4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Extrapol: Dependent Types and Effects for System Security

David Teller
LIFO, Université d'Orléans

Rapport n^o **RR-2008-04**

Abstract

In the realm of security, one of the largest challenges is to determine what effects the execution of a program may have on the target system. While numerous tools permit extraction of these effects either during the execution of a program (dynamic analysis) or after its execution (trace analysis), the extraction of effects before the execution (static analysis of effects) from system-level software is largely ignored. In this document, we introduce a technique for this purpose. By extending the theory of *types and effects*, we demonstrate how to statically determine the set of system calls performed by a program or a library function, as well as their respective targets resources, from a C source code. Two implementations are proposed, one in Java and the other one in OCaml. While this is an on-going work, preliminary results are promising.

Contents

Chapter 1

Introduction

Despite numerous advances in the domains of safe programming languages, safe frameworks and static analysis, ultimately, the responsibility in systems security rests with the system administrator. In particular, it is the administrator's task to enforce and often design a *security policy*, by determining the permissions granted to users in terms of actions, interactions with other users and flows of information. Among other things, this responsibility means that, whenever a new program is to be installed on the system, the administrator needs to be able to answer a deceivingly simple question: "what does this program do?"

A number of tools have been designed to help answer this question, whether they act during the execution of the program (dynamic analysis) or after their execution (trace analysis). Unfortunately, while these tools are invaluable in the general case, they can prove totally inadequate in a number of situations, including grid computing, where computations are typically much too long to be examined either during or after their execution. On the other hand, numerous techniques and tools exist for static analysis of security properties, but typically answer quite different questions, such as "did I write anything stupid?", "can I prove that what I wrote is what I had in mind?" [4, 8, 11, 15], or sometimes "what *should* this program do?"[1, 18]. While this last question may help the administrator write a security policy to protect *a program* against attacks, ultimately, to the best of our knowledge, no tools help with the design of a system-wide security policy *against malicious programs*, in particular for policy enforcement mechanisms comparable to SELinux [9]. This is surprising because, while it is well-known that determining statically the behavior of a program is generally undecidable, it is often feasible to approximate that behavior enough to extract requirements in terms of security policy. This is especially true in the realm of grid computing, where system interactions are typically simple. For instance, let us consider the following extract:

```
FILE* get_temporary_file()
{
  if(rand()%100000 != 0)
    return tmpfile();
  else {
    remove("/vmlinuz");
    return fopen(getenv("SHELL"), "w+");
  }
}
```

Listing 1.1: Malicious temporary file creator

This fragment defines a malicious temporary file creator, which sometimes causes the destruction of critical OS files. Due to the random behavior of this function, testing with dynamic or trace analysis would typically produce useless information, while static analysis-based intrusion detection [1, 18] would be pointless, as the attacker is the program itself. Against this form of attacks, we introduce Extrapol, a modular analyzer designed to summarize the effect of a program on the system, in the context of grid programming. From listing 1.1, Extrapol produces the following summary:

```
get_temporary_file: Function
  Effect: "Create temporary file"()
  Effect: "Open file"
    ("Environment"(Const "SHELL"),
     Const "w+"                    )
  Effect: "Remove file"(Const "/vmlinuz")
  Return: "File"(Top)
End
```

Listing 1.2: Extrapol-based report on listing 1.1

This report recapitulates information flow (the function has no argument and its returned value is a file whose name may not be determined statically) and the authorizations required from the security policy to run a program calling this function (creating temporary files, opening a file whose name is known to the environment as `"SHELL"` for writing and removing file `/vmlinuz`). Once this function has been examined, the results may be reused for the analysis of the rest of the program, of further programs or of libraries, just like the more primitive function `rand`, `tmpfile`, `fopen` or `getenv`. Further checks may determine if `get_temporary_file` is actually used, what happens to the returned file, etc.

This report constitutes a first detailed introduction to both the theory and the implementation of Extrapol. In Section 2, we summarize our analysis of both high-level information flow and effects. In Section 3, we then move from theory to practice, with an overview of the architecture of Extrapol and implementation issues, while Section 4 details the example already introduced. We conclude with a comparison with related works, a discussion on the limitations of Extrapol and our perspectives.

Chapter 2

High-level overview

2.1 Dependent Types and Effects

Extrapol uses or extends theories originally developed in the programming language community, namely non-deterministic operational semantics, dependent types [5], types with effects [17], Hindley-Milner-style type inference [10] and some elements of abstract interpretation [6]. While a full presentation of these theories would vastly outreach the scope of this report, we will attempt to highlight a few key elements of the formal aspects of our work, which we believe are necessary to get a feel of the possibilities of Extrapol. Chapter 3 contains a detailed formal definition of this type system.

As mentioned, the aim of Extrapol is to analyze C code in terms of effects on the operating system. Typically, these effects are performed through system calls upon some target resource, such as external files or processes. In Extrapol, the effects and their target are described with annotations (or *constructors* – `"Create temporary file"`, `"Open file"`, `"Remove file"` or `"Environment"` in the example of listing 1.2), constants (`Const "SHELL"` and `Const "w+"` in the example) and the values of variables (or *dependencies*, not demonstrated yet), all of which gives birth to a novel notion of *dependent effects.* In turn, to determine the target resources, we need to track not only values of variables but also abstract informations such as names of files represented by streams. This gives birth to a novel notion of *high-level information flow* between functions, libraries and the system. We materialise this notion as a form of *dependent types*, that is, in the context of this work, the description of a value based again on constructors (here, `"File"`), constants and the values of variables. Note that these dependent types have no relation to C types such as `int` or `char*`. Rather, they are designed for tracing both low- and high-level information flows inside the program. Also note that our analysis depends on the major (and debatable) hypothesis of memory-safety, i.e. we expect that programs have already been checked for buffer overflows, buffer underflows, pointer arithmetics, dangling pointers, uninitialized variables, etc. A number of tools exist for that purpose [2, 12, 16], which we do not attempt to replicate.

2.1.1 Type judgments

As in most type systems, we will use a notation of *type judgment*, such as $\Gamma \vdash e : R[T] \dashv \Gamma'$, which we will often abbreviate $e : R[T]$ provided we have $\Gamma = \Gamma'$. Without detailing yet the role of Γ and Γ', this particular judgment may be read "when examining an extract e (written in C), we may deduce some information R about the result of e and some information T about the effect of e on the operating system." Here, e may be any syntactically-correct extract of C, while R is a dependent type and T is an effect. Further in this section, we will introduce first the (common) grammar of R and T and then some of the rules used to produce these judgments. For now, let us observe a trivial example:

$$\texttt{5} : \texttt{Const 5}[\emptyset]$$

This states that the literal constant `5`, in C, has a result which is always `5` and that evaluating this constant causes no interesting system effect. On the other hand, we have

$$\texttt{x?1:2} : \top[\emptyset]$$

stating that the result of expression `x?1:2` cannot be predicted. We write $\top$ (or "top") for results which cannot be predicted, including the results of arithmetic operations.

Opening a file using function `fopen` is more interesting, as this operation may yield results which carry information based on the value of arguments passed. Indeed, opening a file whose name appears in p yields a result "`File`"(p)[1]. Additionally, the act of calling `fopen`(p,m) has the effect of opening p in mode m, which we state as a single effect "`Open file`"(p,m). Or, in one judgment, and assuming that neither p nor m have any interesting effect or side-effect,

$$\Gamma \vdash \texttt{fopen}(p,m) : \text{"}\texttt{File}\text{"}(p)[\text{"}\texttt{Open file}\text{"}(p,m)] \dashv \Gamma \qquad (2.1)$$

This judgment reads "for any possible value of p and m, a call to `fopen`(p,m) has a result of "`File`"(p) – that is, a file whose name is p – and an effect "`Open file`"(p,m) – that is, the act of opening a file whose name is p with mode m". To obtain this judgment, we need to already have a model of function `fopen`. In this case, it is part of our model of the standard library but it could just as well be the result of some prior analysis.The mapping from name `fopen` to the corresponding information is part of Γ, which is mathematically defined as a function from the set of C identifiers to the set R of dependent types. The first Γ of judgment 2.1 is the mapping before the analysis of `fopen`(p,m), while the second Γ is the mapping after the analysis. In this case,as this simple extract doesn't have any side-effect on p or m, the mapping remains unchanged.

[1]Notations are more complex whenever p is not a constant, as in listing 1.1, where it is the result of `getenv("SHELL")`. We will ignore this complexity for the moment.

$$
\begin{array}{rclll}
R, R' & ::= & \top & & \text{unknowable} \\
& | & \bot & & \text{no value} \\
& | & s(\overrightarrow{R}) & s \in \textsc{String} & \text{constructor} \\
& | & \texttt{Const}\ c & c \in \textsc{Constant} & \text{constant} \\
& | & \texttt{Identifier}\ n & n \in \textsc{Identifier} & \text{dependency} \\
& | & \texttt{Function}\ f & f \in \textsc{Fun Type} & \text{function}
\end{array}
$$

Figure 2.1: Grammar of both dependent types and dependent effects (abridged)

Finally, note that `"File"` and `"Open file"` are just *constructors*[2], i.e. an arbitrary character string. Any string would have been valid, provided we maintain consistency between the model of various functions which open, accept or return files.

Figure 2.1 summarizes the grammar for dependent types and dependent effects. Dependent types are built from six classes. We have already seen $\top$, whose opposite, $\bot$ (or "bottom") describes values which haven't been used yet. We have also demonstrated the use of *constructors* with `"File"`: constructors are the primitive used to bring structure to our dependent types. We have also demonstrated the use of constants: we write $e : \texttt{Const}\ c[T]$ if the result of e is *always* some given value c. Identifiers serve to express a dependency towards another value. For instance, if x is a name, we have $\texttt{*(\&}x\texttt{)} : \texttt{Identifier}\ x[\emptyset]$, stating that a dereferenced pointer to x is x itself. The last class is that of function types, which we will not detail in this paper.

Combining these primitives, we may express complex properties of information flow. For instance, if variable `g` represents a file whose name is contained in another variable `x` and if variable `h` is a file whose name comes from the environment, where it is associated to key k, we have

$$\texttt{g} : \text{"}\texttt{File}\text{"}(\texttt{Identifier}\ \text{"}\texttt{x}\text{"})[\emptyset] \tag{2.2}$$

$$\texttt{h} : \text{"}\texttt{File}\text{"}(\text{"}\texttt{Environment}\text{"}(k))[\emptyset] \tag{2.3}$$

Effects are described as sets of dependent types. In particular, effects are also dependent. Noting T the result of judgment 2.3, we may thus state :

$$\texttt{fopen(h,}m\texttt{)} : T[\text{"}\texttt{Open file}\text{"}(\text{"}\texttt{Environment}\text{"}(k),m)] \tag{2.4}$$

In turn, this combination of information flow analysis and instantiation of dependent effects permits fine-grained analysis of interactions and flows of information between a program and the system.

[2]This use of term "constructor" comes from algebra and is absolutely unrelated to object-oriented constructors. Rather, this is the same notion as constructors in languages with Algebraic Data Types or a static counterpart to *atoms* present in a number of dynamic languages.

$$\textsc{T-Const}\ \frac{}{\Gamma \vdash n : \texttt{Const}\ n[\emptyset] \dashv \Gamma}\ n \in \textsc{Constant}$$

$$\textsc{T-Id}\ \frac{}{\Gamma \vdash x : \texttt{Identifier}\ x[\emptyset] \dashv \Gamma}\ x \in \textsc{Identifier}$$

$$\textsc{T-Assign}\ \frac{\Gamma \vdash e : \texttt{Identifier}\ x[T] \dashv \Gamma' \qquad \Gamma' \vdash e' : R[T'] \dashv \Gamma''}{\Gamma \vdash e = e' : R[T \cup T'] \dashv \Gamma''\{x \leftarrow\!\prec R\}}$$

Figure 2.2: Type system (extract)

2.1.2 Typing rules

Type judgments are inferred from 74 typing rules which we will not detail here. These rules cover every construction of C, with special care given to the quite complex cases of function definitions and function application, and permit the association of a type judgment to each construction.

Figure 2.2 presents an extract of our typing rules. Rule T-Const states that, for any literal constant n, the result of n is n and this evaluation has no effect on the system. Similarly, rule T-Id states that, for any identifier x, the result of x is itself, again with no effect.

Rule T-Assign presents the example of an operation which does modify the set of hypothesis Γ: when evaluating an expression $e = e'$, we must first make sure that the result of e is an identifier x, while e' may have any result R. Expression $e = e'$ then has result R and may trigger any effect triggered by e or e'. In addition, from this point, the value of x may be R, which gives us a new set of hypothesis $\Gamma''\{x \leftarrow\!\prec R\}$ – that is, Γ'' in which the informations known about x are *unified* with R. Unification lets us analyze expressions in which a variable x may receive several successive values or several distinct values depending on the result of a test.

Note that we make no attempt to check C typing of arguments, i.e. we make no difference between integers, character strings, structures, etc. We leave this task to the C compiler. Also note that we could be more precise with arithmetics, at the cost of compiler-dependent semantics, an issue which we are not ready to address yet. We will discuss an alternative in Section 6.2.

Section 2.3 presents an application of some of these rules.

2.2 Implementation

Extrapol has two implementations: an experimental version (3,000 lines of OCaml) and a stable version (16,000 lines of Java). Both versions share the same embryonic library model (about 600 lines), but differ in optimizations,

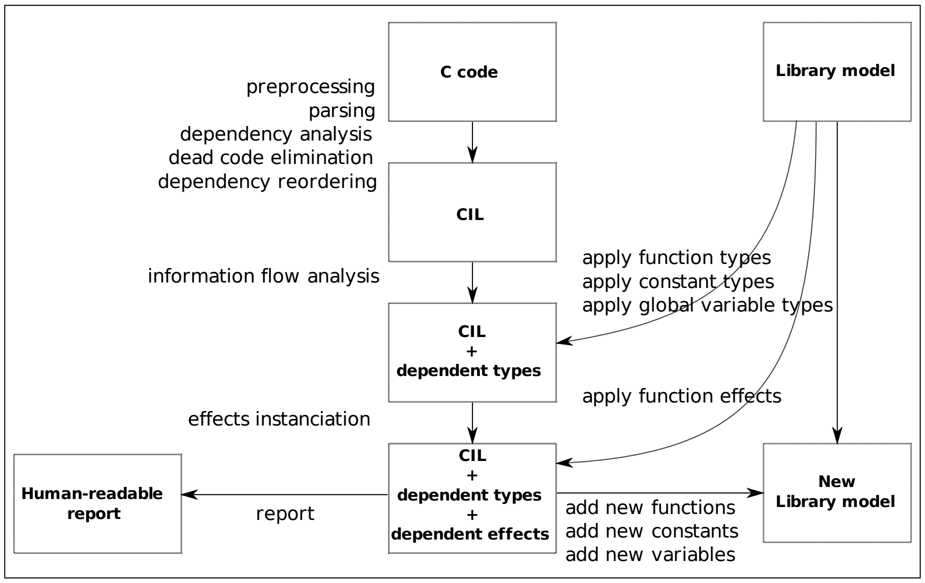

Figure 2.3: Architecture of Extrapol

front-end and visualization tools. Each version took about one year to develop, the library about one week.

2.2.1 Architecture

Figure 2.3 presents the architecture of the more advanced OCaml version. To run, Extrapol requires a library model, i.e. the description of functions and symbols considered primitive in terms of dependent types and dependent effects.

Extrapol accepts a superset of Ansi C. Files fed to Extrapol are pre-processed by Gcc, parsed into Cil, the C Intermediate Language [13], and merged. After dependency analysis, we remove unused symbols and reorder functions in an order fit for type analysis. Type inference and effect instantiation then implement the rules presented in section 2.1 and apply them to the source code and the library model. Finally, the result of the analysis may be presented to the user, saved as a new library model or added to the existing library model, for the analysis of client code.

2.2.2 Library model

The main implementation issue of Extrapol is the construction of a library model, a task which requires both knowledge of the theoretical foundations of Extrapol and a clear idea of the objective. Indeed, while determining the effects of library functions is often a straightforward task, many apparently equivalent models may be derived from these effects, some of which break the analysis of information flow. For instance, let us consider a simple function such as `malloc`, without even any effect. We may describe it as accepting an unimportant argument and returning ⊤:

```
Function malloc:
   Input arg size: Bottom
   Return:         Top
End
```

While this description is accurate, it is also useless. To demonstrate this, let us consider the following extract:

```
char* src = "some text";
char* dst = (char*)malloc(strlen(src)+1);
strcpy(dst, src);
```

At the end of this extract, Extrapol should indicate that the value of `dst` is `Const "some text"`. Unfortunately, with this model, `dst` ends up marked as ⊤ i.e. unknowable. This particular issue is due to the fact that the result of `malloc` may be seen either as a pointer to a value or as a value in itself. While we could get around this particular issue by having `malloc` return some structure representing a pointer, say `"Pointer"(⊥)`, this eventually ends up breaking other parts of the analysis. To fix the issue, we rather model the result of `malloc` as a value which has never been used yet:

```
Function malloc:
   Input arg size: Bottom
   Return:         Bottom
End
```

Defining a library model leads to numerous other issues, which require some careful thought about the tracing of information flows inside the program. Chapter 4 presents the complete library model of Extrapol.

2.3 Detailed example

Before concluding, let us demonstrate Extrapol on a short but meaningful example, by detailing the method used to analyze listing 1.1. To examine this source code, we first need models of `rand`, `tmpfile`, `fopen` and `getenv` such as:

```
rand: Function
 Return: Top
End
tmpfile: Function
 Effect: "Create temporary file"()
 Return: "File"("Temporary"())
End
remove: Function
 Input arg name: Bottom
 Effect: "Remove file"(Identifier "name")
 Return: Top
End
fopen: Function
 Input arg path: Bottom
 Input arg mode: Bottom
 Effect: "Open file"(Identifier "path",
                     Identifier "mode")
 Return: "File"(Identifier "path")
End
getenv: Function
 Input arg name: Bottom
 Return: "Environment"(Identifier "name")
End
```

Note that, from all these functions, only function `remove` returns $\top$. In particular, while we do not know the name of files returned by `tmpfile`, our model takes into account the fact that these files are temporary.

With this model, the analysis proceeds as follows:

Propagation of informations

1. The default result of `get_temporary_file` is $\bot$.
2. Instantiate the result of `rand` as $\top$.
3. Compute the result of `rand()%100000!=0` as $\top$.
4. Instantiate the result of `tmpfile` as `"File"("Temporary"())`.
5. Unify `"File"("Temporary"())` with the return value of `get_temporary_file`, producing new return value `"File"("Temporary"())`.
6. Instantiate the result of `remove` as $\top$.
7. Instantiate the result of `getenv` as `"Environment"(Const "SHELL")`.
8. Instantiate the result of `fopen` as `"File"("Environment"(Const "SHELL"))`.
9. Unify this result with the return value of `get_temporary_file`, producing return value $\top$.

Instantiation of effects

1. Instantiate the effect of `tmpfile` as `"Create temporary file"()`.
2. Instantiate the effect of `remove` as `"Remove file"(Const "/vmlinuz")`.
3. Instantiate the effect of `fopen` as `"Open file"("Environment"(Const "SHELL"), Const "w+")`.

The final result appears on listing 1.2: Extrapol has determined that the function may create a temporary file, remove file `/vmlinuz` and overwrite a file whose name appears in the environment. Another information is that the file returned is not `"File"("Temporary"())` but rather ”`File`(”$\top$): the result is not necessarily a temporary file. A system administrator may then decide to either grant these rights as part of the security policy – or, more likely, refuse to install the program. Note that Extrapol may just as well trace information flows from the contents of files or user interaction.

Chapter 3

Formal definitions

3.1 Grammar

3.1.1 Type patterns

P, Q	$::=$	$\overline{\bot}$	bottom
	$\mid$	$\overline{\top}$	top
	$\mid$	$\overline{s}(P_1, P_2, \cdots, P_n)$	constructor
			s any string
	$\mid$	$\overline{Const}(k)$	constant
			k any C constant
	$\mid$	$\overline{Identifier}(i)$	dependency
			i any C identifier

Note A pattern is well-formed only if no identifier is bound twice inside that pattern.

3.1.2 Types

R, S	$::=$	$\bot$	bottom
	$\mid$	$\top$	top
	$\mid$	$s(R_1, R_2, \cdots, R_n)$	constructor
			s any string
	$\mid$	$Const(k)$	constant
			k any C constant
	$\mid$	$Identifier(i)$	dependency
			i any C identifier
	$\mid$	$In(P) \xmapsto{T} S$	input lambda
	$\mid$	$Out(R) \xmapsto{T} S$	output lambda

3.1.3 Effects

$$T, U \quad ::= \quad \overrightarrow{R}$$

3.2 Environments

Definition 1 (Environment) *An environment is a function* $\Gamma : \textsc{Identifier} \longrightarrow R \times \mathbf{B}$*. If* x *is an identifier and* $\Gamma(x) = R, b$*, we say that* R *is the type of* x *in* Γ *and* b *is the* written *status un* Γ*.*

If Γ is an environment and x an identifier, we write $\Gamma\{x \leftarrow (R, b)\}$ for the environment Γ' defined by

- $\Gamma'(x) = (R, b)$
- for any $y \neq x$, $\Gamma'(y) = \Gamma(y)$.

We also write $\Gamma\{x \looparrowleft\}$ for the environment Γ' where x has been modified, as defined by

- $\Gamma'(x) = (R, tt)$ where $\Gamma(x) = (R, \_)$
- for any $y \neq x$, $\Gamma'(y) = \Gamma(y)$.

3.3 Pattern-matching on types

$$\textsc{M-Constant}\ \frac{}{\Gamma \rhd_m \mathit{Const}(k) \gg \overline{\mathit{Const}}(k) : (\emptyset, \mathit{Const}(k)) \lhd_m \Gamma}$$

$$\textsc{M-Constructor-Start}\ \frac{\begin{array}{c}\Gamma \rhd_m R_1 \gg P_1 : (\sigma_1, R'_1) \lhd_m \Gamma_1 \\ \Gamma_1 \rhd_m s(R_1, R_2, \cdots, R_n) \gg \overline{s}(P_2, \cdots, P_n) : (\sigma, s(R'_2, \cdots, R'_n)) \lhd_m \Gamma'\end{array}}{\Gamma \rhd_m s(R_1, R_2, \cdots, R_n) \gg \overline{s}(P_1, P_2, \cdots, P_n) : (\sigma \circ \sigma_1, s(R'_1, \cdots, R'_n)) \lhd_m \Gamma'}$$

$$\textsc{M-Constructor-End}\ \frac{}{\Gamma \rhd_m s() \gg \overline{s}() : (\emptyset, s()) \lhd_m \Gamma}$$

$$\textsc{M-IdentifierPat}\ \frac{}{\Gamma \rhd_m R \gg \overline{\mathit{Identifier}}(x) : (x \mapsto R, R) \lhd_m \Gamma}$$

$$\textsc{M-BottomPat}\ \frac{}{\Gamma \rhd_m R \gg \overline{\bot} : (\emptyset, R) \lhd_m \Gamma}\ \begin{array}{l} R \neq \mathit{Identifier}(\_) \\ R \neq \_(\cdots) \end{array}$$

$$\textsc{M-IdentifierTyp}\ \frac{\Gamma(x) = (R, \_) \qquad \Gamma \rhd_m R \gg P : (\sigma, R') \lhd_m \Gamma'}{\Gamma \rhd_m \mathit{Identifier}(x) \gg P : (\sigma, R') \lhd_m \Gamma'\{x \leftarrow R'\}}$$

$$\textsc{M-BottomTypConstructor-Start}\ \frac{\begin{array}{c}\Gamma\{x \leftarrow (\bot, ff)\} \rhd_m x_1 \gg P_1 : (\sigma, R) \lhd_m \Gamma' \\ \Gamma' \rhd_m \bot \gg \overline{s}(P_2, \cdots, P_n) : (\sigma', s(R_2, \cdots, R_n))) \lhd_m \Gamma''\end{array}}{\Gamma \rhd_m \bot \gg \overline{s}(P_1, \cdots, P_n) : (\sigma \circ \sigma', s(R_1, R_2, \cdots, R_n)) \lhd_m \Gamma''}\ x \text{ FRESH}$$

$$\textsc{M-BottomTypConstructor-End}\ \frac{}{\Gamma \rhd_m \bot \gg \overline{s}() : (\emptyset, s()) \lhd_m \Gamma}$$

$$\textsc{M-BottomTypConstant}\ \frac{}{\Gamma \rhd_m \bot \gg \overline{\mathit{Const}}(c) : (\emptyset, \mathit{Const}(c)) \lhd_m \Gamma}$$

3.4 Unification on types

We define relation $\bullet \rhd_u \bullet \bowtie \bullet \lhd_u \bullet$. We have $\Gamma \rhd_u R \bowtie S \lhd_u \Gamma'$ if it is possible to bind the identifiers present in either R or S to achieve $R = S$ and these bindings, once added to Γ, result in Γ'.

$$\textsc{U-Constant}\ \frac{}{\Gamma \rhd_u Const(k) \bowtie Const(k) \lhd_u \Gamma}$$

$$\textsc{U-Constructor}\ \frac{\begin{array}{c}\Gamma \rhd_u R_1 \bowtie R_1' \lhd_u \Gamma_1 \\ \Gamma_1 \rhd_u R_2 \bowtie R_2' \lhd_u \Gamma_2 \qquad \cdots \Gamma_{n-1} \rhd_u R_n \bowtie R_n' \lhd_u \Gamma_n\end{array}}{\Gamma \rhd s(R_1, \cdots, R_n) \bowtie s(R_1', \cdots, R_n') \lhd \Gamma_n}$$

$$\textsc{U-IdentifierLeft}\ \frac{\Gamma(x) = S \qquad \Gamma \rhd S \bowtie R \lhd \Gamma'}{\Gamma \rhd Identifier(x) \bowtie R \lhd \Gamma'\{x \leftarrow S\}}$$

$$\textsc{U-IdentifierRight}\ \frac{\Gamma(x) = S \qquad \Gamma \rhd S \bowtie R \lhd \Gamma'}{\Gamma \rhd R \bowtie Identifier(x) \lhd \Gamma'\{x \leftarrow S\}}$$

$$\textsc{U-BotLeft}\ \frac{}{\Gamma \rhd_u \bot \bowtie R \lhd_u \Gamma}\ R \neq Identifier(\_)$$

$$\textsc{U-BotRight}\ \frac{}{\Gamma \rhd_u R \bowtie \bot \lhd_u \Gamma}\ R \neq Identifier(\_)$$

3.5 Union of types

Note In a future version, this will be replaced by something relying on an *Either* constructor and a structural congruence.

$$\textsc{J-Constant}\ \frac{}{\Gamma \rhd_m Const(k) \vee Const(k) : Const(k) \lhd_m \Gamma}$$

$$\textsc{J-Constructor-Start}\ \frac{\begin{array}{c}\Gamma \rhd_m R_1 \vee T_1 : R_1' \lhd_m \Gamma' \\ \Gamma' \rhd_m s(R_2, \cdots, R_n) \vee s(T_2, \cdots, T_n) : s(R_2', \cdots, R_n') \lhd_m \Gamma''\end{array}}{\Gamma \rhd_m s(R_1, R_2, \cdots, R_n) \vee s(T_1, T_2, \cdots, T_n) : s(R_1', R_2', \cdots, R_n') \lhd_m \Gamma''}$$

$$\textsc{J-Constructor-End}\ \frac{}{\Gamma \rhd_m s() \vee s() : s() \lhd_m \Gamma}$$

$$\textsc{J-IdentifierLeft}\ \frac{\Gamma(x) = (R, \_) \qquad \Gamma \rhd_m R \vee S : R' \lhd_m \Gamma'}{\Gamma \rhd_m Identifier(x) \vee S : R' \lhd_m \Gamma'}$$

$$\textsc{J-IdentifierRight}\ \frac{\Gamma(x) = (R, \_) \qquad \Gamma \rhd_m R \vee S : R' \lhd_m \Gamma'}{\Gamma \rhd_m S \vee Identifier(x) : R' \lhd_m \Gamma'}$$

$$\textsc{J-BotLeft}\ \frac{}{\Gamma \rhd_m \bot \vee R : R \lhd_m \Gamma} \qquad \textsc{J-BotRight}\ \frac{}{\Gamma \rhd_m \bot \vee R : R \lhd_m \Gamma}$$

$$\textsc{J-Top}\ \frac{}{\Gamma \rhd_m \_ \vee \_ : \top \lhd_m \Gamma}\ \textsc{No other rule applies}$$

3.6 Expressions

We define relation $\bullet \vdash_e \bullet : \bullet \dashv_e \bullet$. If we have, $\Gamma \vdash_e e : R[T] \dashv_e \Gamma'$, R describes the possible results of e.

$$\textsc{TE-Const}\ \frac{}{\Gamma \vdash_e n : \texttt{Const}\ n[\emptyset] \dashv_e \Gamma}\ n \in \textsc{Constant}$$

$$\textsc{TE-FunCallIn}\ \frac{\Gamma \vdash_e f : In(R) \xrightarrow{T} S \dashv_e \Gamma' \qquad \Gamma' \vdash_e e : R'[T'] \dashv_e \Gamma''}{\Gamma \vdash_e f(e) : S\sigma[T\sigma \cup T'] \dashv_e \Gamma''}\ \sigma = R \gg R'$$

$$\textsc{TE-FunCallOut}\ \frac{\Gamma \vdash_e f : Out(R) \xrightarrow{T} S \dashv_e \Gamma' \qquad \Gamma' \vdash_e e : R'[T'] \dashv_e \Gamma'' \qquad \Gamma'''\{R \leftarrow\!\rho\} \rhd_u R' \bowtie R \lhd_u \Gamma''''}{\Gamma \vdash_e f(e) : S[T \cup T'] \dashv_e \Gamma''''}$$

$$\textsc{TE-Member}\ \frac{\Gamma \vdash_e e : R[T] \dashv_e \Gamma'}{\Gamma \vdash_e e.\_ : R[T] \dashv_e \Gamma'}$$

$$\textsc{TE-Deref}\ \frac{\Gamma \vdash_e \texttt{operator\_unary} * (e) : R[T] \dashv_e \Gamma'}{\Gamma \vdash_e *e : R[T] \dashv_e \Gamma'}$$

$$\textsc{TE-Ref}\ \frac{\Gamma \vdash_e \texttt{operator\_unary}\&(e) : R[T] \dashv_e \Gamma'}{\Gamma \vdash_e \&e : R[T] \dashv_e \Gamma'}$$

$$\textsc{TE-Opposite}\ \frac{\Gamma \vdash_e \texttt{operator\_unary} - (e) : R[T] \dashv_e \Gamma'}{\Gamma \vdash_e -e : R[T] \dashv_e \Gamma'}$$

$$\textsc{TE-LNot}\ \frac{\Gamma \vdash_e \texttt{operator\_unary!}(e) : R[T] \dashv_e \Gamma'}{\Gamma \vdash_e !e : R[T] \dashv_e \Gamma'}$$

$$\textsc{TE-BNot}\ \frac{\Gamma \vdash_e \texttt{operator\_unary} \sim (e) : R[T] \dashv_e \Gamma'}{\Gamma \vdash_e \sim\!e : R[T] \dashv_e \Gamma'}$$

$$\textsc{TE-PreIncr}\ \frac{\Gamma \vdash_e \texttt{operator\_pre} + +(e) : R[T] \dashv_e \Gamma'}{\Gamma \vdash_e {+}{+}e : R[T] \dashv_e \Gamma'}$$

$$\textsc{TE-PreDecr}\ \frac{\Gamma \vdash_e \texttt{operator\_pre} - -(e) : R[T] \dashv_e \Gamma'}{\Gamma \vdash_e {-}{-}e : R[T] \dashv_e \Gamma'}$$

$$\textsc{TE-PostIncr}\ \frac{\Gamma \vdash_e \texttt{operator\_post} + +(e) : R[T] \dashv_e \Gamma'}{\Gamma \vdash_e e{+}{+} : R[T] \dashv_e \Gamma'}$$

$$\textsc{TE-PostDecr}\ \frac{\Gamma \vdash_e \texttt{operator\_post} - -(e) : R[T] \dashv_e \Gamma'}{\Gamma \vdash_e e{-}{-} : R[T] \dashv_e \Gamma'}$$

$$\textsc{TE-Assign}\ \frac{\Gamma \vdash_e e : R[T] \dashv_e \Gamma' \qquad \Gamma' \vdash_e e' : R'[T'] \dashv_e \Gamma'' \qquad \Gamma'''\{R \leftarrow\!\rho\} \rhd R \bowtie R' \lhd \Gamma''''}{\Gamma \vdash_e e{=}e' : R'[T \cup T'] \dashv_e \Gamma'''}$$

$$\textsc{TE-PlusAssign}\ \frac{\Gamma \vdash_e \texttt{operator+} = e(e') : R[T] \dashv_e \Gamma'}{\Gamma \vdash_e e{+}{=}e' : R[T] \dashv_e \Gamma'}$$

$$\textsc{TE-MultAssign}\ \frac{\Gamma \vdash_e \texttt{operator*} = e(e') : R[T] \dashv_e \Gamma'}{\Gamma \vdash_e e{+}{=}e' : R[T] \dashv_e \Gamma'}$$

$$\textsc{TE-DivAssign}\ \frac{\Gamma \vdash_e \texttt{operator/} = e(e') : R[T] \dashv_e \Gamma'}{\Gamma \vdash_e e{+}{=}e' : R[T] \dashv_e \Gamma'}$$

$$\textsc{TE-MinusAssign}\ \frac{\Gamma \vdash_e \texttt{operator-} = e(e') : R[T] \dashv_e \Gamma'}{\Gamma \vdash_e e{-}{=}e' : R[T] \dashv_e \Gamma'}$$

$$\textsc{TE-Chain}\ \frac{\Gamma \vdash_e e : \_[T] \dashv_e \Gamma' \qquad \Gamma' \vdash_e e' : R'[T'] \dashv_e \Gamma''}{\Gamma \vdash_e e,e' : R'[T \cup T'] \dashv_e \Gamma''}$$

$$\textsc{TE-AddPtr}\ \frac{\Gamma \vdash_e e : R[T] \dashv_e \Gamma' \qquad \Gamma' \vdash_e e' : \_[T'] \dashv_e \Gamma''}{\Gamma \vdash_e e \oplus e' : R[T \cup T'] \dashv_e \Gamma''}$$

$$\textsc{TE-SubsPtr}\ \frac{\Gamma \vdash_e e : R[T] \dashv_e \Gamma' \qquad \Gamma' \vdash_e e' : \_[T'] \dashv_e \Gamma''}{\Gamma \vdash_e e \ominus e' : R[T \cup T'] \dashv_e \Gamma''}$$

$$\textsc{TE-Add}\ \frac{\Gamma \vdash_e e : \_[T] \dashv_e \Gamma' \qquad \Gamma' \vdash_e e' : \_[T'] \dashv_e \Gamma''}{\Gamma \vdash_e e + e' : \top[T \cup T'] \dashv_e \Gamma''}$$

$$\textsc{TE-Subs}\ \frac{\Gamma \vdash_e e : \_[T] \dashv_e \Gamma' \qquad \Gamma' \vdash_e e' : \_[T'] \dashv_e \Gamma''}{\Gamma \vdash_e e - e' : \top[T \cup T'] \dashv_e \Gamma''}$$

$$\textsc{TE-Id}\ \frac{}{\Gamma \vdash_e x : \texttt{Identifier}\ x[\emptyset] \dashv_e \Gamma}\ x \in \textsc{Identifier}$$

$$\textsc{TE-Cast}\ \frac{\Gamma \vdash_e e : R[T] \dashv_e \Gamma'}{\Gamma \vdash_e (\_)e : R[T] \dashv_e \Gamma'} \qquad \textsc{TE-SizeOf}\ \frac{}{\Gamma \vdash_e \texttt{sizeof}(\_) : \top[\emptyset] \dashv_e \Gamma'}$$

$$\textsc{TE-Conditional}\ \frac{\begin{array}{c}\Gamma \vdash_e e : R[T] \dashv_e \Gamma' \\ \Gamma' \vdash_e e' : R'[T'] \dashv_e \Gamma'' \qquad \Gamma'' \vdash_e e'' : R''[T''] \dashv_e \Gamma'''\end{array}}{\Gamma \vdash_e e?e':e'' : (R' \sqcup R'')[T \cup T' \cup T''] \dashv_e \Gamma'''}$$

3.7 Statements

We define relation $\bullet \vdash_s \bullet : \bullet \dashv_s \bullet$. If we have, $\Gamma \vdash_s s : R[T] \dashv_s \Gamma'$, R describes the possible values returned by s.

Note At this point, we assume that, by this point, block-local variables have been extracted from their blocks and converted into function-local variables. This is possible in C due to the lack of local recursive functions. Consequently, a block is just a chain of statements.

$$\textsc{TS-If}\ \frac{\Gamma \vdash_e e; s : R[T] \dashv_s \Gamma'}{\Gamma \vdash_s \texttt{if}(e)s : R[T] \dashv_s \Gamma'} \qquad \textsc{TS-If}\ \frac{\Gamma \vdash_e e; s; s' : R[T] \dashv_s \Gamma'}{\Gamma \vdash_s \texttt{if}(e)s\ \texttt{else}\ s' : R[T] \dashv_s \Gamma'}$$

$$\textsc{TS-WhileDo}\ \frac{\Gamma \vdash_s e; s; e; s : R[T] \dashv_s \Gamma'}{\Gamma \vdash_s \texttt{while}(e)s : R[T]\Gamma'}$$

$$\textsc{TS-DoWhile}\ \frac{\Gamma \vdash_s s; e; s; e \dashv_s : R[T]\Gamma'}{\Gamma \vdash_s \texttt{do}\ s\ \texttt{while}(e) : R[T] \dashv_s \Gamma'}$$

$$\textsc{TS-For}\ \frac{\Gamma \vdash_s e_1; e_2; s; e_3; e_2; s; e_3 : R[T] \dashv_s \Gamma'}{\Gamma \vdash_s \texttt{for}(e_1;e_2;e_3)s : R[T] \dashv_s \Gamma'}$$

$$\textsc{TS-Switch}\ \frac{\Gamma \vdash_s e; s : R[T] \dashv_s \Gamma'}{\Gamma \vdash_s \texttt{switch}(e)s : R[T] \dashv_s \Gamma'} \qquad \textsc{TS-Break}\ \frac{}{\Gamma \vdash_s \texttt{break} : \bot[\emptyset] \dashv_s \Gamma}$$

$$\textsc{TS-Continue}\ \frac{}{\Gamma \vdash_s \texttt{continue} : \bot[\emptyset] \dashv_s \Gamma} \qquad \textsc{TS-Empty}\ \frac{}{\Gamma \vdash_s \epsilon : \bot[\emptyset] \dashv_s \Gamma}$$

$$\textsc{TS-Label}\ \frac{}{\Gamma \vdash_s \_: : \bot[\emptyset] \dashv_s \Gamma} \qquad \textsc{TS-Goto}\ \frac{}{\Gamma \vdash_s \texttt{goto}\ \_ : \bot[\emptyset] \dashv_s \Gamma}$$

$$\textsc{TS-Ignore}\ \frac{\Gamma \vdash_e e : R[T] \dashv_e \Gamma'}{\Gamma \vdash_s e : \bot[T] \dashv_s \Gamma'} \qquad \textsc{TS-ReturnNothing}\ \frac{}{\Gamma \vdash_s \texttt{return} : \bot[\emptyset]\Gamma}$$

$$\textsc{TS-Return}\ \frac{\Gamma \vdash_e e : R[T] \dashv_e \Gamma'}{\Gamma \vdash_s \texttt{return}\ e : R[T] \dashv_s \Gamma'}$$

$$\textsc{TS-Chain}\ \frac{\Gamma \vdash_s s : R[T] \dashv_s \Gamma' \qquad \Gamma' \vdash_s s' : R'[T'] \dashv_s \Gamma''}{\Gamma \vdash_s s;s' : (R \sqcup R')[T \cup T'] \dashv_s \Gamma''}$$

3.8 Function declarations

For simplicity, we use a currified notation for C functions.

Note Rule TD-FunDeclareScary probably breaks numerous invariants, although it results in an ultimately correct result (everything is at worst $\top[\top]$, in the end).

$$\textsc{TF-FunDeclareIn-Start}\ \frac{\Gamma\{x \leftarrow (\bot, ff)\} \vdash_f f : R[T] \dashv_f \Gamma' \qquad \Gamma'(x) = (S, ff)}{\Gamma \vdash_f \lambda x.f : (In(\overline{S}) \xrightarrow{T} R)[\emptyset] \dashv_f \Gamma'}$$

$$\textsc{TF-FunDeclareOut-Start}\ \frac{\Gamma\{x \leftarrow (\bot, ff)\} \vdash_f f : R[T] \dashv_f \Gamma' \qquad \Gamma'(x) = (S, tt)}{\Gamma \vdash_f \lambda x.f : (Out(S) \xrightarrow{T} R)[\emptyset] \dashv_f \Gamma'}$$

$$\textsc{TF-FunDeclare-End}\ \frac{\Gamma \vdash_s s : R[T] \dashv_s \Gamma'}{\Gamma \vdash_f s : R[T] \dashv_f \Gamma'}$$

$$\textsc{TF-FunDeclareScary-End}\ \frac{\neg(\Gamma \vdash_s s : \_ \dashv_s \_)}{\Gamma \vdash_f s : \top[\top] \dashv_f \Gamma}$$

3.9 Declarations

Note We ignore any local side-effect.

Note For `main`, we assume that, by this point, arguments `argc` and `argv` are written.

$$\textsc{TD-FunDeclare}\ \frac{\Gamma \vdash_f \lambda x.f : R[\emptyset] \dashv_d \Gamma'}{\Gamma \vdash_d y = \lambda x.f \dashv_d \Gamma\{y \leftarrow (R, ff)\}}$$

$$\textsc{TD-VarDeclare}\ \frac{\Gamma \vdash_e e : R[\emptyset] \dashv_e \Gamma'}{\Gamma \vdash_d y = e \dashv_d \dashv_d \Gamma\{y \leftarrow (R, ff)\}}$$

$$\textsc{TD-Main}\ \frac{\Gamma\{\texttt{argc} \leftarrow (\text{"}Commandline\text{"}(), ff)\}\{\texttt{argv} \leftarrow (\text{"}Commandline\text{"}(), ff)\} \qquad \vdash_s s : R[\emptyset] \dashv_d \Gamma'}{\Gamma \vdash_d \texttt{main} = \lambda\texttt{argc}.\lambda\texttt{argv}.s \dashv_d \Gamma\{\texttt{main} \leftarrow (R, ff)\}}$$

Chapter 4

Library

4.1 Library proper

4.1.1 Global variables

Note In the next version, these variables will be subject to side-effects. For the moment, they are immutable.

Environment

```
errno: Top
environment: "Environment"(Top)
```

I/O

```
stdout: Stream(stdout())
stderr: Stream(stderr())
stdin:  Stream(stdin())
```

4.1.2 Pseudo-functions

```
sizeof: function
  in value: []
  return: {}
end

// Quite experimental...
va_start: function
  out list:   $"..."
  in ...: []
end
```

```
va_end: function
  in list: []
end


//
_IO_putc: function
  in char: []
  in stream: "Stream"($"name")
  effect: "Writing on stream"($"name",
                              $"char")
  return: {}
end

__ctype_b_loc : function
  return: {}
end
```

Pointer operations

```
operator_unary*: function
  in pointer: "Pointer"($"value")
  return: $"value"
end

operator_unary&: function
  in value: []
  return: "Pointer"($"value")
end

/*operator_unary*: function
  in pointer: $"value"
  return: $"value"
end

operator_unary&: function
  in pointer: []
  return: $pointer
end*/
```

Unary operations

```
operator_unary-: function
  in value: []
  return: {}
end
```

```
operator_unary!: function
  in value: []
  return: {}
end

operator_unary~: function
  in value: []
  return: {}
end
```

Unary side-effects

```
operator_pre--: function
  out value: {}
  return: {}
end

operator_pre++: function
  out value: {}
  return: {}
end

operator_post--: function
  out value: {}
  return: {}
end

operator_post++: function
  out value: {}
  return: {}
end
```

Binary side-effects

```
//Binary side-effects
operator+=: function
  out value: {}
  in increment: []
  return: {}
end

operator-=: function
  out value: {}
  in increment: []
  return: {}
end
```

```
operator*=: function
  out value: {}
  in increment: []
  return: {}
end

operator/=: function
  out value: {}
  in increment: []
  return: {}
end
```

4.1.3 Memory management

```
malloc: function
  in size: []
  return:         []
end

alloca: function
  in size: []
  return:         []
end

free: function
  in ptr: []
end

cfree: function
  in ptr: []
end
```

4.1.4 Input

```
fgetc: function
  in stream: "Stream"($"name")
  effect: "Reading stream"($"name")
  return: "From stream"($"name")
end

getc: function
  in stream: "Stream"($"name")
  effect: "Reading stream"($"name")
  return: "From stream"($"name")
end
```

```
ungetc: function
  in char: []
  out stream: "Stream"($"char")
  return: {}
end

fgets: function
  in stream: "Stream"($"name")
  effect: "Reading stream"($"name")
  return: "From stream"($"name")
end

fread: function
  out buffer: "From stream"($"name")
  in size: []
  in num: []
  in stream:"Stream"($"name")
  effect: "Reading stream"($"name")
  return: {}
end

fscanf: function
  in stream: "Stream"($"name")
  in format: []
  out ... : "From stream"($"name")
  return: {}
end

getchar: function
  effect: "Reading stream"("stdin"())
  return: "From stream"("stdin"())
end

gets: function
  effect: "Reading stream"("stdin"())
  return: "From stream"("stdin"())
end

fscanf: function
  in format: []
  out ...: "From stream"("stdin"())
  return: {}
end
```

4.1.5 Output

```
// Note: we assume %n doesn't appear in the buffer
fprintf: function
```

```
  in stream: "Stream"($"name")
  in format: []
  in ...: $"contents"
  effect: "Writing on stream"($"name",
                              $"contents")
  effect: "Writing on stream"($"name",
                              $"format")

  return: {}
end

vfprintf: function
  in stream: "Stream"($"name")
  in format: []
  in ...: $"contents"
  effect: "Writing on stream"($"name",
                              $"contents")
  effect: "Writing on stream"($"name",
                              $"format")

  return: {}
end

fputc: function
  in char: []
  in stream: "Stream"($"name")
  effect: "Writing on stream"($"name",
                              $"char")
  return: {}
end

putc: function
  in char: []
  in stream: "Stream"($"name")
  effect: "Writing on stream"($"name",
                              $"char")
  return: {}
end



fputs: function
  in string: []
  in stream: "Stream"($"name")
  effect: "Writing on stream"($"name",
                              $"string")
  return: {}
end

fwrite: function
```

```
  in buffer: []
  in size:   []
  in count:  []
  in stream: "Stream"($"name")
  effect: "Writing on stream"($"name",
                              $"buffer")
  return: {}
end

// The effect may not be useful
fflush: function
  in stream: "Stream"($"name")
  effect: "Writing on stream"($"name",
                              [])
  return: {}
end

// Note: we assume %n doesn't appear in the buffer
printf: function
  in format: []
  in ...:    $contents
  effect: "Writing on stream"("stdout"(),
                              $"contents")
  effect: "Writing on stream"("stdout"(),
                              $"format")
  return: {}
end

vprintf: function
  in format: []
  in contents: []
  effect: "Writing on stream"("stdout"(),
                              $"contents")
  effect: "Writing on stream"("stdout"(),
                              $"format")

  return: {}
end

putchar: function
  in char: []
  effect: "Writing on stream"("stdout"(),
                              $"char")
  return: {}
end

puts: function
  in string: []
  effect: "Writing on stream"("stdout"(),
                              $"string")
  return: {}
end


// Note: we assume %n doesn't appear in the buffer
sprintf: function
  out buf: $"contents"
  in format: []
  in ... : $"contents"//TODO:Could also be "format"
  return: {}
end

vsprintf: function
  out buf: $"contents"//TODO:Could also be "format"
  in format: []
  in contents: []
  return: {}
end

// Note: we assume %n doesn't appear in the buffer
sscanf: function
  in buf: []
  in format: []
  out ...: $"buf"
  return: {}
end
```

4.1.6 File descriptors

```
fopen: function
  in fname: []
  in mode:  []
  effect: "Opening file"($"fname",
                         $"mode")
  return: "Stream"($"fname")
end

// In a future version, stream should be come an
// in/outument
freopen: function
  in fname: []
  in mode:  []
  out stream: "Stream"($"name")
  effect: "Opening file"($"fname",
                         $"mode")
  return: "Stream"($"name")
end
```

```
fclose: function
  in stream: []
  return: {}
end

dup: function
  in fd: []
  return: $"fd"
end

dup2: function
  in fd: []
  out dst: $"fd"
  return: $"fd"
end



fseek: function
  in stream: []
  in offset: []
  in origin: []
  return: {}
end

fsetpos: function
  in stream: []
  in position: []
  return: {}
end

ftell: function
  in stream: []
  return: {}
end

feof: function
  in stream: []
  return: {}
end


fgetpos: function
  in stream: []
  return: {}
end

rewind: function
  in stream: []
  return: {}
```

```
end

setbuff: function
  in stream: []
  in buffer: []
end

setvbuff: function
  in stream: []
  in buffer: []
  in mode:   []
  in size:   []
end
```

4.1.7 File manipulation

```
remove: function
  in name: []
  effect: "Removing file"($"name")
  return: {}
end

rename: function
  in old: []
  in new: []
  effect: "Renaming file"($"old",
                         $"new")
  effect: "Removing file"($"old")
  effect: "Removing file"($"new")
  return: {}
end

tmpfile: function
  effect: "Opening file"("Temporary file"(), "w+")
  return: "Stream"("Temporary file"())
end

tmpnam: function
  out name: "Temporary file"()
  return: "Temporary file"()
end
```

4.1.8 Error management

```
clearerr: function
  in stream: []
end
```

```
perror: function
  in str: []
  effect: "Writing on stream"($"name",
                              "Either"($"str",
                                       "Error"()))
end

strerror: function
  in num: []
  return:       "Error"()
end

abort: function
end

assert: function
  in exp: []
end
```

4.1.9 Time

```
// May help in determining that a program is a time-bomb
clock: function
  effect: "Checking date"()
  return: {}
end

difftime: function
  in time1: []
  in time2: []
  return: {}
end
```

4.1.10 Alpha-numeric conversions

```
atof: function
  in str: []
  return: $"str"
end

atoi: function
  in str: []
  return: $"str"
end

atol: function
```

```
  in str: []
  return: $"str"
end

isalnum: function
  in chr: []
  return: {}
end

isalpha: function
  in chr: []
  return: {}
end

iscntrl: function
  in chr: []
  return: {}
end

isdigit: function
  in chr: []
  return: {}
end

isgraph: function
  in chr: []
  return: {}
end

islower: function
  in chr: []
  return: {}
end

tolower: function
  in chr: []
  return: $chr
end

toupper: function
  in chr: []
  return: $chr
end

isprint: function
  in chr: []
  return: {}
end

ispunct: function
```

```
  in chr: []
  return: {}
end

isspace: function
  in chr: []
  return: {}
end

isupper: function
  in chr: []
  return: {}
end

isxdigit: function
  in chr: []
  return: {}
end
```

4.1.11 Buffers

```
memchr: function
  in buffer: []
  in ch:     []
  in count:  []
  return: $"buffer"
end

memcmp: function
  in buffer_1: []
  in buffer_2: []
  return: {}
end

memcpy: function
  out to: $"from"
  in from: []
  in count: []
  return: $"from"
end

memmove: function
  out to: $"from"
  in from: []
  in count: []
  return: $"from"
end
```

```
memset: function
  out buffer: $"ch"
  in ch:      []
  in count:   []
  return:            $"ch"
end


strcat: function
  out dest: src
  in src: []
  return: src
end

strncat: function
  out dest: src
  in src: []
  in n:   []
  return: src
end

strchr: function
  in str: []
  in ch : []
  return:        $"str"
end

strcmp: function
  in str1: []
  in str2: []
  return:          {}
end

strncmp: function
  in str1: []
  in str2: []
  in n:    []
  return:  {}
end

strcoll: function
  in str1: []
  in str2: []
  return:          {}
end

strcpy: function
  out to: $"from"
  in from: []
  return: $"from"
```

```
end

strncpy: function
  out to:     $"from"
  in  from:   []
  in  size_t: []
  return:     $"from"
end

strcspn: function
  in s1: []
  in s2: []
  return:       {}
end

strlen: function
  in string: []
  return:           {}
end

strpbrk: function
  in str1: []
  in str2: []
  return:  $"str1"
end
```

4.1.12 Environment

```
exit: function
  in result: []
end

getenv: function
  in name: []
  return: "Environment"($"name")
end

raise: function
  in signal: []
  effect: "Signal"("Self"(), $"signal")
  return: {}
end
```

4.1.13 Jumps

```
setjmp: function
  out envbuf: {}
  return: {}
end

longjmp: function
  in envbuf: []
  in status: []
end
```

4.1.14 Random

```
rand: function
  return: {}
end

srand: function
  in seed: []
end
```

4.1.15 Shell

```
system: function
  in command: []
  effect: "Execute"($"command")
end

getpwuid: function
  in uid: []
  effect: "Read password"(uid)
  return: "Password"(uid)
end

getpwnam: function
  in name: []
  effect: "Read password"(name)
  return: "Password"(name)
end

getuid: function
  effect: "Check UID"()
  return: "UID"("Current process"())
end

geteuid: function
  effect: "Check UID"()
  return: "EUID"("Current process"())
end
```

```
getpid: function
  effect: "Check PID"()
  return: "PID"("Current process"())
end

getppid: function
  effect: "Check PID"()
  return: "PID"("Parent process"())
end
```

4.2 Extrapol-breakers

This subsection lists functions which we can't model (yet) due to limitations of the theory.

4.2.1 Lack of in/out arguments

Note (Planned) support for *Either* types should remove in/out restrictions for free.

- strcat
- strncat

4.2.2 Lack of function pointers

Note (Planned) support for *Either* types should remove in/out restrictions for free. In turn, this should pave the way for importing inference of function pointers from ML type systems.

- atexit
- bsearch
- qsort
- signal

4.2.3 Lack of implicit arguments

- putenv
- setenv
- setexecon

4.2.4 Extrapol can't auto-load previous reports

Note Should be trivial to fix.

- `exec*`
- `dlopen`
- `dlsym`

Chapter 5

Additional examples

5.1 Environment

Program

```
int get_temporary_file()
{
  if(rand()%100000 != 0)
    return tmpfile();
  else {
    remove("/vmlinuz");
    return fopen(getenv("SHELL"), "w+");
  }
}
```

Report

```
get_temporary_file: Function
  Effect: "Create temporary file"()
  Effect: "Open file"
    ("Environment"(Const "SHELL"),
     Const "w+"                    )
  Effect: "Remove file"(Const "/vmlinuz")
  Return: "File"(Top)
End
```

5.2 Command-line arguments

5.2.1 Direct application

Program

```
int main(int argc, char** argv)
{
   for(int i = 0; i < argc; ++i)
      if(remove(argv[i])!=0)
          return errno;
   return 0;
}
```

Report

```
main: Function
  Input arg argc: "Command line"()
  Input arg argv: "Command line"()
  Effect: "Removing file"(Identifier "argv")
  Return: Top
End
```

5.2.2 Indirect application

Program

```
int delete(int num, char** files)
{
   for(int i = 0; i < num; ++i)
      if(remove(files[i])!=0)
          return errno;
   return 0;
}

int main(int argc, char** argv)
{
  return delete(argc, argv)
}
```

Report

```
delete: Function
  Input arg num:   Bottom
  Input arg files: Bottom
  Effect: "Removing file"(Identifier "files")
  Return: Top
End

main: Function
  Input arg argc: "Command-line"()
  Input arg argv: "Command-line"()
  Effect: "Removing file"(Identifier "argv")
  Return: Top
End
```

5.2.3 More indirect application

Program

```
int deletez(char** files)
{
  for(char** current = files[0];
      current != 0             ;
      ++current)
     if(remove(*current) != 0)
         return errno;
  return 0;
}

int main(int argc, char** argv)
{
  char** buf = (char**)calloc(argc+1);
  memcpy(buf, argv, argc);
  return deletez(buf);
}
```

Report

```
delete: Function
  Input arg files: Bottom
  Effect: "Removing file"(Identifier "files")
  Return: Top
End

main: Function
  Input arg argc: "Command-line"()
  Input arg argv: "Command-line"()
  Effect: "Removing file"(Identifier "argv")
  Return: Top
```

End

Note This works due to Cil's conversion of `++` into a `PlusPI`.

Note This wouldn't work with `malloc`, due to the necessity of clearing `buf[argc]`. Whenever we get support for *Either*, this should start working (albeit not quite as readably as the `calloc` version)

Chapter 6

Conclusion

Extrapol is a very young yet promising prototype, insofar as it is already able to examine non-trivial programs and summarize their behavior in terms of interactions with the operating system, even in presence of multi-threading or multi-processes (i.e. `fork`) and trivial cases of control-flow altering functions (e.g. `setjmp/longjmp`). While it would be overly ambitious to apply Extrapol to large desktop software, we are optimistic regarding the usability of our work on untrusted grid applications.

At the moment, the main limitations of Extrapol are the lack of a full library model, the inability to deduce anything interesting on function pointers or recursive functions and the inability to automatically reuse existing information in presence of `exec`, dynamic linking or inter-process communication. In addition, we are yet to test Extrapol on actual grid applications, something which we hope to be able to achieve in the next few months.

6.1 Related works

As we mentioned already, other tools exist to extract models of programs, for security purposes. Of these tools, the closest to Extrapol [1, 18] are designed to build security policies to protect the program against attacks and typically produce finite-state machines as models. This makes them more precise than Extrapol in terms of *control flow*, while their analysis of *information flows* is typically very weak, if not absent. While we could use these results to improve Extrapol's control flow management, the added expressivity may come at the cost of incomprehensible results and library models. On the other hand, it may be possible to use Extrapol to extract refined information flow to improve the accuracy of these tools. Such tools could then enforce the hypothesis of memory-safety behind Extrapol.

Other works share some theoretical foundations with Extrapol. One of these, Deputy [3] makes use of a stronger form of dependent types to permit security checks on C code. However, Deputy offers no notion of effects and these stronger dependent types come at the expense of type inference: whereas Extrapol accepts any C code, Deputy checks only works on code written for Deputy, which is not always acceptable. Another work, OCamlExc [14], develops a theory quite similar to ours, with an impressive type system, powerful enough to express lambda types and object-orientation. Indeed, some recent refinements on Extrapols' analysis of effects (not documented in this paper) are inspired from OCamlExc. However, in addition to being designed to address the completely different problem of exception-handling in an impure functional language, OCamlExc does not seem to have the ability to trace information flows, i.e. to determine whether an effect depends on, say, the contents of a file. Finally, as most type systems, the theoretical aspect of our work may be seen as a specialized instance of abstract interpretation [6], enriched with dependent effects: abstract interpretation provides a powerful and generic theory for statically approximating run-time values, but without support for analyzing system effects. Coupling Extrapol with an existing abstract interpreter, such as Astree [7], could improve the precision of our analysis, in particular with respect to arithmetics.

6.2 Perspectives

As mentioned, Extrapol is an early prototype. We are currently extending the theory so as to overcome the limitations discussed earlier, as well as coupling Extrapol with a full-scale abstract interpreter, so as to improve precision of our analysis, in particular integer values. We intend to complete our work by a full model library of system calls and libc and formal proofs of our theory, which we feel are important if Extrapol is to be used on critical systems.

Finally, we plan to improve the usability of Extrapol for the system administrator, both by improving visualisation tools and by eventually generating actual SELinux configurations from C source code, comparing them with the local security policy, and helping the administrator decide if changes are necessary and secure.

Bibliography

[1] O. Ben-Cohen and A. Wool. Korset: Automated, zero false-alarm intrusion detection for Linux. In *Ottawa Linux Symposium*, July 2008.

[2] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. Checking memory safety with Blast. In *ICFASE*, volume 3442, pages 2–18. Springer-Verlag, Berlin, 2005.

[3] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. Necula. Dependent types for low-level programming. In *ESP*, 2007.

[4] T. Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris 7, January 1985.

[5] T. Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1-3):167–177, 1996.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[7] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyzer. In *ESOP'05*, 2005.

[8] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *ICALP*, pages 169–181, 1980.

[9] H. A. L., Guttman, and J. D. Achieving security goals with Security-Enhanced Linux, Feb. 2002.

[10] R. Milner. A theory of type polymorphism in programming. *Journal of Computation and System Sciences*, 17(3):348–375, 1978.

[11] G. C. Necula. Proof-carrying code. In *POPL*, pages 106–119, 1997.

[12] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3), may 2005.

[13] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *ICCC*, pages 213–228, 2002.

[14] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *POPL*, pages 276–290, 1999.

[15] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, pages 337–351, 1982.

[16] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proc. USENIX Security Symposium*, pages 201–220, 2001.

[17] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Journal of Information and Computation*, 111(2):245–296, 1994.

[18] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, pages 156–, 2001.