\texttt{\textbf{break}}}}$$

$$\text{SContinue}\;\frac{}{\mathrm{env}_P \vdash \langle \texttt{\textbf{continue;}}, \gamma, \mathrm{env}_V, \mu\rangle \rightarrow_S \langle \mu\rangle^{\texttt{\textbf{continue}}}}$$

4.4.6 Expression statements

For expression statements, only the expression is evaluated, and thereby only the store is changed:

$$\text{SExpr}\;\frac{\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle v, \mu'\rangle}{\mathrm{env}_P \vdash \langle e\texttt{;}, \gamma, \mathrm{env}_V, \mu\rangle \rightarrow_S \langle \gamma, \mathrm{env}_V, \mu'\rangle}$$

4.4.7 **if**-statements

For all **if**-statements, first the conditional expression is evaluated. Further action is taken depending on the boolean result of conditional expression.

For **if**-statements with only one branch, that branch is only evaluated if the conditional expression evaluates to **true**. The result state of evaluating the branch is also made the result state of the whole **if**-statement. However, if the branch evaluates to a state in the empty form, its updates to the environments are discarded. If the conditional expression evaluates to **false**, the branch is not evaluated, and the resulting state will be in the empty form with the environments left unchanged. The formal semantic rules for **if**-statements with one branch are as follows:

$$\text{SIfTrue}\;\frac{\begin{array}{c}\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{\textbf{true}} \\ \mathrm{env}_P \vdash \langle S, \gamma, \mathrm{env}_V, \mu'\rangle \rightarrow_S^* \langle \gamma', \mathrm{env}_V', \mu''\rangle\end{array}}{\mathrm{env}_P \vdash \langle \texttt{\textbf{if}}\ \texttt{(}e\texttt{)}\ S, \gamma, \mathrm{env}_V, \mu\rangle \rightarrow_S \langle \gamma, \mathrm{env}_V, \mu''\rangle}$$

$$\text{SIfTrueBreak}\;\frac{\begin{array}{c}\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{\textbf{true}} \\ \mathrm{env}_P \vdash \langle S, \gamma, \mathrm{env}_V, \mu'\rangle \rightarrow_S^* \langle \mu''\rangle^{\texttt{\textbf{break}}}\end{array}}{\mathrm{env}_P \vdash \langle \texttt{\textbf{if}}\ \texttt{(}e\texttt{)}\ S, \gamma, \mathrm{env}_V, \mu\rangle \rightarrow_S \langle \mu''\rangle^{\texttt{\textbf{break}}}}$$

$$\text{SIfTrueContinue}\;\frac{\begin{array}{c}\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{\textbf{true}} \\ \mathrm{env}_P \vdash \langle S, \gamma, \mathrm{env}_V, \mu'\rangle \rightarrow_S^* \langle \mu''\rangle^{\texttt{\textbf{continue}}}\end{array}}{\mathrm{env}_P \vdash \langle \texttt{\textbf{if}}\ \texttt{(}e\texttt{)}\ S, \gamma, \mathrm{env}_V, \mu\rangle \rightarrow_S \langle \mu''\rangle^{\texttt{\textbf{continue}}}}$$

$$\text{SIfTrueReturn}\ \frac{\begin{array}{c}\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \to_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{true} \\ \mathrm{env}_P \vdash \langle S, \gamma, \mathrm{env}_V, \mu'\rangle \to_S^* \langle v, \mu''\rangle^{\texttt{return}}\end{array}}{\mathrm{env}_P \vdash \langle \texttt{if (}e\texttt{)}\ S, \gamma, \mathrm{env}_V, \mu\rangle \to_S \langle v, \mu''\rangle^{\texttt{return}}}$$

$$\text{SIfFalse}\ \frac{\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \to_E \langle l, \mu'\rangle \qquad \mu'(l) = \texttt{false}}{\mathrm{env}_P \vdash \langle \texttt{if (}e\texttt{)}\ S, \gamma, \mathrm{env}_V, \mu\rangle \to_S \langle \gamma, \mathrm{env}_V, \mu'\rangle}$$

The formal semantic rules for **if**-statements with two branches are similar, except that the branch chosen depends on the value of the conditional expression:

$$\text{SIfElseTrue}\ \frac{\begin{array}{c}\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \to_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{true} \\ \mathrm{env}_P \vdash \langle S_1, \gamma, \mathrm{env}_V, \mu'\rangle \to_S^* \langle \gamma', \mathrm{env}'_V, \mu''\rangle\end{array}}{\mathrm{env}_P \vdash \langle \texttt{if (}e\texttt{)}\ S_1\ \texttt{else}\ S_2, \gamma, \mathrm{env}_V, \mu\rangle \to_S \langle \gamma, \mathrm{env}_V, \mu''\rangle}$$

$$\text{SIfElseFalse}\ \frac{\begin{array}{c}\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \to_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{false} \\ \mathrm{env}_P \vdash \langle S_2, \gamma, \mathrm{env}_V, \mu'\rangle \to_S^* \langle \gamma', \mathrm{env}'_V, \mu''\rangle\end{array}}{\mathrm{env}_P \vdash \langle \texttt{if (}e\texttt{)}\ S_1\ \texttt{else}\ S_2, \gamma, \mathrm{env}_V, \mu\rangle \to_S \langle \gamma, \mathrm{env}_V, \mu''\rangle}$$

$$\text{SIfElseTrueBreak}\ \frac{\begin{array}{c}\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \to_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{true} \\ \mathrm{env}_P \vdash \langle S_1, \gamma, \mathrm{env}_V, \mu'\rangle \to_S^* \langle \mu''\rangle^{\texttt{break}}\end{array}}{\mathrm{env}_P \vdash \langle \texttt{if (}e\texttt{)}\ S_1\ \texttt{else}\ S_2, \gamma, \mathrm{env}_V, \mu\rangle \to_S \langle \mu''\rangle^{\texttt{break}}}$$

$$\text{SIfElseFalseBreak}\ \frac{\begin{array}{c}\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \to_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{false} \\ \mathrm{env}_P \vdash \langle S_2, \gamma, \mathrm{env}_V, \mu'\rangle \to_S^* \langle \mu''\rangle^{\texttt{break}}\end{array}}{\mathrm{env}_P \vdash \langle \texttt{if (}e\texttt{)}\ S_1\ \texttt{else}\ S_2, \gamma, \mathrm{env}_V, \mu\rangle \to_S \langle \mu''\rangle^{\texttt{break}}}$$

$$\text{SIfElseTrueContinue}\ \frac{\begin{array}{c}\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \to_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{true} \\ \mathrm{env}_P \vdash \langle S_1, \gamma, \mathrm{env}_V, \mu'\rangle \to_S^* \langle \mu''\rangle^{\texttt{continue}}\end{array}}{\mathrm{env}_P \vdash \langle \texttt{if (}e\texttt{)}\ S_1\ \texttt{else}\ S_2, \gamma, \mathrm{env}_V, \mu\rangle \to_S \langle \mu''\rangle^{\texttt{continue}}}$$

$$\text{SIfElseFalseContinue}\ \frac{\begin{array}{c}\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \to_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{false} \\ \mathrm{env}_P \vdash \langle S_2, \gamma, \mathrm{env}_V, \mu'\rangle \to_S^* \langle \mu''\rangle^{\texttt{continue}}\end{array}}{\mathrm{env}_P \vdash \langle \texttt{if (}e\texttt{)}\ S_1\ \texttt{else}\ S_2, \gamma, \mathrm{env}_V, \mu\rangle \to_S \langle \mu''\rangle^{\texttt{continue}}}$$

$$\text{SIfElseTrueReturn}\ \frac{\begin{array}{c}\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \to_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{true} \\ \mathrm{env}_P \vdash \langle S_1, \gamma, \mathrm{env}_V, \mu'\rangle \to_S^* \langle l', \mu''\rangle^{\texttt{return}}\end{array}}{\mathrm{env}_P \vdash \langle \texttt{if (}e\texttt{)}\ S_1\ \texttt{else}\ S_2, \gamma, \mathrm{env}_V, \mu\rangle \to_S \langle l', \mu''\rangle^{\texttt{return}}}$$

$$\text{SIfElseFalseReturn}\ \frac{\begin{array}{c}\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle e, \mu\rangle \to_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{false} \\ \mathrm{env}_P \vdash \langle S_2, \gamma, \mathrm{env}_V, \mu'\rangle \to_S^* \langle l', \mu''\rangle^{\texttt{return}}\end{array}}{\mathrm{env}_P \vdash \langle \texttt{if (}e\texttt{)}\ S_1\ \texttt{else}\ S_2, \gamma, \mathrm{env}_V, \mu\rangle \to_S \langle l', \mu''\rangle^{\texttt{return}}}$$

4.4.8 **`while`-loops**

As for **`if`**-statements, the conditional expression is evaluated first also for **`while`**-loops. If the conditional expression evaluates to **`false`**, the state is transformed into an empty form:

$$\text{SWhileFalse}\frac{\gamma, \text{env}_V, \text{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle \qquad \mu'(l) = \texttt{false}}{\text{env}_P \vdash \langle \texttt{while (}e\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle \gamma, \text{env}_V, \mu'\rangle}$$

If the conditional expression of the **`while`**-loop evaluates to **`true`**, four different cases must be observed, depending on the outcome of evaluating the body of the loop. If the resulting state is in the return form, it is also returned as the state for the while loop:

$$\text{SWhileReturn}\frac{\begin{array}{c}\gamma, \text{env}_V, \text{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{true} \\ \text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu'\rangle \rightarrow^*_S \langle l, \mu''\rangle^{\texttt{return}}\end{array}}{\text{env}_P \vdash \langle \texttt{while (}e\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle l, \mu''\rangle^{\texttt{return}}}$$

The evaluation of the loop body ends in a state in the empty form, if no **`return`**, **`break`** or **`continue`** statement in the loop body was reached. In this case, all the changes done by the body to the environments are discarded, and the loop is re-evaluated with the changed store. The same holds for the case when a state in the continue form is reached:

$$\text{SWhileTrue}\frac{\begin{array}{c}\gamma, \text{env}_V, \text{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{true} \\ \text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu'\rangle \rightarrow^*_S \langle \gamma', \text{env}'_V, \mu''\rangle\end{array}}{\text{env}_P \vdash \langle \texttt{while (}e\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle \texttt{while (}e\texttt{)}\ S, \gamma, \text{env}_V, \mu''\rangle}$$

$$\text{SWhileContinue}\frac{\begin{array}{c}\gamma, \text{env}_V, \text{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{true} \\ \text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu'\rangle \rightarrow^*_S \langle \mu''\rangle^{\texttt{continue}}\end{array}}{\text{env}_P \vdash \langle \texttt{while (}e\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle \texttt{while (}e\texttt{)}\ S, \gamma, \text{env}_V, \mu''\rangle}$$

If the evaluation of the body of the loop reaches a state in the break form, the changes to the environments are discarded and an empty state is returned:

$$\text{SWhileBreak}\frac{\begin{array}{c}\gamma, \text{env}_V, \text{env}_P \vdash \langle e, \mu\rangle \rightarrow_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{true} \\ \text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu'\rangle \rightarrow^*_S \langle \mu''\rangle^{\texttt{break}}\end{array}}{\text{env}_P \vdash \langle \texttt{while (}e\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle \gamma, \text{env}_V, \mu''\rangle}$$

4.4.9 **`do-while`-loops**

Semantics for the **`do-while`**-loop are similar to the semantics of the **`while`** loop, with the only difference being that the conditional expression is not checked before the very first evaluation of the **`do-while`**-loop body. Hence, if the first execution of the loop body returns a state in the empty or continue form, the rest of the loop can be handled just like a **`while`**-loop as described by the SDoWhile and SDoWhileContinue rules:

$$\text{SDoWhile}\frac{\text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu\rangle \rightarrow^*_S \langle \gamma', \text{env}'_V, \mu'\rangle}{\text{env}_P \vdash \langle \texttt{do}\ S\ \texttt{while (}e\texttt{)}, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle \texttt{while (}e\texttt{)}, \gamma, \text{env}_V, \mu'\rangle}$$

$$\text{SDoWhileContinue}\;\frac{\text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S^* \langle \mu' \rangle^{\texttt{continue}}}{\text{env}_P \vdash \langle \texttt{do}\ S\ \texttt{while}\ \texttt{(}e\texttt{)}, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle \texttt{while}\ \texttt{(}e\texttt{)}, \gamma, \text{env}_V, \mu'\rangle}$$

If the first execution of the **do-while**-loop body results in a state in the return form, that result is also used as the result for the whole loop:

$$\text{SDoWhileReturn}\;\frac{\text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S^* \langle l, \mu' \rangle^{\texttt{return}}}{\text{env}_P \vdash \langle \texttt{do}\ S\ \texttt{while}\ \texttt{(}e\texttt{)}, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle l, \mu'\rangle^{\texttt{return}}}$$

If the first execution of the **do-while**-loop body results in a state in the break form, further evaluation of the loop is skipped and a state in the empty form is returned:

$$\text{SDoWhileBreak}\;\frac{\text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S^* \langle \mu' \rangle^{\texttt{break}}}{\text{env}_P \vdash \langle \texttt{do}\ S\ \texttt{while}\ \texttt{(}e\texttt{)}, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle \gamma, \text{env}_V, \mu'\rangle}$$

4.4.10 **for**-loops

The **for**-loop is the most complex statement in SecreC, since all the subexpressions in its header are optional. Nevertheless, in case the third expression is omitted, the **for**-loop is transformed into a **while**-loop using the following four rules:

$$\text{SForToWhile1}\;\frac{}{\text{env}_P \vdash \langle \texttt{for (}e\texttt{;}\ e'\texttt{;)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle e\texttt{;}\ \texttt{while (}e'\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle}$$

$$\text{SForToWhile2}\;\frac{}{\text{env}_P \vdash \langle \texttt{for (}e\texttt{;;)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle e\texttt{;}\ \texttt{while (true)}\ S, \gamma, \text{env}_V, \mu\rangle}$$

$$\text{SForToWhile3}\;\frac{}{\text{env}_P \vdash \langle \texttt{for (;}\ e'\texttt{;)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle \texttt{while (}e'\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle}$$

$$\text{SForToWhile4}\;\frac{}{\text{env}_P \vdash \langle \texttt{for (;;)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle \texttt{while (true)}\ S, \gamma, \text{env}_V, \mu\rangle}$$

Two of the other cases, where both the third and the first expression are present, can also be eliminated by transforming the **for**-loop to an equivalent compound statement consisting of the expression statement corresponding to the first expression and the rest of the **for**-loop:

$$\text{SForStart}\;\frac{}{\text{env}_P \vdash \langle \texttt{for (}e\texttt{;}\ e'\texttt{;}\ e''\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle e\texttt{;}\ \texttt{for (;}\ e'\texttt{;}\ e''\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle}$$

$$\text{SForStart2}\;\frac{}{\text{env}_P \vdash \langle \texttt{for (}e\texttt{;;}\ e''\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle e\texttt{;}\ \texttt{for (;;}\ e''\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle}$$

But there is still one more case of the **for**-loop which can be eliminated. The case where only the third expression is present, can be transformed into an equivalent **for**-loop with both the third and second expressions present. The second expression will be the constant **true**:

$$\text{SForToFor}\;\frac{}{\text{env}_P \vdash \langle \texttt{for (;;}\ e''\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle \texttt{for (; true;}\ e''\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle}$$

We have eliminated all variants of the **for**-loop except for the one where out of the three optional expressions only the first one is omitted. This variant of the **for**-loop is what actually makes it distinct from the **while**- and **do**-**while**-loops in SecreC.

For this variant of the **for**-loop, first the second expression is evaluated as a conditional expression. If it evaluates to **false**, the state is transformed to an empty form:

$$\text{SForFalse}\ \frac{\gamma, \text{env}_V, \text{env}_P \vdash \langle e', \mu\rangle \rightarrow_E \langle l, \mu'\rangle \qquad \mu'(l) = \texttt{\textbf{false}}}{\text{env}_P \vdash \langle \texttt{\textbf{for}}\ \texttt{(;}\ e'\texttt{;}\ e''\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle \gamma, \text{env}_V, \mu'\rangle}$$

If the second expression evaluates to **true**, we must again observe four different cases based on the outcome of the evaluation of the loop body. First, when the result of that evaluation is in the return form, the result for the evaluation of the whole loop is also in the return form:

$$\text{SForReturn}\ \frac{\begin{array}{c}\gamma, \text{env}_V, \text{env}_P \vdash \langle e', \mu\rangle \rightarrow_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{\textbf{true}} \\ \text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu'\rangle \rightarrow_S^* \langle l', \mu''\rangle^{\texttt{\textbf{return}}}\end{array}}{\text{env}_P \vdash \langle \texttt{\textbf{for}}\ \texttt{(;}\ e'\texttt{;}\ e''\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle l', \mu''\rangle^{\texttt{\textbf{return}}}}$$

Secondly, if the evaluation of the loop body results in a state in the break form, a state in the empty form is returned:

$$\text{SForBreak}\ \frac{\begin{array}{c}\gamma, \text{env}_V, \text{env}_P \vdash \langle e', \mu\rangle \rightarrow_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{\textbf{true}} \\ \text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu'\rangle \rightarrow_S^* \langle \mu''\rangle^{\texttt{\textbf{break}}}\end{array}}{\text{env}_P \vdash \langle \texttt{\textbf{for}}\ \texttt{(;}\ e'\texttt{;}\ e''\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle \gamma, \text{env}_V, \mu''\rangle}$$

If the execution of the loop body results in a state in the empty form, the next iteration of the loop is started. This is done by rewriting the current state into one in the next form, where the statement to execute consists of the third expression of the loop as a statement expression, followed by the whole loop as it is:

$$\text{SForTrue}\ \frac{\begin{array}{c}\gamma, \text{env}_V, \text{env}_P \vdash \langle e', \mu\rangle \rightarrow_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{\textbf{true}} \\ \text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu'\rangle \rightarrow_S^* \langle \gamma', \text{env}_V', \mu''\rangle\end{array}}{\text{env}_P \vdash \langle \texttt{\textbf{for}}\ \texttt{(;}\ e'\texttt{;}\ e''\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle e''\texttt{;}\ \texttt{\textbf{for}}\ \texttt{(;}\ e'\texttt{;}\ e''\texttt{)}\ S, \gamma, \text{env}_V, \mu''\rangle}$$

It is the case when the execution of the loop body results in a state in the continue form, that makes the **for**-loop special. Although its result is analogous to the previous case, it is exactly because of the **continue**-statements that the **for**-loop can't be easily rewritten as a **while** or **do**-**while**-loop.

$$\text{SForContinue}\ \frac{\begin{array}{c}\gamma, \text{env}_V, \text{env}_P \vdash \langle e', \mu\rangle \rightarrow_E \langle l, \mu'\rangle \\ \mu'(l) = \texttt{\textbf{true}} \\ \text{env}_P \vdash \langle S, \gamma, \text{env}_V, \mu'\rangle \rightarrow_S^* \langle \mu''\rangle^{\texttt{\textbf{continue}}}\end{array}}{\text{env}_P \vdash \langle \texttt{\textbf{for}}\ \texttt{(;}\ e'\texttt{;}\ e''\texttt{)}\ S, \gamma, \text{env}_V, \mu\rangle \rightarrow_S \langle e''\texttt{;}\ \texttt{\textbf{for}}\ \texttt{(;}\ e'\texttt{;}\ e''\texttt{)}\ S, \gamma, \text{env}_V, \mu''\rangle}$$

4.5 Evaluation of procedure definitions

The semantic rule for procedures with non-**void** return type and with **void** return type are in principle very similar:

$$\text{Procdef}\frac{}{\mathrm{env}_V \vdash \langle \tau\ \delta\ f\texttt{(}\tau_1\ \delta_1\ p_1\texttt{,}\ \ldots\texttt{,}\ \tau_n\ \delta_n\ p_n\texttt{)}\ \texttt{\{}S\texttt{\}}, \gamma, \mathrm{env}_P\rangle \rightarrow_{DP} \langle \gamma', \mathrm{env}'_P\rangle}$$

$$\text{ProcdefVoid}\frac{}{\mathrm{env}_V \vdash \langle \texttt{void}\ f\texttt{(}\tau_1\ \delta_1\ p_1\texttt{,}\ \ldots\texttt{,}\ \tau_n\ \delta_n\ p_n\texttt{)}\ \texttt{\{}S\texttt{\}}, \gamma, \mathrm{env}_P\rangle \rightarrow_{DP} \langle \gamma', \mathrm{env}'_P\rangle}$$

where

$$\begin{aligned}
\gamma' &= \gamma[f \mapsto \texttt{proc}][\mathcal{M}(f, \delta_1, \ldots, \delta_n) \mapsto \rho] \\
\gamma_c &= \gamma'[\texttt{thisproc} \mapsto \rho][p_1 \mapsto (\tau_1, \delta_1)] \cdots [p_n \mapsto (\tau_n, \delta_n)] \\
\mathrm{env}'_P &= \mathrm{env}_P[\mathcal{M}(f, \delta_1, \ldots, \delta_n) \mapsto ((p_1, \ldots, p_n), S, \gamma_c, \mathrm{env}_V, \mathrm{env}'_P)] \\
\rho &= \begin{cases} ((\tau_1 \times \ldots \times \tau_n) \rightarrow \tau, (\delta_1 \times \ldots \times \delta_n) \rightarrow \delta) & \text{for Procdef} \\ ((\tau_1 \times \ldots \times \tau_n) \rightarrow (), (\delta_1 \times \ldots \times \delta_n) \rightarrow ()) & \text{for ProcdefVoid} \end{cases}
\end{aligned}$$

Both rules properly update the type environment with the identifier and the mangled identifier for the procedure. Another environment, γ_c, is also created for the scope of the procedure by updating γ with the types for the parameters. The procedure environment env'_P is an updated version of env_P. For the mangled identifier of the procedure, it is updated with a 5-tuple consisting of a list of the identifiers $(p_1, \ldots, p_n)$ for the procedure parameters, the function body S, the type environment γ_c for the body, the variable environment env_V at the procedure definition and a reference to env'_P itself. Note that although env'_P is self-referential, this does not cause any trouble in the semantics.

4.6 Evaluation of the program

Sequentially for the whole program, all variable and procedure definitions are read into the environments. We start the evaluation of programs from a state $\langle P, \gamma_0, \mathrm{env}^0_V, \mathrm{env}^0_P, \mu_0\rangle$ where P is the whole program, γ_0, env^0_V and env^0_P are the predefined environments and μ_0 is the predefined store. All these four predefined partial functions may be empty, i.e. have an empty domain. The predefined type environment γ_0 must be the same as used for static checking in Subsection 3.2.5. There are two semantic rules for global variable and procedure definitions:

$$\text{ProgramVarDef}\frac{\mathrm{env}_P \vdash \langle D_V, \gamma, \mathrm{env}_V, \mu\rangle \rightarrow_{DV} \langle \gamma', \mathrm{env}'_V, \mu'\rangle}{\langle D_V\ \texttt{;}\ P, \gamma, \mathrm{env}_V, \mathrm{env}_P, \mu\rangle \rightarrow \langle P, \gamma', \mathrm{env}'_V, \mathrm{env}_P, \mu'\rangle}$$

$$\text{ProgramProcDef}\frac{\mathrm{env}_V \vdash \langle D_P, \gamma, \mathrm{env}_P\rangle \rightarrow_{DP} \langle \gamma', \mathrm{env}'_P\rangle}{\langle D_P\ P, \gamma, \mathrm{env}_V, \mathrm{env}_P, \mu\rangle \rightarrow \langle P, \gamma', \mathrm{env}_V, \mathrm{env}'_P, \mu\rangle}$$

Note, that defining global variables might also change the store because of initializer expressions which can have side-effects in case of assignments and procedure calls.

Once all the global definitions are handled, a procedure call to the procedure `main` is evaluated. When the evaluation of the procedure call expression is completed, the program finishes in an **END** state:

$$\text{ProgramRun}\frac{\gamma, \mathrm{env}_V, \mathrm{env}_P \vdash \langle \texttt{main()}, \mu\rangle \rightarrow_E \langle \texttt{null}, \mu'\rangle}{\langle \varepsilon, \gamma, \mathrm{env}_V, \mathrm{env}_P, \mu\rangle \rightarrow \textbf{END}}$$

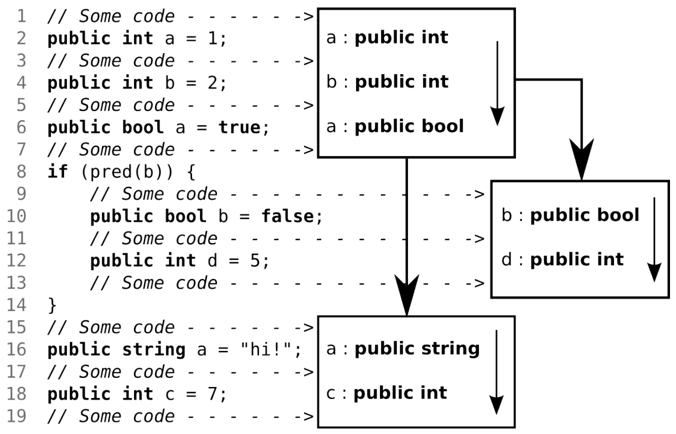
5 Intermediate representation

For analysis purposes, the SecreC Analyzer first compiles all SecreC programs into a much simpler SecreC intermediate representation (SIR). This intermediate representation consists of a symbol table and the intermediate code – a list of SIR instructions. SIR described in this section is only meant to be a data structure inside analysis tools and compilers, although it is not a difficult task to convert it into some simple form of assembly code.

Translation of SecreC programs to the SecreC intermediate representation is done with respect to the techniques described in [?]. More specifically, our analyzer uses quadruples to represent instructions, incremental code generation, backpatching of jump instructions for all SecreC control structures, and short-circuit code generation for the appropriate boolean expressions. Static checking is done in parallel with code generation. Any errors encountered during translation and static checking are logged and the ongoing process is halted.

5.1 Symbol table

The symbol table of the SecreC Analyzer is actually a tree of tables, where each subtree represents a certain scope of the program. This has proven to ease the task of symbol lookup during static checking and incremental code generation. Each node in the abstract syntax tree is statically checked and incrementally translated in the context of the corresponding symbol table. The hierarchical structure of the symbol table simplifies symbol lookup during static checking and intermediate code generation. Figure 1 shows an example fragment of code with its relation to corresponding symbol tables in a simplified form, and a table with definitions in scope for the lines of code.



```
 1 // Some code - - - - - ->
 2 public int a = 1;
 3 // Some code - - - - - ->
 4 public int b = 2;
 5 // Some code - - - - - ->
 6 public bool a = true;
 7 // Some code - - - - - ->
 8 if (pred(b)) {
 9     // Some code - - - - - - - - - - - - ->
10     public bool b = false;
11     // Some code - - - - - - - - - - - - ->
12     public int d = 5;
13     // Some code - - - - - - - - - - - - ->
14 }
15 // Some code - - - - - ->
16 public string a = "hi!";
17 // Some code - - - - - ->
18 public int c = 7;
19 // Some code - - - - - ->
```

Lines of code	Line numbers of definitions in scope
1-2	none
3-4	2
5-6	2, 4
7-10	4, 6
11-12	6, 10
13	6, 10, 12
15-16	4, 6
17-18	4, 16
19	4, 16, 18

Figure 1: The hierarchy of symbol tables during translation and the corresponding table of definitions in scope for each line of code.

The actual symbol tables used in the SecreC Analyzer contain four types of symbols for procedures, regular variables, temporary variables and constants respectively. Symbols for regular variables are accessible by their name, symbols for procedures are accessible by their mangled name. A different kind of name mangling is used for the names of constant and temporary variable symbols. The intermediate code instructions access symbols directly by reference.

Inherently, each symbol in the symbol table contains the SecreC type of the symbol. The types implemented in our analyzer are modeled in a similar fashion with the formal hierarchical structures described in Section 3.1. To track symbol declarations back to the original code, symbols in the analyzer also refer to the node in the abstract syntax tree that caused their generation. This is most useful for providing programmers feedback about the source code.

5.2 Instruction set

SIR instructions are quadruples in the form of (op, d, arg_1, arg_2), where *op* is the type of the instruction, d, arg_1 and arg_2 usually refer to symbols or constants in the symbol table, or other instructions. To track the instructions back to the abstract tree nodes that caused their generation, all instructions in the SecreC Analyzer also have a reference to the node of in the abstract syntax tree of the parsed SecreC program that caused their generation.

The intermediate representation, SIR, described in this section can be used to represent all SecreC programs defined by the grammar, static checking and semantics rules in the previous sections. Complex expressions can be written into simple expression instructions using temporary variables. All control flow structures can be written in SIR by using conditional and unconditional jumps.

Instruction type	Description	Instruction type	Description
MUL	`d = arg1 * arg2;`	DIV	`d = arg1 / arg2;`
MOD	`d = arg1 % arg2;`	ADD	`d = arg1 + arg2;`
SUB	`d = arg1 - arg2;`	EQ	`d = arg1 == arg2;`
NE	`d = arg1 != arg2;`	LE	`d = arg1 <= arg2;`
LT	`d = arg1 < arg2;`	GE	`d = arg1 >= arg2;`
GT	`d = arg1 > arg2;`	LAND	`d = arg1 && arg2;`
LOR	`d = arg1 || arg2;`	ASSIGN	`d = arg1;`
UNEG	`d = !arg1;`	UMINUS	`d = -arg1;`
CLASSIFY	`d = CLASSIFY(arg1);`	DECLASSIFY	`d = DECLASSIFY(arg1);`

Table 1: SIR instructions for expressions.

Instructions for most expressions are given in Table 1. These instructions take at most two arguments, arg_1 and arg_2 which refer to symbols or constants in the symbol table. The result for these expressions is written to the symbol referred to by d.

Instruction type	Description
JUMP	`goto d;`
JT	`if (arg1) goto d;`
JF	`if (!arg1) goto d;`
JE	`if (arg1 == arg2) goto d;`
JNE	`if (arg1 != arg2) goto d;`
JLE	`if (arg1 <= arg2) goto d;`
JLT	`if (arg1 < arg2) goto d;`
JGE	`if (arg1 >= arg2) goto d;`
JGT	`if (arg1 > arg2) goto d;`

Table 2: SIR instructions for control flow.

For control flow structures and logical expressions where the left-hand side subexpression has the **`public`** data type (see Subsection 4.2.4), SIR has an unconditional jump instruction and eight conditional jump instructions, all of which are shown in Table 2. JT and JF only depend on one boolean symbol arg_1, other conditional jump instructions take symbols of types as defined in Appendix A for the corresponding comparison operators. For all these jump instructions in SIR, d refers to the SIR instruction to jump to in case of the unconditional jump or if the expression holds.

Instruction type	Description
POPPARAM	Pops an argument value from the procedure stack to the symbol d which is a procedure parameter.
RETURNVOID	Returns from the current **`void`** procedure.
RETURN	Returns from the current procedure with the value of arg_1.
PUSHPARAM	Pushes the value of symbol arg_1 onto the procedure stack.
CALL	Calls the function identified by the symbol arg_1 using arguments pushed onto the procedure stack and stores the result in the symbol d.
RETCLEAN	This instruction always follows CALL instructions and is the target for RETURN and RETURNVOID.

Table 3: SIR instructions for procedures.

SIR instructions for procedures are summed up in Table 3. For calling procedures, the arguments are pushed to a procedure stack in order from right to left using the PUSHPARAM instruction, with arg_1 as the symbol to push. The procedure itself is called using the CALL instruction, which takes the symbol of the procedure as arg_1 and an optional symbol d to store the return value of the procedure. The CALL instruction passes execution to the first instruction of the procedure. The RETURN and RETURNVOID instructions pass control to the RETCLEAN instruction, which follows each CALL instruction. The RETCLEAN instruction by itself currently does nothing, but the analyzer makes use of it when constructing the graph of basic blocks. The RETCLEAN instruction might later also be used to clean up any leftovers from the stack after returning from the procedure, or to actually assign the returned value to the symbol d of the previous CALL instruction. Inside the procedure, the POPPARAM instruction is used to take procedure arguments from the stack and assign them to local parameter variables.

For example, for a procedure named `add`, which we define as

```
public int add(public int a, public int b) { return a + b; }
```

and its call `result = add(4, 2);`, one of the possible corresponding translations to SIR instructions is shown in Table 4:

POPPARAM a	PUSHPARAM 2
POPPARAM b	PUSHPARAM 4
$t = a + b$	$result$ = CALL add
RETURN t	RETCLEAN

Table 4: Example of a procedure (left) and a procedure call (right) in SIR instructions. Symbol t is a local temporary variable of type (**public**, **int** var).

In addition to the aforementioned SIR instructions, there are two other instruction as shown in Table 5. The COMMENT instruction does absolutely nothing – it is only meant to provide a means to add comments or annotations to the intermediate representation. The END instruction denotes the end of the program – if this instruction is reached by the interpreter, execution is stopped.

Instruction type	Description
COMMENT	Does nothing. arg_1 is reference to a comment string.
END	Stops execution of the program.

Table 5: Miscellaneous SIR instructions.

6 Analysis for SecreC

We are ultimately interested in how much and in what ways private data leaks from SecreC programs via **declassify** statements. Therefore, an appropriate method to use would have to be a data-flow analysis technique. For SecreC and many other programming languages, syntax-oriented approaches are in most cases not very practicable for detecting data flow. Therefore, a more general representation of programs and their execution paths is needed. The most suitable approach is to use control flow graphs. Having developed an intermediate representation for SecreC programs (which we presented in Chapter 5), we can use intermediate representations of SecreC programs to construct these control flow graphs. All further analyses will be done using the control flow graph.

In the first part of this chapter we describe the basic blocks and control flow graphs used by the SecreC Analyzer, the second part describes the implemented general data-flow analysis component of the SecreC Analyzer. The third part of this chapter describes how this framework can be used for detecting information leaks.

6.1 Basic blocks and the control flow graph

For our current and any other future analysis, intermediate code generated by the SecreC Analyzer is partitioned into a graph representation of *basic blocks*. Basic blocks are sequences of consecutive code instructions, into which control flow can only enter through the first instruction, and from which control flow leaves only through the last instruction either by continuing with the next instruction in code (belonging to another basic block) or branching to some arbitrary instruction. Basic blocks are then used to form a *control flow graph* (CFG), whose edges indicate the direction and type of control flow between the blocks. Some basic block A is called the *predecessor* of basic block B, if in the CFG there exists an edge from block A to block B. Under the same conditions, block B is called the *successor* of block A.

The SecreC Analyzer distinguishes between five different types of control flow graph edges. The edge between two basic blocks A and B is a

- **regular edge** if either
  1. the control flow from block A enters block B either by an unconditional jump instruction (JUMP) at the end of block A, or
  2. the last instruction of block A does not alter control flow and in the code the instructions of block B directly follow the instructions of block A;
- **true edge** if the last instruction of block A is a conditional jump which passes the control flow to block B if the conditional expression is found to hold;
- **false edge** if the last instruction of block A is a conditional jump which passes the control flow to block B if the conditional expression is found not to hold;
- **call edge** if the last instruction of block A is a procedure call instruction (CALL) and the code of the procedure it calls starts with the first instruction in block B;
- **return edge** if the last instruction of block A is a procedure return instruction (RETURN or RETURNVOID) and the first instruction of block B is a RETCLEAN instruction which follows

a CALL instruction calling the procedure which contains the code in block A;

- **call pass edge** is an edge which links the block ending with a CALL instruction to the block which holds the corresponding RETCLEAN instruction as its first instruction.

In the control flow graph generated by the SecreC Analyzer all basic blocks have a maximum of two outgoing edges. For all control flow graphs generated by the analyzer, the only basic block with no outgoing edges is the block which follows the call to the `main` procedure (see ProgramRun rule in Section 4.6). The only occurrence of the END instruction in the whole program is at the end of this block. Basic blocks ending with a conditional jump always have two outgoing edges – a true edge and a false edge – one for each branch. If the last instruction of a basic block is a CALL instruction, there are also two outgoing edges – a call edge pointing to the first block of the code of the procedure being called; and a call pass edge pointing to the next block which starts with a RETCLEAN instruction. Blocks ending with a RETURN instructions have outgoing edges to the blocks starting with RETCLEAN which follow the CALL instruction to the procedure which this RETURN instruction is part of. All other blocks have only one outgoing edge. An example fragment of a SecreC control flow graph for the SecreC code fragment in Figure 2 can be seen in Figure 3 where true and false edges are labeled with + and − signs respectively, and regular edges do not have labels.

```
public int a = 42;
public int b;
private int c;
while (b < x) {
    b += y;
    if (b > z)
        c += b;
}
```

Figure 2: Example code fragment from a SecreC program.

```
 8  a = 42
 9  b = 0
10  c = 0
...........................................
11  if (b < x) GOTO 13
...........................................
12  GOTO 19
...........................................
13  b = b + y
14  if (b > z) GOTO 16
...........................................
15  GOTO 11
...........................................
16  t = classify(b)
17  c = c + t
18  GOTO 11
...........................................
19  // Any code following
```

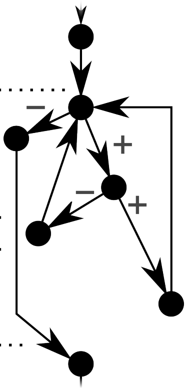


Figure 3: Sequence of SIR instructions (not optimized) and the respective CFG corresponding to the code fragment in Figure 2. On the left, dotted lines represent basic block boundaries.

6.2 Data-flow analysis component in the SecreC Analyzer

The SecreC Analyzer provides a general framework for forward and backward data-flow analyses similar to what is described in [?]. The data-flow analyzer component can be given a number of analysis objects as arguments. Each of these objects represent either a forward or backward data-flow analysis algorithm. The data-flow analyzer then runs these analyses on top of the control-flow graph generated for SecreC programs.

While [?] describes two separate general iterative algorithms for running forward and backward analyses, one at a time, we have found that these two can easily be merged into a single iterative algorithm which would allow a greater number of forward and backward data-flow analyses to be run more efficiently in a single loop. If any of the analyses converge in the loop, it is purged from the list of working analyses and is not considered in next iterations of the loop. The algorithm stops if all analyses converge. The outline for the algorithm used in the SecreC Analyzer is given in Appendix B as pseudocode.

Because the algorithm can differentiate between different types of edges in the control-flow graph, it is able to run path-sensitive analyses, meaning that different kind of flow information can be propagated to different blocks. For example, we know that in case of an **`if`**-statement, if the control flow reaches one of its branches, the conditional expression guard holds in one branch, but does not hold in the other branch. The algorithm used in the SecreC Analyzer allows analyses to catch this kind of information regardless of the direction of analysis.

In our analyzer, the basic blocks in the control flow graph is interconnected with the instructions in the intermediate representation, and those instructions are in turn interconnected with the symbols in the symbol table and the abstract syntax tree. Hence, complex analyses can be implemented on top of the framework provided by the SecreC Analyzer. Because nodes in the abstract syntax tree also provide the locations of language constructs as written in the source code, the analyses have means to provide good feedback for programmers and integrated development environments.

7 Experimental results

The implementation of the SecreC Analyzer is currently a work-in-progress in the form of a C++ library. We have implemented the translation of most of the language described in Chapters 2, 3 and 4 to the intermediate representation presented in Chapter 5. As described in Chapter 6, the analyzer is capable of generating the basic blocks and the control-flow graph for the intermediate representation. It provides the means to run data-flow analyses using the general algorithm described in Appendix B. We have also implemented a preliminary command-line front-end for the library for testing purposes.

At library level, we have started to implement three different forward data-flow analyses. The first is a reaching definitions analysis. We call the second data-flow analysis a reaching jumps analysis. The third analysis is solely focused on information leakage. The implementations are still a work-in-progress and would greatly benefit from constant propagation and region-based constant propagation analysis, which we have not implemented yet.

7.1 Reaching definitions analysis

A reaching definitions analysis discovers for each program location the set of variable definitions which *reach* that program location. A variable definition d of variable v is a statement or expression which modifies the initializes or changes the value of v. The definition d of variable v is said to *reach* the program location l if there exists an execution path from d to l so that there are no other definitions of v on that path.

The reaching definitions analysis we have implemented is a rather simple one. Since SecreC does not yet have pass-by-reference semantics for procedures, the reaching definitions analysis only needs to consider global variables for inter-procedural analysis.

The reaching definitions analysis we implemented does not handle the case of variables going out of scope, therefore the results it yields might in some cases be useless for us. For these cases, it might be practical to hide the definitions of variables for blocks for which these variables are not in scope. This can be done by extending the intermediate representation with explicit notation corresponding to out of scope semantics. Also, a live-variable analysis can be used to hide definitions for variables which are not later used.

Reaching definitions analysis can be used in optimizing compilers. For example, it can be used in a technique called *loop-invariant code motion*, which is used to move loop-invariant expressions outside of the loop if all the reaching definitions for all operands in that expression are outside of that loop.

7.2 Reaching jumps analysis

The *reaching jumps* analysis was originally meant in simple cases to discover which conditional expressions acting as guards in **`if`**-statements and loop headers affect reaching definitions. The current implementation is a simple demonstration of a path-sensitive data-flow analysis. For each basic block B and each conditional jump instruction C that is on some execution path P from the entry node to block B, the reaching jumps analysis calculates which of the branches of the conditional jump C are last taken to reach block B. On each such execution path P only the last occurrence of C is taken into account by the analysis. Potential applications for this analysis can exploit such information when trying to infer conditions which hold at block B.

For example, let us look at the following code fragment:

```
if (c) {
  // Block 1 is here
} else {
  // Block 2 is here
}
// Block 3 is here
```

Our reaching jumps analysis can deduce that block 1 can only be reached if the condition `c` evaluates to true and block 2 can only be reached if `c` evaluates to false. However, block 3 is reached from both branches. For most cases, this tells us, that at the beginning of block 1 the condition `c` holds and that `c` does not hold at the beginning of block 2. In case the conditional expression has side-effects, e.g. `(x += 1) > 2`, then we still know that by reaching one of the branches the expression must evaluated to a certain value.

By itself, this information is not of much use. However, similar path-sensitive analyses can be used for optimizations and for detecting certain programming errors. For example, they can be used to warn the programmer in case the program contains unreachable code in cases like `if (c) { if (!c) doSomething(); }`. Our reaching jumps analysis can be extended to detect some of such errors.

7.3 Detecting information leakage

We have also started work on a data-flow analysis to detect trivial cases of information leakage. This analysis currently considers private procedure parameters and private values returned by procedure calls to be unprocessed sensitive input data. The results of binary operations are considered non-sensitive data. The results of unary operations and regular assignments are considered to be non-sensitive if the operand is a constant. Otherwise sensitivity is transferred from the operand. The analysis detects whether such sensitive data might be declassified.

More specifically, for each **`declassify`** instruction two sets of directly or indirectly (via unary operators) reaching definitions are calculated. The first set contains non-sensitive definitions (by unary operations on constants and by binary operations) and the other contains sensitive definitions (return values from procedure calls and private procedure parameters). If only the set of non-sensitive reaching definitions for some program **`declassify`** instruction is empty, sensitive data (defined by some sensitive definition from the second set) is declassified. If only the second set is empty, no sensitive information is considered to leak. If both sets are non-empty, sensitive data will only leaks if corresponding execution paths are taken.

This kind of analysis suffers from a number of deficiencies which still need to be eliminated. Most importantly, this analysis is not conservative because binary operations should not always be considered to render sensitive data non-sensitive. Trivial examples for this include binary operations where one operand is a known constant. Among other features, a constant propagation analysis is needed to make this analysis more precise for binary operations.

Another deficiency of the current approach is that not all procedures that return private values must be considered to return sensitive input data – only certain predefined procedures must be considered to return sensitive data. For other procedure calls inter-procedural analysis is needed.

8 Conclusion

In this thesis, we present a framework for performing data-flow analyses on SecreC programs. To serve as its theoretical basis, we have developed the first formal specification of the SecreC programming language. Although not yet covering the entire language supported by the present compiler, it provides a sufficient starting point for any further language development. The specification includes the formal grammar of the language, static checking rules, and rules for operational semantics. The static checking rules also provide a simple type system and detection of other programming errors.

We have built a C++ library containing the SecreC Analyzer which provides the means to run data-flow analyses on SecreC programs. While the implementation is still in a development phase, it is the first to use the new SecreC specification we have just provided. The library can be used as a front-end to optimizing compilers and in integrated development environments to provide visual feedback to the programmer. A sample command-line analyzer which uses the library as a back-end has also been implemented.

Three forward data-flow analyses have been implemented as part of the library providing SecreC Analyzer. The first of these is a reaching definitions analysis for SecreC. The second analysis provides information about which branches of conditional jumps the execution flow must follow to reach parts of the code. The third analysis is capable of detecting whether certain kinds of data flows that are deemed to contain sensitive information unsuitable for publication, reach the **`declassify`** operators. This information leakage analysis is capable of detecting information leakages for some simple cases.

We have built a sustainable foundation for further development of the SecreC language and related tools. The language specification and SecreC Analyzer have provided a good starting point for further research and development of the language and of information leakage analyses for SecreC. Full-blown analyses for detecting information leakage and measuring the extent of information leakage are already being researched. In parallel, plans are being made for implementing supportive analyses like constant propagation analysis, and extending the formal specification of SecreC with support for arrays and matrices.

Authors remark after successfully defending this thesis: Mainly due to the lack of time to work on this thesis the author is personally dissatisfied with its submitted version, which in his opinion is not worth a higher grade than D (61-70%).

9 Analüüsiraamistik privaatsust säilitavale programmeerimiskeelele SecreC

Jaak Ristioja

Magistritöö (30 EAP)

Kokkuvõte

Käesolevas magistritöös kirjeldame raamistikku privaatsust säilitava imperatiivse programmeerimiskeele, SecreC, programmide analüüsiks. Programmeerimiskeelt SecreC kasutatakse kõrge-tasemelise programmeerimiskeelena rakenduste loomiseks Sharemind raamistikule. Programmeerimiskeelena eristab SecreC avalikke andmeid salajastest andmetest ning garanteerib, et operatsioonid, mis tehakse salajaste andmete peal ei leki informatsiooni avalikesse väljunditesse. Ometi on programmeerijal kasutada eksplitsiitne keelekonstruktsioon salajaste andmete avaldamiseks, mistõttu kontroll infolekete üle jääb programmeerija vastutuseks. Lihtsustamaks programmeerija ülesannet verifitseerida infovooge nende turvalisuse aspektist, otsustasime luua raamistiku, mis võimaldaks sooritada andmevoo-analüüse SecreC programmide peal.

Loodud raamistiku suurima osa moodustab SecreC programmeerimiskeele esimene formaalne spetsifikatsioon, mis koosneb SecreC kontekstivabast grammatikast, reeglistikust SecreC programmide staatiliseks kontrolliks ning keele olekuteisendussüsteemina kirjeldatud operatsioonsemantikast. Enne käesoleva töö kirjutamist kirjeldas SecreC programmeerimiskeelt ainult vastav kompilaator, mille keelekäsitluses eksisteeris mitu olulist puudust. Loodud formaalne spetsifikatsioon on mõeldud teoreetiliseks aluseks nii keele tööriistadele kui ka keelde lisatavatele laiendustele, millest tähtsamaiks on massiivide ja maatriksite tugi.

Kasutades alusena loodud spetsifikatsiooni, oleme implementeerinud analüsaatori SecreC programmide analüüsiks. Realiseerisime analüsaator programmeerimiskeele C++ teegina üheskoos lihtsa käsurea-põhise testrakendusega, mis jooksutab etteantud SecreC programmitekstil implementeeritud andmevoo-analüüse. Analüsaatori poolt kasutatav algoritm on võimeline jooksutama mõlema-suunalisi rajatundlikke andmevoo-analüüse, mida on võimalik algoritmile edastada argumendina. Loodud analüsaatorit võib kasutada nii optimeerivates kompilaatorites kui ka arenduskeskkondades, mis pakuvad programmeerijale visuaalset tagasisidet.

Implementeerisime analüsaatoris ka kolm lihtsat andmevoo-analüüsi. Esiteks realiseerisime nn *reaching definitions* analüüsi, mis tuvastab muutujatele omistavate avaldiste mõjupiirkonna. Teine loodud analüüs demonstreerib analüsaatori rajatundlikust, tuvastades tingimuslike hargnemiste harud, mille kaudu on võimalik jõuda mõnda programmi punkti. Kolmas analüüs tuvastab üldistatud kujul infolekkeid, määrates protseduuri privaatsed argumendid ning privaatsed tagastusväärtused delikaatseteks andmeteks ning kontrollides, et vastavaid delikaatseid andmeid ei avaldataks **`declassify`** operaatori abil. Sealjuures ei kaota info oma delikaatsust tavalise omistamise või unaarsete operatsioonide kaudu. Olenevalt hargnemistest koodis oskab see analüüs tuvastada, kas delikaatset info lekitatakse igal juhul või ainult mõne valitud täitmisraja puhul.

SecreC formaalne spetsifikatsioon ja analüsaator on asendamatuks vundamendiks keele ja selle tööriistade edasiseks arenduseks. Saavutatu tulemusena alustasime ka võimaluste uurimist täiemahulise infolekete tuvastamise ja mõõtmise analüüsi realiseerimiseks, oleme vastu võtnud olulisi otsuseid SecreC tuleviku suhtes ning alustanud ka meie eesmärke toetavate analüüside implementeerimist.

Autor tänab oma juhendajat Dan Bogdanovit tema rohke pühendumise, kasulike nõuannete ja teema püstituse eest, mis võimaldasid käesoleva magistritöö valmimise; ning kõiki teisi, kes hea nõu ja toega suureks abiks olid, eriti Liina Kammi.

Appendix A. Typing for Binary Operations

The typing rules for binary operators for SecreC are given in the table below. A binary operation is only typeable if there is a row in the table corresponding to the binary operator and the types of its operands. Otherwise, the binary operation is considered to be not typeable.

Binary operators	Type of first 1st operand	Type of first 2nd operand	Type of result
Logical OR: `||` Logical AND: `&&`	`bool`	`bool`	`bool`
Equality: `==` Inequality: `!=` Less-than: `<` Less-or-equal: `<=` Greater-or-equal: `>=` Greater-than: `<=`	`bool`	`bool`	`bool`
	`int`	`int`	`bool`
	`int`	`unsigned int`	`bool`
	`unsigned int`	`int`	`bool`
	`unsigned int`	`unsigned int`	`bool`
	`string`	`string`	`bool`
String concatenation: `+`	`string`	`string`	`string`
Addition: `+` Subtraction: `-` Multiplication: `*` Division: `/` Modulo: `%`	`int`	`int`	`int`
	`unsigned int`	`unsigned int`	`unsigned int`

Appendix B. Algorithm for data-flow analyses in SecreC Analyzer

The followign is a simplified outline of the generic iterative algorithm used for data-flow analysis purposes in the SecreC Analyzer.

INPUT:

- Control flow graph CFG where $(P, T, S) \in CFG$ is an edge of type T from block P to block S.
- A set of data-flow analysis objects $AS = BAS \cup FAS$ where
  - FAS is the set of forward data-flow analysis objects,
  - BAS is the set of backward data-flow analysis objects and
  - each analysis object $A \in AS$ represents some data-flow analysis and has
    1. a direction $D \in \{\text{backward}, \text{forward}\}$ for the analysis,
    2. a set V of data-flow values,
    3. for each basic block $B \in CFG$ values $IN_A[B], OUT_A[B] \in V$ which contain the data-flow values for the beginning and end of that block respectively,
    4. an initialization method $INIT_A$ which takes the CFG as an argument and initializes the state of the analysis object, including the IN_A and OUT_A sets,
    5. a data-flow transfer function f_A,
    6. a data-flow meet operator $\bigwedge^A$ which for path-sensitivity takes operands of type $V \times T$ where the second component is the type of the edge which connects the corresponding blocks,
    7. a method $CLEANUP_A$ which executes any clean-up code after completing the analysis.

ALGORITHM:

```
 1. $BAS^+ = \emptyset$
 2. $FAS^+ = \emptyset$

 3. for each analysis $FA$ in $FAS$
 4.     run $INIT_{FA}(CFG)$
 5.     put $FA$ into $FAS^+$

 6. for each analysis $BA$ in $BAS$
 7.     run $INIT_{BA}(CFG)$
 8.     put $BA$ into $BAS^+$

 9. while $FAS^+ \cup BAS^+ \neq \emptyset$:
10.     for each basic block $B$ in $CFG$:
11.         if $B$ is not the exit block and $BAS^+ \neq \emptyset$:
12.             for each analysis $BA$ in $BAS^+$
13.                 $OUT_{FA}[B] = \bigwedge^{FA}_{(B,T,S) \in CFG} (IN_{FA}[S], T)$
14.                 $IN_{FA}[B] = f_{FA}(B, OUT_{FA}[B])$

15.         if $B$ is not the entry block and $FAS^+ \neq \emptyset$:
16.             for each analysis $FA$ in $FAS^+$
17.                 $IN_{FA}[B] = \bigwedge^{FA}_{(P,T,B) \in CFG} (OUT_{FA}[P], T)$
18.                 $OUT_{FA}[B] = f_{FA}(B, IN_{FA}[B])$

19.     for each analysis $FA$ in $FAS^+$
20.         if any $OUT_{FA}$ wasn't changed
21.             remove $FA$ from $FAS^+$

22.     for each analysis $BA$ in $BAS^+$
23.         if any $IN_{BA}$ wasn't changed
24.             remove $BA$ from $BAS^+$

25. for each analysis $A$ in $AS$
```

```
26.     run $CLEANUP_A()$
```

Appendix C. Source code for SecreC Analyzer library

The source code of the SecreC Analyzer is included on a CD. The code is still a work-in-progress and has only been tested on a 64-bit Gentoo Linux distribution running on a processor with x86-64 architecture. Instructions to compile the library are included in the `README` file. To run the command-line test tool on SecreC programs, the `runsca.sh` Bash shell script can be used. The script expects the file name of a SecreC program as a command-line argument.