Verifying and Synthesizing Constant-Resource Implementations with Types

Van Chan Ngo Mario Dehesa-Azuara Matthew Fredrikson Jan Hoffmann
Carnegie Mellon University, Pittsburgh, Pennsylvania 15213
Email: channgo@cmu.edu, mdehazu@gmail.com, mfredrik@cs.cmu.edu, jhoffmann@cmu.edu

***Abstract*—Side channel attacks have been used to extract critical data such as encryption keys and confidential user data in a variety of adversarial settings. In practice, this threat is addressed by adhering to a constant-time programming discipline, which imposes strict constraints on the way in which programs are written. This introduces an additional hurdle for programmers faced with the already difficult task of writing secure code, highlighting the need for solutions that give the same source-level guarantees while supporting more natural programming models.**

We propose a novel type system for verifying that programs correctly implement constant-resource behavior. Our type system extends recent work on *automatic amortized resource analysis* (AARA), a set of techniques that automatically derive provable upper bounds on the resource consumption of programs. We devise new techniques that build on the potential method to achieve compositionality, precision, and automation.

A strict global requirement that a program always maintains constant resource usage is too restrictive for most practical applications. It is sufficient to require that the program's resource behavior remain constant with respect to an attacker who is only allowed to observe part of the program's state and behavior. To account for this, our type system incorporates information flow tracking into its resource analysis. This allows our system to certify programs that need to violate the constant-time requirement in certain cases, as long as doing so does not leak confidential information to attackers. We formalize this guarantee by defining a new notion of *resource-aware noninterference*, and prove that our system enforces it.

Finally, we show how our type inference algorithm can be used to synthesize a constant-time implementation from one that cannot be verified as secure, effectively repairing insecure programs automatically. We also show how a second novel AARA system that computes *lower bounds* on resource usage can be used to derive quantitative bounds on the amount of information that a program leaks through its resource use. We implemented each of these systems in Resource Aware ML, and show that it can be applied to verify constant-time behavior in a number of applications including encryption and decryption routines, database queries, and other resource-aware functionality.

***Keywords*-Language-based security; timing channels; information flow; resource analysis; static analysis**

I. Introduction

Side-channel attacks extract sensitive information about a program's state through its observable use of resources such as time, network, and memory. These attacks pose a realistic threat to the security of systems in a range of settings, in which the attacker has local access to the native host [1], through multi-tenant virtualized environments [2], [3], or remotely over the network [4]. Side channels have revealed highly-sensitive data such as cryptographic keys [1], [4], [5], [6], [7] and private user data [8], [9], [10], [11], [12].

These attacks are mounted by taking repeated measurements of a program's resource behavior, and comparing the resulting observations against a model that relates the program's secret state to its resource usage. Unlike direct information flow channels that operate over the input/output semantics of a program, the conditions that give rise to side channels are oftentimes subtle and therefore difficult for programmers to identify and mitigate. This also poses a challenge for automated tool support aimed at addressing such problems—whereas direct information flow can be described in terms of standard program semantics, a similar precise treatment of side channels requires incorporating the corresponding resource into the semantics and applying quantitative reasoning.

This difficulty has led previous work in the area to treat resource use indirectly, by reasoning about the flow of secret information into branching control flow or other operations that might affect resource use [13], [14], [15], [16]. These approaches can limit the expressiveness of secure programs and further complicate the development. For example, by requiring programmers to write code using a "constant-time discipline" that forbids the use of variables influenced by secret state in statements that could affect the program's control path [13].

Verifiable constant-resource language: In this paper, we present a novel type system that gives developers the ability to certify that their code is secure against resource side-channel attacks w.r.t. a high-level attack model, in which the resource consumption of each language construct is modeled by a constant. Our approach reduces constraints on the expressiveness of programs that can be verified, and does not introduce general stylistic guidelines that must be followed in order to ensure constant-resource behavior. Programmers write code in typical functional style and annotate variables with standard types. Thus, it does not degrade the readability of the code. At compile time, our verifier performs a quantitative analysis to infer additional type information that characterizes the resource usage. From this, constant-resource behavior w.r.t. the high-level model on all executions of the program is determined automatically.

The granularity with which our resource guarantees hold against an attacker who can measure the total quantity of consumed resources is roughly equivalent to what can be obtained by adhering to a strict constant-time programming discipline. The certified constant-resource programs prevent side-channels that are inherent in implementing algorithms w.r.t. the provided high-level attack model. For example, if the resource under consideration is execution time, measured by the number of language constructs executed by the program (e.g., the total number of arithmetic operations, function calls, etc.), then our system provides a defense against attackers that can observe the same resource measure. To have a stronger guarantee, e.g., against cache side channels, our resource model could in principle incorporate memory-access patterns and instruction caches. Other types of side channels arising from low-level behaviors, such as branch prediction or instructions whose resource usage is influenced by argument values, require corresponding changes to the resource model. Our technique does not currently model such timing differences, so is not a defense against such attacks.

In general, requiring that a program always consumes constant resources is too restrictive. In most settings, it is sufficient to ensure that the resource behavior of a program does not depend on selected confidential parts of the program's state. To account for this, our type system tracks information flow using standard techniques, and uses this information to reason about an adversary who can observe and manipulate public state as well as resource usage through public outputs. Intuitively, *resource-aware noninterference*—the guarantee enforced by this type system—requires that the parts of the program that are both affected by secret data and can influence public outputs, can only make constant use of resources.

To accomplish this without limiting expressiveness or imposing stylistic requirements, the type system must be allowed to freely switch between local and global reasoning. One extreme would be to ignore the information flow of the secret values and prove that the whole program has global constant resource consumption. The other extreme would be to ensure that every conditional that branches on a secret value (*critical* conditionals) uses a constant amount of resources. However, there are constant-resource programs in which individual conditionals are not locally constant-resource (see Section III). As a result, we allow different levels of global and local reasoning in the type system to ensure that every critical conditional occurs in a constant-resource block.

Finally, we show that our type-inference algorithm can be used to automatically repair programs that make inappropriate non-constant use of resources, by *synthesizing* constant-resource ones whose input/output behavior is equivalent. To this end, we introduce a *consume* expression that performs resource padding. The amount of resource padding that is needed is automatically determined by the type system and is parametric in the values held by program variables. An advantage of this technique over prior approaches [17], [18] is that it does not change the worst-case resource behavior of many programs. Of course, it would be possible to perform this transformation by padding resource usage dynamically at the end of the program execution, but this would require instrumenting the program to track at runtime the actual resource usage of the program.

Novel resource type systems: In order to verify constant resource usage, as well as to produce quantitative upper and lower-bounds on information leakage via resource behavior, this work extends the theory behind automatic amortized resource analysis (AARA) [19], [20], [21] to automatically derive lower-bound and constant-resource proofs.

Previous AARA techniques are limited to deriving upper bounds. To this end, the resource potential is used as an *affine* quantity: it must be available to cover the cost of the execution, but excess potential is simply discarded. We show that if potential is treated as a *linear* resource, then corresponding type derivations prove that programs have constant resource consumption, i.e., the resource consumption is independent of the execution path. Intuitively, this amounts to requiring that *all* potential must be used to cover the cost and that excess potential is not wasted. Furthermore, we show that if potential is treated as a *relevant* resource then we can derive lower bounds on the resource usage. Following a similar intuition, this requires that all potential is used, but the available potential does not need to be sufficient to cover the remaining cost of the execution.

We implemented these type systems in Resource Aware ML (RAML) [21], a language that supports user-defined data types, higher-order functions, and other features common to functional languages. Our type inference uses efficient LP solving to characterize resource usage for general-purpose programs in this language. We formalized soundness proofs for these type systems, as well as the one of classic linear AARA [19], in the proof assistant Agda. The soundness is proved w.r.t. an operational cost semantics and, like the type systems themselves, is parametric in the resource of interest.

Contributions: We make the following contributions:

- A security type system that incorporates our novel lower-bound and constant-time type systems to prevent and quantify leakage of secrets through resource side channels, as well as an LP-based method that automatically transforms programs into constant-resource versions.
- An implementation of these systems that extends RAML. We evaluate the implementation on several examples, including encryption routines and data processing programs that were previously studied in the context of timing leaks in differentially-private systems [8].
- A mechanization of the soundness proofs the two new type systems and classic AARA for upper bounds in Agda. To the best our knowledge, this is also the first formalization of the soundness of linear AARA for worst-case bounds.

Technical details including the complete proofs and inference rules can be found on the RAML website [21].

II. LANGUAGE-LEVEL CONSTANT-RESOURCE PROGRAMS

In this section, we define our notion of a constant-resource program. We start with an illustrative example: a login with a username and password. During the login process, the secret password with a high security level is compared with the low-security user input, and the result is sent back to the user. As a result, the pure noninterference property [22], [23] is violated because data flows from high to low. Nevertheless, such a program is often considered secure because it satisfies the *relaxed noninterference* property [24], [25], [26].

Fig. 1 shows an implementation of the login process in a monomorphically-typed purely-functional language. The arguments h and l are lists of integers that are the bytes of the password and the user input (characters of the hashes). The function returns *true* if the input is valid and *false* otherwise.

This implementation is vulnerable against an attacker who measures the execution time of the login function. Because the function returns *false* immediately on finding a mismatched pair of bytes, the resource usage depends on the size of the longest matching prefix. Based on this observation, the attacker can mount an efficient attack to recover the correct password byte-by-byte. For example, if we assume that there is no noise in the measurements, it requires at most $256 = 2^8$ calls to the function to reveal one byte of the secret password. Thus, at most $256 * N$ runs are needed to recover a secret password of N bytes. If noise is added to the measurements then the number of necessary guesses is increased but the attack remains feasible [1], [4].

One method to prevent this sort of attack is to develop a *constant-resource* implementation of the *compare* function that minimizes the information that an attacker can learn from the resource-usage information. Ideally, the resource usage should not be dependent on the content of the secret password, which means it is constant for fixed sizes of all public parameters.

Syntax and semantics: We use the purely-functional first-order language defined in Fig. 2 to formally define the notion of a language-level constant-time implementation. The grammar is written using abstract binding trees [27]. However, equivalent expressions in OCaml syntax are used for examples. The expressions are in *let normal form*, meaning that they are formed from variables whenever it is possible. It makes the typing rules and semantics simpler without losing expressivity. The syntactic form *share* has to be use to introduce multiple occurrences of a variable in an expression. A value is a boolean constant, an integer value n, the empty list nil, a list of values $[v_1, ..., v_n]$, or a pair of values (v_1, v_2). To reason about the resource consumption of programs, we first define the operational cost semantics of the language. It is standard big-step semantics instrumented with a non-negative resource counter that is incremented or decremented by a constant at every step. The semantics is parametric in the cost that is used at each step and we call a particular set of such cost parameters a *cost model*. The constants can be used to indicate the costs of storing or loading a value in the memory, evaluating a primitive operation, binding of a value in the environment, or branching on a Boolean value. It is possible to further parameterize some constants to obtain a more precise cost model. For example, the cost of calling a function may vary according to the number of the arguments. In the following, we will show that the soundness of type systems does not rely on any specific values for these constants. In the examples, we use a cost model in which the constants are 0 for all steps except for calls to the *tick* function where *tick(q)* means that we have resource usage $q \in \mathbb{Q}$. A negative number specifies that resources (such as stack space) become available.

```
let rec compare(h,l) = match h with
| [] →match l with | [] →true
                   | y::ys →false
| x::xs →match l with
  | [] →false
  | y::ys →if (x = y) then compare(xs,ys)
         else false
```

Figure 1. The list comparison function *compare* is not constant resource w.r.t. h and l. This implementation is insecure against an attacker who measures its resource usage.

$$
\begin{array}{lll}
T & ::= & \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid L(T) \mid T * T \\
G & ::= & T \rightarrow T \\
e & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid x \mid \mathrm{op}_\diamond(x_1, x_2) \mid \mathrm{app}(f, x) \\
 & & \mid \mathrm{if}(x, e_t, e_f) \mid \mathrm{let}(x, e_1, x.e_2) \mid \mathrm{pair}(x_1, x_2) \mid \mathrm{nil} \\
 & & \mid \mathrm{match}(x, (x_1, x_2).e) \mid \mathrm{cons}(x_1, x_2) \\
 & & \mid \mathrm{match}(x, e_1, (x_1, x_2).e_2) \mid \mathrm{share}(x, (x_1, x_2).e) \\
v & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid \mathsf{nil} \mid [v_1, ..., v_n] \mid (v_1, v_2) \\
\diamond & \in & \{+, -, *, \mathsf{div}, \mathsf{mod}, =, <>, >, <, \mathsf{and}, \mathsf{or}\}
\end{array}
$$

Figure 2. Syntax of the language

The cost semantics is formulated using an *environment* $E : \mathrm{VID} \rightarrow Val$ that is a finite mapping from a set of variable identifiers to values. Evaluation judgements are of the form $E \vdash^{q}_{q'} e \Downarrow v$ where $q, q' \in \mathbb{Q}_0^+$. The intuitive meaning is that under the environment E and q available resources, e evaluates to the value v without running out of resources and q' resources are available after the evaluation. The evaluation consumes $\delta = q - q'$ resource units. Fig. 12 presents some selected evaluation rules. In the rule E:FUN for function applications, e_g is an expression defining the function's body and x^g is the argument.

Constant-resource programs: Let $\Gamma : \mathrm{VID} \rightarrow \mathscr{T}$ be a context that maps variable identifiers to base types T. We write $\models v : T$ to denote that v is a well-formed value of type T. The typing rules for values are standard [19], [20], [28] and we omit them here. An environment E is *well-formed* w.r.t. Γ, denoted $\models E : \Gamma$, if $\forall x \in \mathrm{dom}(\Gamma).\ \models E(x) : \Gamma(x)$. Below we define the notation of *size equivalence*, written

$$\frac{v = E(x_1) \diamond E(x_2)}{E \vdash^{q+K^{\mathsf{op}}}_{q} \mathrm{op}_\diamond(x_1,x_2) \Downarrow v} \text{(E:BIN)}$$

$$\frac{E[x^g \mapsto E(x)] \vdash^{q}_{q'} e_g \Downarrow v}{E \vdash^{q+K^{\mathsf{app}}}_{q'} \mathrm{app}(g,x) \Downarrow v} \text{(E:FUN)}$$

$$\frac{E \vdash^{q-K^{\mathsf{let}}}_{q'_1} e_1 \Downarrow v_1 \qquad E[x \mapsto v_1] \vdash^{q'_1}_{q'} e_2 \Downarrow v}{E \vdash^{q}_{q'} \mathrm{let}(x, e_1, x.e_2) \Downarrow v} \text{(E:LET)}$$

$$\frac{x \in \mathrm{dom}(E)}{E \vdash^{q+K^{\mathsf{var}}}_{q} x \Downarrow E(x)} \text{(E:VAR)}$$

$$\frac{E(x) = \mathsf{true} \qquad E \vdash^{q-K^{\mathsf{cond}}}_{q'} e_t \Downarrow v}{E \vdash^{q}_{q'} \mathrm{if}(x, e_t, e_f) \Downarrow v} \text{(E:IF-TRUE)}$$

$$\frac{E(x) = [v_1, ..., v_n] \qquad E[x_h \mapsto v_1, x_t \mapsto [v_2, ..., v_n]] \vdash^{q-K^{\mathsf{matchL}}}_{q'} e_2 \Downarrow v}{E \vdash^{q}_{q'} \mathrm{match}(x, e_1, (x_h, x_t).e_2) \Downarrow v} \text{(E:MATCH-L)}$$

Figure 3. Selected evaluation rules of the operational cost semantics

```
let rec p_compare(h,l) =
let rec aux(r,h,l) = match h with
| [] →match l with | [] →tick(1.0); r
      | y::ys →tick(1.0); false
| x::xs →match l with
  | [] →tick(1.0); false
  | y::ys →if (x = y) then
     tick(5.0); aux(r,xs,ys)
    else tick(5.0); aux(false,xs,ys)
in aux(true,h,l)
```

Figure 4. The manually padded function *p_compare* is constant resource w.r.t. h and l. However, it is not constant resource w.r.t. only h.

$|v| \approx |u|$, which is a binary relation relating two values of the same type.

$$\frac{\models v : T \qquad \models u : T \qquad T \in \{\mathrm{unit}, \mathrm{bool}, \mathrm{int}\}}{|v| \approx |u|}$$

$$\frac{|v_1| \approx |u_1| \qquad |v_2| \approx |u_2|}{|(v_1, v_2)| \approx |(u_1, u_2)|} \qquad \frac{m = n \qquad |v_i| \approx |u_i|}{|[v_1, ...v_n]| \approx |[u_1, ...u_m]|}$$

Informally, a program is *constant resource* if it has the same quantitative resource consumption under all environments in which values have the same sizes. Let $X \subseteq \mathrm{dom}(\Gamma)$ be a set of variables and E_1, E_2 be two well-formed environments. Then E_1 and E_2 are *size-equivalent* w.r.t. X, denoted $E_1 \approx_X E_2$, when they agree on the sizes of the variables in X, that is, $\forall x \in X.|E_1(x)| \approx |E_2(x)|$.

Definition 1. *An expression e is constant resource w.r.t. $X \subseteq dom(\Gamma)$, written $const_X(e)$, if for all well-formed environments E_1 and E_2 such that $E_1 \approx_X E_2$, the following statement holds.*

If $E_1 \vdash^{p_1}_{p'_1} e \Downarrow v_1$ and $E_2 \vdash^{p_2}_{p'_2} e \Downarrow v_2$ then $p_1 - p'_1 = p_2 - p'_2$

We say that a function $g(x_1, \ldots, x_n)$ is constant resource w.r.t. X if $\mathrm{const}_X(e_g)$ where e_g is the expression defining the function body. We have the following lemma.

Lemma 1. *For all e, X, and $Y \subseteq X$, if $const_Y(e)$ then $const_X(e)$.*

Example. *The function* p_compare *in Fig. 4 is a manually padded version of* compare*, in which the cost model is defined using* tick *annotations. It is constant resource w.r.t. h and l. However, it is not constant resource w.r.t. h. For instance,* p_compare([1;2;3],[0;1;2]) *has cost* 16 *but* p_compare([1;2;1],[0;1]) *has cost* 12 $\neq$ 16. *If the nil case of the second match on l is padded with* tick(5.0); aux(false,xs,[]) *then the function is constant resource w.r.t. h.*

Intuitively, this implementation is constant w.r.t. the given cost model for fixed sizes of all public parameters, e.g., the lengths of argument lists. However, it might be not constant resource at a lower level, e.g., machine code on modern hardware, because the cost model does not precisely capture the resource consumption of the instructions executed on the hardware. Moreover, the compilation process can interfere with the resource behavior. It may introduce a different type of leakage that could reveal the secret data on the lower level. For instance, memory accesses would allow an attacker with access to the full trace of memory addresses accessed to infer the content of the password. This leakage can be exploited via cache-timing attacks [29], [30]. In addition, in some modern processors, execution time of arithmetic operations may vary depending on the values of their operands and the execution time of conditionals is affected by branch prediction.

III. A RESOURCE-AWARE SECURITY TYPE SYSTEM

In this section we introduce a new type system that enforces *resource-aware noninterference* to prevent the leakage of information in *high-security* variables through *low-security* channels. In addition to preventing leakage over the usual input/output information flow channels, our system incorporates the constant-resource type system discussed in Section IV to ensure that leakage does not occur over resource side channels.

The notion of security addressed by our type system considers an attacker who wishes to learn information about secret data by making observations of the program's public outputs and resource usage. We assume an attacker who is able to control the value of any variable she is capable of observing, and thus to influence the program's behavior and resource consumption. However, in our model the attacker can only observe the program's total resource usage upon termination, and cannot distinguish between intermediate states or between terminating and non-terminating executions.

A. *Security types*

To distinguish parts of the program under the attacker's control from those that remain secret, we annotate types with labels ranging over a lattice $(\mathscr{L}, \sqsubseteq, \sqcup, \bot)$. The elements of $\mathscr{L}$ correspond to security levels partially-ordered by $\sqsubseteq$ with a unique bottom element $\bot$. The corresponding basic security types take the form:

$$\begin{array}{lll} k & \in & \mathscr{L} \\ S & ::= & (\mathrm{unit}, k) \mid (\mathrm{bool}, k) \mid (\mathrm{int}, k) \mid (L(S), k) \mid S * S \end{array}$$

A security context Γ^s is a partial mapping from variable identifiers and the program counter pc to security types. The context assigns a type (unit, k) to pc to track information that may propagate through control flow as a result of branching statements. The security type for lists contains a label $L(S)$ for the elements, as well as a label k for the list's length.

As in other information flow type systems, the partial order $k \sqsubseteq k'$ indicates that the class k' is at least as restrictive as k, i.e., k is allowed to flow to k'. We assume a non-trivial security lattice that contains at least two labels: ℓ (low security) and h (high security), with $\ell \sqsubseteq h$. Following the convention defined in FlowCaml [31], we also make use of a *guard relation* $k \lhd S$, which denotes that all of the labels appearing in S are at least as restrictive as k. The definition is given in Figure 13 along with its dual notion $S \blacktriangleleft k$, called the *collecting relation*, and the standard subtyping relation $S_1 \leq S_2$.

To refer to sets of variables by security class, we write $[\Gamma^s]_{\blacktriangleleft k}$ to denote the set of variable identifiers x in the domain of Γ^s such that $\Gamma^s(x) \blacktriangleleft k$, and define ${}_{k\lhd}[\Gamma^s]$ similarly. This gives us the set of variables upper- and lower-bounded by k, respectively. Conversely, we define $[\Gamma^s]_{\not\blacktriangleleft k} = \{x \in \mathrm{dom}(\Gamma^s) : \Gamma^s(x) \not\blacktriangleleft k\}$, the set of variables more restrictive than k. To refer to the set of variables strictly bounded below by k_1 and above by k_2, we write ${}_{k_1\lhd}[\Gamma^s]_{\blacktriangleleft k_2}$. Given two well-formed environments E_1 and E_2, we say that they are *k-equivalent* w.r.t Γ^s if they agree on all variables with label at most k:

$$E_1 \equiv_k E_2 \Leftrightarrow \forall x \in [\Gamma^s]_{\blacktriangleleft k}.E_1(x) = E_2(x)$$

This relation captures the attacker's *observational equivalence* between the two environments. The first-order security types take the following form. The annotation pc indicates the security level of the program counter, i.e., a lower-bound on the label of any observer who is allowed to learn that a given function has been invoked. The const annotation denotes that the function body respects *resource-aware noninterference*.

$$\mathrm{pc} \in \mathscr{L} \qquad F^s ::= S_1 \xrightarrow{\mathrm{pc/const}} S_2 \mid S_1 \xrightarrow{\mathrm{pc}} S_2$$

$$\frac{k \sqsubseteq k' \quad T \in \text{Atoms}}{k \lhd (T, k')} \qquad \frac{k \sqsubseteq k' \quad k \lhd S}{k \lhd (L(S), k')} \qquad \frac{k \lhd S_1 \quad k \lhd S_2}{k \lhd S_1 * S_2}$$

$$\frac{k' \sqsubseteq k \quad T \in \text{Atoms}}{(T, k') \blacktriangleleft k} \qquad \frac{k' \sqsubseteq k \quad S \blacktriangleleft k}{(L(S), k') \blacktriangleleft k} \qquad \frac{S_1 \blacktriangleleft k \quad S_2 \blacktriangleleft k}{S_1 * S_2 \blacktriangleleft k}$$

$$\frac{k \sqsubseteq k' \quad T \in \text{Atoms}}{(\mathrm{T}, k) \leq (\mathrm{T}, k')} \qquad \frac{k \sqsubseteq k' \quad S \leq S'}{(L(S), k) \leq (L(S'), k')} \qquad \frac{S_1 \leq S_1' \quad S_2 \leq S_2'}{S_1 * S_2 \leq S_1' * S_2'}$$

Figure 5. Guards, collecting security labels, and subtyping (Atoms = $\{\mathrm{unit}, \mathrm{int}, \mathrm{bool}\}$)

A *security signature* $\Sigma^s : \mathrm{FID} \to \wp(\mathscr{F}^s) \setminus \{\emptyset\}$ is a finite partial mapping from a set of function identifiers to a *non-empty sets* of first-order security types.

B. *Resource-aware noninterference*

We consider an adversary associated with label $k_1 \in \mathscr{L}$, who can observe and control variables in $[\Gamma^s]_{\blacktriangleleft k_1}$. Intuitively, we say that a program P satisfies resource-aware noninterference at level (k_1, k_2) w.r.t Γ^s, where $k_1 \sqsubseteq k_2$, if *1)* the behavior of P does not leak any information about the contents of variables more sensitive than k_1, and *2)* does not leak any information about the contents *or sizes* of variables more sensitive than k_2. The definition follows.

Definition 2. *Let E_1 and E_2 be two well-formed environments and Γ^s be a security context sharing their domain. An expression e satisfies resource-aware noninterference at level (k_1, k_2) for $k_1 \sqsubseteq k_2$, if whenever E_1 and E_2 are:*

1) *observationally equivalent at k_1: $E_1 \equiv_{k_1} E_2$,*
2) *size equivalent w.r.t. ${}_{k_1\lhd}[\Gamma^s]_{\blacktriangleleft k_2}$: $E_1 \approx_{{}_{k_1\lhd}[\Gamma^s]_{\blacktriangleleft k_2}} E_2$*

then it follows from $E_1 \mid\frac{p_1}{p_1'}\; e \Downarrow v_1$ and $E_2 \mid\frac{p_2}{p_2'}\; e \Downarrow v_2$ that $v_1 = v_2$ and $p_1 - p_1' = p_2 - p_2'$.

The final condition in Defintion 2 ensures two properties. First, requiring that $v_1 = v_2$ provides noninterference [22], given that E_1 and E_2 are observationally equivalent. Second, the requirement $p_1 - p_1' = p_2 - p_2'$ ensures that the program's resource consumption will remain constant w.r.t changes in variables from the set $[\Gamma^s]_{\not\blacktriangleleft k_1}$. This establishes noninterference w.r.t the program's final resource consumption, and thus prevents the leakage of secret information.

Before moving on, we point out an important subtlety in this definition. We require that all variables in ${}_{k_1\lhd}[\Gamma^s]_{\blacktriangleleft k_2}$ begin with equivalent sizes, but not those in ${}_{k_2\lhd}[\Gamma^s]$. By fixing this quantity in the initial environments, we assume that an attacker is able to control and observe it, so it is not protected by the definition. This effectively establishes three classes of variables, i.e., those whose size and content are observable to the k_1-adversary, those whose size (but *not* content) is observable, and those whose size and content remain secret. In the remainder of the text, we will simplify the technical development by assuming that the third and most-restrictive class is empty, and that all of the secret variables reside in ${}_{k_1 \triangleleft}[\Gamma^s]_{\blacktriangleleft k_2}$.

C. Proving resource-aware noninterference

There are two extreme ways of proving resource-aware noninterference. Assume we already have established classic noninterference, the first way is to additionally prove constant-resource usage *globally* by forgetting the security labels and showing that the program has constant-resource usage. This is a sound approach but it requires us to reason about parts of the programs that are not affected by secret data. It would therefore result in the rejection of programs that have the resource-aware noninterference property but are not constant resource. The second way is to prove constant resource usage *locally* by ensuring that every conditional that branches on secret values is constant time. However, this local approach is problematic because it is not compositional. Consider the following examples where *rev* is the reverse function.

```
let f1(b,x) =
  let z = if b then x else [] in rev z

let f2(b,x,y) =
  let z = if b then let _ = rev y in x
          else let _ = rev x in y in rev z
```

If we assume a cost model in which we count the number of function calls then the cost of *rev(x)* is $|x|$. So *rev* is constant resource w.r.t. its argument. Moreover, the expression *if b then x else []* is constant resource. However, *f1* is not constant resource. In contrast, the conditional in *f2* is not constant resource. But *f2* is a constant-resource function. The function *f2* can be automatically analyzed with the constant-resource type system from Section IV while *f1* is correctly rejected.

The idea of our type system for resource-aware noninterference is to allow both global and local reasoning about resource consumption as well as arbitrary intermediate levels. *We ensure that every expression that is typed in a high security context is part of a constant-resource expression.* In this way, we get the benefits of local reasoning without losing compositionality.

D. Typing rules and soundness

We combine our type system for constant resource usage with a standard information flow type system which based on FlowCaml [32]. The interface between the two type systems is relatively light and the idea is applicable to other cost-analysis methods as well as other security type systems.

In the type judgement, an expression is typed under a type context Γ^s and a label pc. The pc label can be considered an upper bound on the security labels of all values that affect the control flow of the expression and a lower bound on the labels of the function's effects [32]. As mentioned earlier, we will simplify the technical development by assuming that the third and most-restrictive class is empty, that is, the typing rules here guarantee that well-typed expressions provably satisfy the resource-aware noninterference property w.r.t. changes in variables from the set $[\Gamma^s]_{\blacktriangleleft k_1}$, say X. We define two type judgements of the form

$$\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\longmapsto} e : S \quad \text{and} \quad \mathrm{pc}; \Sigma^s; \Gamma^s \longmapsto e : S \, .$$

The judgement with the const annotation states that under a security configuration given by Γ^s and the label pc, e has type S and it satisfies resource-ware noninterference w.r.t. changes in variables from X. The second judgement indicates that e satisfies the noninterference property but does not make any guarantees about resource-based side channels. Selected typing rules are given in Fig. 14. We implicitly assume that the security types and the resource-annotated counterparts have the same base types. We write $[\mathrm{const}]$ to denote it is optional, and $\mathrm{const}_X(e)$ if e is well-typed in the constant-resource type system w.r.t. X (i.e., $\Sigma^r; \Gamma^r \vdash^{q}_{q'} e : A$, $\curlyvee(A \mid A, A)$, and $\forall x \in \mathrm{dom}(\Gamma^r) \setminus X. \curlyvee(\Gamma^r(x) \mid \Gamma^r(x), \Gamma^r(x))$). We will discuss the constant-resource type system in Section IV.

Note that the standard information flow typing rules [33], [32] can be obtained by removing the const annotations from all judgements. Consider for instance the rule SR:IF for conditional expressions. By executing the true or false branches, an adversary could gain information about the conditional value whose security label is k_x. Therefore, the conditional expression must be type-checked under a security assumption at least as restrictive as pc and k_x. This is a standard requirement in any information flow type system. In the following, we will focus on explaining how the rules restrict the observable resource usage instead of these classic noninterference aspects.

The most interesting rules are SR:C-GEN and the rules for *let* and *if* expressions, which block leakage over resource usage when branching on high security data. SR:C-GEN allows us to *globally* reason about constant resource usage for an arbitrary subexpression that has the noninterference property. For example, we can apply SR:IF, the standard rule for conditionals, first and then SR:C-GEN to prove that its super-expression is constant resource. Alternatively, we can use rules such as SR:L-IF and SR:L-LET to *locally* reason about resource use. The rule SR:L-LET reflects the fact that if both e_1 and e_2 have the resource-aware noninterference property and the size of x only depends on low security data then $\mathrm{let}(x, e_1, x.e_2)$ respects resource-aware noninterference. The reasoning is similar for SR:L-IF where we require that the variable x does not depend on high security data.

Leaf expressions such as $\mathrm{op}_\diamond(x_1, x_2)$ and $\mathrm{cons}(x_h, x_t)$ have constant resource usage. Thus their judgements are always associated with the qualifier const as shown in the rule SR:B-OP. The rule SR:C-FUN states that if a function's body has the resource-aware noninterference property then the function application has the resource-aware noninterference property too. If the argument's label is low security data, bounded below by k_1, then the function application has the

$$\text{(SR:GEN)}\quad \frac{\text{pc}; \Sigma^s; \Gamma^s \stackrel{\text{const}}{\vdash} e : S}{\text{pc}; \Sigma^s; \Gamma^s \vdash e : S}$$

$$\text{(SR:C-GEN)}\quad \frac{\text{pc}; \Sigma^s; \Gamma^s \vdash e : S \qquad \text{const}_X(e)}{\text{pc}; \Sigma^s; \Gamma^s \stackrel{\text{const}}{\vdash} e : S}$$

$$\text{(SR:SUBTYPING)}\quad \frac{\text{pc}; \Sigma^s; \Gamma^s \stackrel{[\text{const}]}{\vdash} e : S \qquad S \leq S'}{\text{pc}; \Sigma^s; \Gamma^s \stackrel{[\text{const}]}{\vdash} e : S'}$$

$$\text{(SR:FUN)}\quad \frac{\Sigma^s(f) = S_1 \xrightarrow{\text{pc}'} S_2 \qquad x : S_1 \in \Gamma^s \qquad \text{pc} \sqsubseteq \text{pc}'}{\text{pc}; \Sigma^s; \Gamma^s \vdash \text{app}(f, x) : S_2}$$

$$\text{(SR:L-ARG)}\quad \frac{x : S_1 \in \Gamma^s \qquad \Sigma^s(f) = S_1 \xrightarrow{\text{pc}'} S_2 \qquad \text{pc} \sqsubseteq \text{pc}' \qquad S_1 \blacktriangleleft k_1}{\text{pc}; \Sigma^s; \Gamma^s \stackrel{\text{const}}{\vdash} \text{app}(f, x) : S_2}$$

$$\text{(SR:C-FUN)}\quad \frac{\Sigma^s(f) = S_1 \xrightarrow{\text{pc}'/\text{const}} S_2 \qquad x : S_1 \in \Gamma^s \qquad \text{pc} \sqsubseteq \text{pc}'}{\text{pc}; \Sigma^s; \Gamma^s \stackrel{\text{const}}{\vdash} \text{app}(f, x) : S_2}$$

$$\text{(SR:IF)}\quad \frac{x : (\text{bool}, k_x) \in \Gamma^s \qquad \text{pc} \sqcup k_x; \Sigma^s; \Gamma^s \vdash e_t : S \qquad \text{pc} \sqcup k_x; \Sigma^s; \Gamma^s \vdash e_f : S \qquad \text{pc} \sqcup k_x \lhd S}{\text{pc}; \Sigma^s; \Gamma^s \vdash \text{if}(x, e_t, e_f) : S}$$

$$\text{(SR:L-IF)}\quad \frac{\text{pc} \sqcup k_x; \Sigma^s; \Gamma^s \stackrel{\text{const}}{\vdash} e_t : S \qquad \text{pc} \sqcup k_x; \Sigma^s; \Gamma^s \stackrel{\text{const}}{\vdash} e_f : S \qquad x : (\text{bool}, k_x) \in \Gamma^s \qquad \text{pc} \sqcup k_x \lhd S \qquad k_x \sqsubseteq k_1}{\text{pc}; \Sigma^s; \Gamma^s \stackrel{\text{const}}{\vdash} \text{if}(x, e_t, e_f) : S}$$

$$\text{(SR:LET)}\quad \frac{\text{pc}; \Sigma^s; \Gamma^s \vdash e_1 : S_1 \qquad \text{pc}; \Sigma^s; \Gamma^s, x : S_1 \vdash e_2 : S_2}{\text{pc}; \Sigma^s; \Gamma^s \vdash \text{let}(x, e_1, x.e_2) : S_2}$$

$$\text{(SR:L-LET)}\quad \frac{\text{pc}; \Sigma^s; \Gamma^s \stackrel{\text{const}}{\vdash} e_1 : S_1 \qquad S_1 \blacktriangleleft k_1 \qquad \text{pc}; \Sigma^s; \Gamma^s, x : S_1 \stackrel{\text{const}}{\vdash} e_2 : S_2}{\text{pc}; \Sigma^s; \Gamma^s \stackrel{\text{const}}{\vdash} \text{let}(x, e_1, x.e_2) : S_2}$$

$$\text{(SR:C-MATCH-L)}\quad \frac{x : (L(S), k_x) \in \Gamma^s \qquad \text{pc} \sqcup k_x; \Sigma^s; \Gamma^s \stackrel{\text{const}}{\vdash} e_1 : S_1 \qquad k_x \sqsubseteq k_1 \qquad \text{pc} \sqcup k_x; \Sigma^s; \Gamma^s, x_h : S, x_t : (L(S), k_x) \stackrel{\text{const}}{\vdash} e_2 : S_1 \qquad \text{pc} \sqcup k_x \lhd S_1}{\text{pc}; \Sigma^s; \Gamma^s \stackrel{\text{const}}{\vdash} \text{match}(x, e_1, (x_h, x_t).e_2) : S_1}$$

Figure 6. Selected security typing rules

resource-aware noninterference property since the value of the argument is always the same under any k-equivalent environments. It is reflected by rule SR:L-ARG.

Example. *Recall the functions* compare *and* p_compare *in Fig. 1. Suppose the content of the first list is secret and the length is public. Thus it has type $(L(int, h), \ell)$. While the second list controlled by adversaries is public, hence it has type $(L(int, \ell), \ell)$. Assume that the pc label is ℓ and $[\Gamma^s]_{\blacktriangleleft k_1} = [\Gamma^s]_{\blacktriangleleft \ell}$. The return value's label depends on the content of the elements of the first list whose label is h. Thus it must be assigned the label h to make the functions well-typed.*

$$\text{compare} : ((L(int, h), \ell), (L(int, \ell), \ell)) \xrightarrow{\ell} (bool, h)$$
$$\text{p\_compare} : ((L(int, h), \ell), (L(int, \ell), \ell)) \xrightarrow{\ell/const} (bool, h)$$

Here, both functions satisfy the noninterference property at security label ℓ. However, only p_compare *is a resource-aware noninterference function w.r.t. $[\Gamma^s]_{\blacktriangleleft \ell}$, or the secret list.*

Example. *Consider the following function* cond_rev *in which* rev *is the standard reverse function.*

```
let cond_rev(l1,l2,b1,b2) = if b1 then
  let r = if b2 then rev l1; l2
  else rev l2; l1 in rev r; () else ()
```

Assume that l_1, l_2, b_1 and b_2 have types $(L(int, h), \ell)$, $(L(int, h), \ell)$, $(bool, \ell)$, and $(bool, h)$, respectively. Given the rev *function is constant w.r.t. the argument, the inner conditional does not satisfy resource-aware noninterference. However, the* let *expression satisfies resource-aware noninterference w.r.t. $[\Gamma^s]_{\blacktriangleleft \ell} = \{l_1, l_2, b_2\}$. We can derive this by applying the rule* SR:C-GEN. *By the rule* SR:L-IF, *the outer conditional on low security data satisfies resource-aware noninterference w.r.t. $\{l_1, l_2, b_2\}$ at level ℓ. We derive the following type.*

$$\text{cond\_rev} : ((L(int, h), \ell), (L(int, h), \ell), (bool, \ell), (bool, h)) \xrightarrow{\ell/const} (unit, \ell)$$

We now prove the soundness of the type system w.r.t. the resource-aware noninterference property. It states that if e is a well-typed expression with the const annotation then e is a resource-aware noninterference expression at level k_1.

The following two lemmas are needed in the soundness proof. The first lemma states that the type system satisfies the standard *simple security* property [34] and the second shows that the type system prove classic noninterference.

Lemma 2. *Let $pc; \Sigma^s; \Gamma^s \vdash e : S$ or $pc; \Sigma^s; \Gamma^s \stackrel{const}{\vdash} e : S$. For all variables x in e, if $S \blacktriangleleft k_1$ then $\Gamma^s(x) \blacktriangleleft k_1$.*

Lemma 3. *Let $pc; \Sigma^s; \Gamma^s \vdash e : S$ or $pc; \Sigma^s; \Gamma^s \stackrel{const}{\vdash} e : S$, $E_1 \vdash e \Downarrow v_1$, $E_2 \vdash e \Downarrow v_2$, and $E_1 \equiv_{k_1} E_2$. Then $v_1 = v_2$ if $S \blacktriangleleft k_1$.*

Theorem 1. *If $\models E : \Gamma^s$, $E \vdash e \Downarrow v$, and $pc; \Sigma^s; \Gamma^s \stackrel{const}{\vdash} e : S$ then e is a resource-aware noninterference expression at k_1.*

Proof: The proof is done by induction on the structure of the typing derivation and the evaluation derivation. Let X be the set of variables $[\Gamma^s]_{\blacktriangleleft k_1}$. For all environments E_1, E_2 such that $E_1 \approx_X E_2$ and $E_1 \equiv_{k_1} E_2$, if $E_1 \vdash^{p_1}_{p'_1} e \Downarrow v_1$ and $E_2 \vdash^{p_2}_{p'_2} e \Downarrow v_2$. We then show that $p_1 - p'_1 = p_2 - p'_2$ and

$v_1 = v_2$ if $S \blacktriangleleft k_1$. We illustrate one case of the conditional expression. Suppose e is of the form if(x, e_t, e_f), thus the typing derivation ends with an application of either the rule SR:L-IF or SR:C-GEN. By Lemma 3, if $S \blacktriangleleft k_1$ then $v_1 = v_2$.

- Case SR:L-IF. By the hypothesis we have $E_1(x) = E_2(x)$. Assume that $E_1(x) = E_2(x) = \mathsf{true}$, by the evaluation rule E:IF-TRUE, $E_1 \left|\frac{p_1 - K^{\mathsf{cond}}}{p_1'}\right. e_t \Downarrow v_1$ and $E_2 \left|\frac{p_2 - K^{\mathsf{cond}}}{p_2'}\right. e_t \Downarrow v_2$. By induction for e_t we have $p_1 - p_1' = p_2 - p_2'$. It is similar for $E_1(x) = E_2(x) = \mathsf{false}$.
- Case SR:C-GEN. Since $E_1 \approx_X E_2$ w.r.t. Γ^s, we have $E_1 \approx_X E_2$ w.r.t. Γ^r. By the hypothesis we have $\mathrm{const}_X(e)$. Thus by Theorem 3, it follows $p_1 - p_1' = p_2 - p_2'$. ∎

IV. Type systems for lower bounds and constant resource usage

We now discuss how to automatically and statically verify constant resource usage, upper bounds, and lower bounds. For upper bounds we rely on existing work on automatic amortized resource analysis [19], [21]. This technique is based on an affine type system. For constant resource usage and lower bounds we introduce two new sub-structural resource-annotated type systems: The type system for constant resource usage is linear and the one for lower bounds is relevant.

A. Background

Amortized analysis: To statically analyze a program with the potential method [35], a mapping from program points to potentials must be established. One has to show that the potential at every program point suffices to cover the cost of any possible evaluation step and the potential of the next program point. The initial potential is then an upper bound on the resource usage of the program.

Linear potential for upper bounds: To automate amortized analysis, we fix a format of the potential functions and use LP solving to find the optimal coefficients. To infer linear potential functions, inductive data types are annotated with a non-negative rational numbers q [19]. For example, the type $L^q(\mathsf{bool})$ of Boolean lists with potential q defines potential $q \cdot n$, where n is the number of list's elements.

This idea is best explained by example. Consider the function *filter_succ* below that filters out positive numbers and increments non-positive numbers. As in RAML, we use OCaml syntax and *tick* commands to specify resource usage. If we filter out a number then we have a high cost (8 resource units) since *x* is, e.g., sent to an external device. If *x* is incremented we have a lower cost of 3 resource units. As a result, the worst-case resource consumption of *filter_succ(ℓ)* is $8|\ell| + 1$ (where 1 is for the cost that occurs in the nil case of the match). The function *fs_twice(ℓ)* applies *filter_succ* twice, to ℓ and to the result of *filter_succ(ℓ)*. The worst-case

```
let rec filter_succ(l) =
match l with
| [] →tick(1.0); []
| x::xs →if x > 0 then
    tick(8.0); filter_succ(xs)
  else tick(3.0); (x+1)::filter_succ(xs)

let fs_twice(l) =
  filter_succ(filter_succ(l))
```

Figure 7. Two OCaml functions with linear resource usage. The *worst-case* number of *ticks* executed by *fitler_succ(ℓ)* and *fs_twice(ℓ)* is $8|\ell| + 1$ and $11|\ell| + 2$ respectively. In the *best-case* the functions execute $3|\ell| + 1$ and $6|\ell| + 2$ ticks, respectively. The resource consumption is not constant.

behavior appears if no list element is filtered out in the first call and all elements are filtered out in the second call. The worst-case behavior is thus $11|\ell| + 2$. These upper bounds can be expressed with the following annotated function types, which can be derived using local type rules in Fig. 8.

$$\begin{aligned} \mathsf{filter\_succ} &: \quad L^8(\mathrm{int}) \xrightarrow{1/0} L^0(\mathrm{int}) \\ \mathsf{fs\_twice} &: \quad L^{11}(\mathrm{int}) \xrightarrow{2/0} L^0(\mathrm{int}) \end{aligned}$$

Intuitively, the first function type states that an initial potential of $8|\ell| + 1$ is sufficient to cover the cost of *filter_succ(ℓ)* and there is $0|\ell'| + 0$ potential left where ℓ' is the result of the computation. This is just one possible potential annotation of many. The right choice of the potential annotation depends on the use of the function result. For example, for the inner call of *filter_succ* in *fs_twice* we need the following annotation.

$$\mathsf{filter\_succ} : \quad L^{11}(\mathrm{int}) \xrightarrow{2/1} L^8(\mathrm{int})$$

It states that the initial potential of $11|\ell| + 2$ is sufficient to cover the cost of *filter_succ(ℓ)* and there is $8|\ell'| + 1$ potential left to be assigned to the returned list ℓ'. The potential of the result can then be used with the previous type of *filter_succ* to pay for the cost of the outer call.

$$\mathsf{filter\_succ} : \quad L^p(\mathrm{int}) \xrightarrow{q/q'} L^r(\mathrm{int}) \mid \quad q \geq q'{+}1 \wedge p \geq 8 \wedge p \geq 3{+}r$$

We can summarize all possible types of *filter_succ* with a linear constraint system. In the type inference, we generate such a constraint system and solve it with an off-the-shelf LP solver. To obtain tight bounds, we perform a whole-program analysis and minimize the coefficients in the input potential.

Surprisingly, this approach—as well as the new concepts we introduce here—can be extended to polynomial bounds [36], higher-order functions [37], [21], polymorphism [38], and user-defined inductive types [38], [21].

B. Resource annotations

The resource-annotated types are base types in which the inductive data types are annotated with non-negative rational numbers, called *resource annotations*.

$$A ::= \mathrm{unit} \mid \mathrm{bool} \mid \mathrm{int} \mid L^p(A) \mid A * A \quad (\text{for } p \in \mathbb{Q}_0^+)$$

A type context, $\Gamma^r : \text{VID} \rightarrow \mathscr{A}$, is a partial mapping from variable identifiers to resource-annotated types. The *underlying* base type and context denoted by $\widehat{A}$, and $\widehat{\Gamma^r}$ respectively can be obtained by removing the annotations. We extend all definitions such as $|v|$, $\models E : \Gamma$ and $\approx$ for base data types to resource-annotated data types by ignoring the annotations.

We now formally define the notation of *potential*. The potential of a value v of type A, written $\Phi(v : A)$, is defined by the function $\Phi : \mathit{Val} \rightarrow \mathbb{Q}_0^+$ as follows.

$$\begin{array}{l}\Phi(\,()\,:\text{unit}) = \Phi(b:\text{bool}) = \Phi(n:\text{int}) = 0\\ \Phi((v_1,v_2) : A_1 * A_2) = \Phi(v_1 : A_1) + \Phi(v_2 : A_2)\\ \Phi([v_1,\cdots,v_n] : L^p(A)) = n{\cdot}p + \Sigma_{i=1}^n \Phi(v_i : A)\end{array}$$

Example. *The potential of a list $v = [b_1, \cdots, b_n]$ of type $L^p(bool)$ is $n{\cdot}p$. Similarly, a list of lists of Booleans $v = [v_1, \cdots, v_n]$ of type $L^p(L^q(bool))$, where $v_i = [b_{i1}, \cdots, b_{im_i}]$, has the potential $n{\cdot}p + (m_1 + \cdots + m_n){\cdot}q$.*

Let Γ^r be a context and E be a well-formed environment w.r.t. Γ^r. The potential of $X \subseteq \text{dom}(\Gamma^r)$ under E is defined as $\Phi_E(X : \Gamma^r) = \Sigma_{x\in X}\Phi(E(x) : \Gamma^r(x))$. The potential of Γ^r is $\Phi_E(\Gamma^r) = \Phi_E(\text{dom}(\Gamma^r) : \Gamma^r)$. Note that if $x \notin X$ then $\Phi_E(X : \Gamma^r) = \Phi_{E[x\mapsto v]}(X : \Gamma^r)$. The following lemma states that the potential is the same under two well-formed size-equivalent environments.

Lemma 4. *If $E_1 \approx_X E_2$ then $\Phi_{E_1}(X : \Gamma^r) = \Phi_{E_2}(X : \Gamma^r)$.*

Annotated first-order data types are given as follows, where q and q' are rational numbers.

$$F ::= A_1 \xrightarrow{q/q'} A_2$$

A resource-annotated signature $\Sigma^r : \text{FID} \rightarrow \wp(\mathscr{F}) \setminus \{\emptyset\}$ is a partial mapping from function identifiers to a non-empty sets of annotated first-order types. That means a function can have different resource annotations depending on the context. The *underlying* base types are denoted by $\widehat{F}$. And the underlying base signature is denoted by $\widehat{\Sigma^r}$ where $\widehat{\Sigma^r}(f) = \widehat{\Sigma^r(f)}$.

C. Type system for constant resource consumption

The typing rules of the constant-resource type system define judgements of the form:

$$\Sigma^r; \Gamma^r \mathrel{\vert\!\frac{q}{q'}} e : A$$

where e is an expression and $q, q' \in \mathbb{Q}_0^+$. The intended meaning is that in the environment E, $q + \Phi_E(\Gamma^r)$ resource units are sufficient to evaluate e to a value v with type A and there are *exactly* $q' + \Phi(v : A)$ resource units left over.

The typing rules form a *linear* type system. It ensures that every variable is used exactly once by allowing exchange but not weakening or contraction [39]. The rules can be organized into syntax directed and structural rules.

Syntax-directed rules: The syntax-directed rules are shared among all type systems and selected rules are listed in Fig. 8. Rules like A:VAR and A:B-OP for leaf expressions (e.g., variable, binary operations, pairs) have fixed costs as specified by the constants K^x. Note that we require all available potential to be spent. The cost of the function call is represented by the constant K^{app} in the rule A:FUN and the argument carries the potential to pay for the function execution. In the rule A:LET, the cost of binding is represented by the constant K^{let}. The potentials carried by the contexts Γ_1^r and Γ_2^r are passed sequentially through the sub derivations. Note that the contexts are disjoint since our type system is linear. Multiple uses of variables must be introduced through the rule A:SHARE. Thus, the context split is deterministic. The rule A:IF is the key rule for ensuring constant resource usage. By using the same context Γ^r for typing both e_t and e_f, we ensure that the conditional expression has the same resource usage in size-equivalent environments independent of the value of the Boolean variable x. The rules for inductive data types are crucial for the interaction of the linear potential annotations with the constant potential, in which A:CONS shows how constant potential can be associated with a new data structure. The dual is A:MATCH-L, which shows how potential associated with data can be released. It is important that these transitions are made in a linear fashion: potential is neither lost or gained.

Sharing relation:

$$\frac{A \in \{\text{unit},\text{bool},\text{int}\}}{\curlyvee(A \mid A, A)} \qquad \frac{\curlyvee(A \mid A_1, A_2) \qquad \curlyvee(B \mid B_1, B_2)}{\curlyvee(A * B \mid A_1 * B_1, A_2 * B_2)}$$

$$\frac{\curlyvee(A \mid A_1, A_2) \qquad p = p_1 + p_2}{\curlyvee(L^p(A) \mid L^{p_1}(A_1), L^{p_2}(A_2))}$$

The *share expression* makes multiple uses of a variable explicit. While multiple uses of a variable seem to be in conflict with the linear type discipline, the *sharing relation* $\curlyvee(A \mid A_1, A_2)$ ensures that potential is treated in a linear way. It apportions potential to ensure that the total potential associated with all uses is equal to the potential initially associated with the variable. This relation is only defined for structurally-identical types which differ in at most the resource annotations.

Structural rules: To allow more programs to be typed we add two structural rules to the type system which can be applied to every expression. These rules are specific to the the constant-resource type system.

$$\text{(C:WEAKENING)}\ \frac{\Sigma^r; \Gamma^r \mathrel{\vert\!\frac{q}{q'}} e : B \qquad \curlyvee(A \mid A, A)}{\Sigma^r; \Gamma^r, x : A \mathrel{\vert\!\frac{q}{q'}} e : B} \qquad \text{(C:RELAX)}\ \frac{\Sigma^r; \Gamma^r \mathrel{\vert\!\frac{p}{p'}} e : A \qquad q \geq p \qquad q - p = q' - p'}{\Sigma^r; \Gamma^r \mathrel{\vert\!\frac{q}{q'}} e : A}$$

The rule C:RELAX reflects the fact that if it is sufficient to evaluate e with p available resource units and there are p' resource units left over then e can be evaluated with $p + c$

$$\frac{}{\Sigma^r; x : A \overset{K^{\mathsf{var}}}{\underset{0}{\vdash}} x : A}\;(\text{A:VAR})$$

$$\frac{\diamond \in \{\,\mathsf{and}\,,\,\mathsf{or}\,\}}{\Sigma^r; x_1 : \mathsf{bool}, x_2 : \mathsf{bool} \overset{K^{\mathsf{op}}}{\underset{0}{\vdash}} \mathsf{op}_\diamond(x_1, x_2) : \mathsf{bool}}\;(\text{A:B-OP})$$

$$\frac{\Sigma^r(f) = A_1 \xrightarrow{q/q'} A_2}{\Sigma^r; x : A_1 \overset{q+K^{\mathsf{app}}}{\underset{q'}{\vdash}} \mathsf{app}(f, x) : A_2}\;(\text{A:FUN})$$

$$\frac{\Sigma^r; \Gamma_1^r \overset{q-K^{\mathsf{let}}}{\underset{q_1'}{\vdash}} e_1 : A_1 \qquad \Sigma^r; \Gamma_2^r, x : A_1 \overset{q_1'}{\underset{q'}{\vdash}} e_2 : A_2}{\Sigma^r; \Gamma_1^r, \Gamma_2^r \overset{q}{\underset{q'}{\vdash}} \mathsf{let}(x, e_1, x.e_2) : A_2}\;(\text{A:LET})$$

$$\frac{\Sigma^r; \Gamma^r \overset{q-K^{\mathsf{cond}}}{\underset{q'}{\vdash}} e_t : A \qquad \Sigma^r; \Gamma^r \overset{q-K^{\mathsf{cond}}}{\underset{q'}{\vdash}} e_f : A}{\Sigma^r; \Gamma^r, x : \mathsf{bool} \overset{q}{\underset{q'}{\vdash}} \mathsf{if}(x, e_t, e_f) : A}\;(\text{A:IF})$$

$$\frac{\Sigma^r; \Gamma^r \overset{q-K^{\mathsf{matchN}}}{\underset{q'}{\vdash}} e_1 : A_1 \qquad \Sigma^r; \Gamma^r, x_h : A, x_t : L^p(A) \overset{q+p-K^{\mathsf{matchL}}}{\underset{q'}{\vdash}} e_2 : A_1}{\Sigma^r; \Gamma^r, x : L^p(A) \overset{q}{\underset{q'}{\vdash}} \mathsf{match}(x, e_1, (x_h, x_t).e_2) : A_1}\;(\text{A:MATCH-L})$$

$$\frac{}{\Sigma^r; x_h : A, x_t : L^p(A) \overset{p+K^{\mathsf{cons}}}{\underset{0}{\vdash}} \mathsf{cons}(x_h, x_t) : L^p(A)}\;(\text{A:CONS})$$

$$\frac{\Sigma^r; \Gamma^r, x_1 : A_1, x_2 : A_2 \overset{q}{\underset{q'}{\vdash}} e : B \qquad \curlyvee(A \mid A_1, A_2)}{\Sigma^r; \Gamma^r, x : A \overset{q}{\underset{q'}{\vdash}} \mathsf{share}(x, (x_1, x_2).e) : B}\;(\text{A:SHARE})$$

Figure 8. Selected syntax-directed rules of the resource type systems. They are shared among all type systems.

resource units and there are exactly $p' + c$ resource units left over, where $c \in \mathbb{Q}_0^+$. Rule C:WEAKENING states that an extra variable can be added into the given context if its potential is zero. The condition is enforced by $\curlyvee(A \mid A, A)$ since $\Phi(v : A) = \Phi(v : A) + \Phi(v : A)$ or $\Phi(v : A) = 0$. These rules can be used in branchings such as the conditional or the pattern match to ensure that subexpressions are typed using the same contexts and potential annotations.

Example. *Consider again the function p_compare in Fig. 4 in which the nil case of the second matching on l is padded with tick(5.0); aux(false,xs,[]) and the resource consumption is defined using* tick *annotations. The resource usage of p_compare(h, ℓ) is constant w.r.t. h, that is, it is exactly $5|h| + 1$. This is reflected by the following type.*

$$\mathsf{p\_compare} : \quad (L^5(\mathit{int}), L^0(\mathit{int})) \xrightarrow{1/0} \mathit{bool}$$

It can be understood as follows. If the input list h carries 5 potential units per element then it is sufficient to cover the cost of p_compare(h, ℓ), no potential is wasted, and 0 potential is left.

Soundness: That soundness theorem states that if e is well-typed in the resource type system and evaluates to a value v then the difference between the initial and the final potential is the net resources usage. Moreover, if the potential annotations of the return value and all variables not belonging to a set $X \subseteq \mathrm{dom}(\Gamma^r)$ are zero then e is constant-resource w.r.t. X.

Theorem 2. *If $\models E : \Gamma^r$, $E \vdash e \Downarrow v$, and $\Sigma^r; \Gamma^r \overset{q}{\underset{q'}{\vdash}} e : A$, then for all $p, r \in \mathbb{Q}_0^+$ such that $p = q + \Phi_E(\Gamma^r) + r$, there exists $p' \in \mathbb{Q}_0^+$ satisfying $E \overset{p}{\underset{p'}{\vdash}} e \Downarrow v$ and $p' = q' + \Phi(v : A) + r$.*

Proof: The proof proceeds by a nested induction on the derivation of the evaluation judgement and the typing judgement, in which the derivation of the evaluation judgement takes priority over the typing derivation. We need to induct on both, evaluation and typing derivation. An induction on only the typing derivation would fail for the case of function application, which increases the size of the typing derivation, while the size of the evaluation derivation does not increase. An induction on only the evaluation judgement would fail because of structural rules such as C:WEAKENING. If such a rule is the final step in the derivation then the size of typing derivation decreases while the length of evaluation derivation is unchanged. The additional constant r is needed to make the induction case for the let rule work. ∎

Theorem 3. *If $\models E : \Gamma^r$, $E \vdash e \Downarrow v$, $\Sigma^r; \Gamma^r \overset{q}{\underset{q'}{\vdash}} e : A$, $\curlyvee(A \mid A, A)$, and $\forall x \in \mathit{dom}(\Gamma^r) \setminus X. \curlyvee(\Gamma^r(x) \mid \Gamma^r(x), \Gamma^r(x))$ then e is constant resource w.r.t. $X \subseteq \mathit{dom}(\Gamma^r)$.*

D. Type system for upper bounds

If we treat potential as an *affine* resource then we arrive at the original amortized analysis for upper bounds [19]. To this end, we allow unrestricted weakening and a relax rule in which we can waste potential.

$$\frac{\Sigma^r; \Gamma^r \overset{p}{\underset{p'}{\vdash}} e : A \qquad q \geq p \qquad q - p \geq q' - p'}{\Sigma^r; \Gamma^r \overset{q}{\underset{q'}{\vdash}} e : A}\;(\text{U:RELAX}) \qquad \frac{\Sigma^r; \Gamma^r \overset{q}{\underset{q'}{\vdash}} e : B}{\Sigma^r; \Gamma^r, x : A \overset{q}{\underset{q'}{\vdash}} e : B}\;(\text{U:WEAKENING})$$

Additionally, we can use subtyping to waste linear potential [19]. (See the dual definition for subtyping for lower bounds below.) Similarly to Theorem 2, we can prove the following theorem.

Theorem 4. *If $\models E : \Gamma^r$, $E \vdash e \Downarrow v$, and $\Sigma^r; \Gamma^r \mid\!\frac{q}{q'}\; e : A$, then for all $p, r \in \mathbb{Q}_0^+$ such that $p \geq q + \Phi_E(\Gamma^r) + r$, there exists $p' \in \mathbb{Q}_0^+$ satisfying $E \mid\!\frac{p}{p'}\; e \Downarrow v$ and $p' \geq q' + \Phi(v : A) + r$.*

E. Type system for lower bounds

The type judgements for lower bounds have the same form and data types as the type judgements for constant resource usage and upper bounds. However, the intended meaning of the judgement $\Sigma^r; \Gamma^r \mid\!\frac{q}{q'}\; e : A$ is the following. Under given environment E, less than $q + \Phi_E(\Gamma)$ resource units are not sufficient to evaluate e to a value v so that more than $q' + \Phi(v : A)$ resource units are left over.

The syntax-directed typing rules are the same as the rules in the constant-resource type system as given in Fig. 8. In addition, we have the structural rules in Fig. 9. The rule L:RELAX is dual to U:RELAX. In L:RELAX, potential is treated as a *relevant* resource: We are not allowed to waste potential but we can create potential out of the blue if we ensure that we either use it or pass it to the result. The same idea is formalized for the linear potential with the sub-typing rules L:SUBTYPE and L:SUPERTYPE. The sub-typing relation is defined as follows.

$$\frac{A \in \{\text{unit}, \text{bool}, \text{int}\}}{A <: A} \qquad \frac{A_1 <: A_2 \quad p_1 \leq p_2}{L^{p_1}(A_1) <: L^{p_2}(A_2)} \qquad \frac{A_1 <: A_2 \quad B_1 <: B_2}{A_1 * A_2 <: B_1 * B_2}$$

It holds that if $A <: B$ then $\widehat{A} = \widehat{B}$ and $\Phi(v : A) \leq \Phi(v : B)$. Suppose that it is not sufficient to evaluate e with p available resource units to get p' resource units left over. L:SUBTYPE reflects the fact that we also cannot evaluate e with p resources get more than p' resource units after the evaluation. L:SUPERTYPE says that we also cannot evaluate e with less than p and get p' resource units afterwards.

Example. *Consider again the functions* filter_succ *and* fs_twice *given in Fig. 7 in which the resource consumption is defined using* tick *annotations. The best-case resource usage of* filter_succ(ℓ) *is* $3|\ell| + 1$ *and best-case resource usage of* fs_twice(ℓ) *is* $6|\ell| + 2$. *This is reflected by the following function types for lower bounds.*

$$\begin{aligned} \mathsf{filter\_succ} : \quad & L^3(\mathit{int}) \xrightarrow{1/0} L^0(\mathit{int}) \\ \mathsf{fs\_twice} : \quad & L^6(\mathit{int}) \xrightarrow{2/0} L^0(\mathit{int}) \end{aligned}$$

To derive the lower bound for fs_twice, *we need the same compositional reasoning as for the derivation of the upper bound. For the inner call of* filter_succ *we use the type*

$$\mathsf{filter\_succ} : \quad L^6(\mathit{int}) \xrightarrow{2/1} L^3(\mathit{int})\ .$$

It can be understood as follows. If the input list carries 6 *potential units per element then, for each element, we can either use all* 6 *(if case) or we can use* 3 *and assign* 3 *to the output (else case).*

The type system for lower bounds is a *relevant* type system [39]. That means every variable is used at least once

(L:RELAX)
$$\frac{\Sigma^r; \Gamma^r \mid\!\frac{p}{p'}\; e : A \qquad q \geq p \qquad q - p \leq q' - p'}{\Sigma^r; \Gamma^r \mid\!\frac{q}{q'}\; e : A}$$

(L:WEAKENING)
$$\frac{\Sigma^r; \Gamma^r \mid\!\frac{q}{q'}\; e : B \qquad \curlyvee(A \mid A, A)}{\Sigma^r; \Gamma^r, x : A \mid\!\frac{q}{q'}\; e : B}$$

(L:SUBTYPE)
$$\frac{\Sigma^r; \Gamma^r \mid\!\frac{q}{q'}\; e{:}A \qquad A{<:}B}{\Sigma^r; \Gamma^r \mid\!\frac{q}{q'}\; e : B}$$

(L:SUPERTYPE)
$$\frac{\Sigma^r; \Gamma^r, x{:}B \mid\!\frac{q}{q'}\; e : C \qquad A{<:}B}{\Sigma^r; \Gamma^r, x : A \mid\!\frac{q}{q'}\; e : C}$$

Figure 9. Structural rules for lower bounds.

$$\dfrac{\dfrac{\dfrac{\Sigma^r; \Gamma^r, x_1 : A, x_2 : A \mid\!\frac{q}{q'}\; e : B \qquad A_2 <: A}{\Sigma^r; \Gamma^r, x_1 : A, x_2 : A_2 \mid\!\frac{q}{q'}\; e : B} \qquad A_1 <: A}{\Sigma^r; \Gamma^r, x_1 : A_1, x_2 : A_2 \mid\!\frac{q}{q'}\; e : B} \qquad \curlyvee(A \mid A_1, A_2)}{\Sigma^r; \Gamma^r, x : A \mid\!\frac{q}{q'}\; \mathsf{share}(x, (x_1, x_2).e) : B} \text{ (L:CONTR)}$$

Figure 10. Derivation of the contraction rule for lower-bounds.

by allowing *exchange* and *contraction properties*, but not *weakening*. However, as in the constant-time type system we allow a restricted form of weakening if the potential annotations are zero using the rule L:WEAKENING. The following lemma states formally the contraction property which is derived in Fig. 10.

Lemma 5. *If $\Sigma^r; \Gamma^r, x_1 : A, x_2 : A \mid\!\frac{q}{q'}\; e : B$ then $\Sigma^r; \Gamma^r, x : A \mid\!\frac{q}{q'}\; share(x, (x_1, x_2).e) : B$*

The following theorems establish the soundness of the analysis. The proofs can be found in the TR [28]. Theorem 6 is proved by induction and Theorem 5 follows by contradiction.

Theorem 5. *Let $\models E : \Gamma^r$, $E \vdash e \Downarrow v$, and $\Sigma^r; \Gamma^r \mid\!\frac{q}{q'}\; e : A$. Then for all $p, r \in \mathbb{Q}_0^+$ such that $p < q + \Phi_E(\Gamma^r) + r$, there exists no $p' \in \mathbb{Q}_0^+$ satisfying $E \mid\!\frac{p}{p'}\; e \Downarrow v$ and $p' \geq q' + \Phi(v : A) + r$.*

Theorem 6. *Let $\models E : \Gamma^r$, $E \vdash e \Downarrow v$, and $\Sigma^r; \Gamma^r \mid\!\frac{q}{q'}\; e : A$. Then for all $p, p' \in \mathbb{Q}_0^+$ such that $E \mid\!\frac{p}{p'}\; e \Downarrow v$ we have $q + \Phi_E(\Gamma^r) - (q' + \Phi(v : A)) \leq p - p'$.*

F. Mechanization

We mechanized the soundness proofs for both the two new type systems as well as the classic AARA type system using the proof assistant Agda. The development is roughly 4000 lines of code, which includes the inference rules, the operational cost semantics, a proof of type preservation, and the soundness theorems for each type system.

One notable difference is our implementation of the typing contexts. In Agda our contexts are implemented as lists of pairs of variables and their types. Moreover, in our typing rules whenever a variable is added to the context we require the variable is fresh with respect to the existing context. This requirement is important as it allows us to preserve the invariant that the context is well formed with respect to the environment as we induct over typing and evaluation judgements in our soundness proofs. Furthermore, as our typing contexts are ordered lists we added an *exchange* rule to our typing rules.

Another important detail is in the implementation of potential. Potential $\Phi(v : A)$ for a value only is defined for well formed inputs. Inputs such as $\Phi(\mathsf{nil} : \mathsf{bool})$ are not defined. Agda is total language and as such prohibits users from implementing partial functions. Thus we require in our Agda implementation that when calculating the potential of a value of a given type the user provide a derivation that the value is well formed with respect to that type. Similarly when calculating the potential of a context, $\Phi_E(\Gamma^r)$, with respect to an environment we require that the user provide a derivation that the context is well formed with respect to that environment.

Lastly, whereas the type systems and proofs presented here used positive rational numbers, in the Agda implementation we use natural numbers. This deviation was simply due to the lacking support for rationals in the Agda standard library. By replacing a number of trivial lemmas, mostly related to associativity and commutativity, the proofs and embeddings could be transformed to use rational numbers instead.

V. Quantifying and transforming out leakages

We present techniques to quantify the amount of information leakage through resource usage and transform leaky programs into constant resource programs. The quantification relies on the lower and upper bounds inferred by our resource type systems. The transformation pads the programs with dummy computations so that the evaluations consume the same amount of resource usage and the outputs are identical with the original programs. In the current implementation, these dummy computations are added into programs by users and the padding parameters are automatically added by our analyzer to obtain the optimal values. It would be straightforward to make the process fully automatic but the interactive flavor of our approach helps to get a better understanding of the system.

A. Quantification

Recall from Section III that we assume an adversary at level k_1 who is always able to observe *1)* the values of variables in $[\Gamma^s]_{\blacktriangleleft k_1}$, and *2)* the final resource consumption of the program. For many programs, it may be the case that changes to the secret variables $[\Gamma^s]_{\not\blacktriangleleft k_1}$ effect observable differences in the program's final resource consumption, but only allow the attacker to learn partial information about the corresponding secrets. In this section, we show that the upper and lower-bound information provided by our type systems allow us to derive bounds on the amount of partial information that is leaked.

To quantify the amount of leaked information, we measure the number of distinct environments that the attacker could deduce as having produced a given resource consumption observation. However, because there may be an unbounded number of such environments, we parameterize this quantity on the size of the values contained in each environment. Let $\mathbf{E}^N$ denote the space of environments with values of size characterized by N. Given an environment E and expression e, define $U(E, e) = p_\delta$ such that $E \,{\vdash^{p}_{p'}}\, e \Downarrow v$ and $p_\delta = p - p'$. Then for an expression e and resource observation p_δ, we define the set $R_N(e, p_\delta)$ which captures the attacker's uncertainty about the environment which produced p_δ:

$$R_N(e, p_\delta) = \{E' \in \mathbf{E}^N : U(E, e) = p_\delta\}$$

Notice that when $|R_N(e, p)| = 1$, the attacker can deduce exactly which environment was used, whereas when this quantity is large little additional information is learned from p_δ. This gives us a natural definition of leakage, which is obtained by aggregating the inverse of the cardinality of R_N over the possible initial environments of e:

$$C_N(e) = \left(\sum_{E \in \mathbf{E}^N} \frac{1}{|R_N(e, U(E, e))|} \right) - 1$$

$C_N(e)$ corresponds to our intuition about leakage. When e leaks no information through resource consumption, then each term in the summation will be $1/|\mathbf{E}^{sizes}|$ giving $C_N(e) = 0$, whereas if e leaks perfect information about its starting environment then each term will be 1, leading to $C_N(e) = |\mathbf{E}^N| - 1$.

Theorem 7. *Let P_N^e be the complete set of resource observations producible by expression e under environments of size N, i.e.,*

$$P_N^e = \{p : \exists E \in \mathbf{E}^N . U(E, e) = p\}$$

Then $|P_N^e| = C_N(e) + 1$.

Lemma 6. *Let $l_e(N)$ and $u_e(N)$ be lower and upper-bounds on the resource consumption of e for inputs of size N. If $U(E, e) \in \mathbb{Z}$ for all environments E, then $C_N(e) \leq u_e(N) - l_e(N)$.*

Lemma 7. *Assume that environments are sampled uniform-randomly from $\mathbf{E}^N$. Then the Shannon entropy of P_N^e is given by $C_N(e)$: $H(P_N^e) \leq \log_2(C_N(e) + 1)$*

Lemma 6 leverages Theorem 7 to derive an upper-bound on leakage from upper and lower resource bounds. This result only holds when the resource observations of e are integral, which ensures the interval $[l_e(N), u_e(N)] \supseteq P_N^e$ is finite. Lemma 7 relates $C_N(e)$ to Shannon entropy, which is commonly used to characterize information leakage [40], [41], [42].

B. Transformation

To transform programs into constant resource programs we extend the type system for constant resource use from Section IV. Recall that the type system treats potential in a linear fashion to ensure that potential is not wasted. We will now add *sinks* for potential which will be able to absorb excess potential. At runtime the sinks will consume the exact amount of resources that have been statically-absorbed to ensure that potential is still treated in a linear way. The advantage of this approach is that the worst-case resource consumption is often not affected by the transformation. Additionally, we do not need to keep track of resource usage at runtime to pad the resource usage at the sinks, because the amount of resource that must be discarded is statically-determined by the type system. Finally, we automatically obtain a type derivation that serves as a proof that the transformation is constant resource.

More precisely, the sinks are represented by the syntactic form: $\text{consume}_{(\text{A},p)}(x)$. Here, A is a resource-annotated type and $p \in \mathbb{Q}_{\geq 0}$ is a non-negative rational number. The idea is that A and p define the resource consumption of the expression. In the implementation, the user only has to write $\text{consume}(x)$, and the annotations are added via automatic syntax elaboration during the resource type inference.

Let E be a well-formed environment w.r.t. Γ^r. For every $x \in \text{dom}(\Gamma)$ with $\Gamma^r(x) = A$, the expression $\text{consume}_{(\text{A},p)}(x)$ consumes $\Phi(E(x) : A) + p$ resource units and evaluates to $()$. The evaluation and typing rules for sinks are:

$$\text{(E:CONSUME)}\quad \frac{q = q' + \Phi(E(x) : A) + p}{E \mid\!\frac{q}{q'}\ \text{consume}_{(\text{A},p)}(x) \Downarrow ()}$$

$$\text{(A:CONSUME)}\quad \frac{}{\Sigma^r; x{:}A \mid\!\frac{p}{0}\ \text{consume}_{(A,p)}(x) : \text{unit}}$$

The extension of the proof of Theorem 2 to consume expressions is straightforward.

Adding consume expressions: Let e_i be a subexpression of e and let e_i' be the expression $\text{let}(z, \text{consume}(x_1, \cdots, x_n), z.e_i)$ for some variables x_i. Let e' be the expression obtained from e by replacing e_i with e_i'. We write $e \hookrightarrow e'$ for such a transformation. Note that additional *share* and *let* expressions have to be added to convert e_i' into share-let normal form.

Lemma 8. *If $\Sigma; \Gamma \vdash e : T$, $E \vdash e \Downarrow v$, and $e \hookrightarrow e'$ then $\Sigma; \Gamma \vdash e' : T$ and $E \vdash e' \Downarrow v$.*

To transform an expression e into a constant resource expressions we perform multiple transformations $e \hookrightarrow e'$ which do not affect the type and semantics of e. This can be done automatically but in our implementation it works in an interactive fashion, meaning that users are responsible for the locations where consume expressions are put. The analyzer will infer the annotations A and constants p of the given consume expressions during type inference. If the inference is successful then we have $\text{const}_X(e')$ for the transformed program e'.

Example. *Recall the function* compare *form Fig. 1. To turn* compare *into a constant resource function. We insert* consume *expressions as shown below. Users can insert many* consume *expressions and the analyzer will determine which* consumes *are actually needed.*

```
let rec c_compare(h,l) = match h with
| [] →match l with
  | [] →tick(1.0); true
  | y::ys →tick(1.0); false
| x::xs →match l with
  | [] →tick(1.0); consume(xs); false
  | y::ys →if (x = y) then
      tick(5.0); c_compare(xs,ys)
    else tick(5.0); consume(xs); false
```

We automatically obtain the following typing of the transformed function and the consume expressions:

$$\begin{aligned} \text{c\_compare} &: (L^5(\mathit{int}), L^0(\mathit{int})) \xrightarrow{1/0} \mathit{bool} \\ \text{consume} &: L^5(\mathit{int}) \xrightarrow{5/0} \mathit{unit} && (\textit{at line } 6) \\ \text{consume} &: L^5(\mathit{int}) \xrightarrow{1/0} \mathit{unit} && (\textit{at line } 9) \end{aligned}$$

The worst-case resource consumption of the unmodified function c_compare *is $1 + 5|h|$. Thus the consumption of the first* consume *must be $5 + 5(|h| - 1 - |\ell|)$ when h is longer than l. Otherwise, the consumption is zero. The second one consumes $1 + 5(|h_1| - 1)$, where h_1 is the sub-list of h from the first node which is different from the corresponding node in l.*

VI. IMPLEMENTATION AND EVALUATION

Type Inference: Type inference for the constant resource and lower bound systems are implemented in RAML [21]. RAML is integrated into Inria's OCaml compiler and supports polynomial bounds, user-defined inductive types, higher-order functions, polymorphism, arrays, and references. All features are implemented for the new type systems, as they are straightforward extensions of the simplified rules presented in this paper. The implementation is publicly available in an easy-to-use web interface [43].

The type inference is technically similar to the inference of upper bounds [19]. We first integrate the structural rules of the respective type system in the syntax directed rules. For example, weakening and relaxation is applied at branching points such as conditional. We then compute a type derivation in which all resource annotations are replace by (yet unknown) variables. For each type rule we produce a set of linear constraints that specify the properties of valid annotations. These linear constraints are then solved by the LP solver CLP to obtain a type derivation in which the annotations are rational numbers.

Constant Function	LOC	Metric	Resource Usage	Time
cond_rev : $(L(\text{int}), L(\text{int}), \text{bool}) \to \text{unit}$	20	steps	$13n{+}13x{+}35$	0.03s
trunc_rev : $(L(\text{int}), \text{int}) \to L(\text{int})$	28	function calls	$1n$	0.06s
ipquery : $L(\text{logline}) \to (L(\text{int}), L(\text{int}))$	86	steps	$86n{+}99$	0.86s
kmeans : $L(\text{float}, \text{float}) \to L(\text{float}, \text{float})$	170	steps	$1246n{+}3784$	8.18s
tea_enc : $(L(\text{int}), L(\text{int}), \text{nat}) \to L(\text{int})$	306	ticks	$128n^2z{+}32nxz{+}1184nz{+}96n{+}128z{+}96$	13.73s
tea_dec : $(L(\text{int}), L(\text{int}), \text{nat}) \to L(\text{int})$	306	ticks	$128n^2z{+}32nxz{+}1184nz{+}96n{+}96z{+}96$	14.34s

Function	LOC	Metric	Lower Bound	Time	Upper Bound	Time
compare : $(L(\text{int}), L(\text{int})) \to \text{bool}$	60	steps	7	0.05s	$16n{+}7$	0.09s
find : $(L(\text{int}), \text{int}) \to \text{bool}$	40	steps	5	0.04s	$14n{+}5$	0.02s
rsa : $(L(\text{bool}), \text{int}, \text{int}) \to \text{int}$	42	multiplications	$1n$	0.07s	$2n$	0.05s
filter : $L(\text{int}) \to L(\text{int})$	30	steps	$13n{+}5$	0.05s	$20n{+}5$	0.04s
isortlist : $L(L(\text{int})) \to L(L(\text{int}))$	60	steps	$21n{+}5$	0.13s	$12n^2{+}9n{+}10n^2m{-}10nm{+}5$	0.43s
bfs_tree : $(\text{btree}, \text{int}) \to \text{btree option}$	116	steps	15	0.30s	$92n{+}24$	0.32s

Table I
AUTOMATICALLY-DERIVED BOUNDS WITH RESOURCE AWARE ML

An interesting challenge lies in finding a solution for the linear constraints that leads to the best bound for a given function. For upper bounds, we simply disregard the potential of the result type and provide an objective function that minimizes the annotations of the arguments. The same strategy works the constant-time type systems. An interesting property is that the solution to the linear program is unique if we require that the potential of the result type is zero. To obtain the optimal lower bound we want to maximize the potential of the arguments and minimize the potential of the result. We currently simply maximize the potential of the arguments while requiring the potential of the result to be zero. Another approach would be to first minimize the output potential and then maximize the input potential.

Resource-aware noninterference: We are currently integrating our constant-time type system with FlowCaml [31]. The combined inference is based on the typing rules in Fig. 14. It is possible to derive a set of type inference rules in the same way as for FlowCaml [44], [32]. One of the challenges in the integration is interfacing FlowCaml's type inference with our constant-time type system in rule SR:C-GEN. In the implementation, we intend for each application of SR:C-GEN to generate an intermediate representation of the expression in RAML for the expression under consideration, in which all types are annotated with fresh resource annotations along with the set of variables X. The expression is marked with the qualifier const if RAML can prove that it is constant time. The type inference algorithm always tries to apply the syntax-directed rules first before using SR:C-GEN.

Evaluation: Table I shows the verification and computation of constant resource usage, lower, and upper bounds for different functions, together with the lines of code (LOC) of the analyzed function and the run time of the analysis in seconds. Note that lower and upper bounds are identical when a function is constant. In the computed bounds, n is the size of the first argument, $m = \max_{1 \le i \le n} m_i$ where m_i are the sizes of the first argument's elements, x is the size of the second argument, and z is the value of the third argument.

The cost models are specified by different cost metrics that are appropriate for the respective application, e.g., number of evaluation steps or number of multiplication operations. Note that the computed upper bounds are also the resource usages of functions which are padded using *consume* expressions. The experiments were run on a machine with Intel Core i5 2.4 GHz processor and 8GB RAM under OS X 10.11.5. The run time of the analysis varies from 0.02 to 14.34 seconds depending on the function's code complexity. The example programs that we analyzed consist of commonly-used primitives (cond_rev, trunc_rev, compare, find, filter), functions related to cryptography (tea_enc, tea_dec, rsa), and examples taken from Haeberlan et al. [8] (ipquery, kmeans). The full source code of the examples can be found in the technical report [28].

The encryption functions tea_enc and tea_dec correspond to the encryption and decryption routines of the Corrected Block Tiny Encryption Algorithm [45], a block cipher presented by Needham and Wheeler in an unpublished technical report in 1998. Our implementation correctly identifies these operations as constant-time in the number of primitive operations performed. We applied this cost model for the tea examples due to the presence of bitwise operations in the original algorithm, which are not currently supported in RAML. In order to derive a more meaningful bound, we implemented bitwise operations in the example source and counted them as single operations.

The two examples taken from Haeberlen et al. [8] were originally created in a study of timing attacks in differentially-private data processing systems. ipquery applies pattern matching to a database derived from Apache server logs, counting the number of matches and non-matches. kmeans implements the k-means clustering algorithm [46], which partitions a set of geometric points into k clusters that minimize the total inter-cluster distance between points. Haeberlen et al. demonstrated that when a query applied to a dataset introduces attacker-observable timing variations, then the privacy guarantees provided by differential privacy are negated. To address this, they proposed a mitigation approach that enforces constant-time behavior by aborting or padding the query's runtime. Our implementation is able to determine that the queries, as we implemented them, were constant-time to begin with, and thus did not need black-box mitigation.

VII. Related work

Resource bounds: Our work builds on past research on automatic amortized resource analysis (AARA). AARA has been introduced by Hofmann and Jost for a strict first-order functional language with built-in data types to derive linear heap-memory bounds [19]. It has then been extended to polynomial bounds [47], [36], [48], [59], [60] for strict and higher-order [37], [21] functions. AARA has also been used to derive linear bounds for lazy functional programs [49], [50] and object-oriented programs [51], [52]. In another line of work, the technique has been integrated into separation logic [53] to derive bounds that depend on mutable data-structures, and into Hoare logic to derive linear bounds that depend on integers [54], [55]. Amortized analysis has also been used to manually verify the complexity of algorithms and data-structures using proof assistants [56], [57]. In contrast to our work, these techniques can only derive upper bounds and cannot prove constant resource consumption.

The focus on upper bounds is shared with automatic resource analysis techniques that are based on sized types [61], [62], linear dependent types [63], [64], and other type systems [65], [66], [67]. Similarly, semiautomatic analyses [68], [69], [70], [71] focus on upper bounds too.

Automatic resource bound analysis is also actively studied for imperative languages using recurrence relations [72], [73], [74] and abstract interpretation [75], [76], [77], [78], [79]. While these techniques focus on worst-case bounds, it is possible to use similar techniques for deriving lower bounds [80]. The advantage of our method is that it is compositional, deals well with amortization effects, and works for language features such as user-defined data types and higher-order functions. Another approach to (worst-case) bound analysis is based on techniques from term rewriting [81], [82], [83], which mainly focus on upper bounds. One line of work [84] derives lower bounds on the *worst-case* behavior of programs which is different from our lower bounds on the best-case behavior.

Side channels: Analyzing and mitigating potential sources of side channel leakage is an increasingly well-studied area. Several groups have proposed using type systems or other program analyses to transform programs into constant-time versions by padding with "dummy" commands [85], [86], [87], [40], [88], [16]. Because these systems do not account for timing explicitly, as is the case for our work, this approach will in nearly all cases introduce an unnecessary performance penalty. The most recent system by Zhang et al. [40] describes an approach for mitigating side channels using a combination of security types, hardware assistance, and predictive mitigation [89]. Unlike the type system in Section III, they do not guarantee that information is not leaked through timing. Rather, they show that the amount of this leakage is bounded by the variation of the mitigation commands.

Köpf and Basin [42] presented an information-theoretic model for adaptive side-channel attacks that occur over multiple runs of a program, and an automated analysis for measuring the corresponding leakage. Because their analysis is doubly-exponential in the number of steps taken by the attacker, they describe an approximate version based on a greedy heuristic. Mardziel et al. [90] later generalized this model to probabilistic systems, secrets that change over time, and wait-adaptive adversaries. Pasareanu et al. [91] proposed a symbolic approach for the multi-run setting based on MaxSAT and model counting. Doychev et al. [92] and Köpf et al. [41] consider cache side channels, and present analyses that over-approximate leakage using model-counting techniques. While these analyses are sometimes able to derive useful bounds on the leakage produced by binaries on real hardware, they do not incorporate security labels to distinguish between different sources, and were not applied to verifying constant-time behavior.

FlowTracker [14] and ct-verif [13] are both constant-time analyses built on top of LLVM which reason about timing and other side-channel behavior indirectly through control and address-dependence on secret inputs. VirtualCert [15] instruments CompCert with a constant-time analysis based on similar reasoning about control and address-dependence. These approaches are intended for code that has been written in "constant-time style", and thus impose effective restrictions on the expressiveness of the programs that they will work on. Because our approach reasons about resources explicitly, it imposes no a priori restrictions on program expressiveness.

Information flow: A long line of prior work looks at preventing undesired information flows using type systems. Sabelfeld and Myers [93] present an excellent overview of much of the early work in this area. The work most closely related to our security type system is FlowCaml [32], which provides a type system that enforces noninterference for a core of ML with references, exceptions, and let-polymorphism. The portion of our type system that applies to traditional noninterference coincides with the rules used in FlowCaml. However, the rules in our type system are not only designed to track flows of information, but they are also used to incorporate the information flow and resource usage behavior such as the rules SR:L-IF and SR:L-LET. Moreover, our type system constructs a flexible interface between FlowCaml and the resource type system, which means the rules can be easily adapted to integrate into any information-flow type system.

The primary difference between our work and the prior work on information-flow type systems is best summarized in terms of our attacker model. Whereas prior work assumes an attacker that can manipulate low-security inputs and observe low-security outputs, our type system enhances this attacker by granting the ability to observe the program's final resource consumption. This broadens the relevant class of attacks to include resource side channels, which we prevent by extending a traditional information flow type system with explicit reasoning about the resource behavior of the program.

VIII. Discussion

The definition of resource-aware noninterference given in Definition 2 assumes an adversary whose observations of resource consumption match the cost semantics with respect to the cost model given in Section IV. Depending on how the costs are parameterized, this may not match the actual resource use in a physical environment on modern hardware. Architectural features such as caching and variable-duration instructions need to be accounted for in the cost semantics, or the guarantees might not hold in practice [94], [7], [95]. Moreover, additional artifacts of the compilation process can affect the constant-resource guarantees established by the type system. Certain optimization passes and garbage collectors might affect timing properties in ways that lead to vulnerabilities if not accounted for by the cost semantics.

The cost semantics used in this work is conceptually straightforward, and corresponds to the resource model encapsulated by the high-level programming language. Accordingly, our verifier is oblivious to the machine instructions and operand values that are eventually executed after the high-level code is compiled. In particular, the fact that our cost model effectively counts the total number of language primitives that are executed, and not the corresponding processor instructions with caching and other micro-architectural effects accounted for, means that compiled programs may not satisfy resource-aware noninterference in practice despite being provable within our type system.

Although architectural timing channels are nominally invisible at the source-language level, it may be possible to incorporate these aspects into the cost semantics with specific assumptions about the target platform and compiler toolchain. Doing so with a high degree of precision is challenging, as the semantics may need to track extensive state to accurately reflect the timing behavior of the underlying platform. Another approach is to incorporate dependence on these features indirectly, as in Zhang et al. [40] where security labels are associated with hardware states to track information flow dependencies throughout the hardware environment. This approach is compatible with our resource-aware noninterference type system, but is less flexible for the programmer as it is subject to the same types of imprecision present in information-flow type systems. We leave as future work developing more precise models that remain faithful to the resource-consumption subtleties of hardware platforms.

Another limitation of this work follows from the imprecision of the information-flow type system that is integrated with our constant-resource type system to verify resource-aware noninterference. It is well-known that such type systems are more conservative than the semantics of allowed noninterference [24], [25], [26], and this applies to our work as well. In particular, a variable conservatively identified as high-security could influence resource usage, leading our verifier to conclude that a program which is constant-resource in practice is not. Our approach mitigates this issue since imprecise information-flow tracking does not directly lead to rejections of secure programs but only increases the burden on constant-resource analysis. Another potential mitigation that applies in some cases is to simply prove that the program is constant-resource with respect to all variables. Another approach that we leave to future work is to incorporate declassification mechanisms into our system.

IX. Conclusion

We have introduced new sub-structural type systems for automatically deriving lower bounds and proving constant resource usage. The evaluation with the implementation in RAML shows that the technique extends beyond the core language that we study in this paper and works for realistic example programs. We have shown how the new type systems can interact with information-flow type systems to prove resource-aware noninterference. Moreover, the type system for constant resource can be used to automatically remove side-channel vulnerabilities from programs.

There are many interesting connections between security and (automatic) quantitative resource analysis that we plan to study in the future. Two concrete projects that we already started are the integration of the type systems for upper and lower bounds with information-flow type systems to precisely quantify the resource-based information leakage at certain security levels. Another direction is to more precisely characterize the amount of information that can be obtained about secrets by making one particular resource-usage observation.

Acknowledgments

This article is based on research that has been supported, in part, by AFRL under DARPA STAC award FA8750-15-C-0082 and DARPA Brandeis award FA8750-15-2-028, by NSF under grant 1319671 (VeriQ), and by a Google Research Award. Any opinions, findings, and conclusions contained in this document are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

References

[1] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *16th Annual International Cryptology Conference (CRYPTO'96)*, 1996.

[2] M. K. Reiter, “Side channels in multi-tenant environments,” in *Proceedings of the 2015 ACM Workshop on Cloud Computing Security Workshop*, 2015.

[3] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, “Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[4] D. Brumley and D. Boneh, “Remote timing attacks are practical,” in *Proceedings of the 12th Annual USENIX Security Symposium*, (Berkeley, CA, USA), USENIX Association, 2003.

[5] B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux, *Password Interception in a SSL/TLS Channel*. 2003.

[6] N. J. AlFardan and K. G. Paterson, “Lucky thirteen: Breaking the TLS and DTLS record protocols,” in *2013 IEEE Symposium on Security and Privacy*, 2013.

[7] D. Gullasch, E. Bangerter, and S. Krenn, “Cache games – bringing access-based cache attacks on aes to practice,” in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, 2011.

[8] A. Haeberlen, B. C. Pierce, and A. Narayan, “Differential privacy under fire,” in *Proceedings of the 20th Annual USENIX Security Symposium*, 2011.

[9] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, “On subnormal floating point and abnormal timing,” in *2015 IEEE Symposium on Security and Privacy*, 2015.

[10] E. W. Felten and M. A. Schneider, “Timing attacks on web privacy,” in *Proceedings of the 7th ACM Conference on Computer and Communications Security*, 2000.

[11] A. Bortz and D. Boneh, “Exposing private information by timing web applications,” in *Proceedings of the 16th International Conference on World Wide Web*, 2007.

[12] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-tenant side-channel attacks in paas clouds,” in *Proceedings of the 2014 ACM Conference on Computer and Communications Security*, 2014.

[13] J. C. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, “Verifying constant-time implementations,” in *25th USENIX Security Symposium*, (Austin, TX), Aug. 2016.

[14] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha, “Sparse representation of implicit flows with applications to side-channel detection,” in *Proceedings of the 25th International Conference on Compiler Construction*, 2016.

[15] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie, “System-level non-interference for constant-time cryptography,” in *Proceedings of the 2014 ACM Conference on Computer and Communications Security*, 2014.

[16] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, “The program counter security model: Automatic detection and removal of control-flow side channel attacks,” in *Proceedings of the 8th International Conference on Information Security and Cryptology*, 2006.

[17] A. Askarov, D. Zhang, and A. C. Myers, “Predictive black-box mitigation of timing channels,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[18] B. Köpf and M. Dürmuth, “A provably secure and efficient countermeasure against timing attacks,” in *Proceedings of the 22nd Annual IEEE Computer Security Foundations Symposium*, July 2009.

[19] M. Hofmann and S. Jost, “Static Prediction of Heap Space Usage for First-Order Functional Programs,” in *30th ACM Symp. on Principles of Prog. Langs. (POPL’03)*, 2003.

[20] J. Hoffmann, K. Aehlig, and M. Hofmann, “Multivariate Amortized Resource Analysis,” in *38th ACM Symp. on Principles of Prog. Langs. (POPL’11)*, 2011.

[21] J. Hoffmann, A. Das, and S.-C. Weng, “Towards Automatic Resource Bound Analysis for OCaml,” in *44th Symposium on Principles of Programming Languages (POPL’17)*, 2017.

[22] J. Goguen and J. Meseguer, “Security Policies and Security Models,” in *In Proceedings of the 1982 IEEE Symposium on Security and Privacy*, 1982.

[23] J. McLean, “Security models and information flow,” in *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, May 1990.

[24] A. C. Myers and B. Liskov, “Complete, safe information flow with decentralized labels,” in *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, May 1998.

[25] A. C. Myers and B. Liskov, “Protecting privacy using the decentralized label model,” *ACM Trans. Softw. Eng. Methodol.*, vol. 9, pp. 410–442, Oct. 2000.

[26] P. Li and S. Zdancewic, “Downgrading policies and relaxed noninterference,” in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’05, (New York, NY, USA), pp. 158–170, ACM, 2005.

[27] R. Harper, *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

[28] Anonymous, “Verifying and Synthesizing Constant-Resource Implementations with Types,” Tech. Rep. TR-??-???, Removed for double blind reviewing. Available uppon request., 2016.

[29] D. J. Bernstein, “Cache-timing Attacks on AES.” http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[30] C. Percival, “Cache missing for fun and profit,” in *Proceedings of BSDCan*, 2005.

[31] V. Simonet, “Flow Caml.” http://cristal.inria.fr/~simonet/soft/flowcaml/, 2003.

[32] F. Pottier and V. Simonet, “Information Flow Inference for ML,” in *In Proceedings of the 29th ACM Symposium on Principles Of Programming Languages*, 2002.

[33] N. Heintze and J. G. Riecke, “The SLam calculus: Programming with secrecy and integrity,” in *In Conference Record of the 25th ACM Symposium on Principles of Programming Languages*, 1998.

[34] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," in *Journal of Computer Security*, pp. 167–187, 1996.

[35] R. E. Tarjan, "Amortized Computational Complexity," *SIAM J. Algebraic Discrete Methods*, vol. 6, no. 2, pp. 306–318, 1985.

[36] J. Hoffmann, K. Aehlig, and M. Hofmann, "Multivariate Amortized Resource Analysis," *ACM Trans. Program. Lang. Syst.*, 2012.

[37] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann, "Static Determination of Quantitative Resource Usage for Higher-Order Programs," in *37th ACM Symp. on Principles of Prog. Langs. (POPL'10)*, 2010.

[38] S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann, "Carbon Credits for Resource-Bounded Computations using Amortised Analysis," in *16th Symp. on Form. Meth. (FM'09)*, 2009.

[39] D. Walker, "Substructural Type Systems," *Advanced Topics in Types and Programming Languages. MIT Press*, pp. 3–43, 2002.

[40] D. Zhang, A. Askarov, and A. C. Myers, "Language-Based Control and Mitigation of Timing Channels," in *In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[41] B. Köpf, L. Mauborgne, and M. Ochoa, "Automatic quantification of cache side-channels," in *Proceedings of the 24th International Conference on Computer Aided Verification*, 2012.

[42] B. Köpf and D. Basin, "An information-theoretic model for adaptive side-channel attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[43] J. Hoffmann, "RAML Web Site," 2016. http://raml.co/

[44] M. Sulzmann, M. Müller, and C. Zenger, "Hindley/Milner Style Type Systems in Constraint Form," in *Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science*, 1999.

[45] E. Yarrkov, "Cryptanalysis of XXTEA," 2010.

[46] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, (Berkeley, Calif.), pp. 281–297, University of California Press, 1967.

[47] J. Hoffmann and M. Hofmann, "Amortized Resource Analysis with Polynomial Potential," in *19th Euro. Symp. on Prog. (ESOP'10)*, 2010.

[48] J. Hoffmann and Z. Shao, "Type-Based Amortized Resource Analysis with Integers and Arrays," in *12th International Symposium on Functional and Logic Programming (FLOPS'14)*, 2014.

[49] H. R. Simões, P. B. Vasconcelos, M. Florido, S. Jost, and K. Hammond, "Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs," in *17th Int. Conf. on Funct. Prog. (ICFP'12)*, 2012.

[50] P. B. Vasconcelos, S. Jost, M. Florido, and K. Hammond, "Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages," in *24th European Symposium on Programming (ESOP'15)*, 2015.

[51] M. Hofmann and S. Jost, "Type-Based Amortised Heap-Space Analysis," in *15th Euro. Symp. on Prog. (ESOP'06)*, 2006.

[52] M. Hofmann and D. Rodriguez, "Automatic Type Inference for Amortised Heap-Space Analysis," in *22nd Euro. Symp. on Prog. (ESOP'13)*, 2013.

[53] R. Atkey, "Amortised Resource Analysis with Separation Logic," in *19th Euro. Symp. on Prog. (ESOP'10)*, 2010.

[54] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao, "End-to-End Verification of Stack-Space Bounds for C Programs," in *Conf. on Prog. Lang. Design and Impl. (PLDI'14)*, p. 30, 2014.

[55] Q. Carbonneaux, J. Hoffmann, and Z. Shao, "Compositional Certified Resource Bounds," in *36th Conf. on Prog. Lang. Design and Impl. (PLDI'15)*, 2015.

[56] T. Nipkow, "Amortized Complexity Verified," in *Interactive Theorem Proving - 6th International Conference (ITP'15)*, 2015.

[57] A. Charguéraud and F. Pottier, "Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation," in *Interactive Theorem Proving - 6th International Conference (ITP'15)*, 2015.

[58] B. Campbell, "Amortised Memory Analysis using the Depth of Data Structures," in *18th Euro. Symp. on Prog. (ESOP'09)*, 2009.

[59] M. Hofmann and G. Moser, "Amortised Resource Analysis and Typed Polynomial Interpretations," in *Rewriting and Typed Lambda Calculi (RTA-TLCA;14)*, 2014.

[60] M. Hofmann and G. Moser, "Multivariate Amortised Resource Analysis for Term Rewrite Systems," in *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, 2015.

[61] P. B. Vasconcelos and K. Hammond, "Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs," in *Int. Workshop on Impl. of Funct. Langs. (IFL'03)*, 2003.

[62] P. Vasconcelos, *Space Cost Analysis Using Sized Types*. PhD thesis, School of Computer Science, University of St Andrews, 2008.

[63] U. D. Lago and M. Gaboardi, "Linear Dependent Types and Relative Completeness," in *26th IEEE Symp. on Logic in Computer Science (LICS'11)*, 2011.

[64] U. D. Lago and B. Petit, "The Geometry of Types," in *40th ACM Symp. on Principles Prog. Langs. (POPL'13)*, 2013.

[65] K. Crary and S. Weirich, "Resource Bound Certification," in *27th ACM Symp. on Principles of Prog. Langs. (POPL'00)*, pp. 184–198, 2000.

[66] N. A. Danielsson, "Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures," in *35th ACM Symp. on Principles Prog. Langs. (POPL'08)*, 2008.

[67] E. Çiçek, D. Garg, and U. A. Acar, "Refinement Types for Incremental Computational Complexity," in *24th European Symposium on Programming (ESOP'15)*, 2015.

[68] B. Grobauer, "Cost Recurrences for DML Programs," in *6th Int. Conf. on Funct. Prog. (ICFP'01)*, pp. 253–264, 2001.

[69] R. Benzinger, "Automated Higher-Order Complexity Analysis," *Theor. Comput. Sci.*, vol. 318, no. 1-2, pp. 79–103, 2004.

[70] N. Danner, D. R. Licata, and R. Ramyaa, "Denotational Cost Semantics for Functional Languages with Inductive Types," in *29th Int. Conf. on Functional Programming (ICFP'15)*, 2012.

[71] M. Avanzini, U. D. Lago, and G. Moser, "Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order," in *29th Int. Conf. on Functional Programming (ICFP'15)*, 2012.

[72] D. E. Alonso-Blas and S. Genaim, "On the limits of the classical approach to cost analysis," in *19th Int. Static Analysis Symp. (SAS'12)*, 2012.

[73] A. Flores-Montoya and R. Hähnle, "Resource Analysis of Complex Programs with Cost Equations," in *Programming Languages and Systems - 12th Asian Symposiu (APLAS'14)*, 2014.

[74] E. Albert, J. C. Fernández, and G. Román-Díez, "Non-cumulative Resource Analysis," in *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, (TACAS'15)*, 2015.

[75] S. Gulwani, K. K. Mehra, and T. M. Chilimbi, "SPEED: Precise and Efficient Static Estimation of Program Computational Complexity," in *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, 2009.

[76] R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács, "ABC: Algebraic Bound Computation for Loops," in *Logic for Prog., AI., and Reasoning - 16th Int. Conf. (LPAR'10)*, 2010.

[77] F. Zuleger, M. Sinn, S. Gulwani, and H. Veith, "Bound Analysis of Imperative Programs with the Size-change Abstraction," in *18th Int. Static Analysis Symp. (SAS'11)*, 2011.

[78] M. Sinn, F. Zuleger, and H. Veith, "A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis," in *Computer Aided Verification - 26th Int. Conf. (CAV'14)*, 2014.

[79] P. Cerný, T. A. Henzinger, L. Kovács, A. Radhakrishna, and J. Zwirchmayr, "Segment Abstraction for Worst-Case Execution Time Analysis," in *24th European Symposium on Programming (ESOP'15)*, 2015.

[80] E. Albert, S. Genaim, and A. N. Masud, "On the Inference of Resource Usage Upper and Lower Bounds," *ACM Transactions on Computational Logic*, vol. 14, no. 3, 2013.

[81] M. Avanzini and G. Moser, "A Combination Framework for Complexity," in *24th International Conference on Rewriting Techniques and Applications (RTA'13)*, 2013.

[82] L. Noschinski, F. Emmes, and J. Giesl, "Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs," *J. Autom. Reasoning*, vol. 51, no. 1, pp. 27–56, 2013.

[83] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl, "Alternating Runtime and Size Complexity Analysis of Integer Programs," in *20th Int. Conf. on Tools and Alg. for the Constr. and Anal. of Systems (TACAS'14)*, 2014.

[84] F. Frohn, M. Naaf, J. Hensel, M. Brockschmidt, and J. Giesl, "Lower Runtime Bounds for Integer Programs," in *Automated Reasoning - 8th International Joint Conference (IJCAR'16)*, 2016.

[85] J. Agat, "Transforming out timing leaks," in *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, 2000.

[86] D. Hedin and D. Sands, "Timing aware information flow security for a javacard-like bytecode," *Electron. Notes Theor. Comput. Sci.*, vol. 141, Dec. 2005.

[87] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, 2009.

[88] G. Barthe, T. Rezk, and M. Warnier, "Preventing timing leaks through transactional branching instructions," *Electron. Notes Theor. Comput. Sci.*, vol. 153, May 2006.

[89] D. Zhang, A. Askarov, and A. C. Myers, "Predictive mitigation of timing channels in interactive systems," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[90] P. Mardziel, M. S. Alvim, M. Hicks, and M. R. Clarkson, "Quantifying information flow for dynamic secrets," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[91] C. Pasareanu, Q.-S. Phan, and P. Malacaria, "Multi-run side-channel analysis using symbolic execution and max-smt," in *Proceedings of the 29th IEEE Computer Security Foundations Symposium*, 2016.

[92] G. Doychev, D. Feld, B. Kopf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," in *Proceedings of the 22nd USENIX Security Symposium*, (Washington, D.C.), USENIX, 2013.

[93] A. Sabelfeld and A. C. Myers, "Language-Based Information Flow Security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[94] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *Computer Networks*, 2005.

[95] D. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Topics in Cryptography*, 2006.

APPENDIX

THE LANGUAGE SEMANTICS

The equivalent expressions in OCaml syntax of the language are given as follows.

$$
\begin{aligned}
e \;\; ::= \;& () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid x \\
\mid\;& x_1 \diamond x_2 \\
\mid\;& f(x) \\
\mid\;& \text{let } x = e_1 \text{ in } e_2 \\
\mid\;& \text{if } x \text{ then } e_t \text{ else } e_f \\
\mid\;& (x_1, x_2) \\
\mid\;& \text{match } x \text{ with } (x_1, x_2) \to e \\
\mid\;& [] \\
\mid\;& x_1 :: x_2 \\
\mid\;& \text{match } x \text{ with } \mid [] \to e_1 \mid x_1 :: x_2 \to e_2 \\
\mid\;& \text{share } x = (x_1, x_2) \text{ in } e
\end{aligned}
$$

$$\diamond \in \{+, -, *, \mathsf{div}, \mathsf{mod}, =, <>, >, <, <=, >=, \mathsf{and}, \mathsf{or}\}$$

Fig. 11 and Fig. 12 represent the typing rules for values, the base typing and the evaluation rules for the language, respectively.

TYPING RULES AND PROOFS OF LEMMA 2, LEMMA 3, AND THEOREM 1

Typing rules

The full typing rules of the resource-aware security type system are presented in Fig. 14.

Proof of Lemma 2

The proof is done by induction on the structure of the typing derivation.

SR:UNIT*:* There is no variable thus it follows immediately.

SR:BOOL*:* It is similar to the case SR:UNIT.

SR:INT*:* It is similar to the case SR:UNIT.

SR:VAR*:* Since $\Gamma^s(x) = S$, if $S \blacktriangleleft k_1$ then $\Gamma^s(x) \blacktriangleleft k_1$.

SR:B-OP*:* If $(\text{bool}, k_{x_1} \sqcup k_{x_2}) \blacktriangleleft k_1$ then $\Gamma^s(x_1) = (\text{bool}, k_{x_1}) \blacktriangleleft k_1$ and $\Gamma^s(x_2) = (\text{bool}, k_{x_2}) \blacktriangleleft k_1$.

SR:IB-OP*:* It is similar to the case SR:B-OP.

SR:I-OP*:* It is similar to the case SR:B-OP.

SR:GEN - SR:C-GEN*:* By induction for e in the premise, it follows.

SR:FUN*:* Because e is well-formed program, there exists a well-typed expression e_f such that pc'$;\Sigma^s;\Gamma^s \vdash e_f : S_2$. By induction for e_f, for all variables x in e, if $S \blacktriangleleft k_1$ then $\Gamma^s(x) \blacktriangleleft k_1$. It is similar for SR:L-ARG and SR:C-FUN.

S:LET*:* If $S_2 \blacktriangleleft k_1$ then by induction for e_2, $S_1 \blacktriangleleft k_1$. Thus for all variable x in e, it is a variable in e_1 or e_2. By induction for e_1 and e_2, it follows. It is similar for SR:L-LET.

SR:IF*:* If $S \blacktriangleleft k_1$ then by the hypothesis $(\text{bool}, k_x) \blacktriangleleft k_1$. For all variable y in e, it is a variable in e_t or e_f. By induction for e_t and e_f, it follows. It is similar for SR:L-IF.

SR:PAIR*:* If $S_1 * S_2 \blacktriangleleft k_1$ then $\Gamma^s(x_1) = S_1 \blacktriangleleft k_1$ and $\Gamma^s(x_2) = S_2 \blacktriangleleft k_1$.

SR:MATCH-P*:* If $S \blacktriangleleft k_1$ then by induction for e, $\Gamma^s(x_1) \blacktriangleleft k_1$ and $\Gamma^s(x_2) \blacktriangleleft k_1$. Thus $\Gamma^s(x) \blacktriangleleft k_1$. For all other variables y in e, again by induction for e, if $S \blacktriangleleft k_1$ then $\Gamma^s(y) \blacktriangleleft k_1$. It is similar for SR:C-MATCH-P.

SR:NIL*:* It is similar to the case SR:UNIT.

SR:CONS*:* If $(L(S), k_x) \blacktriangleleft k_1$ then $\Gamma^s(x_h) = S \blacktriangleleft k_1$ and $\Gamma^s(x_t) = (L(S), k_x) \blacktriangleleft k_1$.

SR:MATCH-L*:* If $S_1 \blacktriangleleft k_1$ then by induction for e_2, $\Gamma^s(x_h) = S \blacktriangleleft k_1$ and $\Gamma^s(x_t) = (L(S), k_x) \blacktriangleleft k_1$. Thus $\Gamma^s(x) \blacktriangleleft k_1$. For all other variables y in e, y is a variable in e_1 or e_2. Again by induction for e_1 and e_2, if $S_1 \blacktriangleleft k_1$ then $\Gamma^s(y) \blacktriangleleft k_1$.

SR:SUBTYPING*:* By the subtyping relation, if $S' \blacktriangleleft k_1$ then $S \blacktriangleleft k_1$. Thus by induction for e in the premise, for all variables x in e, if $S \blacktriangleleft k_1$ then $\Gamma^s(x) \blacktriangleleft k_1$. It is similar for SR:C-SUBTYPING.

Proof of Lemma 3

The proof is done by induction on the structure of the evaluation derivation and the typing derivation.

SR:UNIT*:* Suppose the evaluation derivation of e ends with an application of the rule E:UNIT, thus $E_1 \vdash e \Downarrow ()$ and $E_2 \vdash e \Downarrow ()$. Hence, it follows.

SR:BOOL*:* It is similar to the case SR:UNIT.

SR:INT*:* It is similar to the case SR:UNIT.

SR:VAR*:* Suppose the evaluation derivation ends with an application of the rule E:VAR, thus $E_1(x) = v_1$ and $E_2(x) = v_2$. The typing derivation ends with an application of the rule SR:VAR, thus $\Gamma^s(x) = S$. If $S \blacktriangleleft k_1$, by the hypothesis $E_1(x) = E_2(x)$ since $x \in \text{dom}(E_i), i = \{1, 2\}$.

SR:B-OP*:* Suppose the evaluation derivation ends with an application of the rule E:BIN, thus $E_1(x_1) \diamond E_1(x_2) = v_1$ and $E_2(x_1) \diamond E_2(x_2) = v_2$. The typing derivation ends with an application of the rules SR:B-OP or SR:GEN. We have $k_{x_1} \lhd S$ and $k_{x_2} \lhd S$. If $S \blacktriangleleft k_1$ then $k_{x_1} \sqsubseteq k_1$ and $k_{x_2} \sqsubseteq k_1$. By the hypothesis, we have $E_1(x_1) = E_2(x_1)$ and $E_1(x_2) = E_2(x_2)$, thus $v_1 = v_2$.

SR:IB-OP*:* It is similar to the case SR:B-OP.

SR:I-OP*:* It is similar to the case SR:B-OP.

SR:GEN-SR:C-GEN*:* By induction for e in the premise, it follows that if $S \blacktriangleleft k_1$ then $v_1 = v_2$.

SR:FUN*:* Suppose the evaluation derivation ends with an application of the rule E:FUN, thus $\Sigma(g) = T_1 \to T_2$ and $[y^g \mapsto E_i(x)] \vdash e_g \Downarrow v_i$ for $i = \{1, 2\}$. The typing derivation ends with an application of the following rules.

- Case SR:FUN. Because e is well-formed program, there exists a well-typed expression e_f such that pc'$;\Sigma^s;\Gamma^s \vdash e_f : S_2$ and $e_{\hat{f}} = e_g$. By induction for e_f, if $S_2 \blacktriangleleft k_1$ then $v_1 = v_2$.
- Case SR:L-ARG. It is similar to the case SR:FUN.
- Case SR:C-FUN. It is similar to the case SR:FUN.
- Case SR:GEN and SR:C-GEN. It follows.

$$\frac{v = ()}{\models v : \text{unit}}\ (\text{V:UNIT}) \qquad \frac{v \in \{\ \mathsf{true}\ ,\ \mathsf{false}\ \}}{\models v : \text{bool}}\ (\text{V:BOOL}) \qquad \frac{v \in \mathbb{Z}}{\models v : \text{int}}\ (\text{V:INT}) \qquad \frac{\models v_1 : T_1 \qquad \models v_2 : T_2}{\models (v_1, v_2) : T_1 * T_2}\ (\text{V:PAIR}) \qquad \frac{v = \ \mathsf{nil}}{\models v : L(T)}\ (\text{V:NIL}) \qquad \frac{\models v_i : T \qquad \forall i = 1, ..., n}{\models [v_1, ..., v_n] : L(T)}\ (\text{V:LIST})$$

$$\frac{}{\Sigma; \emptyset \vdash () : \text{unit}}\ (\text{T:UNIT}) \qquad \frac{b \in \{\ \mathsf{true}\ ,\ \mathsf{false}\ \}}{\Sigma; \emptyset \vdash b : \text{bool}}\ (\text{T:BOOL}) \qquad \frac{n \in \mathbb{Z}}{\Sigma; \emptyset \vdash n : \text{int}}\ (\text{T:INT}) \qquad \frac{x \in \mathsf{dom}(E)}{\Sigma; x : T \vdash x : T}\ (\text{T:VAR}) \qquad \frac{\diamond \in \{\ \mathsf{and}\ ,\ \mathsf{or}\ \}}{\Sigma; x_1 : \text{bool}, x_2 : \text{bool} \vdash \text{op}_\diamond(x_1, x_2) : \text{bool}}\ (\text{T:B-OP})$$

$$\frac{\diamond \in \{=, <>, >, <, <=, >=\}}{\Sigma; x_1 : \text{int}, x_2 : \text{int} \vdash \text{op}_\diamond(x_1, x_2) : \text{bool}}\ (\text{T:IB-OP}) \qquad \frac{\diamond \in \{+, -, *,\ \mathsf{div}\ ,\ \mathsf{mod}\ \}}{\Sigma; x_1 : \text{int}, x_2 : \text{int} \vdash \text{op}_\diamond(x_1, x_2) : \text{int}}\ (\text{T:I-OP}) \qquad \frac{\Sigma(g) = T_1 \to T_2}{\Sigma; x : T_1 \vdash \mathsf{app}(g, x) : T_2}\ (\text{T:FUN})$$

$$\frac{\Sigma; \Gamma_1 \vdash e_1 : T_1 \qquad \Sigma; \Gamma_2, x : T_1 \vdash e_2 : T_2}{\Sigma; \Gamma_1, \Gamma_2 \vdash \mathsf{let}(x, e_1, x.e_2) : T_2}\ (\text{T:LET}) \qquad \frac{\Sigma; \Gamma \vdash e_t : T \qquad \Sigma; \Gamma \vdash e_f : T}{\Sigma; \Gamma, x : \text{bool} \vdash \mathsf{if}(x, e_t, e_f) : T}\ (\text{T:IF}) \qquad \frac{}{\Sigma; x_1 : T_1, x_2 : T_2 \vdash \mathsf{pair}(x_1, x_2) : T_1 * T_2}\ (\text{T:PAIR})$$

$$\frac{\Sigma; \Gamma, x_1 : T_1, x_2 : T_2 \vdash e : T}{\Sigma; \Gamma, x : T_1 * T_2 \vdash \mathsf{match}(x, (x_1, x_2).e) : T}\ (\text{T:MATCH-P}) \qquad \frac{T \in \mathscr{T}}{\Sigma; \emptyset \vdash\ \mathsf{nil}\ : L(T)}\ (\text{T:NIL}) \qquad \frac{}{\Sigma; x_h : T, x_t : L(T) \vdash \mathsf{cons}(x_h, x_t) : L(T)}\ (\text{T:CONS})$$

$$\frac{\Sigma; \Gamma \vdash e_1 : T_1 \qquad \Sigma; \Gamma, x_h : T, x_t : L(T) \vdash e_2 : T_1}{\Sigma; \Gamma, x : L(T) \vdash \mathsf{match}(x, e_1, (x_h, x_t).e_2) : T_1}\ (\text{T:MATCH-L}) \qquad \frac{\Sigma; \Gamma, x_1 : T, x_2 : T \vdash e : T_1}{\Sigma; \Gamma, x : T \vdash \mathsf{share}(x, (x_1, x_2).e) : T_1}\ (\text{T:SHARE}) \qquad \frac{\Sigma; \Gamma \vdash e : T_1}{\Sigma; \Gamma, x : T \vdash e : T_1}\ (\text{T:WEAKENING})$$

Figure 11. Typing rules for values and base types

S:LET: Suppose the evaluation derivation ends with an application of the rule E:LET, thus $E_i \vdash e_1 \Downarrow v_1^i$ and $E_i[x \mapsto v_1^i] \vdash e_2 \Downarrow v_i$ for $i = \{1, 2\}$. The typing derivation ends with an application of the following rules.

- Case SR:L-LET. If $S_2 \blacktriangleleft k_1$, by the simple security lemma, it holds that $S_1 \blacktriangleleft k_1$. By induction for e_1, we have $v_1^1 = v_1^2$, so $E_1[x \mapsto v_1^1] \equiv_k E_2[x \mapsto v_1^2]$. Again by induction for e_2, we have $v_1 = v_2$.
- Case SR:LET. It is similar to the case SR:L-LET.
- Case SR:GEN and SR:C-GEN. It follows.

SR:IF: Suppose e is of the form $\mathsf{if}(x, e_t, e_f)$, the evaluation derivation ends with an application of the rule E:IF-TRUE or the rule E:IF-FALSE. The typing derivation ends with an application of the following rules.

- Case SR:L-IF. By the hypothesis we have $k_x \sqsubseteq k_1$, thus $E_1(x) = E_2(x)$. Assume that $E_1(x) =\ \mathsf{true}$, then $E_1 \vdash e_t \Downarrow v_1$ and $E_2 \vdash e_t \Downarrow v_2$. By induction for e_t we have $v_1 = v_2$ if $S \blacktriangleleft k_1$. It is similar for $E_1(x) =\ \mathsf{false}$.
- Case SR:IF. If $k_x \sqsubseteq k_1$ the proof is similar to the case SR:L-IF. Otherwise, $k_x \not\sqsubseteq k_1$, thus by the simple security lemma we have $S \not\blacktriangleleft k_1$.
- Case SR:GEN and SR:C-GEN. It follows.

SR:PAIR: Suppose the evaluation derivation ends with an application of the rule E:PAIR, thus $(E_i(x_1), E_i(x_2)) = v_i$ for $i = \{1, 2\}$. The typing derivation ends with an application of the rules SR:PAIR or SR:GEN.

If $S_1 * S_2 \blacktriangleleft k$, then by the simple security lemma we have $S_1 \blacktriangleleft k_1$ and $S_2 \blacktriangleleft k_1$. Hence it follows $v_1 = v_2$.

SR:MATCH-P: Suppose the evaluation derivation ends with an application of the rule E:MATCH-P, thus $E_i(x) = (v_1^i, v_2^i)$ and $E_i[x_1 \mapsto v_1^i, x_2 \mapsto v_2^i] \vdash e \Downarrow v_i$ for $i = \{1, 2\}$. The typing derivation ends with an application of the following rules.

- Case SR:MATCH-P. If $S \blacktriangleleft k_1$, then by the simple security lemma we have $S_1 * S_2 \blacktriangleleft k_1$. By the hypothesis, $E_1(x) = E_2(x)$, thus $v_1^1 = v_1^2$ and $v_2^1 = v_2^2$. Hence, $E_1[x_1 \mapsto v_1^1, x_2 \mapsto v_2^1] \equiv_k E_2[x_1 \mapsto v_1^2, x_2 \mapsto v_2^2]$, by induction for e in the premise, it holds that $v_1 = v_2$.
- Case SR:C-MATCH-P. It is similar to the case SR:MATCH-P.
- Case SR:GEN and SR:C-GEN. It follows.

SR:NIL: It is similar to the case SR:UNIT.

SR:CONS: Suppose the evaluation derivation ends with an application of the rule E:CONS, thus $E_i(x_h) = v_1^i$ and $E_i(x_t) = [v_2^i, \cdots, v_n^i]$ for $i = \{1, 2\}$. The typing derivation ends with an application of the rules SR:CONS or SR:GEN. If $(L(S), k_x) \blacktriangleleft k_1$ then by the hypothesis we have $v_1^1 = v_1^2$ and $[v_2^1, \cdots, v_n^1] = [v_2^2, \cdots, v_n^2]$. Thus $E_1(\ \mathsf{cons}\,(x_h, x_t)) = E_2(\ \mathsf{cons}\,(x_h, x_t))$.

SR:MATCH-L: Suppose e is of the form $\mathsf{match}(x, e_1, (x_h, x_t).e_2)$, the evaluation derivation ends with an application of the rule E:MATCH-N or the rule E:MATCH-L. The typing derivation ends with an application of the following rules.

- Case SR:MATCH-L. If $S_1 \blacktriangleleft k_1$, then by the simple security lemma we have $(L(S), k_x) \blacktriangleleft k_1$. By the

$$\text{(E:UNIT)}\ \frac{}{E \vdash^{q+K^{\mathsf{unit}}}_{q} () \Downarrow ()} \qquad \text{(E:BOOL)}\ \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{E \vdash^{q+K^{\mathsf{bool}}}_{q} b \Downarrow b} \qquad \text{(E:INT)}\ \frac{n \in \mathbb{Z}}{E \vdash^{q+K^{\mathsf{int}}}_{q} n \Downarrow n}$$

$$\text{(E:VAR)}\ \frac{x \in \mathrm{dom}(E)}{E \vdash^{q+K^{\mathsf{var}}}_{q} x \Downarrow E(x)} \qquad \text{(E:BIN)}\ \frac{v = E(x_1) \diamond E(x_2)}{E \vdash^{q+K^{\mathsf{op}}}_{q} \mathrm{op}_{\diamond}(x_1, x_2) \Downarrow v}$$

$$\text{(E:FUN)}\ \frac{\Sigma(g) = T_1 \to T_2 \qquad E[x^g \mapsto E(x)] \vdash^{q}_{q'} e_g \Downarrow v}{E \vdash^{q+K^{\mathsf{app}}}_{q'} \mathrm{app}(g, x) \Downarrow v} \qquad \text{(E:MATCH-N)}\ \frac{E(x) = \mathsf{nil} \qquad E \vdash^{q-K^{\mathsf{matchN}}}_{q'} e_1 \Downarrow v}{E \vdash^{q}_{q'} \mathrm{match}(x, e_1, (x_h, x_t).e_2) \Downarrow v}$$

$$\text{(E:NIL)}\ \frac{}{E \vdash^{q+K^{\mathsf{nil}}}_{q} \mathsf{nil} \Downarrow \mathsf{nil}}$$

$$\text{(E:IF-TRUE)}\ \frac{E(x) = \mathsf{true} \qquad E \vdash^{q-K^{\mathsf{cond}}}_{q'} e_t \Downarrow v}{E \vdash^{q}_{q'} \mathrm{if}(x, e_t, e_f) \Downarrow v} \qquad \text{(E:IF-FALSE)}\ \frac{E(x) = \mathsf{false} \qquad E \vdash^{q-K^{\mathsf{cond}}}_{q'} e_f \Downarrow v}{E \vdash^{q}_{q'} \mathrm{if}(x, e_t, e_f) \Downarrow v}$$

$$\text{(E:PAIR)}\ \frac{x_1, x_2 \in \mathrm{dom}(E) \qquad v = (E(x_1), E(x_2))}{E \vdash^{q+K^{\mathsf{pair}}}_{q} \mathrm{pair}(x_1, x_2) \Downarrow v}$$

$$\text{(E:MATCH-P)}\ \frac{E(x) = (v_1, v_2) \qquad E[x_1 \mapsto v_1, x_2 \mapsto v_2] \vdash^{q-K^{\mathsf{matchP}}}_{q'} e \Downarrow v}{E \vdash^{q}_{q'} \mathrm{match}(x, (x_1, x_2).e) \Downarrow v}$$

$$\text{(E:CONS)}\ \frac{x_h, x_t \in \mathrm{dom}(E) \qquad E(x_h) = v_1 \qquad E(x_t) = [v_2, .., v_n]}{E \vdash^{q+K^{\mathsf{cons}}}_{q} \mathrm{cons}(x_h, x_t) \Downarrow [v_1, ..., v_n]}$$

$$\text{(E:LET)}\ \frac{E \vdash^{q-K^{\mathsf{let}}}_{q_1'} e_1 \Downarrow v_1 \qquad E[x \mapsto v_1] \vdash^{q_1'}_{q'} e_2 \Downarrow v}{E \vdash^{q}_{q'} \mathrm{let}(x, e_1, x.e_2) \Downarrow v} \qquad \text{(E:SHARE)}\ \frac{E(x) = v_1 \qquad E[x_1 \mapsto v_1, x_2 \mapsto v_1] \setminus \{x\} \vdash^{q}_{q'} e \Downarrow v}{E \vdash^{q}_{q'} \mathrm{share}(x, (x_1, x_2).e) \Downarrow v}$$

$$\text{(E:MATCH-L)}\ \frac{E(x) = [v_1, ..., v_n] \qquad E[x_h \mapsto v_1, x_t \mapsto [v_2, ..., v_n]] \vdash^{q-K^{\mathsf{matchL}}}_{q'} e_2 \Downarrow v}{E \vdash^{q}_{q'} \mathrm{match}(x, e_1, (x_h, x_t).e_2) \Downarrow v}$$

Figure 12. Evaluation typing rules

$$\frac{k \sqsubseteq k' \qquad T \in \{\mathrm{unit}, \mathrm{int}, \mathrm{bool}\}}{k \lhd (T, k')} \qquad \frac{k \sqsubseteq k' \qquad k \lhd S}{k \lhd (L(S), k')} \qquad \frac{k \lhd S_1 \qquad k \lhd S_2}{k \lhd S_1 * S_2} \qquad \frac{k' \sqsubseteq k \qquad T \in \{\mathrm{unit}, \mathrm{int}, \mathrm{bool}\}}{(T, k') \blacktriangleleft k}$$

$$\frac{k' \sqsubseteq k \qquad S \blacktriangleleft k}{(L(S), k') \blacktriangleleft k} \qquad \frac{S_1 \blacktriangleleft k \qquad S_2 \blacktriangleleft k}{S_1 * S_2 \blacktriangleleft k} \qquad \frac{k \sqsubseteq k' \qquad T \in \{\mathrm{unit}, \mathrm{int}, \mathrm{bool}\}}{(\mathrm{T}, k) \leq (\mathrm{T}, k')} \qquad \frac{k \sqsubseteq k' \qquad S \leq S'}{(L(S), k) \leq (L(S'), k')}$$

$$\frac{S_1 \leq S_1' \qquad S_2 \leq S_2'}{S_1 * S_2 \leq S_1' * S_2'}$$

Figure 13. Guards, collecting, and subtyping relations

hypothesis we have $E_1(x) = E_2(x)$. Assume that $E_1(x) = E_2(x) = [v_1, \cdots, v_n]$, by the rule E:MATCH-L we have $E_i[x_h \mapsto v_1, x_t \mapsto [v_2, ..., v_n]] \vdash e_2 \Downarrow v_i$ for $i = \{1, 2\}$. Since $E_1[x_h \mapsto v_1, x_t \mapsto [v_2, ..., v_n]] \equiv_k E_2[x_h \mapsto v_1, x_t \mapsto [v_2, ..., v_n]]$, by induction for e_2, it holds that $v_1 = v_2$ if $S_1 \blacktriangleleft k_1$. It is similar for $E_1(x) = E_2(x) = \mathsf{nil}$.

- Case SR:C-MATCH-L.It is similar to the case SR:MATCH-L.
- Case SR:GEN and SR:C-GEN. It follows.

SR:SUBTYPING*:* Suppose the typing derivation ends with the rule SR:SUBTYPING. If $S' \blacktriangleleft k_1$ then $S \blacktriangleleft k_1$. Thus by induction for e in the premise it follows. It is similar for SR:C-SUBTYPING.

Proof of Theorem 1

The proof is done by induction on the structure of the typing derivation and the evaluation derivation. Let X be the set of variables $[\Gamma^s]_{\blacktriangleleft k_1}$. For all environments E_1, E_2 such that $E_1 \approx_X E_2$ and $E_1 \equiv_{k_1} E_2$, if $E_1 \vdash^{p_1}_{p_1'} e \Downarrow v_1$ and $E_2 \vdash^{p_2}_{p_2'} e \Downarrow v_2$. We then show that $p_1 - p_1' = p_2 -$

(SR:UNIT)
$$\frac{}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} () : (\mathrm{unit}, \mathrm{pc})}$$

(SR:BOOL)
$$\frac{b \in \{\, \mathsf{true}\,,\ \mathsf{false}\,\}}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} b : (\mathrm{bool}, \mathrm{pc})}$$

(SR:INT)
$$\frac{n \in \mathbb{Z}}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} n : (\mathrm{int}, \mathrm{pc})}$$

(SR:VAR)
$$\frac{x : S \in \Gamma^s \qquad \mathrm{pc} \lhd S}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} x : S}$$

(SR:B-OP)
$$\frac{x_1 : (\mathrm{bool}, k_{x_1}) \in \Gamma^s \qquad x_2 : (\mathrm{bool}, k_{x_2}) \in \Gamma^s \qquad \mathrm{pc} \sqsubseteq k_{x_1} \sqcup k_{x_2} \qquad \diamond \in \{\, \mathsf{and}\,,\ \mathsf{or}\,\}}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} \mathrm{op}_\diamond(x_1, x_2) : (\mathrm{bool}, k_{x_1} \sqcup k_{x_2})}$$

(SR:C-GEN)
$$\frac{\mathrm{pc}; \Sigma^s; \Gamma^s \vdash e : S \qquad \mathrm{const}_X(e)}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} e : S}$$

(SR:GEN)
$$\frac{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} e : S}{\mathrm{pc}; \Sigma^s; \Gamma^s \vdash e : S}$$

(SR:IB-OP)
$$\frac{\begin{array}{c} x_1 : (\mathrm{int}, k_{x_1}) \in \Gamma^s \qquad x_2 : (\mathrm{int}, k_{x_2}) \in \Gamma^s \\ \mathrm{pc} \sqsubseteq k_{x_1} \sqcup k_{x_2} \qquad \diamond \in \{=, <>, >, <, <=, >=\} \end{array}}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} \mathrm{op}_\diamond(x_1, x_2) : (\mathrm{bool}, k_{x_1} \sqcup k_{x_2})}$$

(SR:PAIR)
$$\frac{x_1 : S_1 \in \Gamma^s \qquad x_2 : S_2 \in \Gamma^s \qquad \mathrm{pc} \lhd S_1 * S_2}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} \mathrm{pair}(x_1, x_2) : S_1 * S_2}$$

(SR:I-OP)
$$\frac{\begin{array}{c} x_1 : (\mathrm{int}, k_{x_1}) \in \Gamma^s \qquad x_2 : (\mathrm{int}, k_{x_2}) \in \Gamma^s \\ \mathrm{pc} \sqsubseteq k_{x_1} \sqcup k_{x_2} \qquad \diamond \in \{+, -, *, \mathsf{div}\,,\ \mathsf{mod}\,\} \end{array}}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} \mathrm{op}_\diamond(x_1, x_2) : (\mathrm{int}, k_{x_1} \sqcup k_{x_2})}$$

(SR:CONS)
$$\frac{\begin{array}{c} x_h : S \in \Gamma^s \qquad x_t : (L(S), k_x) \in \Gamma^s \\ \mathrm{pc} \lhd (L(S), k_x) \end{array}}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} \mathrm{cons}(x_h, x_t) : (L(S), k_x)}$$

(SR:SUBTYPING)
$$\frac{\mathrm{pc}; \Sigma^s; \Gamma^s \vdash e : S \qquad S \leq S'}{\mathrm{pc}; \Sigma^s; \Gamma^s \vdash e : S'}$$

(SR:FUN)
$$\frac{x : S_1 \in \Gamma^s \qquad \Sigma^s(f) = S_1 \xrightarrow{\mathrm{pc}'} S_2 \qquad \mathrm{pc} \sqsubseteq \mathrm{pc}'}{\mathrm{pc}; \Sigma^s; \Gamma^s \vdash \mathrm{app}(f, x) : S_2}$$

(SR:L-ARG)
$$\frac{x : S_1 \in \Gamma^s \qquad \Sigma^s(f) = S_1 \xrightarrow{\mathrm{pc}'} S_2 \qquad \mathrm{pc} \sqsubseteq \mathrm{pc}' \qquad S_1 \blacktriangleleft k_1}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} \mathrm{app}(f, x) : S_2}$$

(SR:C-FUN)
$$\frac{x : S_1 \in \Gamma^s \qquad \Sigma^s(f) = S_1 \xrightarrow{\mathrm{pc}'/\mathrm{const}} S_2 \qquad \mathrm{pc} \sqsubseteq \mathrm{pc}'}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} \mathrm{app}(f, x) : S_2}$$

(SR:NIL)
$$\frac{S \in \mathscr{S} \qquad \mathrm{pc} \lhd S}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} \mathsf{nil} : (L(S), \mathrm{pc})}$$

(SR:C-SUBTYPING)
$$\frac{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} e : S \qquad S \leq S'}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} e : S'}$$

(SR:LET)
$$\frac{\mathrm{pc}; \Sigma^s; \Gamma^s \vdash e_1 : S_1 \qquad \mathrm{pc}; \Sigma^s; \Gamma^s, x : S_1 \vdash e_2 : S_2}{\mathrm{pc}; \Sigma^s; \Gamma^s \vdash \mathrm{let}(x, e_1, x.e_2) : S_2}$$

(SR:L-LET)
$$\frac{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} e_1 : S_1 \qquad \mathrm{pc}; \Sigma^s; \Gamma^s, x : S_1 \overset{\text{const}}{\vdash} e_2 : S_2 \qquad S_1 \blacktriangleleft k_1}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} \mathrm{let}(x, e_1, x.e_2) : S_2}$$

(SR:IF)
$$\frac{x : (\mathrm{bool}, k_x) \in \Gamma^s \qquad \mathrm{pc} \sqcup k_x; \Sigma^s; \Gamma^s \vdash e_t : S \qquad \mathrm{pc} \sqcup k_x; \Sigma^s; \Gamma^s \vdash e_f : S \qquad \mathrm{pc} \sqcup k_x \lhd S}{\mathrm{pc}; \Sigma^s; \Gamma^s \vdash \mathrm{if}(x, e_t, e_f) : S}$$

(SR:L-IF)
$$\frac{x : (\mathrm{bool}, k_x) \in \Gamma^s \qquad \mathrm{pc} \sqcup k_x; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} e_t : S \qquad \mathrm{pc} \sqcup k_x; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} e_f : S \qquad \mathrm{pc} \sqcup k_x \lhd S \qquad k_x \sqsubseteq k_1}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} \mathrm{if}(x, e_t, e_f) : S}$$

(SR:MATCH-P)
$$\frac{x : S_1 * S_2 \in \Gamma^s \qquad \mathrm{pc}; \Sigma^s; \Gamma^s, x_1 : S_1, x_2 : S_2 \vdash e : S}{\mathrm{pc}; \Sigma^s; \Gamma^s \vdash \mathrm{match}(x, (x_1, x_2).e) : S}$$

(SR:C-MATCH-P)
$$\frac{x : S_1 * S_2 \in \Gamma^s \qquad \mathrm{pc}; \Sigma^s; \Gamma^s, x_1 : S_1, x_2 : S_2 \overset{\text{const}}{\vdash} e : S}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} \mathrm{match}(x, (x_1, x_2).e) : S}$$

(SR:MATCH-L)
$$\frac{x : (L(S), k_x) \in \Gamma^s \qquad \mathrm{pc} \sqcup k_x; \Sigma^s; \Gamma^s \vdash e_1 : S_1 \qquad \mathrm{pc} \sqcup k_x; \Sigma^s; \Gamma^s, x_h : S, x_t : (L(S), k_x) \vdash e_2 : S_1 \qquad \mathrm{pc} \sqcup k_x \lhd S_1}{\mathrm{pc}; \Sigma^s; \Gamma^s \vdash \mathrm{match}(x, e_1, (x_h, x_t).e_2) : S_1}$$

(SR:C-MATCH-L)
$$\frac{\begin{array}{c} x : (L(S), k_x) \in \Gamma^s \\ \mathrm{pc} \sqcup k_x; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} e_1 : S_1 \qquad \mathrm{pc} \sqcup k_x; \Sigma^s; \Gamma^s, x_h : S, x_t : (L(S), k_x) \overset{\text{const}}{\vdash} e_2 : S_1 \qquad \mathrm{pc} \sqcup k_x \lhd S_1 \qquad k_x \sqsubseteq k_1 \end{array}}{\mathrm{pc}; \Sigma^s; \Gamma^s \overset{\text{const}}{\vdash} \mathrm{match}(x, e_1, (x_h, x_t).e_2) : S_1}$$

Figure 14. Typing rules for resource-aware security type system

p_2' and $v_1 = v_2$ if $S \blacktriangleleft k_1$. By Lemma 3, e satisfies the noninterference property at security label k_1. Thus we need to prove that $p_1 - p_1' = p_2 - p_2'$.

SR:UNIT*:* Suppose the evaluation derivation of e ends with an application of the rule E:UNIT, thus $p_1 - p_1' = p_2 - p_2' = K^{\mathsf{unit}}$.

SR:BOOL*:* It is similar to the case SR:UNIT.

SR:INT*:* It is similar to the case SR:UNIT.

SR:VAR*:* It is similar to the case SR:UNIT.

SR:B-OP*:* Suppose the evaluation derivation ends with an application of the rule E:BIN, thus $E_1 \left|\frac{p_1' + K^{\mathsf{op}}}{p_1'}\right. e \Downarrow v_1$ and $E_1 \left|\frac{p_2' + K^{\mathsf{op}}}{p_2'}\right. e \Downarrow v_1$. We have $p_1 - p_1' = p_2 - p_2' = K^{\mathsf{op}}$.

SR:IB-OP*:* It is similar to the case SR:B-OP.

SR:I-OP*:* It is similar to the case SR:B-OP.

SR:C-GEN*:* By the hypothesis we have $\text{const}_X(e)$, thus it holds that $\Sigma^r; \Gamma^r \left|\frac{q}{q'}\right. e : A$ and $\curlyvee (A \mid A, A)$. By the constant-resource theorem, for all $p_1, p_1', p_2, p_2' \in \mathbb{Q}_0^+$ such that $E_1 \left|\frac{p_1}{p_1'}\right. e \Downarrow v_1$ and $E_2 \left|\frac{p_2}{p_2'}\right. e \Downarrow v_2$, we have $p_1 - p_1' = q + \Phi_{E_1}(\Gamma^r) - (q' + \Phi(v_1 : A))$ and $p_2 - p_2' = q + \Phi_{E_2}(\Gamma^r) - (q' + \Phi(v_2 : A))$.

Since $E_1 \approx_X E_2$, $\Phi_{E_1}(X) = \Phi_{E_2}(X)$. For all $y \notin X$, $E_1(y) = E_2(y)$ since $E_1 \equiv_{k_1} E_2$, thus $\Phi(E_1(y)) = \Phi(E_2(y))$. Hence, $\Phi_{E_1}(\Gamma^r) = \Phi_{E_2}(\Gamma^r)$, it follows $p_1 - p_1' = p_2 - p_2'$.

SR:FUN*:* Suppose e is of the form $\text{app}(f, x)$, thus the typing derivation ends with an application of either the rule SR:L-ARG, SR:C-FUN, or SR:C-GEN.

- Case SR:L-ARG. By the hypothesis we have $E_1(x) = E_2(x)$, it follows $p_1 - p_1' = p_2 - p_2'$.
- Case SR:C-FUN. Because e is well-formed, there exists a well-typed expression e_f such that pc'; $\Sigma^s; \Gamma^s \left|\frac{\text{const}}{}\right. e_f : S_2$. By induction for e_f which is resource-aware noninterference w.r.t X, $p_1 - K^{\mathsf{app}} - p_1' = p_2 - K^{\mathsf{app}} - p_2'$, it follows.
- Case SR:C-GEN. By the case SR:C-GEN it follows.

SR:LET*:* Suppose e is of the form $\text{let}(x, e_1, x.e_2)$, thus the typing derivation ends with an application of either the rule SR:L-LET or SR:C-GEN.

- Case SR:L-LET. Suppose the evaluations $E_1 \left|\frac{p_1 - K^{\mathsf{let}}}{p'}\right. e_1 \Downarrow v_1^1$, $E_2 \left|\frac{p_2 - K^{\mathsf{let}}}{p"}\right. e_1 \Downarrow v_1^2$, $E_1[x \mapsto v_1^1] \left|\frac{p'}{p_1'}\right. e_2 \Downarrow v_1$, and $E_2[x \mapsto v_1^2] \left|\frac{p"}{p_2'}\right. e_2 \Downarrow v_2$. By induction for e_1 that is resource-aware noninterference w.r.t X, $p_1 - K^{\mathsf{let}} - p' = p_2 - K^{\mathsf{let}} - p"$. By the hypothesis $v_1^1 = v_1^2$. Thus $E_1[x \mapsto v_1^1] \approx_X E_2[x \mapsto v_1^2]$ and $E_1[x \mapsto v_1^1] \equiv_{k_1} E_2[x \mapsto v_1^2]$, by induction for e_2 that is resource-aware noninterference w.r.t X, we have $p' - p_1' = p" - p_2'$. Hence, $p_1 - p_1' = p_2 - p_2'$.
- Case SR:C-GEN. By the case SR:C-GEN it follows.

SR:IF*:* Suppose e is of the form $\text{if}(x, e_t, e_f)$, thus the typing derivation ends with an application of either the rule SR:L-IF or SR:C-GEN.

- Case SR:L-IF. By the hypothesis we have $E_1(x) = E_2(x)$. Assume that $E_1(x) = E_2(x) = \mathsf{true}$, by the evaluation rule E:IF-TRUE, $E_1 \left|\frac{p_1 - K^{\mathsf{cond}}}{p_1'}\right. e_t \Downarrow v_1$ and $E_2 \left|\frac{p_2 - K^{\mathsf{cond}}}{p_2'}\right. e_t \Downarrow v_2$. By induction for e_t that is resource-aware noninterference w.r.t X, we have $p_1 - p_1' = p_2 - p_2'$. It is similar for $E_1(x) = E_2(x) = \mathsf{false}$.
- Case SR:C-GEN. Since $E_1 \approx_X E_2$ w.r.t Γ^s, we have $E_1 \approx_X E_2$ w.r.t Γ^r. By the hypothesis we have $\text{const}_X(e)$. Thus by the soundness theorem of constant resource type system, it follows $p_1 - p_1' = p_2 - p_2'$.

SR:PAIR*:* It is similar to the case SR:B-OP.

SR:MATCH-P*:* Suppose e is of the form $\text{match}(x, (x_1, x_2).e)$, thus the typing derivation ends with an application of either the rule SR:C-MATCH-P or SR:C-GEN.

- Case SR:C-MATCH-P. Let $E_1' = E_1[x_1 \mapsto v_1^1, x_2 \mapsto v_2^1]$ and $E_2' = E_2[x_1 \mapsto v_1^2, x_2 \mapsto v_2^2]$. If $x \in X$ then $|E_1(x)| \approx |E_2(x)|$. Thus $|E_1'(x_1)| \approx |E_2'(x_1)|$ and $|E_1'(x_2)| \approx |E_2'(x_2)|$. Hence, $E_1' \approx_{X \cup \{x_1, x_2\}} E_2'$, by induction for e in the premise which is resource-aware noninterference w.r.t $X \cup \{x_1, x_2\}$, $p_1 - K^{\mathsf{matchP}} - p_1' = p_2 - K^{\mathsf{matchP}} - p_2'$, it follows. If $x \notin X$ then $E_1(x) = E_2(x)$, it is similar.
- Case SR:C-GEN. By the case SR:C-GEN it follows.

SR:NIL*:* It is similar to the case SR:UNIT.

SR:CONS*:* It is similar to the case SR:B-OP.

SR:MATCH-L*:* Suppose e is of the form $\text{match}(x, e_1, (x_h, x_t).e_2)$, thus the typing derivation ends with an application of either the rule SR:C-MATCH-L or SR:C-GEN.

- Case SR:C-MATCH-L. Let $E_1' = E_1[x_h \mapsto v_1^1, x_t \mapsto v_2^1]$ and $E_2' = E_2[x_h \mapsto v_1^2, x_t \mapsto v_2^2]$. If $x \in X$ then $|E_1(x)| \approx |E_2(x)|$. Suppose $E_1(x)$ and $E_2(x)$ are different from nil, $|E_1'(x_h)| \approx |E_2'(x_h)|$ and $|E_1'(x_t)| \approx |E_2'(x_t)|$. Hence, $E_1' \approx_{X \cup \{x_t, x_h\}} E_2'$, by induction for e_2 which is resource-aware noninterference w.r.t $X \cup \{x_t, x_h\}$, we have $p_1 - K^{\mathsf{matchL}} - p_1' = p_2 - K^{\mathsf{matchL}} - p_2'$, thus $p_1 - p_1' = p_2 - p_2'$. If $E_1(x) = E_2(x) = \mathsf{nil}$ then by induction for e_1 that is resource-aware noninterference w.r.t X, it follows. If $x \notin X$ then $E_1(x) = E_2(x)$, it is similar.
- Case SR:C-GEN. By the case SR:C-GEN it follows.

SR:SUBTYPING*:* The typing derivation ends with an application of either the rule SR:C-SHARE or SR:C-GEN.

- Case SR:C-SUBTYPING. By induction for e in the premise, $p_1 - p_1' = p_2 - p_2'$.
- Case SR:C-GEN. By the case SR:C-GEN it follows.

TYPE SYSTEMS FOR LOWER BOUNDS AND CONSTANT RESOURCE

The common syntax-directed typing rules for all of three type systems; upper bounds, constant resource, and lower bounds are represented in Fig. 15. While the different structural rules are shown in Fig. 16, Fig. 17, and Fig. 18. We can see that the relax rules are consistent among these type systems in sense of satisfying the following.

$$
\begin{array}{ll}
 & (q \geq p \wedge q - p \leq q' - p') \wedge (q \geq p \wedge q - p \geq q' - p') \\
\Leftrightarrow & (q \geq p \wedge q - p = q' - p')
\end{array}
$$

That means the constraints for upper bounds and lower bounds imply the constraints for constant resource and vice versa.

The type systems for upper bounds, constant resource, and lower bounds are *affine*, *linear*, and *relevant* sub-structural type systems, respectively.

- Type system for upper bounds allows exchange and weakening, but not contraction properties.
- Type system for constant resource allows exchange but not weakening or contraction properties.
- Type system for lower bounds allows exchange and contraction, but not weakening properties.

PROOFS OF LEMMA 4, THEOREM 2, THEOREM 3, THEOREM 5, AND THEOREM 6

Proof of Lemma 4

The claim is proved by induction on the definitions of potential and size-equivalence, in which $|E_1(x)| \approx |E_2(x)|$ implies $\Phi(E_1(x) : \Gamma^r(x)) = \Phi(E_2(x) : \Gamma^r(x))$.

Proof of Theorem 2

The proof is done by induction on the length of the derivation of the evaluation judgement and the typing judgement with lexical order, in which the derivation of the evaluation judgement takes priority over the typing derivation. We need to do induction on the length of both evaluation and typing derivations since on one hand, an induction of only typing derivation would fail for the case of function application, which increases the length of the typing derivation, while the length of the evaluation derivation never increases. On the other hand, if the rule C:WEAKENING is the final step in the derivation, then the length of typing derivation decreases, while the length of evaluation derivation is unchanged.

A:SHARE*:* Assume that the typing derivation ends with an application of the rule A:SHARE, thus $\Sigma^r; \Gamma^r, x_1 : A_1, x_2 : A_2 \vdash^{q}_{q'} e : B$ and $\curlyvee(A | A_1, A_2)$.

Let $E_1 = E \setminus \{x\} \cup \{[x_1 \mapsto E(x), x_2 \mapsto E(x)]\}$. Since $\models E : \Gamma^r, x : A$ and following the property of the share relation we have $\models E_1 : \Gamma^r, x_1 : A_1, x_2 : A_2$. By the induction hypothesis for e, it holds that for all $p, r \in \mathbb{Q}_0^+$ such that $p = q + \Phi_{E_1}(\Gamma^r, x_1 : A_1, x_2 : A_2) + r$, there exists $p' \in \mathbb{Q}_0^+$ satisfying $E_1 \vdash^{p}_{p'} e \Downarrow v$ and $p' = q' + \Phi(v : B) + r$.

Because $\Phi(E(x) : A) = \Phi(E_1(x_1) : A_1) + \Phi(E_1(x_2) : A_2)$ and $\Phi_E(\Gamma^r) = \Phi_{E_1}(\Gamma^r) = \Phi_{E \setminus \{x\}}(\Gamma^r)$, thus $p = q + \Phi_E(\Gamma^r, x : A) + r$ and there exists p' satisfying $E \vdash^{p}_{p'} \text{share}(x, (x_1, x_2).e) \Downarrow v$.

C:WEAKENING*:* Suppose that the typing derivation ends with an application of the rule C:WEAKENING. Thus we have $\Sigma^r; \Gamma^r \vdash^{q}_{q'} e : B$, in which the data type A satisfies $\curlyvee(A \mid A, A)$.

Since $\models E : \Gamma^r, x : A$, it follows $\models E : \Gamma^r$. By the induction hypothesis for e, it holds that for all $p, r \in \mathbb{Q}_0^+$ such that $p = q + \Phi_E(\Gamma^r) + r$, there exists $p' \in \mathbb{Q}_0^+$ satisfying $E \vdash^{p}_{p'} e \Downarrow v$ and $p' = q' + \Phi(v : B) + r$. By the property of the share relation, $\Phi(a : A) = 0$, then we have $p = q + \Phi_E(\Gamma^r, x : A) + r$, $E \vdash^{p}_{p'} e \Downarrow v$ and $p' = q' + \Phi(v : B) + r$ as required.

C:RELAX*:* Suppose that the typing derivation ends with an application of the rule C:RELAX, thus we have $\Sigma^r; \Gamma^r \vdash^{q_1}_{q'_1} e : A$, $q \geq q_1$, and $q - q_1 = q' - q'_1$.

For all $p, r \in \mathbb{Q}_0^+$ such that $p = q + \Phi_E(\Gamma^r) + r = q_1 + \Phi_E(\Gamma^r) + (q - q_1) + r$, we have $\models E : \Gamma^r$. By the induction hypothesis for e in the premise, there exists $p' \in \mathbb{Q}_0^+$ satisfying $E \vdash^{p}_{p'} e \Downarrow v$ and $p' = q'_1 + \Phi(v : A) + (q - q_1) + r = q' + \Phi(v : A) + r$.

A:VAR*:* Assume that e is a variable x. If $\Sigma^r; x : A \vdash^{K^{\mathsf{var}}}_{0} x : A$. Thus for all $p, r \in \mathbb{Q}_0^+$ such that $p = K^{\mathsf{var}} + \Phi(v : A) + r$, there exists $p' = \Phi(v : A) + r$ satisfying $E \vdash^{p}_{p'} e \Downarrow v$.

A:UNIT*:* It is similar to the case A:VAR.

A:BOOL*:* It is similar to the case A:VAR.

A:INT*:* It is similar to the case A:VAR.

A:B-OP*:* Assume that e is an expression of the form $\text{op}_{\diamond}(x_1, x_2)$, where $\diamond = \{\,\mathsf{and}\,,\,\mathsf{or}\,\}$. Thus $\Sigma^r; x_1 : \text{bool}, x_2 : \text{bool} \vdash^{K^{\mathsf{op}}}_{0} e : \text{bool}$ and $\models E : \{x_1 : \text{bool}, x_2 : \text{bool}\}$. We have $E \vdash^{K^{\mathsf{op}}}_{0} e \Downarrow v$, thus for all $p, r \in \mathbb{Q}_0^+$ such that $p = K^{\mathsf{op}} + r = K^{\mathsf{op}} + \Phi_E(x_1 : \text{bool}, x_2 : \text{bool}) + r$, there exists $p' = \Phi(v : \text{bool}) + r = r$ satisfying $E \vdash^{p}_{p'} e \Downarrow v$.

A:I-OP*:* It is similar to the case A:B-OP.

A:IB-OP*:* It is similar to the case A:B-OP.

A:CONS*:* If e is of the form $\text{cons}(x_1, x_2)$, then the type derivation ends with an application of the rule A:CONS and the evaluation ends with the application of the rule E:CONS. Thus $\Sigma^r; x_1 : A, x_2 : L^{p_1}(A) \vdash^{p_1 + K^{\mathsf{cons}}}_{0} e : L^{p_1}(A)$ and $\models E : \{x_1 : A, x_2 : L^{p_1}(A)\}$.

We have $E \vdash^{K^{\mathsf{cons}}}_{0} e \Downarrow [v_1, ..., v_n]$, where $E(x_1) = v_1$ and $E(x_2) = [v_2, \cdots, v_n]$. Let $\Gamma^r = x_h : A, x_t : L^{p_1}(A)$, for all $p, r \in \mathbb{Q}_0^+$ such that $p = p_1 + K^{\mathsf{cons}} + \Phi_E(\Gamma^r) + r$, there exists $p' \in \mathbb{Q}_0^+$ satisfying $p' = \Phi([v_1, ..., v_n] : L^{p_1}(A)) + r = \Phi_E(\Gamma^r) + p_1 + r$ and $E \vdash^{p}_{p'} e \Downarrow [v_1, ..., v_n]$.

A:PAIR*:* It is similar to the case A:CONS.

A:NIL*:* It is similar to the case A:CONS.

A:MATCH-P*:* Suppose that the typing derivation $\Sigma^r; \Gamma^r, x : A_1 * A_2 \vdash^{q}_{q'} \text{match}(x, (x_1, x_2).e) : A$ ends with

$$\text{(A:UNIT)}\quad \frac{}{\Sigma^r;\emptyset \vdash^{K^{\mathsf{unit}}}_{0} () : \mathrm{unit}}$$

$$\text{(A:B-OP)}\quad \frac{\diamond \in \{\mathsf{and}, \mathsf{or}\}}{\Sigma^r; x_1 : \mathrm{bool}, x_2 : \mathrm{bool} \vdash^{K^{\mathsf{op}}}_{0} \mathrm{op}_\diamond(x_1, x_2) : \mathrm{bool}}$$

$$\text{(A:IB-OP)}\quad \frac{\diamond \in \{=, <>, >, <, <=, >=\}}{\Sigma^r; x_1 : \mathrm{int}, x_2 : \mathrm{int} \vdash^{K^{\mathsf{op}}}_{0} \mathrm{op}_\diamond(x_1, x_2) : \mathrm{bool}}$$

$$\text{(A:INT)}\quad \frac{n \in \mathbb{Z}}{\Sigma^r;\emptyset \vdash^{K^{\mathsf{int}}}_{0} n : \mathrm{int}}$$

$$\text{(A:I-OP)}\quad \frac{\diamond \in \{+, -, *, \mathsf{div}, \mathsf{mod}\}}{\Sigma^r; x_1 : \mathrm{int}, x_2 : \mathrm{int} \vdash^{K^{\mathsf{op}}}_{0} \mathrm{op}_\diamond(x_1, x_2) : \mathrm{int}}$$

$$\text{(A:FUN)}\quad \frac{\Sigma^r(f) = A_1 \xrightarrow{q/q'} A_2}{\Sigma^r; x : A_1 \vdash^{q + K^{\mathsf{app}}}_{q'} \mathrm{app}(f, x) : A_2}$$

$$\text{(A:LET)}\quad \frac{\Sigma^r;\Gamma^r_1 \vdash^{q - K^{\mathsf{let}}}_{q'_1} e_1 : A_1 \qquad \Sigma^r;\Gamma^r_2, x : A_1 \vdash^{q'_1}_{q'} e_2 : A_2}{\Sigma^r;\Gamma^r_1,\Gamma^r_2 \vdash^{q}_{q'} \mathrm{let}(x, e_1, x.e_2) : A_2}$$

$$\text{(A:IF)}\quad \frac{\Sigma^r;\Gamma^r \vdash^{q - K^{\mathsf{cond}}}_{q'} e_t : A \qquad \Sigma^r;\Gamma^r \vdash^{q - K^{\mathsf{cond}}}_{q'} e_f : A}{\Sigma^r;\Gamma^r, x : \mathrm{bool} \vdash^{q}_{q'} \mathrm{if}(x, e_t, e_f) : A}$$

$$\text{(A:PAIR)}\quad \frac{}{\Sigma^r; x_1 : A_1, x_2 : A_2 \vdash^{K^{\mathsf{pair}}}_{0} \mathrm{pair}(x_1, x_2) : A_1 * A_2}$$

$$\text{(A:MATCH-P)}\quad \frac{\Sigma^r;\Gamma^r, x_1 : A_1, x_2 : A_2 \vdash^{q - K^{\mathsf{matchP}}}_{q'} e : A}{\Sigma^r;\Gamma^r, x : A_1 * A_2 \vdash^{q}_{q'} \mathrm{match}(x, (x_1, x_2).e) : A}$$

$$\text{(A:BOOL)}\quad \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Sigma^r;\emptyset \vdash^{K^{\mathsf{bool}}}_{0} b : \mathrm{bool}}$$

$$\text{(A:VAR)}\quad \frac{}{\Sigma^r; x : A \vdash^{K^{\mathsf{var}}}_{0} x : A}$$

$$\text{(A:NIL)}\quad \frac{A \in \mathscr{A}}{\Sigma^r;\emptyset \vdash^{K^{\mathsf{nil}}}_{0} \mathsf{nil} : L^p(A)}$$

$$\text{(A:CONS)}\quad \frac{}{\Sigma^r; x_h : A, x_t : L^p(A) \vdash^{p + K^{\mathsf{cons}}}_{0} \mathrm{cons}(x_h, x_t) : L^p(A)}$$

$$\text{(A:MATCH-L)}\quad \frac{\Sigma^r;\Gamma^r \vdash^{q - K^{\mathsf{matchN}}}_{q'} e_1 : A_1 \qquad \Sigma^r;\Gamma^r, x_h : A, x_t : L^p(A) \vdash^{q + p - K^{\mathsf{matchL}}}_{q'} e_2 : A_1}{\Sigma^r;\Gamma^r, x : L^p(A) \vdash^{q}_{q'} \mathrm{match}(x, e_1, (x_h, x_t).e_2) : A_1}$$

$$\text{(A:SHARE)}\quad \frac{\Sigma^r;\Gamma^r, x_1 : A_1, x_2 : A_2 \vdash^{q}_{q'} e : B \qquad \curlyvee(A \mid A_1, A_2)}{\Sigma^r;\Gamma^r, x : A \vdash^{q}_{q'} \mathrm{share}(x, (x_1, x_2).e) : B}$$

Figure 15. Common syntax-directed typing rules for upper bounds, constant, and lower bounds

$$\text{(U:RELAX)}\quad \frac{\Sigma^r;\Gamma^r \vdash^{p}_{p'} e : A \qquad q \geq p \qquad q - p \geq q' - p'}{\Sigma^r;\Gamma^r \vdash^{q}_{q'} e : A}$$

$$\text{(U:WEAKENING)}\quad \frac{\Sigma^r;\Gamma^r \vdash^{q}_{q'} e : B}{\Sigma^r;\Gamma^r, x : A \vdash^{q}_{q'} e : B}$$

$$\text{(U:SUBTYPE)}\quad \frac{\Sigma^r;\Gamma^r \vdash^{q}_{q'} e : A \qquad B <: A}{\Sigma^r;\Gamma^r \vdash^{q}_{q'} e : B}$$

$$\text{(U:SUPERTYPE)}\quad \frac{\Sigma^r;\Gamma^r, x : B \vdash^{q}_{q'} e : C \qquad B <: A}{\Sigma^r;\Gamma^r, x : A \vdash^{q}_{q'} e : C}$$

Figure 16. Relax and structural typing rules for upper bounds

$$\text{(C:RELAX)}\quad \frac{\Sigma^r;\Gamma^r \vdash^{p}_{p'} e : A \qquad q \geq p \qquad q - p = q' - p'}{\Sigma^r;\Gamma^r \vdash^{q}_{q'} e : A}$$

$$\text{(C:WEAKENING)}\quad \frac{\Sigma^r;\Gamma^r \vdash^{q}_{q'} e : B \qquad \curlyvee(A \mid A, A)}{\Sigma^r;\Gamma^r, x : A \vdash^{q}_{q'} e : B}$$

Figure 17. Relax and structural typing rules for constant-resource

an application of the rule A:MATCH-P. Thus $\Sigma^r;\Gamma^r, x_1 : A_1, x_2 : A_2 \vdash^{q - K^{\mathsf{matchP}}}_{q'} e : A$ and $\models E : \Gamma^r, x : A_1 * A_2$.

Let $E_1 = E[x_1 \mapsto v_1, x_2 \mapsto v_2]$ and $\Gamma^r_1 = \Gamma^r, x_1 : A_1, x_2 : A_2$. Since $\models v_1 : A_1$, $\models v_2 : A_2$, and $\models E : \Gamma^r$ it holds that $\models E_1 : \Gamma^r_1$. For all $p, r \in \mathbb{Q}^+_0$ such that $p = q + \Phi_E(\Gamma^r, x : A_1 * A_2) + r$, thus $p - K^{\mathsf{matchP}} = q - K^{\mathsf{matchP}} + \Phi_{E_1}(\Gamma^r_1) + r$, by the induction hypothesis for e, there exists $p' \in \mathbb{Q}^+_0$ satisfying $p' = q' + \Phi(v : A) + r$ and $E_1 \vdash^{p - K^{\mathsf{matchP}}}_{p'} e \Downarrow v$. Hence, by the rule E:MATCH-P, there exists $p' = q' + \Phi(v : A) + r$ satisfying $E \vdash^{p}_{p'} \mathrm{match}(x, (x_1, x_2).e) \Downarrow v$.

A:FUN: Assume that e is a function application of the form $\mathrm{app}(f, x)$. Thus $\Sigma^r; x : A_1 \vdash^{q + K^{\mathsf{app}}}_{q'} e : A_2$ and $\Sigma^r(f) = A_1 \xrightarrow{q/q'} A_2$. Because the considering program is well-formed, there exists a well-typed expression e_f under the typing context $\Gamma^r_1 = y^{\widehat{f}} : A_1$ and the signature Σ^r, or

$$\frac{\Sigma^r; \Gamma^r \mid\frac{p}{p'}\ e : A \qquad q \geq p \qquad q - p \leq q' - p'}{\Sigma^r; \Gamma^r \mid\frac{q}{q'}\ e : A}\ \text{(L:RELAX)}$$

$$\frac{\Sigma^r; \Gamma^r \mid\frac{q}{q'}\ e : B \qquad \curlyvee(A \mid A, A)}{\Sigma^r; \Gamma^r, x : A \mid\frac{q}{q'}\ e : B}\ \text{(L:WEAKENING)}$$

$$\frac{\Sigma^r; \Gamma^r \mid\frac{q}{q'}\ e : A \qquad A <: B}{\Sigma^r; \Gamma^r \mid\frac{q}{q'}\ e : B}\ \text{(L:SUBTYPE)}$$

$$\frac{\Sigma^r; \Gamma^r, x : B \mid\frac{q}{q'}\ e : C \qquad A <: B}{\Sigma^r; \Gamma^r, x : A \mid\frac{q}{q'}\ e : C}\ \text{(L:SUPERTYPE)}$$

$$\frac{\Sigma^r; \Gamma^r, x_1 : A, x_2 : A \mid\frac{q}{q'}\ e : B}{\Sigma^r; \Gamma^r, x : A \mid\frac{q}{q'}\ \mathsf{share}(x, (x_1, x_2).e) : B}\ \text{(L:CONTRACTION)}$$

Figure 18. Relax and structural typing rules for lower bounds

$\Sigma^r; \Gamma_1^r \mid\frac{q}{q'}\ e_f : A_2$.

Let $\Gamma^r = x : A_1$, $E(x) = v_1$ and $E_1 = [y^{\widehat{f}} \mapsto v_1]$, since $\models E : \Gamma^r$, it follows that $\models E_1 : \Gamma_1^r$. For all $p, r \in \mathbb{Q}_0^+$ such that $p = q + K^{\mathsf{app}} + \Phi_E(\Gamma^r) + r$, since $\Phi_{E_1}(\Gamma_1^r) = \Phi(E_1(y^{\widehat{f}}) : A_1) = \Phi_E(\Gamma^r) = \Phi(E(x) : A_1)$, it holds that $p - K^{\mathsf{app}} = q + \Phi_{E_1}(\Gamma_1^r) + r$. By the induction hypothesis for e_f, there exists $p' \in \mathbb{Q}_0^+$ satisfying $p' = q' + \Phi(v : A_2) + r$ and $E_1 \mid\frac{p_1}{p_1'}\ e_{\widehat{f}} \Downarrow v$. Hence, $E \mid\frac{p}{p'}\ e \Downarrow v$.

A:IF*:* Suppose that e is an expression of the form $\mathsf{if}(x, e_t, e_f)$. Then one of the rules E:IF-TRUE and E:IF-FALSE has been applied in the evaluation derivation depending on the value of x.

Assume that the variable x is assigned the value $\mathtt{true}$ in E, or $E(x) = \mathtt{true}$. The typing rule for e has been derived by an application of the rule A:IF using the premise on the left thus $\Sigma^r; \Gamma^r \mid\frac{q - K^{\mathsf{cond}}}{q'}\ e_t : A$.

Let $\Gamma_1^r = \Gamma^r, x : \mathsf{bool}$, since $\models E : \Gamma_1^r$, it follows that $\models E : \Gamma^r$. For all $p, r \in \mathbb{Q}_0^+$ such that $p = q + \Phi_E(\Gamma_1^r) + r$, since $\Phi_E(\Gamma^r) = \Phi_E(\Gamma_1^r)$ thus $p_1 = p - K^{\mathsf{cond}} = q - K^{\mathsf{cond}} + \Phi_E(\Gamma^r) + r$. By the induction hypothesis for e_t, there exists $p_1' \in \mathbb{Q}_0^+$ satisfying $E \mid\frac{p_1}{p_1'}\ e_t \Downarrow v$ and $p_1' = q' + \Phi(v : A)$. Hence, by the rule E:IF-TRUE, there exists $p' = p_1'$ satisfying $E \mid\frac{p}{p'}\ e \Downarrow v$ and $p' = q' + \Phi(v : A)$. If x is assigned the value $\mathtt{false}$ in E then it is similar to the case $E(x) = \mathtt{true}$.

A:MATCH-L*:* It is the same as the case of a conditional expression. The evaluation derivation applies one of the rules E:MATCH-N and E:MATCH-L depending on the value of x.

Assume that x is assigned the value $[v_1, ..., v_n]$ under E, or $E(x) = [v_1, ..., v_n]$. Then, the evaluation derivation ends with an application of the rule E:MATCH-L. Let $E_1 = E[x_h \mapsto v_1, x_t \mapsto [v_2, ..., v_n]]$ and $\Gamma_1^r = \Gamma^r, x_h : A, x_t : L^{p_1}(A)$, the typing derivation ends with an application of the rule A:MATCH-L, thus $\Sigma^r; \Gamma_1^r \mid\frac{q + p_1 - K^{\mathsf{matchL}}}{q'}\ e_2 : A_1$.

Since $\models [v_1, ..., v_n] : L^{p_1}(A)$, we have $\models v_i : A, \forall i = 1, ..., n$. Hence, it holds that $\models v_1 : A$ and $\models [v_2, ..., v_n] : L^{p_1}(A)$. Finally, we have $\models E_1 : \Gamma_1^r$ (since $\models E : \Gamma^r$ implies $\models E_1 : \Gamma^r$).

For all $p, r \in \mathbb{Q}_0^+$ such that $p = q + \Phi_E(\Gamma^r, x : L^{p_1}(A)) + r$, because $\Phi_E(\Gamma^r, x : L^{p_1}(A)) = \Phi_E(\Gamma^r) + n.p_1 + \Sigma_{i=1}^n \Phi(v_i : A)$, $\Phi_{E_1}(\Gamma_1^r) = \Phi_{E_1}(\Gamma^r) + (n-1).p_1 + \Sigma_{i=1}^n \Phi(v_i : A)$ and $\Phi_{E_1}(\Gamma^r) = \Phi_E(\Gamma^r)$, thus we have $\Phi_{E_1}(\Gamma_1^r) = \Phi_E(\Gamma^r, x : L^{p_1}(A)) - p_1$. Thus $p_2 = p - K^{\mathsf{matchL}} = q + p_1 - K^{\mathsf{matchL}} + \Phi_{E_1}(\Gamma_1^r) + r$. By the induction hypothesis for e_2, there exists $p_2' \in \mathbb{Q}_0^+$ satisfying $E_1 \mid\frac{p_2}{p_2'}\ e_2 \Downarrow v$ and $p_2' = q' + \Phi(v : A_1)$. Hence, there exists $p' = p_2'$ such that $E \mid\frac{p}{p'}\ e \Downarrow v$. If $E(x) = \mathsf{nil}$ then it is similar to the case A:MATCH-P.

A:LET*:* Assume that e is an expression of the form $\mathsf{let}(x, e_1, x.e_2)$. Hence, the evaluation derivation ends with an application of the rule E:LET. Let $E_1 = E[x \mapsto v_1]$ and $\Gamma^r = \Gamma_1^r, \Gamma_2^r$. The typing derivation ends with an application of the rule A:LET, thus $\Sigma^r; \Gamma_1^r \mid\frac{q - K^{\mathsf{let}}}{q_1'}\ e_1 : A_1$ and $\Sigma^r; \Gamma_2^r, x : A_1 \mid\frac{q_1'}{q'}\ e_2 : A_2$.

For all $p, r \in \mathbb{Q}_0^+$ such that $p = q + \Phi_E(\Gamma^r) + r$, thus $p_1 = p - K^{\mathsf{let}} = q - K^{\mathsf{let}} + \Phi_E(\Gamma_1^r) + \Phi_E(\Gamma_2^r) + r$. Since $\models E : \Gamma^r$, we have $\models E : \Gamma_1^r$. By the induction hypothesis for e_1, there exists $p_1' \in \mathbb{Q}_0^+$ satisfying $E \mid\frac{p_1}{p_1'}\ e_1 \Downarrow v_1$ and $p_1' = q_1' + \Phi(v_1 : A_1) + \Phi_E(\Gamma_2^r) + r$.

We have $\models E : \Gamma_2^r$, thus $\models E_1 : \Gamma_2^r, x : A_1$. Again by the induction hypothesis for e_2, with $p_2 = p - K^{\mathsf{let}} - (p_1 - p_1') = p_1' = q_1' + \Phi_{E_1}(\Gamma_2^r, x : A_1) + r$, there exists $p_2' \in \mathbb{Q}_0^+$ satisfying $E_1 \mid\frac{p_2}{p_2'}\ e_2 \Downarrow v$ and $p_2' = q' + \Phi(v : A_2) + r$. Hence, by the rule E:LET, there exists $p' = p_2'$ satisfying $E \mid\frac{p}{p'}\ e \Downarrow v$ and $p' = q' + \Phi(v : A_2)$.

Proof of Theorem 3

First, we prove that if $E \mid\frac{p}{p'}\ e \Downarrow v$ then $p - p' = q + \Phi_E(\Gamma^r) - (q' + \Phi(v : A))$. Suppose $p - p' \neq q + \Phi_E(\Gamma^r) - (q' + \Phi(v : A))$, there exists always some $r_1, r_2 \in \mathbb{Q}_0^+$ such that $p + r_1 = q + \Phi_E(\Gamma^r) + r_2$. Since $E \mid\frac{p}{p'}\ e \Downarrow v$, we have $E \mid\frac{p + r_1}{p' + r_1}\ e \Downarrow v$. By Theorem 2, $p' + r_1 = q' + \Phi(v : A) + r_2$, thus the assumption is contradictory.

Consider any E_1 and E_2 such that $E_1 \approx_X E_2$, hence $E_1 \vdash e \Downarrow v_1$ and $E_2 \vdash e \Downarrow v_2$. For all $p_1, p_1' \in \mathbb{Q}_0^+$ such that $E_1 \mid\frac{p_1}{p_1'}\ e \Downarrow v_1$, we have $p_1 - p_1' = q + \Phi_{E_1}(\Gamma^r) - (q' + \Phi(v_1 : A))$. Similarly, for all $p_2, p_2' \in \mathbb{Q}_0^+$ such that $E_2 \mid\frac{p_2}{p_2'}\ e \Downarrow v_2$, $p_2 - p_2' = q + \Phi_{E_2}(\Gamma^r) - (q' + \Phi(v_2 : A))$. Since $\Phi_{E_1}(X) = \Phi_{E_2}(X)$ by Lemma 4, $\forall x \in \mathrm{dom}(\Gamma^r) \setminus X.\Phi(E_i(x) : \Gamma^r(x)) = 0$, and $\Phi(v_i : A) = 0$, $i = 1, 2$. Thus $p_1 - p_1' = p_2 - p_2'$.

Proof of Theorem 5

The proof is relied on Theorem 6. For all $p, r \in \mathbb{Q}_0^+$ such that $p < q + \Phi_E(\Gamma^r) + r$, assume that there exists some

$p' \in \mathbb{Q}_0^+$ such that $E \vdash^{p}_{p'} e \Downarrow v$ and $p' \geq q' + \Phi(v : A) + r$. Thus we have $p - p' < q + \Phi_E(\Gamma^r) - (q' + \Phi(v : A))$. On the other hand, it holds that $q + \Phi_E(\Gamma^r) - (q' + \Phi(v : A)) \leq p - p'$. The assumption is contradictory.

Proof of Theorem 6

The proof is done by induction on the length of the derivation of the evaluation judgement $E \vdash^{p}_{p'} e \Downarrow v$ and the typing judgement $\Sigma; \Gamma \vdash^{q}_{q'} e : A$ with lexical order, in which the derivation of the evaluation judgement takes priority over the typing derivation. We need to do induction on the length of both evaluation and typing derivations since on one hand, an induction of only typing derivation would fail for the case of function application, which increases the length of the typing derivation, while the length of the evaluation derivation never increases. On the other hand, if the rules L:WEAKENING and A:SHARE are final step in the derivation, then the length of typing derivation decreases, while the length of evaluation derivation is unchanged.

A:SHARE: Assume that the typing derivation ends with an application of the rule A:SHARE, thus $\Sigma^r; \Gamma^r, x_1 : A_1, x_2 : A_2 \vdash^{q}_{q'} e : B$ and $\curlyvee(A|A_1, A_2)$. Let $E_1 = E \setminus \{x\} \cup \{[x_1 \mapsto E(x), x_2 \mapsto E(x)]\}$. Since $\models E : \Gamma^r, x : A$ and following the property of the share relation we have $\models E_1 : \Gamma^r, x_1 : A_1, x_2 : A_2$.

For all $p, p' \in \mathbb{Q}_0^+$ such that $E \vdash^{p}_{p'}$ share$(x, (x_1, x_2).e \Downarrow v$, by the rule E:SHARE we have $E_1 \vdash^{p}_{p'} e \Downarrow v$. Hence, by the induction hypothesis for e in the premise, it holds that $q + \Phi_{E_1}(\Gamma^r, x_1 : A_1, x_2 : A_2) - (q' + \Phi(v : B)) \leq p - p'$.

Because $\Phi(E(x) : A) = \Phi(E_1(x_1) : A_1) + \Phi(E_1(x_2) : A_2)$ and $\Phi_E(\Gamma^r) = \Phi_{E_1}(\Gamma^r) = \Phi_{E \setminus \{x\}}(\Gamma^r)$, we have $q + \Phi_E(\Gamma^r, x : A) - (q' + \Phi(v : B)) \leq p - p'$.

L:WEAKENING: Suppose that the typing derivation $\Sigma^r; \Gamma^r, x : A \vdash^{q}_{q'} e : B$ ends with an application of the rule L:WEAKENING. Thus we have $\Sigma^r; \Gamma^r \vdash^{q}_{q'} e : B$, in which the data type A satisfies $\curlyvee(A \mid A, A)$. Since $\models E : \Gamma^r, x : A$, it follows that $\models E : \Gamma^r$.

For all $p, p' \in \mathbb{Q}_0^+$ such that $E \vdash^{p}_{p'} e \Downarrow v$, by the induction hypothesis for e in the premise, it holds that $q + \Phi_E(\Gamma^r) - (q' + \Phi(v : B)) \leq p - p'$. By the property of the share relation, $\Phi(a : A) = 0$, hence we have $q + \Phi_E(\Gamma^r, x : A) - (q' + \Phi(v : B)) \leq p - p'$.

L:RELAX: Suppose that the typing derivation ends with an application of the rule L:RELAX, thus we have $\Sigma^r; \Gamma^r \vdash^{q_1}_{q_1'} e : A$, $q \geq q_1$, and $q - q_1 \leq q' - q_1'$.

For all $p, p' \in \mathbb{Q}_0^+$ such that $E \vdash^{p}_{p'} e \Downarrow v$, we have $\models E : \Gamma^r$, hence by the induction hypothesis for e in the premise, it holds that $q_1 + \Phi_E(\Gamma^r) - (q_1' + \Phi(v : A)) \leq p - p'$. We have $q + \Phi_E(\Gamma^r) - (q' + \Phi(v : A)) = q_1 + \Phi_E(\Gamma^r) - (q_1' + \Phi(v : A)) + ((q - q_1) - (q' - q_1'))$. Since $q - q_1 \leq q' - q_1'$, it holds that $q + \Phi_E(\Gamma^r) - (q' + \Phi(v : A)) \leq q_1 + \Phi_E(\Gamma^r) - (q_1' + \Phi(v : A)) \leq p - p'$.

A:VAR: Assume that e is a variable x. If $\Sigma^r; x : A \vdash^{K^{\mathsf{var}}}_{0} x : A$. Thus for all $p, p' \in \mathbb{Q}_0^+$ such that $E \vdash^{p}_{p'} e \Downarrow v$, we have $p = p' + K^{\mathsf{var}}$, hence $K^{\mathsf{var}} + \Phi(E(x) : A) - \Phi(v : A) \leq p - p' = K^{\mathsf{var}}$.

A:UNIT: It is similar to the case A:VAR.

A:BOOL: It is similar to the case A:VAR.

A:INT: It is similar to the case A:VAR.

A:B-OP: Assume that e is an expression of the form $\mathrm{op}_{\diamond}(x_1, x_2)$, where $\diamond = \{\mathsf{and}, \mathsf{or}\}$. Thus $\Sigma^r; x_1 : \mathsf{bool}, x_2 : \mathsf{bool} \vdash^{K^{\mathsf{op}}}_{0} e : \mathsf{bool}$ and $\models E : \{x_1 : \mathsf{bool}, x_2 : \mathsf{bool}\}$.

For all $p, p' \in \mathbb{Q}_0^+$ such that $E \vdash^{p}_{p'} e \Downarrow v$, we have $K^{\mathsf{op}} + \Phi_E(x_1 : \mathsf{bool}, x_2 : \mathsf{bool}) - \Phi(v : \mathsf{bool}) = K^{\mathsf{op}} \leq p - p' = K^{\mathsf{op}}$.

A:IB-OP: It is similar to the case A:B-OP.

A:I-OP: It is similar to the case A:B-OP.

A:CONS: If e is of the form $\mathrm{cons}(x_1, x_2)$, then the typing derivation ends with an application of the rule A:CONS and the evaluation derivation ends with the application of the rule E:CONS. Thus $\Sigma^r; x_1 : A, x_2 : L^{p_1}(A) \vdash^{p_1 + K^{\mathsf{cons}}}_{0} e : L^{p_1}(A)$ and $\models E : \{x_1 : A, x_2 : L^{p_1}(A)\}$.

For all $p, p' \in \mathbb{Q}_0^+$ such that $E \vdash^{p}_{p'} e \Downarrow v$, we have $p - p' = K^{\mathsf{cons}}$, $E(x_1) = v_1$ and $E(x_2) = [v_2, \cdots, v_n]$. Let $\Gamma^r = x_h : A, x_t : L^{p_1}(A)$, it holds that $p_1 + K^{\mathsf{cons}} + \Phi_E(\Gamma^r) - (\Phi([v_1, ..., v_n] : L^{p_1}(A))) = K^{\mathsf{cons}} \leq p - p'$.

A:PAIR: It is similar to the case A:CONS.

A:NIL: It is similar to the case A:CONS.

A:MATCH-P: Suppose that the typing derivation $\Sigma^r; \Gamma^r, x : A_1 * A_2 \vdash^{q}_{q'} \mathrm{match}(x, (x_1, x_2).e) : A$ ends with an application of the rule A:MATCH-P. Thus $\Sigma^r; \Gamma^r, x_1 : A_1, x_2 : A_2 \vdash^{q - K^{\mathsf{matchP}}}_{q'} e : A$ and $\models E : \Gamma, x : A_1 * A_2$.

Let $E_1 = E[x_1 \mapsto v_1, x_2 \mapsto v_2]$ and $\Gamma_1^r = \Gamma^r, x_1 : A_1, x_2 : A_2$, since $\models v_1 : A_1$, $\models v_2 : A_2$, and $\models E : \Gamma^r$ it holds that $\models E_1 : \Gamma_1^r$.

For all $p, p' \in \mathbb{Q}_0^+$ such that $E \vdash^{p}_{p'} e \Downarrow v$, by the rule E:MATCH-P we have $E_1 \vdash^{p - K^{\mathsf{matchP}}}_{p'} e \Downarrow v$. Hence, by the induction hypothesis for e in the premise, it holds that $q - K^{\mathsf{matchP}} + \Phi_{E_1}(\Gamma_1^r) - (q' + \Phi(v : A)) \leq p - K^{\mathsf{matchP}} - p'$.

Since $\Phi_E(\Gamma^r, x : A_1 * A_2) = \Phi_{E_1}(\Gamma_1^r)$, it follows that $q + \Phi_E(\Gamma^r, x : A_1 * A_2) - (q' + \Phi(v : A)) \leq p - p'$.

A:FUN: Assume that e is a function application of the form $\mathrm{app}(f, x)$. Thus $\Sigma^r; x : A_1 \vdash^{q + K^{\mathsf{app}}}_{q'} e : A_2$ and $\Sigma^r(f) = A_1 \xrightarrow{q/q'} A_2$. Because the considering program is well-formed, there exists a well-typed expression e_f under the typing context $\Gamma_1^r = y^{\widehat{f}} : A_1$ and the signature Σ^r, or $\Sigma^r; \Gamma_1^r \vdash^{q}_{q'} e_f : A_2$.

Let $\Gamma^r = x : A_1$, $E(x) = v_1$ and $E_1 = [y^{\widehat{f}} \mapsto v_1]$, since $\models E : \Gamma^r$, it follows that $\models E_1 : \Gamma_1^r$. For all $p, p' \in \mathbb{Q}_0^+$ such that $E \vdash^{p}_{p'} e \Downarrow v$, we have $E_1 \vdash^{p - K^{\mathsf{app}}}_{p'} e_{\widehat{f}} \Downarrow v$. Hence, by the induction hypothesis for e_f, it holds that $q + \Phi_{E_1}(\Gamma_1^r) - (q' + \Phi(v : A_2)) \leq p - K^{\mathsf{app}} - p'$.

Since $\Phi_{E_1}(\Gamma_1^r) = \Phi(E_1(y^{\widehat{f}}) : A_1) = \Phi_E(\Gamma^r) = \Phi(E(x) : A_1)$, it follows that $q + K^{\mathsf{app}} + \Phi(E(x) : A_1) - (q' + \Phi(v : A_2)) \leq p - p'$.

A:IF: Suppose that e is an expression of the form if(x, e_t, e_f). Then one of the rules E:IF-TRUE and E:IF-FALSE has been applied in the evaluation derivation depending on the value of x.

Assume that the variable x is assigned the value $\mathtt{true}$ in E, or $E(x) = \mathtt{true}$. The typing rule for e has been derived by an application of the rule A:IF using the premise on the left thus $\Sigma^r; \Gamma^r \left|\frac{q - K^{\mathsf{cond}}}{q'}\right. e_t : A$. Let $\Gamma_1^r = \Gamma^r, x : \text{bool}$, since $\models E : \Gamma_1^r$, it follows that $\models E : \Gamma^r$.

For all $p, p' \in \mathbb{Q}_0^+$ such that $E \left|\frac{p}{p'}\right. e \Downarrow v$, by the rule E:IF-TRUE we have $E \left|\frac{p - K^{\mathsf{cond}}}{p'}\right. e_t \Downarrow v$. Hence, by the induction hypothesis for e_t, it holds that $q - K^{\mathsf{cond}} + \Phi_E(\Gamma^r) - (q' + \Phi(v : A)) \leq p - K^{\mathsf{cond}} - p'$.

Because $\Phi_E(\Gamma^r) = \Phi_E(\Gamma_1^r)$, it follows $q + \Phi_E(\Gamma_1^r) - (q' + \Phi(v : A)) \leq p - p'$. If $E(x) = \mathsf{false}$ then the proof is similar.

A:MATCH-L: It is the same as the case of a conditional expression. The evaluation derivation applies one of the rules E:MATCH-N and E:MATCH-L depending on the value of x.

Assume that x is assigned the value $[v_1, ..., v_n]$ under E, or $E(x) = [v_1, ..., v_n]$. Then, the evaluation derivation ends with an application of the rule E:MATCH-L. Let $E_1 = E[x_h \mapsto v_1, x_t \mapsto [v_2, ..., v_n]]$ and $\Gamma_1^r = \Gamma^r, x_h : A, x_t : L^{p_1}(A)$, the typing derivation ends with an application of the rule A:MATCH-L, thus $\Sigma^r; \Gamma_1^r \left|\frac{q + p_1 - K^{\mathsf{matchL}}}{q'}\right. e_2 : A_1$.

Since $\models [v_1, ..., v_n] : L^{p_1}(A)$, we have $\models v_i : A, \forall i = 1, ..., n$. Hence, it holds that $\models v_1 : A$ and $\models [v_2, ..., v_n] : L^{p_1}(A)$. Finally, we have $\models E_1 : \Gamma_1^r$ (since $\models E : \Gamma^r$ implies $\models E_1 : \Gamma^r$).

For all $p, p' \in \mathbb{Q}_0^+$ such that $E \left|\frac{p}{p'}\right. e \Downarrow v$, by the rule E:MATCH-L we have $E_1 \left|\frac{p - K^{\mathsf{matchL}}}{p'}\right. e_2 \Downarrow v$. By the induction hypothesis for e_2, it holds that $q + p_1 - K^{\mathsf{matchL}} + \Phi_{E_1}(\Gamma_1^r) - (q' + \Phi(v : A_1)) \leq p - K^{\mathsf{matchL}} - p'$.

Because $\Phi_E(\Gamma^r, x : L^p(A)) = \Phi_E(\Gamma^r) + n.p_1 + \Sigma_{i=1}^n \Phi(v_i : A)$, $\Phi_{E_1}(\Gamma_1^r) = \Phi_{E_1}(\Gamma^r) + (n-1).p_1 + \Sigma_{i=1}^n \Phi(v_i : A)$ and $\Phi_{E_1}(\Gamma^r) = \Phi_E(\Gamma^r)$, thus we have $\Phi_{E_1}(\Gamma_1^r) = \Phi_E(\Gamma^r, x : L^{p_1}(A)) - p_1$. Therefore, $q + \Phi_E(\Gamma^r, x : L^{p_1}(A)) - (q' + \Phi(v : A_1)) \leq p - p'$. If $E(x) = \mathsf{nil}$ then it is similar to the case A:MATCH-P.

A:LET: Assume that e is an expression of the form let$(x, e_1, x.e_2)$. Hence, the evaluation derivation ends with an application of the rule E:LET. Let $E_1 = E[x \mapsto v_1]$ and $\Gamma^r = \Gamma_1^r, \Gamma_2^r$. The typing derivation ends with an application of the rule A:LET, thus $\Sigma^r; \Gamma_1^r \left|\frac{q - K^{\mathsf{let}}}{q_1'}\right. e_1 : A_1$ and $\Sigma^r; \Gamma_2^r, x : A_1 \left|\frac{q_1'}{q'}\right. e_2 : A_2$.

For all $p, p' \in \mathbb{Q}_0^+$ such that $E \left|\frac{p}{p'}\right. e \Downarrow v$, by the rule E:LET we have $E \left|\frac{p - K^{\mathsf{let}}}{p_1'}\right. e_1 \Downarrow v_1$ and $E_1 \left|\frac{p_1'}{p'}\right. e_2 \Downarrow v$. Since $\models E : \Gamma^r$, we have $\models E : \Gamma_1^r$. By the induction hypothesis for e_1, it holds that $q - K^{\mathsf{let}} + \Phi_E(\Gamma_1^r) - (q_1' + \Phi(v_1 : A_1)) \leq p - K^{\mathsf{let}} - p_1'$.

We have $\models E : \Gamma_2^r$, thus $\models E_1 : \Gamma_2^r, x : A_1$. Again by the induction hypothesis for e_2, we derive that $q_1' + \Phi_{E_1}(\Gamma_2^r, x : A_1) - (q' + \Phi(v : A_2)) \leq p_1' - p'$.

Sum two in-equations above, it follows $q + \Phi_E(\Gamma_1^r) - \Phi(v_1 : A_1) + \Phi_{E_1}(\Gamma_2^r, x : A_1) - (q' + \Phi(v : A_2)) = q + \Phi_E(\Gamma_1^r, \Gamma_2^r) - (q' + \Phi(v : A_2)) \leq p - p'$.

L:SUBTYPE: Assume that the typing derivation ends with an application of the rule L:SUBTYPE, thus $\Sigma^r; \Gamma^r \left|\frac{q}{q'}\right. e : A$ and $A <: B$.

By the induction hypothesis for e in the premise, for all $p, p' \in \mathbb{Q}_0^+$ such that $E \left|\frac{p}{p'}\right. e \Downarrow v$ it holds that $q + \Phi_E(\Gamma^r) - (q' + \Phi(v : A)) \leq p - p'$.

Because $\Phi(E(x) : A) \leq \Phi(E(x) : B)$ we have $q + \Phi_E(\Gamma^r) - (q' + \Phi(v : B)) \leq p - p'$.

L:SUPERTYPE: Assume that the typing derivation ends with an application of the rule L:SUPERTYPE, thus $\Sigma^r; \Gamma^r, x : B \left|\frac{q}{q'}\right. e : C$ and $A <: B$. Since $\models E : \Gamma^r, x : A$ and following the property of the subtyping relation we have $\models E : \Gamma^r, x : B$.

By the induction hypothesis for e in the premise, for all $p, p' \in \mathbb{Q}_0^+$ such that $E \left|\frac{p}{p'}\right. e \Downarrow v$ it holds that $q + \Phi_E(\Gamma^r, x : B) - (q' + \Phi(v : C)) \leq p - p'$.

Because $\Phi(E(x) : A) \leq \Phi(E(x) : B)$ we have $q + \Phi_E(\Gamma^r, x : A) - (q' + \Phi(v : C)) \leq p - p'$.