Linear Capabilities for Fully Abstract Compilation of Separation-Logic-Verified Code

Technical Report

THOMAS VAN STRYDONCK, KU Leuven, Belgium

FRANK PIESSENS, KU Leuven, Belgium

DOMINIQUE DEVRIESE, Vrije Universiteit Brussel, Belgium

This technical report accompanies the JFP 2020 submission *Linear Capabilities for Fully Abstract Compilation of Separation-Logic-Verified Code* and is an updated version of the technical report accompanying the ICFP 2019 paper of the same name [Van Strydonck et al. 2019]. It aims to lay out the full abstraction proof summarized there in all details here. In other words, this report proves in all detail that our compilation from the subset of separation-logic-verified C-code to linear capability-enhanced unverified C code is fully abstract. The proof is structured as follows. First, section 1 describes the grammar of the source and target languages. These grammars build statements out of expressions, components out of statements and programs out of components. Before detailing the operational semantics of source and target programs in section 2, section 2.1 first explains how the smallest building blocks, namely expressions, behave. This includes, but is not limited to, the semantics regarding source and target language expression. Next, section 3 describes how source-level separation logic proofs are constructed by providing all possible separation logic inference rules (or axioms). Section 4 builds further upon these separation logic axioms and shows how the compiler compiles all different source statements, components and entire programs in a separation-logic-proof-directed fashion, by using the separation logic axioms from the previous section as input. After the compiler has been detailed in this section, the bulk of the proof can finally commence. Section 5 defines what exact formulation of full abstraction we are proving and dissects the proof, before diving into it. Section 6 proves the correctness direction of the fully abstract compilation by constructing a simulation relation and proving that any well-typed source program and its compilation perform a simulation. Section 7 proves the security direction of the fully abstract compilation by constructing a back-translation (a kind of compilation, but now from target to source) and another simulation relation, after which simulation between target and source can again be proven.

Authors' addresses: Thomas Van Strydonck, KU Leuven, Belgium, thomas.vanstrydonck@cs.kuleuven.be; Frank Piessens, KU Leuven, Belgium, frank.piessens@cs.kuleuven.be; Dominique Devriese, Vrije Universiteit Brussel, Belgium, dominique.devriese@vub.be.

Contents

1 GRAMMAR

This section details the legal input programs in both the source and target languages by means of a grammar in BNF. Both grammars build statements out of expressions, components out of statements and programs out of components. The source language also contains functions annotated with separation logic contracts, for which a third kind of expression *exp* is needed, next to the regular source and target expressions *sexp* and *texp*, as can be seen below.

In the below grammar and the rest of the technical report k denotes an integer, id_{log} a logical variable identifier, id_{prog} a program variable identifier, f a function identifier, τ_{s} a source type identifier, τ_{t} a target type identifier and n a heap chunk identifier. During execution, the set of all previously used program variables is from now on denoted ID_{prog}, whereas the set of all logical variables is denoted ID_{log}.

Chunks can either represent a single variable-length array resource, or encode an entire range of resources and/or pure conditions. This second case is useful when we have universal quantification over heap resources, ie. in the case where we do not know the length of an array resource that has nested array resources. It is then not possible to represent each nested array resource individually, and we are forced to bundle them together in a containing structure. The expression $exp \mapsto_\tau exp$ represents a flat array resource, and $[assert \mid exp \leq i < exp]$ a range resource. We also define the simplified notation $exp \mapsto_\tau [exp_1, \ldots, exp_k]$, which desugars to $exp \mapsto_\tau l * \overline{l[i] = exp_i} * \text{length}(l) = k$, to represent fixed-length array resources.

An important restriction on range resources is that we cannot allow commutativity on different impure parts of the range resource, because of the way these resources are reified. The resource $[a : b \mapsto [\text{int}] * c : d \mapsto [\text{int}*] \mid \ldots]$ will reify to $(\text{int}*, \text{int}**)$, whereas the resource $[c : d \mapsto [\text{int}*] * a : b \mapsto [\text{int}] \mid \ldots]$ reifies to $(\text{int}**, \text{int}*)$. Both are hence not simply interchangeable. If we would define an order on resources, we could allow some form of commutativity, but we choose the simpler path here.

Chunks are assumed named using a heap chunk identifier n ; this is needed in the compilation process later on. We use the syntax n : to denote a single name, and $\overline{n}$: to denote a tuple of names. If a name n does not appear in the precondition of a Hoare triple, it is assumed to be a fresh name. Chunks are assumed named in the programmer's code as well, seen as chunk names also appear in separation logic contracts. We assume the existence of a relation $assert \approx_{\text{Names}} assert$ ranging over two *assert*s, that is true if the assertions are identical up to chunk name substitution. We also define a function CN(*assert*) (short for ChunkNames) that returns all chunk names present in a separation logic assertion as a tuple in order of appearance.

We could introduce a convenience-based chunk renaming rule, to allow eg. the Hoare triple $\{\overline{n} : chunks\}$ skip $\{\overline{m} : chunks\}$ to be a sound separation logic proof. This chunk renaming will be handled by the classical separation logic Consequence rule.

⟨op1⟩	::=		(Unary Operators)
		−	*negative*
	|	!	*negation*
⟨op2⟩	::=		(Binary Operators)
		!=	*inequality*
	|	==	*equality*
	|	+	*addition*

	$\mid \wedge$	*conjunction*
	$\mid \vee$	*disjunction*
$\langle exp \rangle$	::=	(Logical Expressions)
	id_{log}	*logical var*
	$\mid k$	*int*
	$\mid$ op1 *exp*	*unary op*
	$\mid$ *exp* op2 *exp*	*binary op*
	$\mid$ null	*logical null pointer*
	$\mid (exp^*)$	*tuple creation*
	$\mid exp.k$	*tuple projection*
	$\mid$ length(*exp*)	*logical list length*
	$\mid exp[exp]$	*logical list indexing*
	$\mid$ emp	*empty logical list*
	$\mid \text{cons}(id_{\text{log}}, id_{\text{log}})$	*logical list cons*
	$\mid \forall id_{\text{log}} : \tau.\ exp$	*forall*
	$\mid \exists id_{\text{log}} : \tau.\ exp$	*exists*
	$\mid \ldots$	*extensions allowed*
$\langle sexp \rangle$	::=	(Source Expressions)
	id_{prog}	*program var*
	$\mid k$	*int*
	$\mid$ op1 *sexp*	*unary op*
	$\mid$ *sexp* op2 *sexp*	*binary op*
	$\mid \text{null}_0$	*null pointer*
	$\mid (sexp^*)$	*tuple creation*
	$\mid sexp.k$	*tuple projection*
$\langle texp \rangle$	::=	(Target Expressions)
	id_{prog}	*program var*
	$\mid k$	*int*
	$\mid$ op1 *texp*	*unary op*
	$\mid$ *texp* op2 *texp*	*binary op*
	$\mid$ null	*null pointer*
	$\mid \text{null}_0$	*address null pointer*
	$\mid$ addr(*texp*)	*address of pointer*
	$\mid$ length(*texp*)	*length of linear capability*
	$\mid (texp^*)$	*tuple creation*
	$\mid texp.k$	*tuple projection*
$\langle sstm \rangle$	::=	(Source Statements)
	skip	*skip*
	$\mid id_{\text{prog}} = \text{malloc}(sexp * \text{sizeof}(\tau_s))$	*malloc statement*
	$\mid$ //@split n[*sexp*]	*ghost split*
	$\mid$ //@join n n	*ghost join*
	$\mid$ //@flatten n	*ghost flatten*
	$\mid$ //@collect $n^* \cdot \ldots \cdot n^*$	*ghost collect*
	$\mid$ foreach($sexp \le i < sexp$){*sstm*}	*foreach loop*
	$\mid$ *sstm*; *sstm*	*sequencing*

		$\mid$ if $sexp$ then $sstm$ else $sstm$	*if*
		$\mid$ τ id_{prog}	*variable decl*
		$\mid$ $id_{\text{prog}} = sexp$	*var assign*
		$\mid$ $(id^*_{\text{prog}}) = f(sexp^*)$	*function app*
		$\mid$ $id_{\text{prog}}[sexp] = sexp$	*array mut*
		$\mid$ $id_{\text{prog}} = sexp[sexp]$	*array lookup*
		$\mid$ guard($sexp$)	*guarding*
$\langle tstm \rangle$	::=		(Target Statements)
		skip	*skip*
		$\mid$ $id_{\text{prog}} = \text{malloc}(texp * \text{sizeof}(\tau_{\text{t}}))$	*malloc statement*
		$\mid$ $(id_{\text{prog}}, id_{\text{prog}}) = \text{split}(id_{\text{prog}}, texp)$	*code split*
		$\mid$ $id_{\text{prog}} = \text{join}(id_{\text{prog}}, id_{\text{prog}})$	*code join*
		$\mid$ foreach$(texp \le i < texp)\{tstm\}$	*foreach loop*
		$\mid$ $tstm$; $tstm$	*sequencing*
		$\mid$ if $texp$ then $tstm$ else $tstm$	*if*
		$\mid$ τ id_{prog}	*variable decl*
		$\mid$ $id_{\text{prog}} = texp$	*var assign*
		$\mid$ $(id^*_{\text{prog}}) = f(texp^*)$	*function app*
		$\mid$ $id_{\text{prog}}[texp] = texp$	*array mut*
		$\mid$ $id_{\text{prog}} = texp[texp]$	*array lookup*
		$\mid$ guard($texp$)	*guarding*
$\langle assert \rangle$	::=		(Outer Separation Logic Assertion)
$\langle assert \rangle$	::=		(Inner Separation Logic Assertion)
		exp	*expression*
		$\mid$ $assert * assert$	*sep conjunction*
		$\mid$ exp ? $assert$	*conditional assertion*
		$\mid$ $n : exp \mapsto_{\tau_{\text{s}}} exp$	*array resource*
		$\mid$ $n : [assert \mid exp \le id_{\text{log}} < exp]$	*range resource*
		$\mid$ $exp \mapsto_{\tau_{\text{s}}} exp$	*array resource*
		$\mid$ $[assert \mid exp \le id_{\text{log}} < exp]$	*range resource*
$\langle \tau \rangle$	::=		(Logical Type)
		int	*integer*
		$\mid$ $\tau*$	*pointer*
		$\mid$ (τ^*)	*tuple*
		$\mid$ list_τ	*list*
$\langle \tau_{\text{s}} \rangle$	::=		(Source Type)
		int	*integer*
		$\mid$ $\tau_{\text{s}}*$	*pointer*
		$\mid$ (τ^*_{s})	*tuple*
$\langle \tau_{\text{t}} \rangle$	::=		(Target Type)
		int	*integer*
		$\mid$ $\tau_{\text{t}}*$	*pointer*
		$\mid$ (τ^*_{t})	*tuple*
		$\mid$ $\tau_{\text{t}}*_0$	*0-length address capability*

$\langle isfunc \rangle$ ::= (Implemented Source Function)
$\tau_s^*\ f((\tau_s\ id)^*)$ //@pre *assert* //@post *assert*
$\{sstm; \text{return } sexp^*\}$

$\langle csfunc \rangle$ ::= (Context Source Function)
$\tau_s^*\ f((\tau_s\ id)^*)$ //@pre *assert* //@post *assert*

$\langle itfunc \rangle$ ::= (Implemented Target Function)
$\tau_t^*\ f((\tau_t\ id)^*)\ \{tstm; \text{return } texp^*\}$

$\langle ctfunc \rangle$ ::= (Context Target Function)
$\tau_t^*\ f((\tau_t\ id)^*)$

$\langle scomp \rangle$ ::= (Source Component)
$isfunc^+$ //@import $csfunc^*$ //@export $csfunc^*$

$\langle sprog \rangle$ ::= (Source Program)
$scomp^+$ //@main = id

$\langle tcomp \rangle$ ::= (Target Component)
$itfunc^+$ //@import $ctfunc^*$ //@export $ctfunc^*$

$\langle tprog \rangle$ ::= (Target Program)
$tcomp^+$ //@main = id

Note that neither the source nor the target language contains a *free* statement, although both contain a *malloc* statement. Free has been removed, because it required (to uphold the C semantics of only being able to free a block that has been malloced and only if the current pointer points to the first word of data of that block) a *malloc chunk* in separation logic, essentially a simple abstract predicate, that had to be compiled into a sealed capability. This, however, caused problems during the back-translation, as operations on this chunk had to be representable as well. Also, as adding in the free statement essentially boils down to adding support for abstract predicates that compile to sealed capabilities, we leave this bit for when we extend the current formalization with a more general form of abstract predicates. From this discussion, the formalization of the free statement will then follow more easily. Removing the free statement entails that we have to allow leaking of separation logic heap chunks, and our separation logic axioms (particularly the Consequence rule, as will be apparent later) should hence allow for this.

Note that there is no explicit boolean type present, neither in the source language, nor in the target languages. The booleans are assumed embedded within the integers, where any value of 0 is considered false and all other integers are considered true. This is a minor technicality that makes the formalization easier in a couple respects. Pair types are included in both the source and target language to make the back-translation possible. The target-level addr function takes a pointer of type $\tau*$ and returns an $\tau*_0$-value for the same location that describes the address of this pointer. This function is needed to be able to tell whether a target-level linear capability and a target-level length-0 capability correspond to the same address. It is back-translated to left projection.

Note that the addr function can also be applied to e.g. the type $\tau\ *_0\ *$ of a reified resource with integer pointer contents and would then return a $\tau\ *_0\ *_0$-type address.

All chunk names, program variable names and logical variable names are assumed mutually disjoint throughout the whole program. This makes some compilation rules easier because it avoids name

clashes between the two groups of identifiers. For all input programs (both in target and source), static type checking is assumed to be performed. We assume that this includes checking for redeclarations of variables or usages of undeclared variables (although these first 2 checks can be left out without consequences, as they are worked into the operational semantics too), disjointness of variable names and chunk names, types of expressions used as input to statements and received back from statements to be assigned to variables. Contracts are type-checked as well insofar possible, as we do not want contracts of eg. the following form: $a \mapsto [b, c] * b == 2 * c \mapsto [\_]_L$.

Another important remark is that we do not strictly need the above foreach loop: every instance of foreach in this report can be replaced by a call to a generated, recursive function that performs the same task. This is, however, a less elegant and concise solution. Additionally, foreach loops do not cause a lot of difficulties during the formalization, which is why they are employed.

We subscript null-pointers that correspond to the address of a 0-length address capability, rather than a regular pointer, with a 0, as some inference rules will need this. Logical pointers work like 0-length address capabilities and are hence subscripted in this way as well. This forms no actual language enhancement, as this type information would be available from the type checker anyway.

To be able to handle unknown-length separation logic lists, we introduce logical variables of list type. These logical variables have type list_τ, have a length attribute associated with them, are prepended with values using cons, allow indexing $[i]$ and the following Consequence rules hold for them:

$$
\begin{aligned}
&l : \text{list}_\tau \Vdash \text{length}(l) = n \wedge n > 0 \Leftrightarrow \\
&\qquad \exists x : \tau, l' : \text{list}_\tau . \, l = \text{cons}(x, l') \wedge \text{length}(l') = n - 1 \\
&l : \text{list}_\tau \Vdash \text{length}(l) = 0 \Leftrightarrow l = ()
\end{aligned}
$$

We also define the following functions as a shorthand (inspired by Z3), to make the notation throughout this report easier to read:

$$
\begin{aligned}
&l : \text{list}_\tau \Vdash l == \text{repeat}(k, exp) \Leftrightarrow \\
&\qquad \forall 0 \leq i < \text{length}(l). \, l[i] == k \wedge \\
&\qquad \text{length}(l) == exp \\
&l : \text{list}_\tau, l' : \text{list}_\tau, l'' : \text{list}_\tau \Vdash l == \text{append}(l', l'') \Leftrightarrow \\
&\qquad \forall 0 \leq i < \text{length}(l'). \, l[i] = l'[i] \wedge \\
&\qquad \forall \, \text{length}(l') \leq i < \text{length}(l') + \text{length}(l''). \, l[i] = l''[i - \text{length}(l')] \wedge \\
&\qquad \text{length}(l) = \text{length}(l') + \text{length}(l'') \\
&l : \text{list}_\tau, l' : \text{list}_\tau \Vdash l == \text{take}(l', k_1, k_2) \Leftrightarrow \\
&\qquad \forall k_1 \leq i < k_2 . \, l[i - k_1] = l'[i] \wedge \\
&\qquad \text{length}(l) == k_2 - k_1 \\
&l : \text{list}_\tau, l' : \text{list}_\tau \Vdash l == \text{update}(l', k, exp) \Leftrightarrow \\
&\qquad l[k] == exp \wedge \\
&\qquad \forall 0 < i < k . \, l[i] == l'[i] \wedge \\
&\qquad \forall k < i < \text{length}(l'). \, l[i] == l'[i]
\end{aligned}
$$

$$\text{length}(l) == \text{length}(l')$$

The Consequence rule allows switching between the logical functions and their implementations in both directions. The above definitions can also be instantiated with logical expressions *exp*, *exp′* and *exp″* instead of logical lists l, l' and l'' and integers k. Note that the above definitions also allow using any of the introduced shorthands as expressions, because any separation logic assertion $P(\mathit{shorthand})$ can always be rewritten as $P(l) * \mathit{shorthand} == l$ with l fresh, through the use of the use of the Consequence rule.

For the entirety of this report, we assume the existence of a TypeOfVar meta-function, which, given the name of a variable, returns its type (if it is given a non-variable, it returns int, because pointer arithmetic or other pointer operations are not present in the target). The variable type is entirely statically known and hence forms no limiting assumption for the formalization. The function TypeOfVar can both be used on the source, target level and the context usually clarifies the applied version.

2 OPERATIONAL SEMANTICS

Now that the previous section has detailed how to manipulate expressions, we have the tools to formalize the behavior of larger grammatical entities in our source and target language. This section describes the operational semantics of source and target level statements, components and entire programs. The first subsections defines operational semantics for the source language, whereas the second section details operational semantics for the target language.

A component-wise environment Σ_i of functions in component C_i is assumed implicitly given for both the target and the source languages. A similar assumption is made in the next section when formalizing the compilation. This environment Σ_i is indexed by function name and contains all information on functions, namely the separation-logic contracts in the case of source functions, the tuple of return types, the tuple of argument types with argument identifiers, the type of function: component-internal/exported/imported and the function body in the case of a non-imported function. The environment Σ_i can easily be automatically parsed from a given source program consisting of components $C_1, \ldots, C_m$, because each component contains a full declaration of its own functions and of the functions it exports and imports.

Since we are executing a whole program, consisting of different components, scoping comes into play and has to be factored in when executing the program, especially during function calls. Assume the program P consists of components C_1,…,C_i, …,C_k, all with their own function environment Σ_i. The different component environments are assumed completely disjoint for a given source or target program, as will be apparent from the separation logic axioms in the next section. For the operational semantics, we just need the overall operational program environment Σ_{op}, defined as the union of the exported function parts of the function environments Σ_i where only the information on function bodies and arguments is kept, as all checks concerning proper function application are already performed in the separation logic axioms. We also define the functions $\text{PRE}(f)$ and $\text{POST}(f)$, which return a function's pre-, respectively postcondition, as we will need this for the FApp and Return rules in the operational semantics. In the separation logic axioms, the separate Σ_i are used, but the function bodies are not needed and hence dropped there. The resulting environments are denoted Σ_i^{ax} and if clear from the context which component environment we mean, we abbreviate to simply Σ or drop the environment entirely.

Before the operational semantics rules can be given, some notation technicalities need to be delineated first. As usually the case with operational semantics, our semantics will make use of a stack $\overline{s}$ and a heap h. The stack is a list of partial functions from the set of program variables ID_{prog} to values of source or target types (depending on the language we are defining semantics for). The partial functions are called *stack frames* and the full stack is split into stack frames using the concatenation notation $\overline{s} = s :: \overline{s'}$. Evaluation of a source or target expression e in the current stack frame s is denoted $[\![e]\!]_s$, as mentioned in the previous section. The heap h is a partial function from a pair value (l, i), with l a location from a set of locations L and i an integer index. We write $h(l) = [v_0, \ldots, v_m]$ or $l \rightarrow [v_0, \ldots, v_m] \in h$ to express that heap h, at location l, contains the value v_j at index j ($0 \leq j \leq m$; heap allocation always starts at index 0). Alternatively, the notation $h(l, j) = v_j$ can be used to single out one of these values, and (l, j) to pinpoint any one heap value. The use of locations makes the formalization easier, because separately malloced variables are logically separate as well.

Given that the stack is a list of partial functions with as range the values of source or target types, we need to have a notion of what the values that populate types are. The values $v_s \in \text{VAL}_s$ that populate the source types τ_s are:

- Integers k for int
- Heap location-index pairs (l,i) and null_0 for pointers τ_s*
- Pairs $(v_{s1},\ldots,v_{sk})$ for types $(\tau_{s,1},\ldots, \tau_{s,k})$ with v_{s1} a value of type $\tau_{s,1},\ldots, v_{sk}$ a value of type $\tau_{s,k}$

The values $v_t \in \text{VAL}_t$ that populate the source types τ_t are:

- Integers k for int
- Pointers in the target language will always be linear capabilities (except if they have length 0, see below), and should hence be formalized as more than just a heap location-index pair, as is the case in the source. We choose the notation $l^{[i_1, i_2]}$ (with $i_1 \leq i_2$) to denote that there is a capability pointing to heap location l with permissions for the region in the closed heap address interval $[i_1, i_2]$. There is no need to keep track of the index a linear capability currently points to, because of the way integer arithmetic is handled during compilation; the memory location a capability points to is never altered, but the index used for dereferencing it is. Pointer arithmetic can hence just be disallowed in the target language. We hence have heap location-index pair values $l^{[i_1, i_2]}$ for pointers τ_t*, and the null-pointer null.
- Pairs $(v_{t1},\ldots,v_{tk})$ for types $(\tau_{t,1},\ldots, \tau_{t,k})$ with v_{t1} a value of type $\tau_{t,1},\ldots,$ and v_{tk} a value of type $\tau_{t,k}$
- Capabilities that cannot be dereferenced anywhere are so-called *length-0 capabilities* and are of type τ_s*_0. These types of values are not linear, as they do not allow any permissions on the heap anyway. Values of length 0 capabilities do have an index, contrary to the regular target-level linear capabilities, as we need to perform pointer arithmetic on them. Length-0 capabilities are not linear and do not have a range because they cannot be dereferenced anywhere. Hence the notation l_0^i, with l the location and i the index of the capability, for values of type τ_t*. We also have the null-pointer null_0.

For the small-step operational semantics, we use the notation $\langle \overline{s}, h \rangle \mid \overline{c}$ to describe a state, with $\overline{s}$ the stack, modeled as a list of stack frames, h the heap and $\overline{c}$ a list of the currently executing functions bodies. Each stack frame s_i will correspond to a currently executing function body c_i. The splitting of the executing code into executing function bodies c_i and the correspondence between these c_i and stack frames s_i will make the correctness proof of section 6 easier to perform compared

to the case where the executing code is not split up. A transition from program state $\langle \overline{s}, h\rangle \mid \overline{c}$ to program state $\langle \overline{s'}, h'\rangle \mid \overline{c'}$ by means of the small-step operational semantics will be denoted using an inference rule as follows:

$$\frac{Precondition}{\langle \overline{s}, h\rangle \mid \overline{c} \hookrightarrow \langle \overline{s'}, h'\rangle \mid \overline{c'}} \quad \text{(RuleName)}$$

For every statement present in the grammar of the source and target language in section 1, there will be at least one rule in the operational semantics of the respective language. There is also an extra rule ProgExec for the execution of entire programs.

The sequencing statement is the only statement that cannot be immediately executed in a top-down fashion, as it is our only statement that contains other statements. To avoid having a Sequence congruence rule, we define an *evaluation context* to capture the non-immediately executable statements. The evaluation context does not have a single hole as is usually the case, but it has a hole for every function body c_i currently in execution. The evaluation context is formalized as follows, where the return statements denote a transition in stack frame s_i and in body c_i and stm_{lan} is a *sstm* for the source language and a *tstm* for the target language:

$$\langle F\rangle \quad ::= \text{return } exp \mid stm_{\text{lan}}; F \quad \text{(Executing Component)}$$

$$\langle C\rangle \quad ::= \emptyset \mid C :: F \quad \text{(Execution Context)}$$

In the first case of Executing Component, the hole will always be filled with a return statement, whereas in the second case, a regular non-return statement will fill the hole. This will be apparent from the operational semantics below.

Nonsensical input programs in both target and source get stuck because no operational semantics rules apply. There is no such thing as en *error* state in our formalization.

Notation and rules for the operational semantics are based on the notation of Agten et al. in their paper on sound modular verification of C code [Agten et al. 2015].

2.1 Expression evaluation

This section details how expressions are formally evaluated. With the inclusions of pairs to the expressions in source and target language, it becomes increasingly important to properly define the semantics of expression evaluation with respect to the current stack frame s, denoted $[\![\cdot]\!]_s$, in both source and target language.

We make abstraction of the difference between *sexp* and *texp* and denote both by *exp*, except for the parts where a distinction has to be made.

$$\frac{s(id_{\text{prog}}) = v}{[\![id_{\text{prog}}]\!]_s = v} \quad \text{(ProgramVar)}$$

$$\frac{}{[\![k]\!]_s = k} \quad \text{(Int)}$$

$$\frac{[\![exp]\!]_s = v_1 \qquad \mathrm{op1}(v_1) = v_2}{[\![\mathrm{op1}\ \ exp]\!]_s = v_2} \tag{UnaryOp}$$

$$\frac{[\![exp_1]\!]_s = v_1 \qquad [\![exp_2]\!]_s = v_2 \qquad \mathrm{op2}\ (v_1, v_2) = v_3}{[\![exp_1\ \ \mathrm{op2}\ \ exp_2]\!]_s = v_3} \tag{BinaryOp}$$

$$\frac{}{[\![\mathrm{null}]\!]_s = \mathrm{null}} \tag{Null}$$

$$\frac{}{[\![\mathrm{null}_0]\!]_s = \mathrm{null}_0} \tag{Null0}$$

This rule is target language only:

$$\frac{[\![exp]\!]_s = v_1 \qquad \mathrm{addr}(v_1) = v_2}{[\![\mathrm{addr}(exp)]\!]_s = v_2} \tag{PtrAddress}$$

This rule is target language only:

$$\frac{[\![exp]\!]_s = l^{[a,b]} \qquad v = b - a + 1}{[\![\mathrm{length}(exp)]\!]_s = v} \tag{LincapLength}$$

$$\frac{[\![exp_1]\!]_s = v_1 \qquad [\![exp_2]\!]_s = v_2}{[\![(exp_1, exp_2)]\!]_s = (v_1, v_2)} \tag{PairCreate}$$

$$\frac{[\![exp]\!]_s = (v_1, \_)}{[\![exp.1]\!]_s = v_1} \tag{LeftProject}$$

$$\frac{[\![exp]\!]_s = (\_, v_2)}{[\![exp.2]\!]_s = v_2} \tag{RightProject}$$

2.2 Source Language

$$\frac{}{\langle \overline{s}, h\rangle \mid C :: \mathrm{skip}; F \hookrightarrow \langle \overline{s}, h\rangle \mid C :: F} \tag{Skip}$$

$$\frac{}{\mathrm{int} \rightsquigarrow_{\mathrm{def}} 0} \tag{DefInt}$$

For the following rule, there is a difference between the source and target languages, because a pointer only represents a permission in the target language.

$$\frac{}{\tau* \rightsquigarrow^{\mathrm{src}}_{\mathrm{def}} \mathrm{null}_0} \tag{DefPtrSrc}$$

$$\frac{}{\tau* \rightsquigarrow^{\mathrm{tgt}}_{\mathrm{def}} \mathrm{null}} \tag{DefPtrTgt}$$

The following case is only used in the target language:

$$\frac{}{\tau *_0 \rightsquigarrow_{\text{def}} \text{null}_0} \quad \text{(DefSrcPtr)}$$

$$\frac{\begin{array}{c}\tau_1 \rightsquigarrow_{\text{def}} def_1 \\ \dots \\ \tau_k \rightsquigarrow_{\text{def}} def_k\end{array}}{(\tau_1, \dots, \tau_k) \rightsquigarrow_{\text{def}} (def_1, \dots, def_k)} \quad \text{(DefTuple)}$$

We also define the corresponding function $\text{NULL}(\tau)$ that we define as $\text{NULL}(\tau) = v \Leftrightarrow \tau \rightsquigarrow_{\text{def}} v$. This function can be used both in the target and the source language, and is appropriately annotated if the concrete usage is unclear, eg. in the source language we would then write $\text{NULL}^{\text{src}}(\tau)$.

As seen in the below rule, any malloc'ed pointer starts out at index 0 of the heap location it is malloced at.

$$\frac{\begin{array}{cc}\overline{s} = s :: \overline{s_t} & \overline{s'} = s' :: \overline{s_t} \\ s' = s[id_{\text{prog}} \to (l, 0)] & l \notin \text{dom}(h) \quad \tau \rightsquigarrow_{\text{def}} v \\ [\![sexp]\!]_s = k \quad k > 0 & h' = h[l \to [v, \dots, v]_k]\end{array}}{\langle \overline{s}, h\rangle \mid C :: id_{\text{prog}} = \text{malloc}(sexp * \text{sizeof}(\tau)); F \hookrightarrow \langle \overline{s'}, h'\rangle \mid C :: \text{skip}; F} \quad \text{(Malloc)}$$

Skip and join only have an effect on the separation logic proof and hence do not have any effect on the source-level operational semantics.

$$\frac{}{\langle \overline{s}, h\rangle \mid C :: \text{//@split } n_1[sexp_2]; F \hookrightarrow \langle \overline{s}, h\rangle \mid C :: \text{skip}; F} \quad \text{(Split)}$$

$$\frac{}{\langle \overline{s}, h\rangle \mid C :: \text{//@join } n_1\ n_2; F \hookrightarrow \langle \overline{s}, h\rangle \mid C :: \text{skip}; F} \quad \text{(Join)}$$

$$\frac{}{\langle \overline{s}, h\rangle \mid C :: \text{//@flatten } n_1; F \hookrightarrow \langle \overline{s}, h\rangle \mid C :: \text{skip}; F} \quad \text{(Flatten)}$$

$$\frac{}{\langle \overline{s}, h\rangle \mid C :: \text{//@collect } \overline{n_{0j}} \cdot \dots \cdot \overline{n_{(k-1)j}}; F \hookrightarrow \langle \overline{s}, h\rangle \mid C :: \text{skip}; F} \quad \text{(Collect)}$$

$$\frac{\overline{s} = s :: \overline{s_t} \qquad [\![sexp_1]\!]_s = \text{true}}{\langle \overline{s}, h\rangle \mid C :: \text{if } sexp_1 \text{ then } sstm_1 \text{ else } sstm_2; F \hookrightarrow \langle \overline{s}, h\rangle \mid C :: sstm_1; F} \quad \text{(IfTrue)}$$

$$\frac{\overline{s} = s :: \overline{s_t} \qquad [\![sexp_1]\!]_s = \text{false}}{\langle \overline{s}, h\rangle \mid C :: \text{if } sexp_1 \text{ then } sstm_1 \text{ else } sstm_2; F \hookrightarrow \langle \overline{s}, h\rangle \mid C :: sstm_2; F} \quad \text{(IfFalse)}$$

To declare a variable, it shouldn't have been declared yet.

$$\frac{\begin{array}{cc}\overline{s} = s :: \overline{s_t} & \overline{s'} = s' :: \overline{s_t} \\ \tau \rightsquigarrow_{\text{def}} v \qquad & s' = s[id_{\text{prog}} \to v]\end{array}}{\langle \overline{s}, h\rangle \mid C :: \tau\ id_{\text{prog}}; F \hookrightarrow \langle \overline{s'}, h\rangle \mid C :: \text{skip}; F} \quad \text{(VarDecl)}$$

To assign a variable, it has to already have been declared earlier.

$$\frac{\begin{array}{cc}\overline{s} = s :: \overline{s_t} & \overline{s'} = s' :: \overline{s_t} \\ s' = s[id_{\text{prog}} \rightarrow v] & [\![sexp]\!]_s = v\end{array}}{\langle \overline{s}, h\rangle \mid C :: id_{\text{prog}} = sexp; F \hookrightarrow \langle \overline{s'}, h\rangle \mid C :: \text{skip}; F} \quad \text{(VarAsgn)}$$

The approach for the FApp rule below follows the usual operational semantics-formalization of function calls. Usually, after execution of a function call of the form $\overline{id_{\text{prog}}} = f(\overline{sexp})$, the concrete function identifier and the supplied arguments are not needed anymore and replaced with a hole. To avoid notational confusion with the context C and its holes, we will denote a hole as $\bullet$ instead of $[\cdot]$ in this technical report. The result of the function call would be the following statement: $\overline{id_{\text{prog}}} = \bullet$. This would suffice for the proof, *if* we were compiling unverified source code.

This does not suffice, however, to perform the full abstraction proof in our case, since we have to incorporate the fact that our source code is verified and remains verified while running it (ie. we use a type of *lifted operational semantics*). In the full abstraction proof, it will be necessary to specify the separation-logic proof of a partly-executed program, which includes programs with a code-level hole (such programs cannot be written by a programmer, but they can result from execution). The next paragraphs discuss how to handle this case, but use a lot of concepts from further on in the report. The reader is advised to skip this discussion for now, and neglect the lifted semantics (green) parts of the rules, sticking to the simplified versions for now and revisiting the proof-related extensions when reading about the correctness proof in section 6.

To be able to make a separation logic proof for a partly executed program, a hole $\bullet$ needs to be able to correspond to a separation logic triple. We hence annotate this hole with the original function f's postcondition in order to be able to use it in a separation logic proof. This is, however, still not sufficient. The hole has to be annotated with the arguments supplied to f and their names as well, since in section 3 the FApp function application separation logic axiom will use the function arguments in the construction of the separation logic proof.

The final notation for a hole in the lifted semantics is hence $\bullet^{\overline{id_{\text{arg}}}=\overline{sexp}}_{\text{POST}(f)}$, with $\overline{sexp}$ the arguments supplied to f and $\text{POST}(f)$ f's postcondition. In the target language, holes do not need to be annotated as no separation logic proof has to be made, and we can just write $\bullet^{\overline{id_{\text{arg}}}=\overline{texp}}$. The argument expressions are necessary for the back-translation later on.

Note that the above discussion necessitates two versions for both the FApp and the Return rules below: one for the regular and one for the lifted operational semantics. The additions required for the lifted operational semantics rules (ie. hole annotations) are highlighted in green. It is obvious that the lifted and non-lifted versions of the rules will equi-terminate and that the additions serve solely separation-logic- and proof-related functions, causing no problems for coherence between non-verified and verified code.

$$\frac{\begin{array}{c}\Sigma_{\text{op}}(f) = \{\overline{\tau_{\text{arg}}\ id_{\text{arg}}}\ \{BODY; \text{return } \overline{sexp'}\}\} \\ [\![\overline{sexp}]\!]_s = \overline{n} \quad \overline{s} = s :: \overline{s_t} \quad \overline{s'} = [\overline{id_{\text{arg}}} \rightarrow \overline{n}] :: \overline{s}\end{array}}{\begin{array}{c}\langle \overline{s}, h\rangle \mid C :: \overline{id_{\text{prog}}} = f(\overline{sexp}); F \hookrightarrow \langle \overline{s'}, h\rangle \mid \\ C :: \{\overline{id_{\text{prog}}}\} = \bullet^{\overline{id_{\text{arg}}}=\overline{sexp}}_{\text{POST}(f)}; F :: BODY; \text{return } \overline{sexp'}\end{array}} \quad \text{(FAppLifted)}$$

$$\frac{\overline{s} = s :: s' :: \overline{s_t} \qquad \overline{s'} = s'[\overline{id_{\text{prog}}} \rightarrow \overline{n}] :: \overline{s_t} \qquad [\![\overline{sexp'}]\!]_s = \overline{n}}{\langle \overline{s}, h\rangle \mid C :: \{\overline{id_{\text{prog}}}\} = \bullet^{\overline{id_{\text{arg}} = sexp}}_{\text{POST}(f)}; F :: \text{return } \{\overline{sexp'}\} \hookrightarrow \langle \overline{s'}, h\rangle \mid C :: \text{skip}; F} \quad \text{(ReturnLifted)}$$

For the final return statement (which is the return statement created by the ProgExec rule below), there is no s_i - c_i pair to return to. The execution will hence not end in a skip statement, but in a return statement.

$$\frac{\overline{s} = s :: \overline{s_t} \qquad s(id_{\text{prog}}) = (l, i) \qquad h(l) = [v_0, \ldots, v_k] \qquad [\![sexp_1]\!]_s = n \qquad [\![sexp_2]\!]_s = v \qquad 0 \leq i + n \leq k \qquad h' = h[l \rightarrow h(l)[i + n : v]]}{\langle \overline{s}, h\rangle \mid C :: id_{\text{prog}}[sexp_1] = sexp_2; F \hookrightarrow \langle \overline{s}, h'\rangle \mid C :: \text{skip}; F} \quad \text{(ArrayMut)}$$

$$\frac{\overline{s} = s :: \overline{s_t} \qquad \overline{s'} = s' :: \overline{s_t} \qquad h(l) = [v_0, \ldots, v_k] \qquad [\![sexp_1]\!]_s = (l, i) \qquad [\![sexp_2]\!]_s = n \qquad 0 \leq i + n \leq k \qquad s' = s[id_{\text{prog}} \rightarrow v_{i+n}]}{\langle \overline{s}, h\rangle \mid C :: id_{\text{prog}} = sexp_2; F \hookrightarrow \langle \overline{s'}, h\rangle \mid C :: \text{skip}; F} \quad \text{(ArrayLkup)}$$

There is no explicit check that i is part of the current stack frame, as we suppose, for the sake of simplicity, that this check is handled during type checking, as was the case for variable declaration and lookup.

$$\frac{\overline{s} = s :: \overline{s_t} \qquad [\![sexp]\!]_s = n \qquad [\![sexp']\!]_s = n' \qquad n < n'}{\langle \overline{s}, h\rangle \mid C :: \text{foreach}(sexp \leq i < sexp')\{sstm\}; F \hookrightarrow \langle \overline{s}, h\rangle \mid C :: i = n; sstm; i = n + 1; \ldots; sstm; i = n'; F} \quad \text{(ForUnroll)}$$

$$\frac{\overline{s} = s :: \overline{s_t} \qquad [\![sexp]\!]_s = \text{true}}{\langle \overline{s}, h\rangle \mid C :: \text{guard}(sexp); F \hookrightarrow \langle \overline{s}, h\rangle \mid C :: \text{skip}; F} \quad \text{(GuardTrue)}$$

The below rule describes the transition from the initial state of any function execution. Notice the distinction in notation between an empty stack frame/a stack with one empty frame/an empty heap $\bullet$, versus a stack without frames ϵ.

$$\frac{\Sigma_{\text{op}}(id) = \{\epsilon\ \{BODY; \text{return } \{\overline{sexp}\}\}\}}{\langle \epsilon, \epsilon\rangle \mid (C_1\ \ldots C_k\ //@\text{main} = id\ ) \hookrightarrow \langle \bullet, \epsilon\rangle \mid BODY; \text{return } \{\overline{sexp}\}} \quad \text{(ProgExec)}$$

2.3 Target Language

Whenever the value of a pointer variable is stored in a different variable in the target language, the linear capability present in this pointer variable (except if it has the value null of course), has to be *erased* in the copied variable. This erasure was described and formally defined in section 2.1 and will be used in a couple instances in the operational semantics of this subsection.

First, an auxiliary concept, then the rules.

2.3.1 Linear Capability Erasure. This section describes the environment that has to be created to erase the correct linear capabilities when evaluating expressions in the target language. The environment in question is given by the output of StoreLinCap.

We first define a judgment $\rightsquigarrow_{\text{ValErase}}$ to replace stack values by stack values that are identical, except for the fact that all linear capabilities have been replaced by null pointers. This judgment is used in the judgment $\rightsquigarrow_{\text{GatherLinCap}}$, that gathers expressions for all linear capabilities that need to be erased as a tuple of the form $(exp, v) \ldots (exp, v)$, where each exp is the expression that accesses the linear capability that has to be erased, and v is the value to replace it with. Finally, the judgment $\rightsquigarrow_{\text{StoreLincap}}$ transforms this tuple of gathered linear capabilities into a proper stack assignment that can be used in the operational semantics.

$$\frac{}{k \rightsquigarrow_{\text{ValErase}} k} \quad \text{(IntToNull)}$$

$$\frac{}{l^a \rightsquigarrow_{\text{ValErase}} l^a} \quad \text{(SrcPtrToNull)}$$

$$\frac{}{l^{[a,b]} \rightsquigarrow_{\text{ValErase}} \text{null}} \quad \text{(PtrToNull)}$$

$$\frac{m_1 \rightsquigarrow_{\text{ValErase}} v_1 \quad \ldots \quad m_k \rightsquigarrow_{\text{ValErase}} v_k}{(m_1, \ldots, m_k) \rightsquigarrow_{\text{ValErase}} (v_1, \ldots, v_k)} \quad \text{(PairToNull)}$$

$$\frac{[\![id_{\text{prog}}]\!]_s \rightsquigarrow_{\text{ValErase}} erase}{id_{\text{prog}}, s \rightsquigarrow_{\text{GatherLinCap}} (id_{\text{prog}}, erase)} \quad \text{(IDProgStore)}$$

$$\frac{}{k, s \rightsquigarrow_{\text{GatherLinCap}} \bullet} \quad \text{(IntStore)}$$

$$\frac{}{\text{op1}\ \ exp, s \rightsquigarrow_{\text{GatherLinCap}} \bullet} \quad \text{(Op1Store)}$$

No pointer arithmetic is allowed, so the expression operations with arity 2 can never result in capability erasure.

$$\frac{}{exp_1\ \ \text{op2}\ \ exp_2, s \rightsquigarrow_{\text{GatherLinCap}} \bullet} \quad \text{(Op2Store)}$$

$$\frac{}{\text{null}, s \rightsquigarrow_{\text{GatherLinCap}} \bullet} \quad \text{(NullStore)}$$

$$\frac{}{\text{null}_0, s \rightsquigarrow_{\text{GatherLinCap}} \bullet} \quad \text{(Null0Store)}$$

$$\frac{}{\text{addr}(exp), s \rightsquigarrow_{\text{GatherLinCap}} \bullet} \quad \text{(AddressStore)}$$

$$\frac{}{\text{length}(exp), s \rightsquigarrow_{\text{GatherLinCap}} \bullet} \quad \text{(LengthStore)}$$

Any target-level tuple values causing the erasure of the same capability twice are disallowed. This is not a situation that can occur in a compiled program, so it is only relevant to the back-translation. We make this evaluation stuck using the below CheckStuck predicate, so that this stuck state will also cause the operational semantics to get stuck in the places where capability erasure is applicable. We did not use this CheckStuck in the TupleStore here, as that would be inefficient, but use it once at the end in the StoreLinCap rule below.

$$\frac{}{\text{CheckStuck}(\epsilon)} \quad \text{(CheckEps)}$$

$$\frac{\begin{array}{c}(id_{\text{prog}}, \_) \notin \overline{v_{\text{rest}}} \\ \forall i.\, (id_{\text{prog}}.i, \_) \notin \overline{v_{\text{rest}}} \\ \text{CheckStuck}(\overline{v_{\text{rest}}})\end{array}}{\text{CheckStuck}((id_{\text{prog}}, val)\ \overline{v_{\text{rest}}})} \quad \text{(CheckIdprog)}$$

$$\frac{\begin{array}{c}(id_{\text{prog}}, \_) \notin \overline{v_{\text{rest}}} \\ (id_{\text{prog}}.i, \_) \notin \overline{v_{\text{rest}}} \\ \text{CheckStuck}(\overline{v_{\text{rest}}})\end{array}}{\text{CheckStuck}((id_{\text{prog}}.i, val)\ \overline{v_{\text{rest}}})} \quad \text{(CombineIdprogIndex)}$$

$$\frac{\begin{array}{c}exp_1, s \rightsquigarrow_{\text{GatherLinCap}} v_1 \\ \dots \\ exp_k, s \rightsquigarrow_{\text{GatherLinCap}} v_k\end{array}}{(exp_1, \dots, exp_k), s \rightsquigarrow_{\text{GatherLinCap}} v_1\ \dots\ v_k} \quad \text{(TupleStore)}$$

The following 2 rules should not erase capabilities on the elements that are not being used for the projection, as this would leak capability values. We also have to make an exception for the case where *exp* is actually a program variable id_{prog}, as program variables are handled differently by GatherLinCap.

$$\frac{exp = (exp_1, \dots, exp_i, \dots, exp_k) \qquad exp_i, s \rightsquigarrow_{\text{GatherLinCap}} v}{exp.i, s \rightsquigarrow_{\text{GatherLinCap}} v} \quad \text{(ProjectTupleStore)}$$

$$\frac{exp = id_{\text{prog}} \qquad id_{\text{prog}}, s \rightsquigarrow_{\text{GatherLinCap}} (id_{\text{prog}}, (erase_1, \dots, erase_k))}{exp.i, s \rightsquigarrow_{\text{GatherLinCap}} (id_{\text{prog}}.i, erase_i)} \quad \text{(ProjectIDProgStore)}$$

Lastly, we define a judgment $\rightsquigarrow_{\text{TupleToSAsgn}}$ to convert the assignments to linear capabilities that we gathered using $\rightsquigarrow_{\text{GatherLincCap}}$ into proper stack assignments.

An auxiliary filter judgment is used, that filters out all assignments to the same id_{prog}. This is used in $\rightsquigarrow_{\text{TupleToSAsgn}}$, that converts the previously created list of tuples to a list of stack assignments.

$$\frac{}{\epsilon, id_{\text{prog}} \rightsquigarrow_{\text{IDFilter}} \epsilon, \epsilon} \quad \text{(FilterBase)}$$

$$\frac{\overline{v_{\text{rest}}}, id_{\text{prog}} \rightsquigarrow_{\text{IDFilter}} v_1, v_2}{(id_{\text{prog}}.i, v)\ \overline{v_{\text{rest}}}, id_{\text{prog}} \rightsquigarrow_{\text{IDFilter}} (id_{\text{prog}}.i, v)\ v_1, v_2} \quad \text{(FilterFound)}$$

$$\frac{\overline{v_{\text{rest}}}, id_{\text{prog}} \rightsquigarrow_{\text{IDFilter}} v_1, v_2 \qquad exp \neq id_{\text{prog}}.i}{(exp, v)\ \overline{v_{\text{rest}}}, id_{\text{prog}} \rightsquigarrow_{\text{IDFilter}} v_1, (exp, v)\ v_2} \quad \text{(FilterNotFound)}$$

$$\frac{}{\epsilon, s \rightsquigarrow_{\text{TupleToSAsgn}} \epsilon} \quad \text{(TupleToSAsgnEps)}$$

$$\frac{\overline{v_{\text{rest}}}, s \rightsquigarrow_{\text{TupleToSAsgn}} asgn}{(id_{\text{prog}}, v)\ \overline{v_{\text{rest}}}, s \rightsquigarrow_{\text{TupleToSAsgn}} asgn[id_{\text{prog}} : v]} \quad \text{(TupleToSAsgnID)}$$

$$\frac{\begin{array}{c} v', s \rightsquigarrow_{\text{TupleToSAsgn}} asgn \\ \overline{v_{\text{rest}}} \rightsquigarrow_{\text{IDFilter}} (id_{\text{prog}}.i_2, v_2)\ \ldots (id_{\text{prog}}.i_k, v_k), v' \\ \text{TypeOfVar}(id_{\text{prog}}) = (\tau_1, \ldots, \tau_k) \qquad v = (v'_1, \ldots v'_k) \\ \forall i_j \in \{i_1, \ldots i_k\}.\ v'_{i_j} = v_j \\ \forall i \notin \{i_1, \ldots i_k\}.\ v'_i = [\![id_{\text{prog}}.i]\!]_s \end{array}}{(id_{\text{prog}}.i_1, v_1)\ \overline{v_{\text{rest}}}, s \rightsquigarrow_{\text{TupleToSAsgn}} asgn[id_{\text{prog}} : v]} \quad \text{(TupleToSAsgnIDIndex)}$$

$$\frac{\begin{array}{c} exp, s \rightsquigarrow_{\text{GatherLinCap}} tuple \\ tuple, s \rightsquigarrow_{\text{TupleToSAsgn}} asgn \\ \text{CheckStuck}(tuple) \end{array}}{exp, s \rightsquigarrow_{\text{StoreLinCap}} asgn} \quad \text{(StoreLinCap)}$$

2.3.2 *Operational rules.*

$$\frac{}{\langle \overline{s}, h \rangle \mid C :: \text{skip}; F \hookrightarrow \langle \overline{s}, h \rangle \mid C :: F} \quad \text{(Skip)}$$

The variable id_{prog} is of type $\tau*$.

$$\frac{\begin{array}{c} \overline{s} = s :: \overline{s_{\text{t}}} \qquad \overline{s'} = s' :: \overline{s_{\text{t}}} \\ s' = s[id_{\text{prog}} \rightarrow l^{[0,k-1]}] \\ l \notin \text{dom}(h) \qquad \tau \rightsquigarrow_{\text{def}} v \\ [\![texp]\!]_s = k \qquad k > 0 \qquad h' = h[l \rightarrow [v, \ldots, v]_k] \end{array}}{\langle \overline{s}, h \rangle \mid C :: id_{\text{prog}} = \text{malloc}(texp * \text{sizeof}(\tau)); F \hookrightarrow \langle \overline{s'}, h' \rangle \mid C :: \text{skip}; F} \quad \text{(Malloc)}$$

The variable id_{prog3} is erased in the rule below. We do not need the erasure rules from section 2.1, however, since we know we are dealing with a target variable and not a more general target expression.

$$\frac{\begin{array}{c}\overline{s} = s :: \overline{s_t} \qquad \overline{s'} = s' :: \overline{s_t} \\ [\![texp]\!]_s = n \qquad 1 \le n \le b - a \\ s(id_{prog3}) = l^{[a,b]} \qquad s' = s[id_{prog1} \to l^{[a,a+n]}][id_{prog2} \to l^{[a+n,b]}][id_{prog3} \to \text{null}]\end{array}}{\langle \overline{s}, h\rangle \mid C :: \{id_{prog1},\ id_{prog2}\} = split(id_{prog3}, texp); F \hookrightarrow \langle \overline{s'}, h\rangle \mid C :: \text{skip}; F} \quad \text{(Split)}$$

$$\frac{\begin{array}{c}\overline{s} = s :: \overline{s_t} \qquad \overline{s'} = s' :: \overline{s_t} \\ s(id_{prog2}) = l^{[a,n]} \qquad s(id_{prog3}) = l^{[n,b]} \\ s' = s[id_{prog1} \to l^{[a,b]}][id_{prog2} \to \text{null}][id_{prog3} \to \text{null}]\end{array}}{\langle \overline{s}, h\rangle \mid C :: id_{prog1} = join(id_{prog2}, id_{prog3}); F \hookrightarrow \langle \overline{s}, h\rangle \mid C :: \text{skip}; F} \quad \text{(Join)}$$

$$\frac{\overline{s} = s :: \overline{s_t} \qquad [\![texp_1]\!]_s = \text{true}}{\langle \overline{s}, h\rangle \mid C :: \text{if } texp_1 \text{ then } tstm_1 \text{ else } tstm_2; F \hookrightarrow \langle \overline{s}, h\rangle \mid C :: tstm_1; F} \quad \text{(IfTrue)}$$

$$\frac{\overline{s} = s :: \overline{s_t} \qquad [\![texp_1]\!]_s = \text{false}}{\langle \overline{s}, h\rangle \mid C :: \text{if } texp_1 \text{ then } tstm_1 \text{ else } tstm_2; F \hookrightarrow \langle \overline{s}, h\rangle \mid C :: tstm_2; F} \quad \text{(IfFalse)}$$

$$\frac{\begin{array}{c}\overline{s} = s :: \overline{s_t} \qquad \overline{s'} = s' :: \overline{s_t} \\ \tau \rightsquigarrow_{\text{def}} v \qquad s' = s[id_{\text{prog}} \to v]\end{array}}{\langle \overline{s}, h\rangle \mid C :: \tau\ id_{\text{prog}}; F \hookrightarrow \langle \overline{s'}, h\rangle \mid C :: \text{skip}; F} \quad \text{(VarDecl)}$$

Self-assignment is no problem below because env is added to the stack frame before id_{prog}.

$$\frac{\begin{array}{c}\overline{s} = s :: \overline{s_t} \qquad \overline{s'} = s' :: \overline{s_t} \\ [\![texp]\!]_s = n \qquad texp, s \rightsquigarrow_{\text{StoreLinCap}} [env] \qquad s' = s[env][id_{\text{prog}} \to n]\end{array}}{\langle \overline{s}, h\rangle \mid C :: id_{\text{prog}} = texp; F \hookrightarrow \langle \overline{s'}, h\rangle \mid C :: \text{skip}; F} \quad \text{(VarAsgn)}$$

In the below rule, we consider StoreLinCap to be applied to the entire tuple $\overline{texp}$ simultaneously, so that even if different arguments use the same linear capability, execution still gets stuck.

As described in the source operational semantics, we use the notation $\bullet^{\overline{id_{\text{arg}}=texp}}$ for target-level holes, because the argument expressions $\overline{texp}$ will be required information when performing the back-translation later on. Technically, we should define a 'non-lifted' (ie. one without proof-related clutter) target operational semantics as well, where holes are simply represented as $\bullet$ and prove that this operational semantics equi-terminates with our current one. As this is trivial, we leave this last uncluttering step implicit in the rest of the full abstraction proof.

$$\frac{\begin{array}{c}\Sigma_{\text{op}}(f) = \{\overline{\tau_{\text{arg}}\ id_{\text{arg}}}\ \{BODY; \text{return } \{\overline{texp'}\}\}\} \\ [\![\overline{texp}]\!]_s = \overline{k} \qquad \overline{texp}, s \rightsquigarrow_{\text{StoreLinCap}} [\overline{env}] \\ \overline{s} = s :: \overline{s_t} \qquad \overline{s'} = [\overline{id_{\text{arg}}} \to \overline{k}] :: s[\overline{env}] :: \overline{s_t}\end{array}}{\begin{array}{c}\langle \overline{s}, h\rangle \mid C :: \{\overline{id_{\text{prog}}}\} = f(\overline{texp}); F \hookrightarrow \langle \overline{s'}, h\rangle \mid \\ C :: \{\overline{id_{\text{prog}}}\} = \bullet^{\overline{id_{\text{arg}}=texp}}; F :: BODY; \text{return } \{\overline{texp'}\}\end{array}} \quad \text{(FApp)}$$

No linear capability erasure needed below, as the current stack frame is erased anyway. We add StoreLinCap to the precondition anyway, because it fails if the same capability is erased twice.

$$\frac{\overline{s} = s :: s' :: \overline{s_t} \qquad \overline{s'} = s'[\overline{id_{\text{prog}}} \rightarrow \overline{n}] :: \overline{s_t} \qquad [\![\overline{texp'}]\!]_s = \overline{n} \qquad \overline{texp}, s \rightsquigarrow_{\text{StoreLinCap}} -}{\langle \overline{s}, h\rangle \mid C :: \{\overline{id_{\text{prog}}}\} = \bullet^{\overline{id_{\text{arg}}=texp}}; F :: \text{return } \{\overline{texp'}\} \hookrightarrow \langle \overline{s'}, h\rangle \mid C :: \text{skip}; F} \quad \text{(Return)}$$

Note that $0 \leq n \leq b - a$ also implies $n \in \text{dom}(h(l))$ given $l \in \text{dom}(h)$, because $l^{[a,b]}$ is obtained by splitting and combining intervals from the original allocated linear capability. The same applies for the ArrayLkup rule.

$$\frac{\begin{array}{ccc} \overline{s} = s :: \overline{s_t} & \overline{s'} = s' :: \overline{s_t} & s(id_{\text{prog}}) = l^{[a,b]} \\ [\![texp_1]\!]_s = n & & [\![texp_2]\!]_s = m \\ l \in \text{dom}(h) \quad 0 \leq n \leq b - a & & h' = h[l \rightarrow h(l)[a + n : m]] \\ texp, s \rightsquigarrow_{\text{StoreLinCap}} [env] & & s' = s[env] \end{array}}{\langle \overline{s}, h\rangle \mid C :: id_{\text{prog}}[texp_1] = texp_2; F \hookrightarrow \langle \overline{s'}, h'\rangle \mid C :: \text{skip}; F} \quad \text{(ArrayMut)}$$

$$\frac{\begin{array}{ccc} \overline{s} = s :: \overline{s_t} & & \overline{s'} = s' :: \overline{s_t} \\ [\![texp_1]\!]_s = l^{[a,b]} & & [\![texp_2]\!]_s = m \\ h(l, a + m) = val \quad s' = s[id_{\text{prog}} \rightarrow val] & & 0 \leq m \leq b - a \\ val \rightsquigarrow_{\text{ValErase}} val' & & h' = h[l \rightarrow h(l)[a + m : val']] \end{array}}{\langle \overline{s}, h\rangle \mid C :: id_{\text{prog}} = texp_1[texp_2]; F \hookrightarrow \langle \overline{s'}, h'\rangle \mid C :: \text{skip}; F} \quad \text{(ArrayLkup)}$$

$$\frac{\begin{array}{c} \overline{s} = s :: \overline{s_t} \\ [\![texp]\!]_s = n \quad [\![texp']\!]_s = n' \quad n < n' \end{array}}{\begin{array}{c} \langle \overline{s}, h\rangle \mid C :: \text{foreach}(texp \leq i < texp')\{sstm\}; F \hookrightarrow \langle \overline{s}, h\rangle \mid C :: \\ i = n; sstm; i = n + 1; \ldots; sstm; i = n'; F \end{array}} \quad \text{(ForUnroll)}$$

$$\frac{\overline{s} = s :: \overline{s_t} \qquad [\![texp]\!]_s = \text{true}}{\begin{array}{c} \langle \overline{s}, h\rangle \mid C :: \\ guard(texp); F \hookrightarrow \langle \overline{s}, h\rangle \mid C :: \text{skip}; F \end{array}} \quad \text{(GuardTrue)}$$

$$\frac{\Sigma_{\text{op}}(id) = \{\epsilon\ \{BODY; \text{return } \{\overline{texp}\}\}\}}{\langle \epsilon, \epsilon\rangle \mid (C_1\ \ldots C_k\ //@\text{main} = id\ ) \hookrightarrow \langle \bullet, \epsilon\rangle \mid BODY; \text{return } \{\overline{texp}\}} \quad \text{(ProgExec)}$$

3 AXIOMS

This section details how the separation logic proofs that are used as input to our compiler are constructed from the separation-logic annotated source code. The building blocks for these proof are the separation logic axioms for each source level statement, presented in this section in the form of inference rules. Before being able to present detailed separation logic axioms for source statements, components and entire source programs, this section first formalizes some separation-logic proof notation and defines what separation logic contracts and separation logic proofs look like exactly.

3.1 Separation-logic proofs and contracts

Classical separation logic uses *Hoare triple notation* of the form $\{P\}\ c\ \{Q\}$ to construct proofs of Q given precondition P for the piece of source code c. We use c to denote a *command*, ie. either a source expression or a source expression followed by a return statement. A proof $\{P\}\ c\ \{Q\}$ is only sound if it can be constructed as a proof tree from the individual separation logic axioms. For this technical report, we use the partial correctness interpretation of separation logic, which allows non-termination of separation logic-verified code [Reynolds 2002].

In our formalization of separation logic, we do not use the classical Hoare triple notation $\{P\}\ c\ \{Q\}$. Instead, we split the conditions P and Q into two separate parts, namely:

(1) The *symbolic heap P (often Q in a postcondition)* is a $*$-separated list of separation logic expressions, ranging over symbolic variables only. It contains both spatial and pure parts, and both types of parts can be combined within one separation logic expression. This is identical to the separation logic assertion defined above.
We will sometimes (in the case of boundary functions, cfr. later on) require the symbolic assertion P to be separable into a *spatial or chunk heap* P_c, consisting of separation logic array resources only, and a pure *pure heap* P_p.
The spatial heap P_c (sometimes just P if the context is clear) is a list of array chunks and contains the *spatial* part of the separation logic conditions. As the different chunks are connected using separating conjunctions $*$, facts in the spatial heap cannot be copied without changing the spatial heap's meaning. To stress the fact that chunks are named within our formalization, often $\overline{n} : P_\mathrm{c}$ or $\overline{n : P_n}$ is used to denote the spatial heap. The spatial heap is constructed from non-desugared array chunks only.
The pure heap P_P contains assumptions that are non-spatial or heap-independent. Pure assumptions can be copied any number of times within the pure heap. Control flow decisions and the pure parts of function postconditions are added to the pure heap. The pure heap is constructed from the first half of the expressions *exp* in the BNF grammar.
(2) The *environment γ* for all variable assigments. Our separation logic makes a distinction between program variables which appear in source programs and logical (or symbolic) variables, which only appear in separation logic triples. The environment γ provides a mapping from program variables to logical variables, and hence relates the separation logic proof (ie. the symbolic assertion) to the concrete program that is being verified.

The notation for these aspects of our extended version of Hoare triples and the notation for the extended Hoare triples themselves is defined by the following BNF grammar:

$\langle P \rangle$	::=		(Symbolic Heap)
		assert	*separation logic assertion*
$\langle \gamma \rangle$	::=		(Symbolic Stack)
		$\bullet[id_\mathrm{prog} : exp]^*$	*map*
$\langle c \rangle$	::=		(Command)
		sstm | *sstm*; return *sexp*	
$\langle triple \rangle$	::=		(Hoare Triple)
		$\{P\}_\gamma\ c\ \{P\}_\gamma$	

In the extended Hoare triple $\{P\}_\gamma\ c\ \{Q\}_{\gamma'}$, the subscripts can be left out in a given separation logic axiom if they are not altered and do not influence the pre- or postcondition in any way. The extended tuple notation can easily be desugared to the classical separation logic notation, resulting in the following classical triple: $\{P * *\{x == \gamma(x) \mid x \in dom(\gamma)\}\}\ c\ \{Q' * *\{x == \gamma(x) \mid x \in dom(\gamma')\}\}$. The only requirement for this desugaring is that the namespaces for program and symbolic variables are disjoint, to avoid name clashes in the desugared separation logic triple. This condition can easily be achieved by renaming logical variables or requiring them to be fresh.

A triple $\{P\}_\gamma\ c\ \{Q\}_{\gamma'}$ is always (often implicitly) constructed in the context of a component environment Σ_i^{ax} of the type described in section 2, as we need called function's contracts in order to create Hoare triples for function calls. We denote this contextual triple as $\Sigma_i^{ax} \vdash \{P\}_\gamma\ c\ \{Q\}_{\gamma'}$, but mostly leave the component environment implicit in the arguments.

The reason for using this extended form of Hoare tiples instead of the classical notation is that is makes multiple separation logic axioms easier and different parts of the formalization more coherent. First of all, name clashes are avoided between program and logical variables by making a clear conceptual distinction. Because of this distinction between variable assignment and declaration, for example, only have to change the environment, whereas eg. array mutation and lookup only change the symbolic heap. This is easier to formalize and more coherent than in the case where the Hoare triple consists of a single piece of information. The same holds true when we define simulation relations during the full abstraction proof: the conceptually distinct parts make it easier to define a relation between the separation logic contract and the state of the program itself.

Having talked about the formalization of separation logic triples, we can now have a more in-depth discussion of what we are trying to prove using these triples: the separation logic contracts. A separation logic contract consists of a pre- and a postcondition, both of which are a separation logic assertion. Function contracts are hence very similar to separation logic triples. One difference we will digress on in the following 3 paragraphs is the fact that we have 3 extra form-restraints for certain separation logic contracts compared to regular triples. These restrictions only hold for contracts of so called *boundary functions*. We define *import boundary functions* as functions that are imported by some module from another module and *export boundary functions* as functions that are exported by a module. Boundary functions play an important role in the compilation, as will be explained in section 4. For import boundary functions, the conditions in principle hold on the precondition, whereas for export boundary functions, the same conditions now hold for the postcondition in principle. An important caveat is that from the perspective of another component and during the back-translation, the roles of import and export boundary functions are swapped, also swapping the conditions from pre- to postcondition and the other way around. This implies that we have to require the below conditions to hold on both the pre- and postcondition of both import and export boundary functions, otherwise the back-translated program would not be a valid source program. We hence make no difference between the restrictions on import and export boundary functions.

Firstly, we require the contract to consist of two separable parts in both the pre- and postcondition. The first part consists of a list of fixed-length, non-conditional (conditionals make reification harder) array chunks over symbolic variables (which is the spatial heap mentioned before) named PRE_s, resp. $POST_\mathrm{s}$, whereas the second part is a restricted form of expression over symbolic variables (which is the pure heap mentioned before) named PRE_p, resp. $POST_\mathrm{p}$. These parts reflect 1 of the 2 parts present in our extended Hoare triples, which means that contracts only lack the environment part when compared to extended Hoare triples. The reason for this is that at the start of the function, all function arguments are assumed to map to symbolic variables with the same name. The function

with declaration $\overline{\tau}\ f(\overline{id_{\text{arg}}\ \tau_{\text{arg}}})$ thus has the environment $[\overline{id_{\text{arg}}}{:}\overline{id_{\text{arg}}}]$ as starting environment. Function contracts are hence made up of logical variables that represent the original values of the function's program arguments, seen as the value of logical variables does not change. For the returned program expressions, the privileged symbolic variables $result_i$ are provided, corresponding to the ith return value. These variables are only allowed to appear in the pure heap of the contract's post-condition. For example, the expression $result_3 == 3$ means that the third returned value must be equal to 3.

The second restriction is in the fact that separation logic contracts have to be *linear*. This means that variable names cannot be repeated between the function arguments, the chunks in the precondition and the chunks in the postcondition. This is formalized in the FDeclWellFormed rule below. Repetitions of e.g. the variable a are to be avoided by replacing a by a variable a' on the second occurrence of a and adding $a == a'$ to the linear equalities. Also, integers k (and later constants in general) are avoided inside array chunks, by replacing them with a new variable id and adding the equality $id == k$. The *result* variables that were mentioned before were automatically made linear by definition, as they do not appear in the arguments nor the chunks. Additionally, the pure heap should not introduce new symbolic variables in this case, and in the case of the postcondition not use symbolic variables that were defined in the pure heap of the precondition.

The third restriction states that all logical variables denoting addresses of the chunks in the contract of f are bound to a single expression in the pure heap, where this single expression contains only logical variables that we can reify into program variables. This condition is necessary because of the compilation of boundary functions in section 4 and will be detailed there in the BoundaryContrWF rule.

Another important caveat concerns the implicit assumptions on quantifier use in function contracts that we make. Similarly to what the VeriFast tool does, we assume that any logical variables appearing in a precondition are implicitly universally quantified, whereas fresh logical variables appearing in a postcondition are existentially quantified. As an example, consider the following source function:

$$\begin{array}{l}\text{int } f(\text{int}* x)\\ \text{//@pre } n : x \mapsto y \ \text{//@post } n : x \mapsto z * result == 0\\ \quad \{*x = 3; \text{return } 0\}\end{array}$$

Then we assume that x and y are universally quantified, whereas z is existentially quantified. This allows any function possessing the right precondition to call f, and use the results without concern.

3.2 Some additional notation

An expression that often appears in the separation logic axioms of this section is $sexp_\gamma$, which means that the environment γ is applied to the source expression *sexp*. Concretely, all program variables id_{prog} in *sexp* are replaced by $\gamma(id_{\text{prog}})$.

The notations <pointer> $\mapsto_\tau$ <array> and <pointer> $\Mapsto_\tau$ <array>, [...] are used for heap chunks to specify that the contents of the array pointed to are of type τ. This syntax is necessary because nested arrays are possible and the separation logic syntax could otherwise leave the type of the array contents ambiguous. In principle, this typed points-to chunk is not necessary, as static type checking of the source program should be able to derive the proper types for all chunks. As the issue of type checking has been intensively investigated and is orthogonal to the challenges presented in

this paper, it has not been applied. In most occasions, the τ is left out because it is irrelevant to the formalization. The only place where this notation really plays a role is when chunks in function contracts are reified during compilation. This occurs in the ResDecl rule.

Abstract heap locations, which are the separation logic equivalent of the operational semantics heap locations and indexes, are denoted as l_a, to avoid confusion with the concrete locations l from the operational semantics. In the current memory model, each chunk in a function contract is assumed to be positioned at a separate memory location l_a.

3.3 Separation logic axioms

3.3.1 *Basic statement rules.*

$$\frac{}{\{\}\ \text{skip}\ \{\}} \qquad \text{(Skip)}$$

Notice that the malloc function used in the below axiom doesn't correspond to the vanilla malloc function in C, as it guarantees a fresh location for the allocated variable. Our malloc call can be seen as a wrapper (depending on architecture and implementation) around the regular malloc function in C, with the extra guarantee that any newly allocated space hasn't been used before by the context (which could otherwise have kept references). An alternative would be to introduce a specific capability-based memory manager, which does allow reallocation of a region in memory if all capabilities corresponding to this region have gone out of scope. This would, however make the formalization more difficult and has hence been left out of scope. No Free statement has been formalized for similar reasons: it forces us to formalize some kind of memory management in the operational semantics and hence in the separation logic, and does not add a lot the the power and relevance of the full abstraction result.

Note that we often use the short-hand notation $\forall 0 \leq i < \text{length}(l).$, which desugars to $\forall i : \text{int}.\ 0 \leq i < \text{length}(l) \Rightarrow$.

$$\frac{\begin{array}{cc}\tau \rightsquigarrow_{\text{def}} v & n, id_{\text{log}}\ \text{fresh} \\ id_{\text{prog}} \in \text{dom}(\gamma) & \gamma' = \gamma[id_{\text{prog}} : id_{\text{log}}]\end{array}}{\begin{array}{c}\{sexp_\gamma > 0\}_\gamma\ id_{\text{prog}} = \text{malloc}(sexp * \text{sizeof}(\tau)) \\ \{n : id_{\text{log}} \mapsto_\tau \text{repeat}(sexp_\gamma, v)\}_{\gamma'}\end{array}} \qquad \text{(Malloc)}$$

To define the Flatten rule below, we need to be able to add fresh chunknames to a given assertion *assert*. We introduce the judgment $assert \rightsquigarrow_{\text{NameChunks}} assert, \overline{n}$ to transform an inner separation logic assertion to an outer assertion (ie. including chunk names). The names $\overline{n}$ contain the target level declarations for the freshly named chunks we need to create during compilation.

The only two non-identity rules (identity rules create no names) for this judgment are the following:

$$\frac{}{\begin{array}{c}[assert \mid exp \leq id_{\text{log}} < exp'] \\ \rightsquigarrow_{\text{NameChunks}} n : [assert \mid exp \leq id_{\text{log}} < exp'], n\end{array}} \qquad \text{(NameNestedChunks)}$$

$$\frac{}{\begin{array}{c} exp \mapsto_\tau exp' \\ \rightsquigarrow_{\text{NameChunks}} n : exp \mapsto_\tau exp', n \end{array}} \quad \text{(NameFlatChunks)}$$

We also introduce the notation $assert_{i \mapsto 0 \ldots k}$ to mean the tuple of assertions obtained by substituting i in $assert$ for all values in the range $0 \ldots k$ (including k).

$$\frac{\begin{array}{c} assert[i \mapsto k_1 \ldots k_2 - 1] \rightsquigarrow_{\text{NameChunks}} \overline{assert'}, \overline{n_{ij}} \\ \overline{n_{ij}} \text{ fresh} \end{array}}{\begin{array}{c} \{n : [assert \mid k_1 \leq i < k_2]\}_\gamma \\ //@\text{flatten } n \\ \{\overline{assert'}\}_\gamma \end{array}} \quad \text{(Flatten)}$$

When collecting chunks, the reverse of $assert \rightsquigarrow_{\text{NameChunks}} assert, \overline{vardecl}$ needs to happen: $assert \rightsquigarrow_{\text{UnNameChunks}} assert, \overline{exp}, \overline{\tau}, \overline{n}$, where chunk names are erased. The chunk addresses $\overline{exp}$, types $\overline{\tau}$ and chunk names $\overline{n}$ for each assertion are also part of the output, as these play a role in the Collect rule. This judgment again has only two non-identity rules:

$$\frac{}{\begin{array}{c} n : [exp \mid exp' \leq \tau < assert][id_{\log}] \\ \rightsquigarrow_{\text{UnNameChunks}} ([exp \mid exp' \leq \tau < assert][id_{\log}]), n \end{array}} \quad \text{(UnnameRangeResource)}$$

$$\frac{}{\begin{array}{c} n : exp \mapsto_\tau exp' \\ \rightsquigarrow_{\text{UnNameChunks}} (exp \mapsto_\tau exp'), n \end{array}} \quad \text{(UnnameFlatResource)}$$

Notice that collect is non-deterministic wrt the inclusion of pure conditions. This does not matter, as these conditions can be moved in and out of the tree anyway.

To avoid ambiguity, the collect rule has the range for i start out on 0. The Consequence rule can be used to adjust this value, if need be.

$$\frac{\begin{array}{c} assert[i \mapsto 0 \ldots k-1] \rightsquigarrow_{\text{UnNameChunks}} assert'[i \mapsto 0 \ldots k-1], \overline{n_{ij}} \\ n \text{ fresh} \end{array}}{\begin{array}{c} \{assert[i \mapsto 0] * \ldots * assert[i \mapsto k-1] * k > 0\}_\gamma \\ //@\text{collect } \overline{n_{0j}} \cdot \ldots \cdot \overline{n_{(k-1)j}} \\ \{n : [assert' \mid 0 \leq i < k]\}_\gamma \end{array}} \quad \text{(Collect)}$$

In the below rule, the separation logic assertion $Inv[i_s]$ and the environment $\gamma[i_s]$ are both functions of some general parameter i_s, which is the logical variable corresponding to i.

Note that we do not disallow assignments to the variable i inside a for loop; both in the target and in the source. This is not necessary, as the operational semantics guarantee that i has the correct value at the start of each for loop. This is the reason why the expression $\gamma[i_s + 1][i : \_]$ allows any expression instead of i_s in the below rule, as i's mapping might have been changed by the for loop in the general expression $\gamma(i_s)$.

$$\frac{\begin{array}{c}\{sexp_{\gamma_{\text{pre}}} \leq i_{\text{s}} < sexp'_{\gamma_{\text{pre}}} * Inv[i_{\text{s}}]\}_{\gamma[i_{\text{s}}]}\ sstm\ \{Inv[i_{\text{s}}+1]\}_{\gamma[i_{\text{s}}+1][i:\_]} \\ \gamma[i_{\text{s}}](i) = i_{\text{s}} \qquad \gamma[sexp_{\gamma_{\text{pre}}}] = \gamma_{\text{pre}}[i : sexp_{\gamma_{\text{pre}}}] \qquad \gamma[sexp'_{\gamma_{\text{pre}}}] = \gamma_{\text{post}}\end{array}}{\{Inv[sexp_{\gamma_{\text{pre}}}]\}_{\gamma_{\text{pre}}}\ \text{foreach}(sexp \leq i < sexp')\{sstm\}\ \{Inv[sexp'_{\gamma_{\text{pre}}}]\}_{\gamma_{\text{post}}}} \quad \text{(For)}$$

The below split and join rules are also defined on range-shaped resources. This **does** add expressive power, as both cannot be emulated easily be a consecutive flatten and two collect statements in the case of split, or by 2 flatten and one collect statement in the case of join: we do not know what the length of the collect should be, as we do not know the value of the split index or any of the lengths beforehand, as they might not be constants. We will need this version of split and join in the back-translation, because all manipulated chunks are tree chunks.

Furthermore, we also use the resource name and not the address in the case where we split a flat array resource, to keep the rules for flat resources more uniform with the rules for range resources, and because it makes sense to use the name of the heap resource as a handle to access this resource in ghost commands.

Notice that the target language only has 1 set of built-in split and join functions. The reason for this is that both types of source-language split and join just come down to merging or splitting linear capabilities in the target language.

$$\frac{n', n'' \text{ fresh}}{\begin{array}{c}\{n : exp_{\text{a}} \mapsto_{\tau} l * \text{length}(l) == exp_{\text{l}} * 0 < sexp_{\gamma} < exp_{\text{l}}\}_{\gamma} \\ //@\text{split } n[sexp] \\ \{n' : exp_{\text{a}} \mapsto_{\tau} \text{take}(l, 0, sexp_{\gamma}) \\ * n'' : (exp_{\text{a}} + sexp_{\gamma}) \mapsto_{\tau} \text{take}(l, sexp_{\gamma}, exp_{\text{l}})\}_{\gamma}\end{array}} \quad \text{(Split)}$$

$$\frac{n', n'' \text{ fresh}}{\begin{array}{c}\{n : [assert \mid exp \leq i < exp'] * 0 < sexp_{\gamma} < exp' - exp\}_{\gamma} \\ //@\text{split } n[sexp] \\ \{n' : [assert \mid exp \leq i < exp + sexp_{\gamma}] * \\ n'' : [assert \mid exp + sexp_{\gamma} \leq i < exp']\}_{\gamma}\end{array}} \quad \text{(SplitRange)}$$

$$\frac{n, l \text{ fresh}}{\begin{array}{c}\{n' : exp'_{\text{a}} \mapsto_{\tau} l' * \text{length}(l') == exp'_{\text{l}} \\ * \ n'' : exp''_{\text{a}} \mapsto_{\tau} l'' * \text{length}(l'') == exp''_{\text{l}} \\ * exp''_{\text{a}} = exp'_{\text{a}} + exp'_{\text{l}} + 1\}_{\gamma} \\ //@\text{join } n'\ n'' \\ \{n : exp'_{\text{a}} \mapsto_{\tau} \text{append}(l', l'')\}_{\gamma}\end{array}} \quad \text{(Join)}$$

$$\frac{n \text{ fresh}}{\begin{array}{c}\{n' : [assert \mid exp \leq i < exp'] * \\ n'' : [assert \mid exp' \leq i < exp'']\}_\gamma \\ //@\text{join } n'\ n''' \\ \{n : [assert \mid exp \leq i < exp'']\}_\gamma\end{array}} \quad \text{(JoinRange)}$$

$$\frac{\begin{array}{c}\{P\}_\gamma\ sstm_1\ \{Q\}_{\gamma'} \\ \{Q\}_{\gamma'}\ sstm_2\ \{R\}_{\gamma''}\end{array}}{\{P\}_\gamma\ sstm_1; sstm_2\ \{R\}_{\gamma''}} \quad \text{(Seq)}$$

The idea behind the conditional rule is to hand the responsibility of matching the postconditions of $sstm_1$ and $sstm_2$ to the programmer or the verification tool. This requires that the power to perform this actual matching be put in the Consequence rule below.

$$\frac{\begin{array}{c}\{P * sexp_\gamma\}_\gamma\ sstm_1\ \{Q\}_{\gamma'} \\ \{P * !sexp_\gamma\}_\gamma\ sstm_2\ \{Q\}_{\gamma'}\end{array}}{\{P\}_\gamma\ \text{ if } sexp \text{ then } sstm_1 \text{ else } sstm_2\ \{Q\}_{\gamma'}} \quad \text{(If)}$$

$$\frac{\begin{array}{c}id_{\text{prog}} \notin \text{dom}(\gamma) \qquad \tau \rightsquigarrow_{\text{def}} v \\ \gamma' = \gamma[id_{\text{prog}} : v]\end{array}}{\{\}_\gamma\ \tau\ id_{\text{prog}}\ \{\}_{\gamma'}} \quad \text{(VarDecl)}$$

$$\frac{id_{\text{prog}} \in \text{dom}(\gamma) \qquad \gamma' = \gamma[id_{\text{prog}} : sexp_\gamma]}{\{\}_\gamma\ id_{\text{prog}} = sexp\ \{\}_{\gamma'}} \quad \text{(VarAsgn)}$$

The below 2 rules force the programmer to use appropriate split statements to ensure that array mutation and lookup happen on the first element of a heap chunk. This is a simplifying assumption without loss of generality. We might just as well allow mutation and lookup to happen in any element of a heap chunk, but do not do this for simplicity's sake and because it isn't allowed for split and join statements either.

Array mutation and lookup are supposed to be performed on non-tree chunks, as the values that are assigned in the target language are also compiled values, and we are hence manipulating the reified flat array chunks, and not the tree chunks. Appropriate split commands on trees are hence in order, to allow for a flatten to happen in order to manipulate a tree chunk.

Array mutation and lookup require the program variable to correspond to the address of the corresponding chunk, disallowing them to differ by an expression *sexp*. This requires a programmer to also perform a split after pointer arithmetic, if he still wishes to manipulate the result. This simplifies the below 2 axioms slightly, but mostly simplifies the back-translation.

$$\frac{}{\begin{array}{c}\{n : id_{\text{prog},\gamma} \mapsto exp * \text{length}(exp) == exp_1 \\ * \ 0 \leq sexp_{1,\gamma} < exp_1\}_\gamma \\ id_{\text{prog}}[sexp_1] = sexp_2 \ \{n : id_{\text{prog},\gamma} \mapsto \text{update}(exp, sexp_{1,\gamma}, sexp_{2,\gamma})\}_\gamma\end{array}} \quad \text{(ArrayMut)}$$

$$\frac{id_{\text{prog}} \in \text{dom}(\gamma) \qquad \gamma' = \gamma[id_{\text{prog}} : exp_{\text{read}}]}{\begin{array}{c}\{n : sexp_{1,\gamma} \mapsto exp * exp[sexp_{2,\gamma}] == exp_{\text{read}}\}_\gamma \\ id_{\text{prog}} = sexp_1[sexp_2] \ \{n : sexp_{1,\gamma} \mapsto exp\}_{\gamma'}\end{array}} \quad \text{(ArrayLkup)}$$

$$\frac{}{\{P\} \ \text{guard}(sexp) \ \{P * sexp_\gamma\}} \quad \text{(Guard)}$$

3.3.2 Function application axiom. The variables $\overline{n}$ in the below rule are chosen fresh so we do not have to care about redeclaring existing chunks. Renaming chunks has no impact on the proof, as there are no conditions over the chunk names themselves; they are only used during compilation. Renaming can happen through the Consequence rule. The rule renames the $\overline{result}$ variables to fresh dummy variables $\overline{id_{\text{res}}}$.

$$\frac{\begin{array}{c}\Sigma(f) = \{PRE_f, POST_f, \overline{id_{\text{arg}}}\} \\ PRE_f \approx_{\text{Names}} PRE \qquad POST_f \approx_{\text{Names}} POST \\ \overline{id} \in \text{dom}(\gamma) \qquad \gamma' = \gamma[\overline{id} : \overline{id_{\text{res}}}] \qquad \overline{id_{\text{res}}}, \overline{n} \text{ fresh} \\ [subst_{\text{pre}}] = [\overline{id_{\text{arg}}} \mapsto \overline{sexp}_\gamma] \\ [subst_{\text{post}}] = [subst_{\text{pre}}][\overline{result} \mapsto \overline{id_{\text{res}}}]\end{array}}{\{PRE[subst_{\text{pre}}]\}_\gamma \ \overline{id} = f(\overline{sexp}) \ \{POST[subst_{\text{post}}]\}_{\gamma'}} \quad \text{(FApp)}$$

Code-level holes $\bullet^{\overline{id}=\overline{sexp}}_{POST,PRE}$, resulting from FApp calls in the operational semantics above, are handled as a special kind of ad-hoc function call by the above FApp rule. They are seen as functions with name $\bullet$, for which the following ad-hoc environment entry is used: $\Sigma(\bullet) = \{\text{true}, POST, \overline{id}\}$ and where the expressions $\overline{sexp}$ are used as arguments to the function call. No separate separation logic axiom is needed for them.

3.3.3 Rules for return, consequence and frame. The below rules are needed as extra rules to be able to complete proof trees, but are pretty standard in separation logic formalizations and hence presented last.

- *Return rule:* As every function has one explicit return statement, written as just a $\overline{sexp}$, there is no need for an explicit return rule. We do, however, have to define a rule for sequencing a *sexp* and a *sstm*.
  In this case, *sstm* is allowed to be the empty statement as well, this is for the case where a function's body consists only of a return statement.

$$\frac{\{P\}_\gamma \ sstm \ \{Q\}_{\gamma'}}{\{P\}_\gamma \ sstm;\ \text{return}\ \{\overline{sexp}\}\ \{Q * \overline{result} == \overline{sexp_{\gamma'}}\}_{\gamma'}} \quad \text{(Return)}$$

- *Consequence* The Consequence rule is the glue between different axioms. It works by means of strengthening preconditions and weakening postconditions. Strengthening preconditions is not required here, as proofs can just be described start-to-end. The consequence rule is applied modulo some logical theory, and hence formalizes a full-fledged SMT solver. Luckily, we do not need to formalize this SMT solver, as we assume the proof to be given.
  Consequence also includes chunk leaking.
  Actually, the Consequence rule below should also be applicable to the Return axiom results, as it needs to drop the result variables again if they do not appear in the function contract's postcondition. For this reason, we write c instead of *sexp* in the below 2 rules, where c represents an *sstm* or a *sstm*; return $\overline{sexp}$.
  We do not allow forgetting parts of γ as we do not allow different branches of an If-statement or For-loops to contain declarations. Declarations also cause problems because variables should go out of scope after an if, which is what happens automatically at the logical level because the symbolic variables of the different if branches will be different, but does not happen automatically at the level of the operational semantics, and is a pain to formalize in the current operational semantics.
  Another **important assumption** (without loss of generality) we make is that the values of symbolic variables are never reused, within the same function's body, when they have gone unused in the current separation logic assertion. The reason for this is that parts of the formalization (ie. the δ function defined further on) depend on the fact that logical variables always refer to the same symbolic concept. If at any point in any of the proofs in this report a symbolical variable reappears, this will happen for aesthetic reasons and because it makes explanations easier, but the reader should assume implicit renamings for all these cases. This also means that after a function call, any logical variables that appear in the called function's postcondition but not in the precondition are implicitly renamed if the name in question already existed as a logical variable name in the caller's separation logic proof before the call.

$$\frac{\begin{array}{c}\{P\}_\gamma \ c \ \{Q\}_{\gamma'} \\ \text{dom}(\gamma_{\text{post}}) == \text{dom}(\gamma') \qquad \text{dom}(\gamma_{\text{pre}}) == \text{dom}(\gamma) \\ \forall x \in \text{dom}(\gamma_{\text{post}}).\ Q \vdash \gamma'(x) == \gamma_{\text{post}}(x) \\ \forall x \in \text{dom}(\gamma).\ P \vdash \gamma_{\text{pre}}(x) == \gamma(x) \\ P_{\text{leak}} \subseteq P_{\text{pre}} \qquad\qquad Q_{\text{leak}} \subseteq Q \\ P_{\text{leak}} \approx_{\text{Names}} P_{\text{rename}} \qquad Q_{\text{leak}} \approx_{\text{Names}} Q_{\text{rename}} \\ \vdash P_{\text{rename}} \Rightarrow P \qquad \vdash Q_{\text{rename}} \Rightarrow Q_{\text{post}}\end{array}}{\{P_{\text{pre}}\}_{\gamma_{\text{pre}}} \ c \ \{Q_{\text{post}}\}_{\gamma_{\text{post}}}} \quad \text{(Conseq)}$$

- *Frame rule* This is one of the most classical separation logic axioms out there and one of the reasons for the power and general usability of separation logic.

$$\frac{\begin{array}{c}\{P\}_\gamma \ c \ \{Q\}_{\gamma'} \\ \text{CN}(R) = \overline{n} \qquad \overline{n} \text{ fresh} \\ \gamma_s = \gamma \uplus \gamma_{\text{frame}} \qquad \gamma_s' = \gamma' \uplus \gamma_{\text{frame}}\end{array}}{\{P * R\}_{\gamma_s} \ c \ \{Q * R\}_{\gamma_s'}} \quad \text{(Frame)}$$

3.3.4 Rules for functions.

$$\frac{\begin{array}{c} isfunc = \overline{\tau_{\text{ret}}}\ f(\overline{\tau_{\text{arg}}\ id_{\text{arg}}})//@\text{pre}\ PRE\ //@\text{post}\ POST \\ \{PRE\}_{[\overline{id_{\text{arg}}:id_{\text{arg}}}]}\ sstm; \text{return}\ \overline{sexp}\ \{POST\}_\gamma \end{array}}{\vdash isfunc\ \{sstm; \text{return}\ \overline{sexp}\}} \quad \text{(ImplFVerif)}$$

$$\frac{csfunc = \overline{\tau_{\text{ret}}}\ f(\overline{\tau_{\text{arg}}\ id_{\text{arg}}})//@\text{pre}\ PRE\ //@\text{post}\ POST}{\vdash csfunc} \quad \text{(ContFVerif)}$$

3.3.5 Rules for components and programs.

Well-formdness/well-scopedness rules. The rules below are well-formdness/well-scopedness rules for separation logic contracts of boundary functions. Also, $\vdash_{name}$ denotes affine closedness (all variables to the left are used 0 or 1 times to the right).

$$\frac{x, x_1, \ldots, x_k\ distinct}{x, x_1, \ldots, x_k \vdash_{\text{s}} n : x \mapsto [x_1, \ldots, x_k]} \quad \text{(ArrayWF)}$$

The operation $\oplus$ is disjoint environment concatenation.

$$\frac{\Gamma \vdash_{\text{s}} C \qquad \Gamma' \vdash_{\text{s}} C'}{\Gamma \oplus \Gamma' \vdash_{\text{s}} C * C'} \quad \text{(SepConjChunkWF)}$$

The function $FV(\cdot)$ returns a set of all variable names id_{log} appearing free in an expression *exp* or a symbolic assertion P.

$$\frac{\begin{array}{c} FV(exp) = \{x_1, \ldots, x_k\} \\ \{x_1, \ldots, x_k\} \subseteq \Gamma \end{array}}{\Gamma \vdash_{\text{p}} exp} \quad \text{(ConditionWF)}$$

$$\frac{\Gamma \vdash_{\text{p}} C \qquad \Gamma \vdash_{\text{p}} C'}{\Gamma \vdash_{\text{p}} C * C'} \quad \text{(SepConjCondWF)}$$

Remember; the contract of any boundary function has to be separable into a spatial heap $PRE/POST_{\text{s}}$ and a pure heap $PRE/POST_{\text{s}}$, as described before.

$$\frac{\begin{array}{c}\Gamma = \overline{id_{\text{arg}}} \\ \Delta_{\text{pre}} \vdash_{\text{s}} \overline{m} : PRE_{\text{s}} \qquad \Gamma \oplus \Delta_{\text{pre}} \vdash_{\text{p}} PRE_{\text{p}} \\ \Delta_{\text{post}} \vdash_{\text{s}} \overline{n} : POST_{\text{s}} \qquad \Gamma \oplus \Delta_{\text{pre}} \oplus \Delta_{\text{post}} \vdash_{\text{p}} POST_{\text{p}} \\ PRE_{\text{s}} = \overline{id_{\text{pre}} \mapsto [\ldots]} \qquad PRE_{\text{s}} = \overline{id_{\text{post}} \mapsto [\ldots]} \\ (\overline{id_{\text{pre}}} == \overline{exp_m}) \subseteq PRE_{\text{p}} \\ (\overline{id_{\text{post}}} == \overline{exp_n}) \subseteq POST_{\text{p}} \\ FV(\overline{exp_m}) = v_1 \qquad FV(\overline{exp_n}) = v_2 \\ v_1 \subseteq \overline{id_{\text{arg}}} \qquad v_2 \subseteq \overline{id_{\text{arg}}} \cup \overline{result} \cup \Delta_{\text{pre}}\end{array}}{\vdash_{\text{WFBD}} \overline{\tau_{\text{ret}}}\ f(\overline{\tau_{\text{arg}}\ id_{\text{arg}}})//@\text{pre}\ \overline{m} : PRE_{\text{s}} * PRE_{\text{p}}\ //@\text{post}\ \overline{n} : POST_{\text{s}} * POST_{\text{p}}}$$

(BoundaryContrWF)

Component/program rules. The imported and exported functions are forced to be disjoint by the UniqueId requirement combined with the Subset requirement. The predicate UniqueId($\overline{fdecl}$) enforces the function names in the given set of function declarations to be different. The predicate Subset($\overline{fdecl}, \overline{f}$) enforces all left-hand function declarations to reappear in the right-hand function bodies.

$$\frac{\begin{array}{c}\text{UniqueId}(\overline{isfunc}\ \overline{csfunc_{\text{i}}}) \qquad \text{UniqueId}(\overline{ctfunc_{\text{e}}}) \\ \text{Subset}(\overline{ctfunc_{\text{e}}}, \overline{isfunc}) \\ \vdash_{\text{WFBD}} \overline{csfunc_{\text{i}}} \qquad \vdash_{\text{WFBD}} \overline{ctfunc_{\text{e}}}\end{array}}{\vdash_{\text{WF}} \overline{isfunc}\ //@\text{import}\ \overline{csfunc_{\text{i}}}\ //@\text{export}\ \overline{ctfunc_{\text{e}}}} \quad \text{(CompWF)}$$

The component environment Σ_i has to be passed downwards from this rule, because the component falls apart at this point. This operation of passing the environment throughout the separation logic derivation is performed implicitly; for every rule in the proof tree below this one, a subscript Σ can be added mentally to the precondition of every triple.

$$\frac{\begin{array}{c}scomp = \overline{isfunc}//@\text{import}\ \overline{csfunc_{\text{i}}}\ //@\text{export}\ \overline{csfunc_{\text{e}}} \\ \vdash_{\text{WF}} scomp \\ \forall x \in \overline{isfunc}.\vdash x \\ \forall x \in \overline{csfunc_{\text{i}}}.\vdash x \\ \forall x \in \overline{csfunc_{\text{e}}}.\vdash x\end{array}}{\vdash scomp} \quad \text{(CompVerif)}$$

Every function needs a return type, so main does too here. Because we are not really interested in the concrete value our main programs output, we restrict the main function to have the void type as its return type. It has the trivial precondition as precondition, and we do not care about its

postcondition.

$$\frac{\begin{array}{c}\overline{C} = C_1\ \ldots\ C_j\ \ldots\ C_k \qquad C_j = \overline{isfunc_j}\ \texttt{//@import}\ \overline{csfunc_{\mathrm{i}j}}\ \texttt{//@export}\ \overline{csfunc_{\mathrm{e}j}} \\ \forall j \in \{1..k\}.\, \forall l \in \{1..k\}.\, j \neq l \Rightarrow \mathrm{UniqueId}(\overline{isfunc_j}\ \overline{isfunc_l}) \\ \forall j \in \{1..k\}.\, \mathrm{Subset}(\overline{csfunc_{\mathrm{i}j}}, \overline{csfunc_{\mathrm{e}1}}\ \ldots\ \overline{csfunc_{\mathrm{e}k}}) \\ \exists j \in \{1..k\}.\, \mathrm{Subset}((\mathrm{void}\ id()\ \texttt{//@pre true //@post \_ }), \overline{csfunc_{\mathrm{e}j}})\end{array}}{\vdash_{\mathrm{WF}} \overline{C}\ \texttt{//@main} = id} \quad (\textsc{ProgWF})$$

$$\frac{sprog = \overline{C}\ \texttt{//@main} = id \qquad \vdash_{\mathrm{WF}} sprog \qquad \forall X \in \overline{C}.\, \vdash X}{\vdash sprog} \quad (\textsc{ProgVerif})$$

4 COMPILATION RULES

The axioms written down in section 3 demonstrate what separation logic proofs look like. On the other hand, they are also an aid to write down the compilation rules presented in this section; all axioms are implicitly present in the compilation rules, as the astute reader will immediately notice.

4.1 Basic statement rules

$$\frac{}{\{\}\ \mathrm{skip}\ \{\} \rightsquigarrow \mathrm{skip}} \quad (\textsc{Skip})$$

The below CompileType rules can be used to get the correct type for the contents of a reified linear capability.

$$\frac{}{\mathrm{int} \rightsquigarrow_{\mathrm{CompileType}} \mathrm{int}} \quad (\textsc{CompileInt})$$

$$\frac{}{\tau* \rightsquigarrow_{\mathrm{CompileType}} \tau*_0} \quad (\textsc{CompileSrcPtr})$$

$$\frac{\tau_1 \rightsquigarrow_{\mathrm{CompileType}} \tau_1' \qquad \ldots \qquad \tau_k \rightsquigarrow_{\mathrm{CompileType}} \tau_k'}{(\tau_1, \ldots, \tau_k) \rightsquigarrow_{\mathrm{CompileType}} (\tau_1', \ldots, \tau_k')} \quad (\textsc{CompileTuple})$$

$$\frac{\begin{array}{c}\tau \rightsquigarrow_{\mathrm{def}} v \qquad \tau \rightsquigarrow_{\mathrm{CompileType}} \tau' \qquad n, id_{\mathrm{log}}\ \mathrm{fresh} \\ id_{\mathrm{prog}} \in \mathrm{dom}(\gamma) \qquad \gamma' = \gamma[id_{\mathrm{prog}} : id_{\mathrm{log}}]\end{array}}{\begin{array}{c}\{sexp_\gamma > 0\}_\gamma\ id_{\mathrm{prog}} = \mathrm{malloc}(sexp * \mathrm{sizeof}(\tau)) \\ \{n : id_{\mathrm{log}} \mapsto_\tau \mathrm{repeat}(sexp_\gamma, v)\}_{\gamma'} \\ \rightsquigarrow \tau' * n;\ n = \mathrm{malloc}(sexp * \mathrm{sizeof}(\tau')); id_{\mathrm{prog}} = \mathrm{addr}(n)\end{array}} \quad (\textsc{Malloc})$$

The judgment $resource \rightsquigarrow_{\mathrm{ChunkType}} \overline{\tau_{\mathrm{t}}}$ returns the tuple of types the given separation-logic *resource* will reify to.

$$\frac{}{exp \rightsquigarrow_{\mathrm{ChunkType}} \epsilon} \quad (\textsc{ExpResType})$$

$$\frac{C \rightsquigarrow_{\text{ChunkType}} \tau_1 \qquad C' \rightsquigarrow_{\text{ChunkType}} \tau_2}{C * C' \rightsquigarrow_{\text{ChunkType}} \tau_1, \tau_2} \tag{ChunkAssertToDecl}$$

$$\frac{assert \rightsquigarrow_{\text{ChunkType}} \tau}{exp\ ?\ assert \rightsquigarrow_{\text{ChunkType}} \tau} \tag{CondResType}$$

$$\frac{\tau \rightsquigarrow_{\text{CompileType}} \tau'}{exp \mapsto_{\tau} exp' \rightsquigarrow_{\text{ChunkType}} \tau'*} \tag{ArrayResType}$$

$$\frac{assert \rightsquigarrow_{\text{ChunkType}} \tau_1, \ldots, \tau_k}{[assert \mid exp \leq i < exp'] \rightsquigarrow_{\text{ChunkType}} (\tau_1, \ldots, \tau_k)*} \tag{RangeResType}$$

Notice how we do not need built-in functions for flatten and collect. The reason for this is that we are just moving a known amount of capabilities outside of a wrapping capability. There is no manipulation of the capabilities itself needed. This simplifies the back-translation, because we do not have to create back-translation rules for flatten and collect.

$$\frac{\begin{array}{c} assert[i \mapsto k_1 \ldots k_2 - 1] \rightsquigarrow_{\text{NameChunks}} \overline{assert'}, \overline{n_{ij}} \\ \overline{n_{ij}}\ \text{fresh} \qquad assert \rightsquigarrow_{\text{ChunkType}} \overline{\tau_j} \qquad \overline{\tau_{ij}} = \overline{\tau_j}^{(k_2 - k_1)} \end{array}}{\begin{array}{c} \{n : [assert \mid k_1 \leq i < k_2]\}_{\gamma} \\ //@\text{flatten}\ n \\ \{\overline{assert'}\}_{\gamma} \\ \rightsquigarrow \overline{\tau_{ij}\ n_{ij}}; \overline{(\tau_j)\ n_i}; \overline{n_i = n[i]}; \overline{n_{ij} = n_i.j} \end{array}} \tag{Flatten}$$

$$\frac{\begin{array}{c} assert[i \mapsto 0 \ldots k - 1] \rightsquigarrow_{\text{UnNameChunks}} assert'[i \mapsto 0 \ldots k - 1], \overline{n_{ij}} \\ n\ \text{fresh} \qquad [assert' \mid 0 \leq i < k] \rightsquigarrow_{\text{ChunkType}} \tau* \end{array}}{\begin{array}{c} \{assert[i \mapsto 0] * \ldots * assert[i \mapsto k - 1] * k > 0\}_{\gamma} \\ //@\text{collect}\ \overline{n_{0j}} \cdot \ldots \cdot \overline{n_{(k-1)j}} \\ \{n : [assert' \mid 0 \leq i < k]\}_{\gamma} \\ \rightsquigarrow \tau* n; n = \text{malloc}(k * \text{sizeof}(\tau)); n[0] = (\overline{n_{0j}}); \ldots; n[k-1] = (\overline{n_{(k-1)j}}) \end{array}} \tag{Collect}$$

To avoid errors because we redeclare newly-reified target declarations in every iteration of a compiled for loop, we need to define declaration hoisting for the compilation of for loops.

$$\frac{}{\epsilon \rightsquigarrow_{\text{ExtractDecl}} \epsilon, \epsilon} \tag{GatherDecl}$$

$$\frac{\overline{stm} \rightsquigarrow_{\text{ExtractDecl}} decl, nondecl}{\tau\ id; \overline{stm} \rightsquigarrow_{\text{ExtractDecl}} \tau\ id; decl, nondecl} \tag{GatherDecl}$$

$$\frac{\begin{array}{c} stm \neq \tau\ id \\ \overline{stm} \rightsquigarrow_{\text{ExtractDecl}} decl, nondecl \end{array}}{stm; \overline{stm} \rightsquigarrow_{\text{ExtractDecl}} decl, stm; nondecl} \tag{GatherNonDecl}$$

$$\frac{\begin{array}{c}\{sexp_{\gamma_{\text{pre}}} \leq i_{\text{s}} < sexp'_{\gamma_{\text{pre}}} * Inv[i_{\text{s}}]\}_{\gamma[i_{\text{s}}]}\ sstm\ \{Inv[i_{\text{s}}+1]\}_{\gamma[i_{\text{s}}+1][i:\_]} \rightsquigarrow p \\ \gamma[i_{\text{s}}](i) = i_{\text{s}} \qquad \gamma[sexp_{\gamma_{\text{pre}}}] = \gamma_{\text{pre}}[i : sexp_{\gamma_{\text{pre}}}] \qquad \gamma[sexp'_{\gamma_{\text{pre}}}] = \gamma_{\text{post}} \\ p \rightsquigarrow_{\text{ExtractDecl}} decl, nondecl\end{array}}{\begin{array}{c}\{Inv[sexp_{\gamma_{\text{pre}}}]\}_{\gamma_{\text{pre}}}\ \text{foreach}(sexp \leq i < sexp')\{sstm\}\ \{Inv[sexp'_{\gamma_{\text{pre}}}]\}_{\gamma_{\text{post}}} \rightsquigarrow \\ decl; \text{foreach}(sexp \leq i < sexp')\{nondecl\}\end{array}} \quad \text{(For)}$$

$$\frac{n', n'' \text{ fresh} \qquad \tau \rightsquigarrow_{\text{CompileType}} \tau'}{\begin{array}{c}\{n : exp_{\text{a}} \mapsto_{\tau} l * \text{length}(l) == exp_{\text{l}} * 0 < sexp_{\gamma} < exp_{\text{l}}\}_{\gamma} \\ //@\text{split}\ n[sexp] \\ \{n' : exp_{\text{a}} \mapsto_{\tau} \text{take}(l, 0, sexp_{\gamma}) \\ * n'' : (exp_{\text{a}} + sexp_{\gamma}) \mapsto_{\tau} \text{take}(l, sexp_{\gamma}, exp_{\text{l}})\}_{\gamma} \\ \rightsquigarrow \tau' * n'; \tau' * n''; \{n', n''\} = \text{split}(n, sexp)\end{array}} \quad \text{(Split)}$$

$$\frac{n', n'' \text{ fresh} \qquad [assert \mid exp \leq i < exp'] \rightsquigarrow_{\text{ChunkType}} \tau}{\begin{array}{c}\{n : [assert \mid exp \leq i < exp'] * 0 < sexp_{\gamma} < exp' - exp\}_{\gamma} \\ //@\text{split}\ n[sexp] \\ \{n' : [assert \mid exp \leq i < exp + sexp_{\gamma}] * \\ n'' : [assert \mid exp + sexp_{\gamma} \leq i < exp']\}_{\gamma} \\ \rightsquigarrow \tau\ n'; \tau\ n''; \{n', n''\} = \text{split}(n, sexp)\end{array}} \quad \text{(SplitRange)}$$

$$\frac{n, l \text{ fresh} \qquad \tau \rightsquigarrow_{\text{CompileType}} \tau'}{\begin{array}{c}\{n' : exp'_{\text{a}} \mapsto_{\tau} l' * \text{length}(l') == exp'_{\text{l}} \\ * \ n'' : exp''_{\text{a}} \mapsto_{\tau} l'' * \text{length}(l'') == exp''_{\text{l}} \\ * exp''_{\text{a}} = exp'_{\text{a}} + exp'_{\text{l}} + 1\}_{\gamma} \\ //@\text{join}\ n'\ n'' \\ \{n : exp'_{\text{a}} \mapsto_{\tau} \text{append}(l', l'')\}_{\gamma} \\ \rightsquigarrow \tau' * n; n = \text{join}(n', n'')\end{array}} \quad \text{(Join)}$$

Notice that we cannot use the built-in target language join statement for the below compilation rule. The reason is that we cannot be sure that the two given linear capabilities, that both represent the reifications of arbitrary range statements, are adjacent in memory. It is hence necessary to allocate a new linear capability and copy the contents of the other two linear capabilities to this one. Notice that we could have used this technique in all of the above compilation rules, to avoid any uses of split and join, but this would be more memory-inefficient, and would forgo part of the power of linear capabilities and the capability machines they are implemented on.

$$\frac{n \text{ fresh} \qquad [assert \mid exp \leq i < exp'] \rightsquigarrow_{\text{ChunkType}} \tau *}{\begin{array}{c}\{n' : [assert \mid exp \leq i < exp'] * \\ n'' : [assert \mid exp' \leq i < exp'']\}_{\gamma} \\ //@\text{join}\ n'\ n''' \\ \{n : [assert \mid exp \leq i < exp'']\}_{\gamma} \\ \rightsquigarrow \tau * n; n = \text{malloc}((exp'' - exp) * \text{sizeof}(\tau)); \\ \text{foreach}(exp \leq i < exp')\{n[i] = n'[i]\}; \\ \text{foreach}(exp \leq i < exp'')\{n[i] = n''[i - exp']\}\end{array}} \quad \text{(JoinRange)}$$

$$\frac{\{P\}_\gamma\ sstm_1\ \{Q\}_{\gamma'} \rightsquigarrow p_1 \quad \{Q\}_{\gamma'}\ sstm_2\ \{R\}_{\gamma''} \rightsquigarrow p_2}{\{P\}_\gamma\ sstm_1; sstm_2\ \{R\}_{\gamma''} \rightsquigarrow p_1; p_2} \tag{Seq}$$

For the below compilation rule to work, we forbid the appearance of the same resource with different ReifiesToType-values in different If statement branches in the source language. This makes sure that the union $decl_1 \cup decl_2$ does not contain a double declaration for the same variable. The declarations in the below rule are hoisted outside of the If statement to make sure that no declarations occur inside of If statements in the target language either. Notice that this hoisting works for nested If statements as well. This causes no problems with the correctness relation R, as it allows for extra null-variables in the target language that do not (yet) correspond to resources in the source language proof. Hoisting happens analogously to the case for For statements.

$$\frac{\begin{array}{c}\{P * sexp_\gamma\}_\gamma\ sstm_1\ \{Q\}_{\gamma'} \rightsquigarrow p_1 \\ \{P * !sexp_\gamma\}_\gamma\ sstm_2\ \{Q\}_{\gamma'} \rightsquigarrow p_2 \\ p_1 \rightsquigarrow_{\text{ExtractDecl}} decl_1, nondecl_1 \\ p_2 \rightsquigarrow_{\text{ExtractDecl}} decl_2, nondecl_2 \\ decl = decl_1 \cup decl_2\end{array}}{\begin{array}{c}\{P\}_\gamma\ \text{if } sexp \text{ then } sstm_1 \text{ else } sstm_2\ \{Q\}_{\gamma'} \\ \rightsquigarrow decl; \text{if } sexp \text{ then } nondecl_1 \text{ else } nondecl_2\end{array}} \tag{If}$$

$$\frac{id_{\text{prog}} \notin \text{dom}(\gamma) \quad \tau \rightsquigarrow_{\text{def}} v \quad \tau \rightsquigarrow_{\text{CompileType}} \tau' \quad \gamma' = \gamma[id_{\text{prog}} : v]}{\{\}_\gamma\ \tau\ id_{\text{prog}}\ \{\}_{\gamma'} \rightsquigarrow \tau'\ id_{\text{prog}}} \tag{VarDecl}$$

$$\frac{id_{\text{prog}} \in \text{dom}(\gamma) \quad \gamma' = \gamma[id_{\text{prog}} : sexp_\gamma]}{\{\}_\gamma\ id_{\text{prog}} = sexp\ \{\}_{\gamma'} \rightsquigarrow id_{\text{prog}} = sexp} \tag{VarAsgn}$$

Statements like $a = a + 1$ for a source pointer a (which corresponds to a chunk n) are left untouched during compilation, as can be seen in the VarAsgn compilation rule. Only during array access (mutation or lookup) is the pointer arithmetic calculated in, because a's value is used. Because arithmetic isn't translated on a pointer level, the linear capabilities we use for arrays will *always* point to the first element in their domain, rendering their concrete location useless. This is the reason we were able to formalize linear capabilities without an index i, and it makes the operational semantics and the further proof somewhat easier.

$$\frac{}{\begin{array}{c}\{n : id_{\text{prog},\gamma} \mapsto exp * \text{length}(exp) == exp_1 \\ * 0 \leq sexp_{1,\gamma} < exp_1\}_\gamma \\ id_{\text{prog}}[sexp_1] = sexp_2\ \{n : id_{\text{prog},\gamma} \mapsto \text{update}(exp, sexp_{1,\gamma}, sexp_{2,\gamma})\}_\gamma \\ \rightsquigarrow n[sexp_1] = sexp_2\end{array}} \tag{ArrayMut}$$

$$\frac{id_{\text{prog}} \in \text{dom}(\gamma) \qquad \gamma' = \gamma[id_{\text{prog}} : exp_{\text{read}}]}{\begin{array}{c}\{n : sexp_{1,\gamma} \mapsto exp * exp[sexp_{2,\gamma}] == exp_{\text{read}}\}_{\gamma} \\ id_{\text{prog}} = sexp_1[sexp_2]\ \{n : sexp_{1,\gamma} \mapsto exp\}_{\gamma'} \\ \rightsquigarrow id_{\text{prog}} = n[sexp_2]\end{array}} \quad \text{(ArrayLkup)}$$

$$\frac{}{\{P\}\ \text{guard}(sexp)\ \{P * sexp_{\gamma}\} \rightsquigarrow \text{guard}(sexp)} \quad \text{(Guard)}$$

4.2 Function application compilation rule

Before we can get into the details of how the function application compilation rule works, we have to introduce a new kind of function that pops up during compilation: the *stub* function. Generally speaking, stubs are target-level functions that enforce the separation logic contracts present at the source level when a transition between trust domains (in this case components) occurs. There are 2 kinds of stubs; *incall and outcall stubs*.

Outcall stub For each function outcall from one component to another, a function *outcall stub* has to be created and added to the outcalling components' function context Σ_i. This stub is a wrapper around the original function call, that enforces the called function's postcondition. Given the called function is f, this stub will be called f_{comp} (for compiled function) so that the *csfunc* can keep its name. The outcall stub asserts whether all constraints from the called function's postcondition hold after the function call. To do this, variables have to be stored before the function outcall to avoid them being overwritten. The precondition does not need to be checked, as an outcall stub is a tool for a caller to check whether a callee behaves properly. The calling component of course trusts itself. Outcall stubs are formalized in the Outcall rule below, when we have all necessary notation.

Incall stub For each function incall from one component to another, a function *incall stub* has to be created and added to the incalled components' function context Σ_i. The concept is symmetrical to that of an outcall stub, but now the untrusted party is the caller and not the callee. When an implemented function is called from an unverified context, we are unsure whether or not its precondition will be upheld. The stub is a wrapper around the called function, that enforces the called function's precondition. The postcondition does not need to be checked, as an incall stub is a tool for a callee to check whether a caller behaves properly. The called component of course trusts itself. Incall stubs are formalized in the Incall rule below, when we have all necessary notation. Given an implemented function f, this incall stub will be named f (to avoid changing the caller code) and the old function f_{comp}. The new stub function f will now be exported instead of f_{comp}.

We use the new name f_{comp} for both incall and outcall stubs to make compilation of function names easier and more uniform. Every function f mentioned in a component, will now have a compiled version f_{comp}, and also a stub-related f if it is an imported or exported function.

To define the FApp rule, we first need rules that allow us to reify separation logic assertions to target-level program variable declarations. The rule *assert* $\rightsquigarrow_{\text{ResDecl}}$ *decl* reifies a target-level declaration from a given impure separation-logic assertion. The auxiliary judgment *resource* $\rightsquigarrow_{\text{ChunkType}} \overline{\tau_t}$ was defined above and returns the tuple of types the given separation-logic *resource* will reify to.

$$\frac{}{exp \rightsquigarrow_{\text{ResDecl}} \epsilon} \quad \text{(ExpToDecl)}$$

$$\frac{C \rightsquigarrow_{\text{ResDecl}} decl_1 \qquad C' \rightsquigarrow_{\text{ResDecl}} decl_2}{C * C' \rightsquigarrow_{\text{ResDecl}} decl_1\ decl_2} \quad \text{(ChunkAssertToDecl)}$$

$$\frac{assert \rightsquigarrow_{\text{ResDecl}} decl}{exp\ ?\ assert \rightsquigarrow_{\text{ResDecl}} decl} \quad \text{(CondToDecl)}$$

$$\frac{exp \mapsto_\tau exp' \rightsquigarrow_{\text{ChunkType}} \tau}{n : exp \mapsto_\tau exp' \rightsquigarrow_{\text{ResDecl}} \tau\ n;} \quad \text{(FixChunkToDecl)}$$

$$\frac{[assert \mid exp \le i < exp'] \rightsquigarrow_{\text{ChunkType}} \tau}{n : [assert \mid exp \le i < exp'] \rightsquigarrow_{\text{ResDecl}} \tau\ n;} \quad \text{(RangeChunkToDecl)}$$

We also define the function ReifiesToType, that takes a resource name n as an argument and returns the type of the chunk it reifies to, based on the types in the rules FixChunkToDecl and RangeChunkToDecl for ResDecl: pointers for regular points-to chunks and tuples for array chunks.

$$\frac{\begin{array}{c}\Sigma(f) = \{PRE_f, POST_f, \overline{id_{\text{arg}}}\} \\ PRE_f \approx_{\text{Names}} PRE \qquad POST_f \approx_{\text{Names}} POST \\ \overline{id} \in \text{dom}(\gamma) \qquad \gamma' = \gamma[\overline{id} : \overline{id_{\text{res}}}] \qquad \overline{id_{\text{res}}}, \overline{n} \text{ fresh} \\ [subst_{\text{pre}}] = [\overline{id_{\text{arg}}} \mapsto \overline{sexp}_\gamma] \\ [subst_{\text{post}}] = [subst_{\text{pre}}][\overline{result} \mapsto \overline{id_{\text{res}}}] \\ \text{CN}(PRE) = \overline{m} \qquad POST \rightsquigarrow_{\text{resDecl}} \overline{\tau_n\ n}\end{array}}{\begin{array}{c}\{PRE[subst_{\text{pre}}]\}_\gamma\ \overline{id} = f(\overline{sexp})\ \{POST[subst_{\text{post}}]\}_{\gamma'} \rightsquigarrow \\ \overline{\tau_n\ n};\ \{\overline{id}, \overline{n}\} = f_{\text{comp}}(\overline{sexp}, \overline{m})\end{array}} \quad \text{(FApp)}$$

Code-level holes $\bullet^{\overline{id}=\overline{sexp}}_{POST, PRE}$ in the source will compile to regular non-annotated code-level holes $\bullet^{\overline{id}=\overline{sexp}}$ in the target.

4.3 Rules for return, consequence and frame

$$\frac{\{P\}_\gamma\ sstm\ \{Q\}_{\gamma'} \rightsquigarrow p \qquad \text{CN}(Q) = \overline{n}}{\begin{array}{c}\{P\}_\gamma\ sstm;\ \text{return}\ \{\overline{sexp}\}\ \{Q * \overline{result} == \overline{sexp_{\gamma'}}\}_{\gamma'} \\ \rightsquigarrow p; \text{return}\ \{\overline{sexp}, \overline{n}\}\end{array}} \quad \text{(Return)}$$

The following rules for conditional chunks should also be part of consequence:

- Addition or removal of a condition that is equivalent to true to/from any chunk. This is eg. needed to deconstruct/reconstruct universal contracts containing chunks.

$$exp == \text{true} \vdash assert \Leftrightarrow exp\ ?\ assert$$

- Adding a condition that is equivalent to false, around a freshly made-up separation logic assertion, or removal of such a condition. This first case results in a null-assignment of all (necessarily fresh) chunk names in the made-up assertion after compilation. This rule equivalence is eg. needed to deconstruct/reconstruct universal contracts containing null-valued variables (although deconstruction can happen through chunk leaking anyway).

$$exp == \text{false} \vdash \text{true} \Leftrightarrow exp \ ?\ assert$$

  The last two lines in the below Consequence rule create the declarations that should be reified in the target language because of this 'making up' of chunks.

The judgment $\rightsquigarrow_{\text{RenameDecl}}$ used below creates declarations in the target language, that allow a transition from the first set of names to the second set of names by using declarations and assignments. Herein, it is assumed that any name that changes is fresh. A name is still fresh if it is only used in a different branch of an if statement. This allows us to rename chunks in different If branches to identical names, which is often required to satisfy the If axiom.

$$\frac{}{\epsilon, \epsilon \rightsquigarrow_{\text{RenameDecl}} \epsilon} \quad \text{(RenameEmpt)}$$

$$\frac{\overline{names_1}, \overline{names_2} \rightsquigarrow_{\text{RenameDecl}} rename}{n_1 :: \overline{names_1}, n_1 :: \overline{names_2} \rightsquigarrow_{\text{RenameDecl}} rename} \quad \text{(RenameEq)}$$

$$\frac{n_1 \neq n_2 \qquad n_2 \text{ fresh} \qquad \text{ReifiesToType}(n_1) = \tau \qquad \overline{names_1}, \overline{names_2} \rightsquigarrow_{\text{RenameDecl}} rename}{n_1 :: \overline{names_1}, n_2 :: \overline{names_2} \rightsquigarrow_{\text{RenameDecl}} \tau\ n_2; n_2 = n_1; rename} \quad \text{(RenameNeqFresh)}$$

Important note: the below rule is the *only* way to reintroduce non-fresh chunk names! In all other places, chunk names are required to be fresh. If they do not appear to be fresh in any place in this report, that means that this rules has been implicitly applied. Allowing non-fresh chunks to pop up in a single place is a simplifying measure that avoids formalizing the same measures on chunk names in multiple places.

$$\frac{n_1 \neq n_2 \qquad n_2 \in \text{dom}(\gamma) \qquad \text{ReifiesToType}(n_1) = \text{ReifiesToType}(n_2) \qquad \overline{names_1}, \overline{names_2} \rightsquigarrow_{\text{RenameDecl}} rename}{n_1 :: \overline{names_1}, n_2 :: \overline{names_2} \rightsquigarrow_{\text{RenameDecl}} n_2 = n_1; rename} \quad \text{(RenameNeqNotFresh)}$$

$$\frac{\begin{array}{c}\{P\}_{\gamma}\ c\ \{Q\}_{\gamma'} \rightsquigarrow p \\ \mathrm{dom}(\gamma_{\mathrm{post}}) == \mathrm{dom}(\gamma') \qquad \mathrm{dom}(\gamma_{\mathrm{pre}}) == \mathrm{dom}(\gamma) \\ \forall x \in \mathrm{dom}(\gamma_{\mathrm{post}}).\ Q \vdash \gamma'(x) == \gamma_{\mathrm{post}}(x) \\ \forall x \in \mathrm{dom}(\gamma).\ P \vdash \gamma_{\mathrm{pre}}(x) == \gamma(x) \\ P_{\mathrm{leak}} \subseteq P_{\mathrm{pre}} \qquad Q_{\mathrm{leak}} \subseteq Q \\ P_{\mathrm{leak}} \approx_{\mathrm{Names}} P_{\mathrm{rename}} \qquad Q_{\mathrm{leak}} \approx_{\mathrm{Names}} Q_{\mathrm{rename}} \\ \mathrm{CN}(P_{\mathrm{leak}}), \mathrm{CN}(P_{\mathrm{rename}}) \rightsquigarrow_{\mathrm{RenameDecl}} \mathit{rename}_{\mathrm{pre}} \\ \mathrm{CN}(Q_{\mathrm{leak}}), \mathrm{CN}(Q_{\mathrm{rename}}) \rightsquigarrow_{\mathrm{RenameDecl}} \mathit{rename}_{\mathrm{post}} \\ \vdash P_{\mathrm{rename}} \Rightarrow P \qquad \vdash Q_{\mathrm{rename}} \Rightarrow Q_{\mathrm{post}} \qquad \overline{n_{\mathrm{pre}}}, \overline{n_{\mathrm{post}}}\ \text{fresh} \\ \mathrm{CN}(P) \setminus \mathrm{CN}(P_{\mathrm{rename}}) = \overline{n_{\mathrm{pre}}} \qquad \mathrm{CN}(Q_{\mathrm{post}}) \setminus \mathrm{CN}(Q_{\mathrm{rename}}) = \overline{n_{\mathrm{post}}} \\ \mathrm{ReifiesToType}(\overline{n_{\mathrm{pre}}}) = \overline{\tau_{\mathrm{pre}}} \qquad \mathrm{ReifiesToType}(\overline{n_{\mathrm{post}}}) = \overline{\tau_{\mathrm{post}}}\end{array}}{\{P_{\mathrm{pre}}\}_{\gamma_{\mathrm{pre}}}\ c\ \{Q_{\mathrm{post}}\}_{\gamma_{\mathrm{post}}} \rightsquigarrow \mathit{rename}_{\mathrm{pre}}; \overline{\tau_{\mathrm{pre}}\ n_{\mathrm{pre}}}; p; \mathit{rename}_{\mathrm{post}}; \overline{\tau_{\mathrm{post}}\ n_{\mathrm{post}}}} \quad (\textsc{Conseq})$$

$$\frac{\begin{array}{c}\{P\}_{\gamma}\ c\ \{Q\}_{\gamma'} \rightsquigarrow p \\ \mathrm{CN}(R) = \overline{n} \qquad \overline{n}\ \text{fresh} \\ \gamma_{\mathrm{s}} = \gamma \uplus \gamma_{\mathrm{frame}} \qquad \gamma'_{\mathrm{s}} = \gamma' \uplus \gamma_{\mathrm{frame}}\end{array}}{\{P * R\}_{\gamma_{\mathrm{s}}}\ c\ \{Q * R\}_{\gamma'_{\mathrm{s}}} \rightsquigarrow p} \quad (\textsc{Frame})$$

4.4 Rules for functions

$$\frac{\begin{array}{c}\mathit{PRE} \rightsquigarrow_{\mathrm{ResDecl}} \overline{\tau_{\mathrm{pre}}\ m} \qquad \mathit{POST} \rightsquigarrow_{\mathrm{ResDecl}} \overline{\tau_{\mathrm{post}}\ n} \\ \overline{\tau_{\mathrm{arg}}} \rightsquigarrow_{\mathrm{CompileType}} \overline{\tau'_{\mathrm{arg}}} \qquad \overline{\tau_{\mathrm{ret}}} \rightsquigarrow_{\mathrm{CompileType}} \overline{\tau'_{\mathrm{ret}}}\end{array}}{\begin{array}{c}\overline{\tau_{\mathrm{ret}}}\ f(\overline{\tau_{\mathrm{arg}}\ id_{\mathrm{arg}}})//@\mathrm{pre}\ \mathit{PRE}\ //@\mathrm{post}\ \mathit{POST} \\ \rightsquigarrow_{\mathrm{Decl}} \{\overline{\tau'_{\mathrm{ret}}}, \overline{\tau_{\mathrm{post}}}\}\ f\ (\overline{\tau'_{\mathrm{arg}}\ id_{\mathrm{arg}}}, \overline{\tau_{\mathrm{pre}}\ m})\end{array}} \quad (\textsc{FDecl})$$

Compilation of functions is handled by the following rules (detailed below) during compilation:

- *Implemented functions*: each implemented function f is compiled to the function f_{comp}, as detailed in ImplFVerif, and if it is called from an unverified context, an incall stub f is also generated, as detailed in Incall.
- *Context functions*: each context function f is compiled to the function f, as detailed in ContFVerif, and if it is called from a verified context, an outcall stub f_{comp} is also generated, as detailed in Outcall.

We should recheck the set $\overline{n}$ below, because Conseq might have leaked resources by now. A simple fix is to disallow Conseq from leaking, renaming or fabricating resources after a return, which we do from now on.

$$\frac{\begin{array}{c}\mathit{isfunc} = \overline{\tau_{\mathrm{ret}}}\ f(\overline{\tau_{\mathrm{arg}}\ id_{\mathrm{arg}}})//@\mathrm{pre}\ \mathit{PRE}\ //@\mathrm{post}\ \mathit{POST} \\ \mathit{isfunc} \rightsquigarrow_{\mathrm{Decl}} \{\overline{\tau'_{\mathrm{ret}}}, \overline{\tau_{\mathrm{post}}}\}\ f(\overline{\tau'_{\mathrm{arg}}\ id_{\mathrm{arg}}}, \overline{\tau_{\mathrm{pre}}\ m}) \\ \{\mathit{PRE}\}_{[\overline{id_{\mathrm{arg}}:id_{\mathrm{arg}}}]}\ \mathit{sstm}; \mathrm{return}\ \overline{\mathit{sexp}}\ \{\mathit{POST}\}_{\gamma} \\ \rightsquigarrow p_1; \mathrm{return}\ \{\overline{\mathit{texp}}, \overline{n}\}\end{array}}{\begin{array}{c}\vdash \mathit{isfunc}\ \{\mathit{sstm}; \mathrm{return}\ \overline{\mathit{sexp}}\} \\ \rightsquigarrow \{\overline{\tau'_{\mathrm{ret}}}, \overline{\tau_{\mathrm{post}}}\}\ f_{\mathrm{comp}}(\overline{\tau'_{\mathrm{arg}}\ id_{\mathrm{arg}}}, \overline{\tau_{\mathrm{pre}}\ m})\ \{p_1; \mathrm{return}\ \{\overline{\mathit{texp}}, \overline{n}\}\}\end{array}} \quad (\textsc{ImplFVerif})$$

$$\frac{csfunc = \overline{\tau_{\text{ret}}}\ f(\overline{\tau_{\text{arg}}\ id_{\text{arg}}})//@\text{pre } PRE\ //@\text{post } POST \qquad csfunc \rightsquigarrow_{\text{Decl}} \{\overline{\tau'_{\text{ret}}}, \overline{\tau_{\text{post}}}\}\ f(\overline{\tau'_{\text{arg}}\ id_{\text{arg}}}, \overline{\tau_{\text{pre}}\ m})}{\vdash csfunc \rightsquigarrow \{\overline{\tau'_{\text{ret}}}, \overline{\tau_{\text{post}}}\}\ f(\overline{\tau'_{\text{arg}}\ id_{\text{arg}}}, \overline{\tau_{\text{pre}}\ m})} \quad (\textsc{ContFVerif})$$

Separation logic assertion desugaring/elaboration The below rules define the necessary checks, declarations and assignment (*store*) statements that are generated from a given separation logic contract. These different pieces are then used in the rules for incall and outcall stubs below.

The names of the variables that are stored are the same as the logical variables that are used in the contracts. This causes no problems, since logical variables are required to be fresh at malloc time, contracts are linear and no argument variables (that would already exist) will be assigned in this way.

Note that in case a variable is not stored and we do have a condition over it, the guard statement created for this condition gets stuck. This enforces the restriction that the pure heap of the reified contract cannot contain symbolic variables that don't appear in either the arguments or in the spatial heap, as was mentioned before when we defined boundary contracts in section 3.

$$\frac{\begin{array}{c}\tau \rightsquigarrow_{\text{CompileType}} \tau' \\ check = (\text{guard}(n\ !{=}\ \text{null}); \text{guard}(\text{length}(n) == k)) \\ decl = (\tau *_0 x;\ \tau'\ x_1;\ \ldots;\ \tau'\ x_k) \\ assign = (x = \text{addr}(n);\ x_1 = n[0];\ \ldots;\ x_k = n[k-1])\end{array}}{n : x \mapsto_\tau [x_1, \ldots, x_k] \rightsquigarrow_{\text{s}} (check, decl, assign)} \quad (\textsc{ResReify})$$

$$\frac{C \rightsquigarrow_{\text{s}} (c_1, d_1, a_1) \qquad C' \rightsquigarrow_{\text{s}} (c_2, d_2, a_2)}{C * C' \rightsquigarrow_{\text{s}} (c_1; c_2, d_1; d_2, a_1; a_2)} \quad (\textsc{SepConjCReify})$$

$$\frac{}{exp \rightsquigarrow_{\text{p}} \text{guard}(exp)} \quad (\textsc{ConditionReify})$$

$$\frac{C \rightsquigarrow_{\text{p}} c_1 \qquad C' \rightsquigarrow_{\text{p}} c_2}{C * C' \rightsquigarrow_{\text{p}} c_1; c_2} \quad (\textsc{SepConjPReify})$$

With the introduction of the reification rules above, we have enough to define incall and outcall stubs in the following two rules. The condition $\overline{n}$ fresh is in place to avoid name clashes between n and m, as we did in FApp.

Reification of symbolic variables in the below rules causes no name clashes, since the only program variables present are the argument and the return values, and they already correctly correspond to logical variables. There is no need to store the argument variables, not even pointers, as their addresses cannot be changed by the call to f, only their contents can, and that is no problem here.

The check cc_{pre} happens before any assignments in s_{pre}, because it makes sense to check if the size of the chunks is correct and if they are not null before starting to assign values. This order is actually required to have a correct back-translation later on. Analogous order issues for cc_{post} versus s_{post} in the Outcall rule below.

To avoid having to declare a renamed version $\overline{n'}$ of $\overline{n}$ that's fresh in the below two rules (due to possible overlap with $\overline{m}$), we introduce the extra requirement that chunk names in the precondition versus the postcondition of function contracts need to be disjoint. This forms no restriction, as we can always rename chunks using the Conseq rule. We will therefore not always explicitly mention or visibly follow this restriction, as it is solely name clash related.

$$\frac{\begin{array}{c} f_\mathrm{e} = \overline{\tau_\mathrm{ret}}\ f(\overline{\tau_\mathrm{arg}\ id_\mathrm{arg}})\text{//@pre } \overline{m} : PRE_\mathrm{s} * PRE_\mathrm{p}\ \text{//@post } \overline{n} : POST_\mathrm{s} * POST_\mathrm{p} \\ p_\mathrm{e} = \{\overline{\tau'_\mathrm{ret}}, \overline{\tau_\mathrm{post}}\}\ f\ (\overline{\tau'_\mathrm{arg}\ id_\mathrm{arg}}, \overline{\tau_\mathrm{pre}\ m}) \\ \vdash f_\mathrm{e} \rightsquigarrow p_\mathrm{e} \\ \overline{m} : PRE_\mathrm{s} \rightsquigarrow_\mathrm{s} (cs_\mathrm{pre}, d_\mathrm{pre}, a_\mathrm{pre}) \\ PRE_\mathrm{p} \rightsquigarrow_\mathrm{p} cp_\mathrm{pre} \\ stub_\mathrm{incall} = \\ \left\{\begin{array}{l} \{\overline{\tau'_\mathrm{ret}}, \overline{\tau_\mathrm{post}}\}\ f\ (\overline{\tau'_\mathrm{arg}\ id_\mathrm{arg}}, \overline{\tau_\mathrm{pre}\ m})\{ \\ cs_\mathrm{pre}; \\ d_\mathrm{pre}; a_\mathrm{pre}; cp_\mathrm{pre}; \\ \overline{\tau'_\mathrm{ret}\ result}; \overline{\tau_\mathrm{post}\ n}; \\ \{\overline{result}, \overline{n}\} = f_\mathrm{comp}(\overline{id_\mathrm{arg}}, \overline{m}); \\ \text{return } \{\overline{result}, \overline{n}\}\} \end{array}\right\} \end{array}}{\vdash f_\mathrm{e} \rightsquigarrow_\mathrm{Incall} p_\mathrm{e}, stub_\mathrm{incall}}\ (\textsc{Incall})$$

$$\frac{\begin{array}{c} f_i = \overline{\tau'_\mathrm{ret}}\ f\ (\overline{\tau_\mathrm{arg}\ id_\mathrm{arg}})\ \text{//@pre } \overline{m} : PRE_\mathrm{s} * PRE_\mathrm{p}\ \text{//@post } \overline{n} : POST_\mathrm{s} * POST_\mathrm{p} \\ p_i = \{\overline{\tau'_\mathrm{ret}}, \overline{\tau_\mathrm{post}}\}\ f\ (\overline{\tau'_\mathrm{arg}\ id_\mathrm{arg}}, \overline{\tau_\mathrm{pre}\ m}) \\ \vdash f_i \rightsquigarrow p_i \\ \overline{m} : PRE_\mathrm{s} \rightsquigarrow_\mathrm{s} (cs_\mathrm{pre}, d_\mathrm{pre}, a_\mathrm{pre}) \\ \overline{n} : POST_\mathrm{s} \rightsquigarrow_\mathrm{s} (cs_\mathrm{post}, d_\mathrm{post}, a_\mathrm{post}) \\ POST_\mathrm{p} \rightsquigarrow_\mathrm{p} cp_\mathrm{post} \\ stub_\mathrm{outcall} = \\ \left\{\begin{array}{l} \{\overline{\tau'_\mathrm{ret}}, \overline{\tau_\mathrm{post}}\}\ f_\mathrm{comp}\ (\overline{\tau'_\mathrm{arg}\ id_\mathrm{arg}}, \overline{\tau_\mathrm{pre}\ m})\{ \\ d_\mathrm{pre}; a_\mathrm{pre}; \\ \overline{\tau'_\mathrm{ret}\ result}; \overline{\tau_\mathrm{post}\ n}; \\ \{\overline{result}, \overline{n}\} = f(\overline{id_\mathrm{arg}}, \overline{m}); \\ cs_\mathrm{post}; \\ d_\mathrm{post}; a_\mathrm{post}; cp_\mathrm{post}; \\ \text{return } \{\overline{result}, \overline{n}\}\} \end{array}\right\} \end{array}}{\vdash f_i \rightsquigarrow_\mathrm{Outcall} p_i, stub_\mathrm{outcall}}\ (\textsc{Outcall})$$

4.5 Rules for components and programs

$$\frac{\begin{array}{c} scomp = \overline{isfunc}\,//@\text{import}\ \overline{csfunc_{\text{i}}}\ //@\text{export}\ \overline{csfunc_{\text{e}}} \\ \vdash_{\text{WF}} scomp \\ \forall x \in \overline{isfunc}.\ \vdash x \rightsquigarrow pv_x \\ \forall x \in \overline{csfunc_{\text{i}}}.\ \vdash x \rightsquigarrow_{\text{Outcall}} pi_x, stub_{\text{out},x} \\ \forall x \in \overline{csfunc_{\text{e}}}.\ \vdash x \rightsquigarrow_{\text{Incall}} pe_x, stub_{\text{in},x} \end{array}}{\begin{array}{c} \vdash scomp \\ \rightsquigarrow (\overline{pv_x}\ \overline{stub_{\text{out},x}}\ \overline{stub_{\text{in},x}})\ //@\text{import}\ \overline{pi_x}\ //@\text{export}\ \overline{pe_x} \end{array}} \quad (\textsc{CompVerif})$$

No need to put id_{old} here, as the incall stub for main will do nothing useful anyway, because its contract is empty.

$$\frac{sprog = \overline{C}\ //@\text{main} = id \qquad \vdash_{\text{WF}} sprog \qquad \forall X \in \overline{C}.\ \vdash X \rightsquigarrow C_X}{\vdash sprog \rightsquigarrow \overline{C_X}\ //@\text{main} = id} \quad (\textsc{ProgVerif})$$

5 FULL ABSTRACTION RESULT

This section describes a summary of the full abstraction proof that will be further detailed in the coming two sections. In this section, notions relating to the source and target language are typeset in **green** and pink respectively.

The first subsection formally defines both directions of full abstraction. Subsection 5.2 outlines how the full abstraction proof is conceptually split up.

5.1 Definition

To formally define the notion of full abstraction, we require the notion of contextual equivalence $\simeq$.

Terms x and x' are contextually equivalent, denoted $x \simeq_{\text{ctx}} x'$, if $\forall C : C[x] \Downarrow \Leftrightarrow C[x'] \Downarrow$ where $\Downarrow$ denotes termination of execution (starting from an *empty* heap and stack) and C is any program context with a hole that x and x' can be plugged into. Both x and x' will be either source or target components in our case. A context C consists of two parts in both our source and target languages: a *component context* $\mathfrak{C}_{\text{s}}$ or $\mathfrak{C}_{\text{t}}$, which is a sequence of components, and a *main function identifier* $//@\text{main} = id$, identifying the main function to execute when starting execution of the full program. A context is hence denoted $(\mathfrak{C}, id)$ and an entire program $\mathfrak{C}[x]//@\text{main} = id$. Correctly applying the notion of *plugging* from the contextual equivalence definition above to the source language also requires (next to the usual well-formedness constraints) that given the source component proof $\vdash \mathbf{s}$ and the context $(\mathfrak{C}_{\text{s}}, id)$, a program proof $\vdash \mathfrak{C}_{\text{s}}[\mathbf{s}]//@\text{main} = id$ is constructible.

Full abstraction is then defined as the reflection and preservation of contextual equivalence $\simeq_{\text{ctx}}$ [Abadi 1999]. Given source components $\mathbf{s}$ and $\mathbf{s}'$ and target components t and t', we hence have that fully abstract compilation is achieved iff. $\vdash \mathbf{s} \rightsquigarrow \mathrm{t} \ \wedge \vdash \mathbf{s}' \rightsquigarrow \mathrm{t}' \Rightarrow (\mathrm{t} \simeq_{\text{ctx}} \mathrm{t}' \Leftrightarrow \mathbf{s} \simeq_{\text{ctx}} \mathbf{s}')$. This statement depends on the concrete proofs $\vdash$ of $\mathbf{s}$ and $\mathbf{s}'$ that were chosen, but has to hold for *any* such choice. Notice how our formulation of full abstraction does not make a distinction between code that gets stuck and code that diverges. This is, however, not a problem; since our compiler does not alter control flow, it is easy to prove that it preserves divergence and stuck-ness, as expected.

Fully abstract compilation proofs are usually split up in a correctness proof direction $\Rightarrow$ that states (by contraposition) that non-equivalent source programs should yield non-equivalent target programs and a security proof direction $\Leftarrow$ that (by contraposition) states that any non-equivalence in the target programs should already have been there in the source programs, and hence attackers have no more power in the absence of contracts than they do in their presence. Both proof directions are summarized by the following equations:

$$\forall\, \mathbf{s}, \mathbf{s}', \mathbf{t}, \mathbf{t}'.\, \vdash \mathbf{s} \rightsquigarrow \mathbf{t} \;\wedge \vdash \mathbf{s}' \rightsquigarrow \mathbf{t}' \Rightarrow (\mathbf{t} \simeq_{\mathrm{ctx}} \mathbf{t}' \Rightarrow \mathbf{s} \simeq_{\mathrm{ctx}} \mathbf{s}') \qquad \textbf{(Correctness)}$$
$$\forall\, \mathbf{s}, \mathbf{s}', \mathbf{t}, \mathbf{t}'.\, \vdash \mathbf{s} \rightsquigarrow \mathbf{t} \;\wedge \vdash \mathbf{s}' \rightsquigarrow \mathbf{t}' \Rightarrow (\mathbf{t} \simeq_{\mathrm{ctx}} \mathbf{t}' \Leftarrow \mathbf{s} \simeq_{\mathrm{ctx}} \mathbf{s}') \qquad \textbf{(Security)}$$

5.2 Proof Decomposition

This subsection decomposes the correctness and security proof directions into the full correctness and security proof outline schemata of figures 1 and 2 (following the schema's of [Devriese et al. 2016]) that are used to make the full abstraction proof more modular and manageable. It also demonstrates the similarities between the techniques used and concrete lemma's proven in both proof directions. For compactness' sake, the invariant part //@main = *id* has been left out of the source and target program descriptions in both schemata, abbreviating e.g. $\vdash \mathfrak{C}_s[\mathbf{s}]$//@main = *id* to $\vdash \mathfrak{C}_s[\mathbf{s}]$ and $\mathfrak{C}_t[\mathbf{t}]$//@main = *id* to $\mathfrak{C}_t[\mathbf{t}]$.

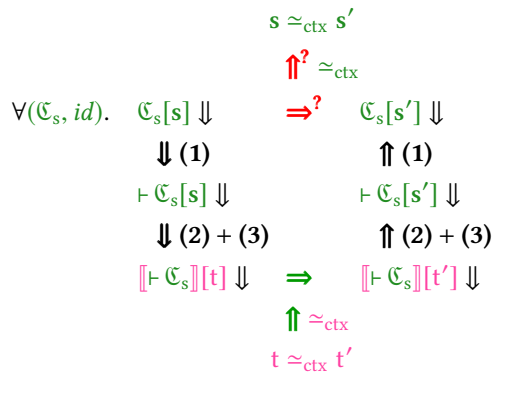
The notation $[\![\bullet]\!]$ is used as a functional shorthand for the compilation of proven source code and $\langle\!\langle \bullet \rangle\!\rangle$ is used as a functional shorthand for a new notion, called the *back-translation*, taking target code as input.

This *back-translation* is a standard tool in full abstraction proofs. It is a type of opposite-direction compilation that features in the security proof and is denoted $\rightsquigarrow_b$. It is necessary, because we need a source-level representation of the target program during the security proof, and we cannot just invert compilation. The back-translation relates a target statement, a target function or a target component $\mathbf{c}_t$ to a behaviourally-equivalent source-level proof $\vdash \mathbf{c}_s$, i.e., $\mathbf{c}_t \rightsquigarrow_b \vdash \mathbf{c}_s$. Behavioural equivalence is taken to mean equi-termination of a target program and its back-translation, as this is required in the security proof below. The back-translation $\rightsquigarrow_b$ is rule-based like the compilation $\rightsquigarrow$, also building back-translations from single statement to entire programs.

Contrary to what compilation does in the opposite direction, we cannot just back-translate entire programs freely given a target context $(\mathfrak{C}_t, id)$, as a verified source component $\vdash \mathbf{s}$ is always given and constrains the back-translation. Back-translation of entire programs is hence described by the rule $\mathfrak{C}_t[\mathbf{t}]$//@main = *id* $\rightsquigarrow_b \vdash \mathfrak{C}_s[\mathbf{s}]$//@main = *id* , given $\vdash \mathbf{s}$, with $\vdash \mathbf{s} \rightsquigarrow \mathbf{t}$, and a context $(\mathfrak{C}_t, id)$, that $\mathbf{t}$ can be plugged into. The back-translation derives a verified source context $\vdash \mathfrak{C}_s$ from the context $\mathfrak{C}_t$ and constructs the proof $\vdash \mathfrak{C}_s[\mathbf{s}]$//@main = *id* given the proof $\vdash \mathbf{s}$. Again, a program and its back-translation will be equi-terminating.

For reasons of presentational clarity in figure 2, the notation $\langle\!\langle \bullet \rangle\!\rangle$ only denotes the *code* output by the back-translation, not the proof. The full back-translation, ie. including the source proof, is denoted $\vdash\langle\!\langle \bullet \rangle\!\rangle$. We also denote $\mathfrak{C}_s$ from the previous paragraph as $\langle\!\langle \mathfrak{C}_t \rangle\!\rangle$, even though the back-translated context $\mathfrak{C}_s$ actually depends on $\vdash \mathbf{s}$ too.

In the schemata of figures 1 and 2 The proof in both proof schemata starts at the left side of $\Rightarrow^?$ and goes full circle before arriving at its right side. For both correctness and security, the proof steps $\Rightarrow$ are explained by either the definition of contextual equivalence $\simeq_{\mathrm{ctx}}$, or one of a set of



(1) $\forall \vdash \mathbf{c_s}. \vdash \mathbf{c_s} \Downarrow \Leftrightarrow \mathbf{c_s} \Downarrow$ **(Coherence)**

(2) $\forall \vdash \mathbf{s}, (\mathfrak{C}_s, id).$

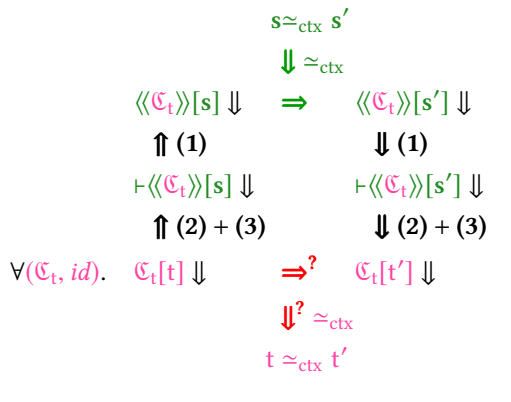
$\vdash \mathfrak{C}_s[\mathbf{s}]//@main = id$
$\rightsquigarrow \mathfrak{C}_t[t]//@main = id$
$\Rightarrow (\langle \cdot, \epsilon \rangle \mid \vdash \mathfrak{C}_s[\mathbf{s}]//@main = id\ )\ R$
$(\langle \cdot, \epsilon \rangle \mid \mathfrak{C}_t[t]//@main = id\ )$

(CompilIsCorr)

(3) $\forall\ \mathbf{s_s}, \mathbf{h_s} \vdash \mathbf{c_s}, \mathbf{s_t}, \mathbf{h_t}, \mathbf{c_t}.$
$(\langle \mathbf{s_s}, \mathbf{h_s} \rangle \mid \vdash \mathbf{c_s}) R (\langle \mathbf{s_t}, \mathbf{h_t} \rangle \mid \mathbf{c_t})$
$\Rightarrow \vdash \mathbf{c_s} \Downarrow \Leftrightarrow \mathbf{c_t} \Downarrow$

(CorrToEquiTermin)

Fig. 1. Correctness proof outline



(1) $\forall \vdash \mathbf{c_s}. \vdash \mathbf{c_s} \Downarrow \Leftrightarrow \mathbf{c_s} \Downarrow$ **(Coherence)**

(2) $\forall \vdash \mathbf{s}, (\mathfrak{C}_t, id).$

$\mathfrak{C}_t[t]//@main = id$
$\rightsquigarrow_b \vdash \mathfrak{C}_s[\mathbf{s}]//@main = id$
$\Rightarrow (\langle \cdot, \epsilon \rangle \mid \vdash \mathfrak{C}_s[\mathbf{s}]//@main = id\ )\ S$
$(\langle \cdot, \epsilon \rangle \mid \mathfrak{C}_t[t]//@main = id\ )$

(CompilIsSec)

(3) $\forall\ \mathbf{s_s}, \mathbf{h_s} \vdash \mathbf{c_s}, \mathbf{s_t}, \mathbf{h_t}, \mathbf{c_t}.$
$(\langle \mathbf{s_s}, \mathbf{h_s} \rangle \mid \vdash \mathbf{c_s})\ S\ (\langle \mathbf{s_t}, \mathbf{h_t} \rangle \mid \mathbf{c_t})$
$\Rightarrow \vdash \mathbf{c_s} \Downarrow \Leftrightarrow \mathbf{c_t} \Downarrow$

(SecToEquiTermin)

Fig. 2. Security proof outline

three proof implications. These three implications are similar between correctness and security and numbered **(1)**, **(2)** and **(3)** in both. We discuss both types of proof step annotations in order.

The arrows annotated with $\simeq_{ctx}$ and $\simeq_{ctx}$ denote an application of the definition of source- and target-level contextual equivalence, respectively. Contextual equivalence features in both proofs in the following way. Correctness proves that $\forall(\mathfrak{C}_s, id).\ \mathfrak{C}_s[\mathbf{s}] \Downarrow \Rightarrow^? \mathfrak{C}_s[\mathbf{s}'] \Downarrow$ by taking any $(\mathfrak{C}_s, id)$ and proving that if $\mathfrak{C}_s[\mathbf{s}]$ terminates, then so does $\mathfrak{C}_s[\mathbf{s}']$. This single implication is sufficient to prove that $\mathbf{s} \simeq_{ctx} \mathbf{s}'$, since the proof is not dependent on the identities of s and s' and hence reversible. On the other side of the schema, correctness uses the given $\simeq_{ctx}$ to prove the implication $\llbracket \vdash \mathfrak{C}_s \rrbracket[t] \Downarrow \Rightarrow \llbracket \vdash \mathfrak{C}_s \rrbracket[t'] \Downarrow$. Security uses contextual equivalence in a dual way with respect to the source and target language and proves a similar implication on the target level.

Before we can get into what the number-annotated $\Rightarrow$ mean, we first have to introduce two relations S and R. Both are binary relations, used in the proof of correctness and the proof of security, respectively. The relations R and S both relate source states to target states. These related states are the same $\langle \overline{s}, h \rangle \mid \overline{c}$-states, where the states from the lifted operational semantics are used for the source language, ie. $\overline{c}$ is a sequence of separation logic proofs of partially executed function bodies.

Execution of a full program *prog* starts by executing the function whose *id* is given in //@main = *id*. The initial state in this case is $\langle \epsilon, \bullet \rangle \mid prog$: it contains an empty stack ϵ ($\bullet$ is an empty stack frame) and an empty heap $\bullet$. We denote an arbitrary value of $\overline{c}$ by $\vdash \mathbf{c}_s$ for the source language and by $\mathbf{c}_t$ for the target. This includes the aforementioned case where $\overline{c} = prog$.

The relations R and S are proven to be *simulation relations*, meaning that these relations admit a bisimulation between the executions of the source and target states they relate. These bisimulation arguments are used to prove equi-termination in the respective directions of the full abstraction proof below. Both the definitions of R and S and their simulation relation proofs are extensive and hence only detailed in the technical report.

We now have the context necessary to discuss the sets of proof implications **(1)**, **(2)** and **(3)**.

The Coherence rule **(1)** is used in both correctness and security to add proofs to source programs and conversely strip away proofs while preserving termination. This conversion is necessary since $\simeq_{ctx}$ is defined on proof-less code, whereas the compiler requires a source proof as input and the back-translation creates a source proof as output.

Rules **(2)** and **(3)** are used in combination to prove equi-termination between a source program and its compilation or between a target program and its back-translation, respectively. Rule **(2)** proves that any source program and its compilation are related by R for correctness, and that any target program and its back-translation are related by S for security. Rule **(3)** finishes the combined equi-termination argument by stating that if source and target code is related by S and R, respectively, then the source and target code equi-terminates (from the given stack and heap). The proof of rule **(3)** follows immediately from the fact that S and R are proven to be simulation relations. Together, rules **(2)** and **(3)** for security indeed prove that the back-translation preserves (non-)termination between the target and the source languages, as it was designed to do.

By combining the rules **(1)**, **(2)** and **(3)** in both proof schemata, we arrive at two equi-termination statements, namely CorrectnessEquiTermination and SecurityEquiTermination below, that are sufficient to prove respectively Correctness and Security, as justified by figures 1 and 2. Once the individual statements **(1)**, **(2)** and **(3)** are proven, these resulting statements make completing both directions of the full abstraction proof trivial, as they link contextual equivalence in the target language with contextual equivalence in the source.

$$\forall \vdash \mathbf{s}, (\mathfrak{C}_s, id).\vdash \mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = id \rightsquigarrow \mathfrak{C}_t[\mathbf{t}]//@\mathrm{main} = id$$
$$\Rightarrow \mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = id \Downarrow \Leftrightarrow \mathfrak{C}_t[\mathbf{t}]//@\mathrm{main} = id \Downarrow$$
(CorrectnessEquiTermination)

$$\forall \vdash \mathbf{s}, (\mathfrak{C}_t, id).\mathfrak{C}_t[\mathbf{t}]//@\mathrm{main} = id \rightsquigarrow_b \vdash \mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = id$$
$$\Rightarrow \mathfrak{C}_s[\mathbf{s}]//@\mathrm{main} = id \Downarrow \Leftrightarrow \mathfrak{C}_t[\mathbf{t}]//@\mathrm{main} = id \Downarrow$$
(SecurityEquiTermination)

6 COMPILER CORRECTNESS

6.1 Definitions

Before we can introduce this section's central inference rule we aim to prove, we require the notion of *equi-termination* and the notion of *contextual equivalence* $\simeq_{ctx}$:

Definition (Equi-termination)**.**
Terms x and x' equi-terminate, denoted $x \Downarrow \Leftrightarrow x' \Downarrow$, if either x and x' reduce to an irreducible return

statement using the operational semantics in a finite number of steps starting from empty stack ϵ and empty heap $\cdot$, or neither x nor x' reduce to an irreducible return *in a finite number of steps.*

Definition (Contextual equivalence)**.**
Terms x and x' are contextually equivalent, denoted $x \simeq_{\text{ctx}} x'$, if $\forall C : C[x] \Downarrow \Leftrightarrow C[x'] \Downarrow$ where C is any program context with a hole that x and x' can be plugged into.

It is important to see that a context C in both the source and target languages consists of two parts throughout the full abstraction proof: a *component context* $\mathfrak{C}_\text{s}$ or $\mathfrak{C}_\text{t}$, which is just a sequence of components, and a *main function identifier* //@main = *id* , identifying the main function to execute when starting the run of the full program. A context is denoted ($\mathfrak{C}$, *id*) .

The notion of *plugging* from the contextual equivalence definition applied to the source language, also requires (next to the usual well-typedness/formedness and scoping constraints) that for source component proof $\vdash s$ and context ($\mathfrak{C}_\text{s}$, *id*), a proof $\vdash \mathfrak{C}_\text{s}$//@main = *id* exists.

Proving compiler correctness entails proving that given the compiler $\rightsquigarrow$ from the previous section and source components with separation logic derivations s and s', the following holds:

$$\frac{t \simeq_{\text{ctx}} t' \qquad \vdash s \rightsquigarrow t \quad \vdash s' \rightsquigarrow t'}{s \simeq_{\text{ctx}} s'} \qquad \text{(CompilerCorrectness)}$$

To prove the CompilerCorrectness rule, it suffices to show that for any source context ($\mathfrak{C}_\text{s}$, *id*) and any proof $\vdash \mathfrak{C}_\text{s}[s]$//@main = *id* constructed using the concrete proof $\vdash s$ of s used in CompilerCorrectness, the following inference rule holds:

$$\frac{\vdash s \rightsquigarrow t \qquad \vdash \mathfrak{C}_\text{s}[s]\text{//@main} = id \rightsquigarrow \mathfrak{C}_\text{t}[t]\text{//@main} = id}{\mathfrak{C}_\text{s}[s]\text{//@main} = id \Downarrow \Leftrightarrow \mathfrak{C}_\text{t}[t]\text{//@main} = id \Downarrow} \qquad \text{(CompilerEquiTermination)}$$

This rule suffices because it allows using the given target-level equi-termination result by linking equi-termination of source and target programs.

6.2 Simulation

We will prove the above statement using a simulation argument. This argument employs a simulation relation to relate source to target *states*, denoted as follows: $(\langle s_\text{s}, h_\text{s}\rangle \mid s)\ R\ (\langle s_\text{t}, h_\text{t}\rangle \mid t)$, s is the current separation logic derivation (which also implicitly contains the source code) in the source language and t is the corresponding target level code. The variable s is a sequence of the currently executing function bodies c_i and their separation logic proofs. It is hence a sequence of proof trees. The variables s_x and h_x denote the stack and heap, where x = s for the source language and x = t for the target. The full notation for the relation R is in fact $sprog \vdash (\langle s_\text{s}, h_\text{s}\rangle \mid s)\ R\ (\langle s_\text{t}, h_\text{t}\rangle \mid t)$ with *sprog* the original source program that is being executed. This program is necessary because the FApp rule uses the component environments Σ_i. These can be derived from *sprog* in the source and from its compilation in the target. Usually, the concrete *sprog* is either unimportant or clear from the context and hence omitted. Given *sprog*, R can thus be seen as a per-program relation, or it can be seen as one element of a larger relational family $R_{sprogram}$. We often write $s_1\ R\ s_2$ as a

shorthand notation, where s_1 denotes the source state consisting of the current stack, heap and proven code and s_2, analogously, denotes the target state.

For reasons that become clear below, we define R using an alternative notion of compilation:

Definition.
Given the relation $\rightsquigarrow$ above, we define $\rightsquigarrow_{NoStubs}$. $\rightsquigarrow_{NoStubs}$ is identical to $\rightsquigarrow$, except for the generation of stubs: all target-level functions keep their source-level names during compilation (so no f_{comp} name-change is needed) and no stubs are generated (the rules Incall *and* Outcall *from the compilation become obsolete).*

The *definition of the relation R* consists of a listing of state interrelation invariants and is detailed below. Two cases can be distinguished in the definition of R, one for full programs that have not yet taken a step, and one for partially-executed programs.

The relation R (and the definition of linking below) contains an environment δ_i, defined for each different executing function body i. The function δ_i should be seen as a kind of inverse to γ, that now maps logical variables to program values. Applying δ to any structure is formalized as replacing any logical variable id_{log} present in this structure by $\delta(id_{\text{log}})$. Notice that the range of δ is the set $\text{VAL}_\text{s} \cup [\text{VAL}_\text{s}(, \text{VAL}_\text{s})^*]$ and not just VAL_s. The type $[\text{VAL}_\text{s}(, \text{VAL}_\text{s})^*]$ can be seen as an alternative notation for $\text{cons}(\text{VAL}_\text{s}, \text{cons}(\ldots, ()))$ that fits more closely to fixed-length chunk notation. The reason for this is that the logical types contain logical lists, whereas the physical types do not, ie. we need to emulate them ourselves. We add the set of values $[\text{VAL}_\text{s}(, \text{VAL}_\text{s})^*]$ as a sort of ad-hoc source-level list, so we can represent the values of eg. a source-mapped list variable l_δ. This environment is defined using the following predicate (using γ, P, s_s and h_s, ie. every aspect of both the symbolic and source program state of the execution - together they sufficiently determine the target program state, which is hence not required to narrow down the set of satisfying functions δ, but will be used in R_{comp} later on):

$$
\begin{aligned}
\text{InverseMap}(\delta, P, \gamma, s_\text{s}, h_\text{s}) \Leftrightarrow \\
&\delta : \mathit{ID}_{\text{log}} \rightarrow \text{VAL}_\text{s} \cup [\text{VAL}_\text{s}(, \text{VAL}_\text{s})^*] \\
&\forall x \in \text{dom}(\gamma).\, s_\text{s}(x) = [\gamma(x)]_\delta \\
&h_\text{s} \vdash^{\text{src}}_{\delta} P
\end{aligned}
$$

where $h_\text{s} \vdash^{\text{src}}_{\delta} P$ is defined below, and intuitively forces the separation logic state and physical heap to correspond. This last correspondence is an important part of all 4 types of relations (one of which is R_{comp}) we will be defining for the security proof.

Another important aspect is the appearance of ID_{log}; the set of all previously and currently used symbolic variable names for the current frame. This set is used for scoping purposes and during frame linking below and is implicitly constructed during execution of the lifted operational semantics (we never explicitly denote this). It necessarily contains any variables in $\text{V}(P) \cup \text{range}(\gamma)$, all arguments $\overline{id_{\text{arg}}}$ provided to the current frame, but also any variables in the set $\text{V}(\mathit{POST}) \cap \mathit{ID}'_{\text{log}}$, with POST the current frame's postcondition and $\mathit{ID}'_{\text{log}}$ the previous frame's ID_{log} set, because these are the logical variables that were passed in from the previous frame (actually, the full set is $\text{V}(\mathit{PRE}) \cap \mathit{ID}'_{\text{log}}$, but we do not retain the function's precondition after the function call). This

condition on the contents of ID_{log} will implicitly be assumed to hold for any frame for each relation defined further on in this report.

Before being able to define the correctness relation, we need to define the auxiliary concept of *separation logic linking* between the function bodies s_i/t_i of the executing stack frames i. Without any constraints between different consecutive stack frames, the code executing in the next frame could be chosen freely and wouldn't have to match the function invoked by the previous frame. This can break the relation R (and other relations in the security proof later on) when the next stack frame returns to the non-matching, previous one. An example follows in the next paragraph.

Some subtleties have to be watched out for when considering the correspondence of the separation logic state with the program state. For example, imagine the only separation logic linking condition we apply is that the separation logic postcondition of the callee frame and of the caller's function need to correspond. Then a function with argument int a that has $result == a$ in its postcondition could be run with $a == 3$ as an argument value, whereas the previous stack frame could have $a : a$ stored in the environment and $a == 2$ in the symbolic heap. This would, because of the FApp rule, result in a contract containing $a == 2 \wedge a == 3$, which is a contradiction and clearly undesired behavior. An extra condition on the arguments prohibits this behavior, by requiring that $a_{\delta_i} = 3 == a_{\delta_{i-1}} = 2$, which is clearly not the case here.

The question then is: what constraints are sufficient in general? We do not have to require the callee frame to be an exact execution of what the caller suspects. It suffices that any information returned from the callee frame matches what the caller would expect (ie. does not break the relation R in the caller's frame after the return). The more general condition will then (intuitively speaking) require that any information that is passed from the callee to the caller at the end of function execution matches any preconceptions the caller had about this information, no matter what execution we substitute the callee frame with. The expectations of the caller frame are captured by the hole's postcondition *POST*, which had to match the callee's postcondition. Equality on a logical level is not enough, however, as was shown in the example above. Any variables appearing in *POST* that already existed in the caller frame (ie. were part of its ID_{log}, the set of previously used symbolic variables) before the call, should have the same program value in the callee frame; this forces a correspondence between the information caller and callee communicate at the end of function execution. The same condition also has to hold over any function arguments appearing in the postcondition and the expression that was used for them by the caller. Formally then, linking states that δ applied to any predeclared ($\in ID_{\text{log}}$) logical variable appearing in *POST* should still have the same value in the callee frame (through its δ). The rationale behind the statement of this condition is that logical variables don't change value (this is the reason for our earlier assumption that logical variable names are never reused); δ hence has to map back the logical variables that existed from the beginning of execution to the originally supplied values at any point during execution. Separation logic linking is formally defined as the following predicate (with s_{i-1} and s_i two consecutive executing source function bodies and δ_{i-1}, δ_i their inverse environment mappings):

$$\text{Link}(s_{i-1}, s_i, \delta_{i-1}, \delta_i) \Leftrightarrow$$

Form constraints

$$s_{i-1} = \vdash \{P'\}_{\gamma_{i-1}} \overline{id_{\text{prog}}} = \bullet^{\overline{id_{\text{arg}}}=\overline{sexp}}_{POST}; sstm\ \{Q'\}_{\gamma'_{i-1}}$$
$$s_i = \vdash \{P\}_{\gamma_i}\ s_{\text{c}}\ \{Q\}_{\gamma'_i}$$

$$\underline{\text{Actual frame linking constraints}}$$
$$Q == POST$$
$$[\overline{sexp}_{\gamma_{i-1}}]_{\delta_{i-1}} == \delta_i(\overline{id_{\text{arg}}})$$
$$\overline{id_{\text{link}}} = (ID_{\text{log},\,i-1} \setminus \overline{id_{\text{arg}}}) \cap \text{V}(POST)$$
$$\delta_{i-1}(\overline{id_{\text{link}}}) == \delta_i(\overline{id_{\text{link}}})$$

Actually, not all function arguments $\overline{id_{\text{arg}}}$ strictly have to correspond in the above definition; only the variables $\overline{id^{\text{arg}}_{\text{link}}} = \overline{id_{\text{arg}}} \cap \text{V}(POST)$ do. There is, however, no need to use this more complicated condition. Also note that the appearance of ID_{log} could have been avoided, had we used explicit quantifiers for new variables appearing in contracts, as is the case in VeriFast, because we would then immediately know what variables to enforce consistency on.

We will also need to have a formal way of partitioning source and target heaps, defined as follows:

Definition (partition of the heap h)**.**
The set of heaps $\{h_1, \ldots, h_k\}$ is a partition of the heap h if the following conditions are met:

$$\bigcap\nolimits_{\emptyset}\{h_1, \ldots, h_k\} \triangleq \forall i, j.\, i \neq j \Rightarrow$$
$$l \in \text{dom}(h_i) \Rightarrow l \notin \text{dom}(h_j) \vee$$
$$l \in \text{dom}(h_j) \wedge \text{dom}(h_i(l)) \cap \text{dom}(h_j(l)) = \emptyset$$
$$\bigcup\{h_1, \ldots, h_k\} = h$$

The previous definition is written as $h_1 \uplus \ldots \uplus h_k = h$ or $\biguplus \overline{h_i} = h$.

Lastly, we define a function $\text{Loc}(\overline{s})$ that obtains a list of all locations l appearing in the range of the set of stack frames $\overline{s}$. This function can either take source or target frames as input. We use it to define a bijection b between locations l_{s} in the source language and a subset of the locations l_{t} used in the target language. The subset considered is the subset of *real* locations, ie. non-*reified* location that have an equivalent in the source language as well and do not purely arise from the reification of logical concepts (ie. the target locations corresponding to the reification of a range resource). This bijection is useful when mapping (source or heap) values or locations between the source and target language. Its domain defined as follows:

$$\text{Loc}(b, \overline{s_{\text{s}}}, h_{\text{s}}, \overline{s_{\text{t}}}, h_{\text{t}}) \Leftrightarrow$$
$$loc_{\text{s}} = \text{Loc}(\overline{s_{\text{s}}}) \cup \text{dom}(h_{\text{s}})$$
$$loc_{\text{t}} = \text{Loc}(\overline{s_{\text{t}}}) \cup \text{dom}(h_{\text{t}})$$
$$loc_{\text{t}} = loc_{\text{real}} \uplus loc_{\text{reify}}$$
$$b : loc_{\text{s}} \leftrightarrow loc_{\text{real}}$$

The function b's contents are further constrained by each individual frame. It thereby enforces consistency constraints between different frames. In the security proof later on, it will eg. enforce the fact that the mapping between locations in source and target language should be preserved by locations that are passed through an in- or outcall frame.

$$
\begin{aligned}
&\mathit{sprog}, \mathit{tprog} \vdash (\langle s_\mathrm{s}, h_\mathrm{s}\rangle \mid s)\ R\ (\langle s_\mathrm{t}, h_\mathrm{t}\rangle \mid t) \Leftrightarrow \\
&\qquad \vdash \mathit{sprog} \rightsquigarrow_{\mathrm{NoStubs}} \mathit{tprog} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \underline{\text{Initial case}} \\
&\qquad s = \vdash \mathit{sprog}, t = \mathit{tprog} \\
&\qquad h_\mathrm{s} = h_\mathrm{t} = \bullet, s_\mathrm{s} = s_\mathrm{t} = \epsilon \\
&\qquad\qquad\qquad\qquad\qquad\qquad \underline{\text{Executing case}} \\
&\qquad s = \overline{s_i},\ t = \overline{t_i},\ s_\mathrm{s} = \overline{s_{\mathrm{s}i}},\ s_\mathrm{t} = \overline{s_{\mathrm{t}i}} \\
&\qquad \textstyle\bigcup\!\!\!\!+ \overline{h_{\mathrm{s}i}} = h_\mathrm{s},\ \bigcup\!\!\!\!+ \overline{h_{\mathrm{t}i}} = h_\mathrm{t}, \\
&\qquad \mathrm{Loc}(b, \overline{s_{\mathrm{s}i}}, h_\mathrm{s}, \overline{s_{\mathrm{t}i}}, h_\mathrm{t}) \\
&\qquad \forall i. \\
&\qquad\quad \underline{\text{Component relation}} \\
&\qquad\quad \mathrm{InverseMap}(\delta_i, P_i, \gamma_i, s_{\mathrm{s}i}, h_{\mathrm{s}i}) \text{ given } s_i = \vdash \{P_i\}_{\gamma_i}\ s_{\mathrm{c}i}\ \{Q_i\}_{\gamma_i'} \\
&\qquad\quad \mathit{sprog}, \mathit{tprog} \vdash^{\delta_i}_{b} (\langle s_{\mathrm{s}i}, h_{\mathrm{s}i}\rangle \mid s_i)\ R_\mathrm{comp}\ (\langle s_{\mathrm{t}i}, h_{\mathrm{t}i}\rangle \mid t_i) \\
&\qquad\quad \underline{\text{Separation logic linking}} \\
&\qquad\quad i > 0 \Rightarrow \mathrm{Link}(s_{i-1}, s_i, \delta_{i-1}, \delta_i)
\end{aligned}
$$

In the second case, R is composed of frame-wise R_comp relations. This second component-level relation is defined below.

From now on, we call the state R_comp is dependent on (the source and target heap and stack and the source and target function body code) a *frame*. The relation R then holds over a set of frames in the non-trivial case.

6.3 Assertion semantics

Another auxiliary definition is needed:

Definition (δ applied to expression *exp*)**.**
Applying the mapping δ to a given separation logic assertion exp is defined as follows:

$$\frac{FV(\mathit{exp}) = \{x_1, \ldots, x_k\}}{\mathit{exp}_\delta \triangleq \mathit{exp}[x_1, \ldots, x_k \mapsto \delta(x_1), \ldots, \delta(x_k)]} \quad (\textsc{DeltaApp})$$

We use the following rules to enforce correspondence between the symbolic state P and the source heap h_s. The judgment $h_\mathrm{s} \vdash^{\mathrm{src}}_{\delta} \mathit{assert}$ holds if the program-level assertion resulting from applying δ to every logical variable in the separation logic assertion *assert* is satisfied in the source-level program state h_s. The judgment hence tests for Logical state to Source heap Correspondence, abbreviated to LogSrcCorr.

The below rule's precondition is not rigorously defined, but the meaning of the precondition is that δ applied to the pure expression holds true in the current stack frame. Here, 'holding true' means that all logical expression constructs get a parallel interpretation for source-level values as they have for logical values, and that the resulting boolean expression results in the value true. The interpretation of the first block of *exp*'s maps nicely to existing source expressions *sexp*, whereas

the logical lists operate on the ad-hoc source-level list values $[\mathrm{VAL_s}(, \mathrm{VAL_s})^*]$ and quantifications are straightforward to adjust to a program setting. We have to reinterpret cons, $\forall$, ... for program values, whereas these concepts were originally used for logical values. Their meanings stay the same, however, and these concepts are now interpreted on the physical heap, for physical values, rather than on the symbolic heap and for symbolic values. This permits us not to change any keywords when applying δ.

$$\frac{\vdash exp_\delta}{h_\mathrm{s} \vdash^{\mathrm{src}}_\delta exp} \quad \text{(LogSrcCorrExp)}$$

$$\frac{h_{\mathrm{s}1} \uplus h_{s_2} = h_\mathrm{s} \quad h_{\mathrm{s}1} \vdash^{\mathrm{src}}_\delta assert_1 \quad h_{\mathrm{s}2} \vdash^{\mathrm{src}}_\delta assert_2}{h_\mathrm{s} \vdash^{\mathrm{src}}_\delta assert_1 * assert_2} \quad \text{(LogSrcCorrSConj)}$$

$$\frac{exp_\delta == \mathrm{false}}{h_\mathrm{s} \vdash^{\mathrm{src}}_\delta exp\ ?\ assert} \quad \text{(LogSrcCorrCondFalse)}$$

$$\frac{exp_\delta == \mathrm{true} \quad h_\mathrm{s} \vdash^{\mathrm{src}}_\delta assert}{h_\mathrm{s} \vdash^{\mathrm{src}}_\delta exp\ ?\ assert} \quad \text{(LogSrcCorrCondTrue)}$$

$$\frac{exp_\delta == (l, i) \quad exp'_\delta == [v_0, \ldots, v_k] \quad h_\mathrm{s}(l, [i, , \ldots, i+k]) = [v_0, \ldots, v_k]}{h_\mathrm{s} \vdash^{\mathrm{src}}_\delta n : exp \mapsto exp'} \quad \text{(LogSrcCorrArrName)}$$

$$\frac{h_{\mathrm{s}, exp_\delta} \uplus \ldots \uplus h_{\mathrm{s}, exp'_\delta - 1} = h_\mathrm{s} \quad \forall exp_\delta \le i < exp'_\delta.\, h_{\mathrm{s},i} \vdash^{\mathrm{src}}_\delta assert_i}{h_\mathrm{s} \vdash^{\mathrm{src}}_\delta n : [assert \mid exp \le i < exp']} \quad \text{(LogSrcCorrRangeName)}$$

Another check we need is the check that chunk names appearing in (outer) separation logic assertions get reified to the correct locations (according to b) and ranges in the target-stack s_t and, in the case of range chunks (since those do not have a counterpart in the source heap and are hence not checked in the judgment above) to some sensible location in the heap h_t for reified heap chunks, because these are not present in the source heap. We denote this check as the judgment $s_\mathrm{t}, h_\mathrm{t} \vdash^{\mathrm{tgt}}_{\delta,b} assert$, or as $s_\mathrm{t}, h_\mathrm{t} \vdash^{\mathrm{tgt}}_{\delta,b} assert, \overline{v_\mathrm{t}}$ in the case of inner assertions. We use the color red for $\overline{v_\mathrm{t}}$ if the judgment is applicable to both outer and inner assertions. In this case, the red part only applies in the case of inner assertions. The reason for this extra target-level output value, is that it is necessary to correctly restrict the target heap for range assertions. $\overline{v_\mathrm{t}}$ contains all addresses of the enclosed assertions.

$$\frac{}{s_\mathrm{t}, h_\mathrm{t} \vdash^{\mathrm{tgt}}_{\delta,b} exp, ()} \quad \text{(LogTgtCorrExp)}$$

$$\frac{h_{t1} \uplus h_{t2} = h_t \quad s_t, h_{t1} \vdash^{tgt}_{\delta,b} assert_1, \overline{v_t} \quad s_t, h_{t2} \vdash^{tgt}_{\delta,b} assert_2, \overline{v'_t}}{s_t, h_t \vdash^{tgt}_{\delta,b} assert_1 * assert_2, \overline{v_t\ v'_t}} \text{(LogTgtCorrSConj)}$$

$$\frac{exp_\delta == \text{false} \quad assert \rightsquigarrow_{\text{ChunkType}} \overline{\tau}}{s_t, h_t \vdash^{tgt}_{\delta,b} exp\ ?\ assert, \text{NULL}^{tgt}(\overline{\tau})} \text{(LogTgtCorrCondFalse)}$$

$$\frac{exp_\delta == \text{true} \quad s_t, h_t \vdash^{tgt}_{\delta,b} assert, \overline{v_t}}{s_t, h_t \vdash^{tgt}_{\delta,b} exp\ ?\ assert, \overline{v_t}} \text{(LogTgtCorrCondTrue)}$$

$$\frac{exp_\delta == (l, i) \qquad exp'_\delta == [v_0, \ldots, v_k]}{s_t, h_t \vdash^{tgt}_{\delta,b} exp \mapsto exp', b(l)^{[i, i+k]}} \text{(LogTgtCorrArr)}$$

$$\frac{exp_\delta == (l, i) \qquad exp'_\delta == [v_0, \ldots, v_k] \quad s_t(n) = b(l)^{[i, i+k]}}{s_t, h_t \vdash^{tgt}_{\delta,b} n : exp \mapsto exp'} \text{(LogTgtCorrArrName)}$$

$$\frac{h_{\text{range}} \uplus h_{t, exp_\delta} \uplus \ldots \uplus h_{t, exp'_\delta - 1} = h_t \qquad l \notin \text{range}(b) \quad \forall exp_\delta \leq i < exp'_\delta.\ s_t, h_{t,i} \vdash^{tgt}_{\delta,b} assert_i, \overline{v_{t,i}} \quad h_{\text{range}}(l, [k + exp_\delta, k + exp'_\delta - 1]) = [(\overline{v_{t, exp_\delta}}), \ldots, (\overline{v_{t, exp'_\delta}})]}{s_t, h_t \vdash^{tgt}_{\delta,b} [assert \mid exp \leq i < exp'], l^{[k+exp_\delta, k+exp'_\delta - 1]}} \text{(LogTgtCorrRange)}$$

$$\frac{h_{\text{range}} \uplus h_{t, exp_\delta} \uplus \ldots \uplus h_{t, exp'_\delta - 1} = h_t \qquad l \notin \text{range}(b) \quad \forall exp_\delta \leq i < exp'_\delta.\ s_t, h_{t,i} \vdash^{tgt}_{\delta,b} assert_i, \overline{v_{t,i}} \quad s_t(n) = l^{[k+exp_\delta, k+exp'_\delta - 1]} \quad h_{\text{range}}(l, [k + exp_\delta, k + exp'_\delta - 1]) = [(\overline{v_{t, exp_\delta}}), \ldots, (\overline{v_{t, exp'_\delta}})]}{s_t, h_t \vdash^{tgt}_{\delta,b} n : [assert \mid exp \leq i < exp']} \text{(LogTgtCorrRangeName)}$$

Another auxiliary function, valmap_b, maps source values to target values. It is defined as follows (with b some bijection between locations):

$$\begin{aligned}
&\text{valmap}_b : \text{VAL}_s \to \text{VAL}_t \\
&\text{valmap}_b(k) = k \\
&\text{valmap}_b(\text{null}_0) = \text{null}_0 \\
&\text{valmap}_b((l, i)) = b(l)^i_0 \\
&\text{valmap}_b((v_1, \ldots, v_k)) = (\text{valmap}_b(v_1), \ldots, \text{valmap}_b(v_k))
\end{aligned}$$

An important property of valmap_b is its compositionality:

Theorem 1 (Compositionality)**.**
The function valmap_b *is compositional. We define this as follows: if* $s_\text{t}(\overline{id_\text{t}}) = \text{valmap}_b(s_\text{s}(\overline{id_\text{s}}))$ *for some sets of source and target variables* $\overline{id_\text{s}}$ *and* $\overline{id_\text{t}}$ *, then for any parametrized source (and hence also target) expression* $sexp[\overline{id_\text{s}}]$*, it holds that* $[\![sexp[\overline{id_\text{t}}]]\!]_{s_\text{t}} = \text{valmap}_b([\![sexp[\overline{id_\text{s}}]]\!]_{s_\text{s}})$*. In other words, applying the same expression to variables related by valmap, keeps the results related.*

Proof.

Trivial case-based analysis on the form of source expressions *sexp*. □

6.4 Component simulation relation

$$
\begin{aligned}
&sprog, tprog \vdash^{\delta}_{b} (\langle s_\text{s}, h_\text{s}\rangle \mid s)\ R_\text{comp}\ (\langle s_\text{t}, h_\text{t}\rangle \mid t) \Leftrightarrow \\
&\quad s \rightsquigarrow_\text{NoStubs} t \\
&\quad s = \Sigma^{ax} \vdash \{P\}_\gamma\ sstm; \text{return } \overline{sexp}\ \{Q\}_{\gamma'} \wedge \Sigma^{ax} \in \Sigma^{ax}_{sprog} \\
&\quad CN(P) = \overline{m} \\
&\qquad\qquad \underline{\text{Relating target to source}} \\
&\quad \underline{s_\text{t}} \approx \underline{s_\text{s}} \\
&\quad \text{dom}(s_\text{t}) = \text{dom}(s_\text{s}) \uplus \overline{m} \uplus \overline{n_\text{null}} \uplus \overline{id_\text{aux}} \\
&\quad \forall id \in \text{dom}(s_\text{s}).\ s_\text{t}(id) = \text{valmap}_b(s_\text{s}(id)) \\
&\quad \underline{h_\text{t}} \approx \underline{h_\text{s}} \\
&\quad h_\text{t} = h_{\text{t,real}} \uplus h_{\text{t,reify}} \\
&\quad \text{dom}(h_{\text{t,real}}) \subseteq \text{range}(b) \qquad \text{dom}(h_{\text{t,reify}}) \cap \text{range}(b) == \emptyset \\
&\quad \forall l.\ h_\text{s}(l) = [v_0, \ldots, v_k] \\
&\qquad \Leftrightarrow h_{\text{t,real}}(b(l)) = [\text{valmap}_b(v_0), \ldots, \text{valmap}_b(v_k)] \\
&\qquad\qquad \underline{\text{Relating logical state to reification}} \\
&\quad s_\text{t}, h_{\text{t,reify}} \vdash^{\text{tgt}}_{\delta, b} P
\end{aligned}
$$

In the above, we do not require any condition of the form $\text{dom}(s_\text{s}) == \text{dom}(\gamma)$, because if the first is a subset of the second, nothing happens and execution can go through as usual, because any values $\text{dom}(\gamma) \setminus \text{dom}(s_\text{s})$ do not correspond to the stack anymore, whereas if the second is a subset of the first, we know all variables in $\text{dom}(s_\text{s}) \setminus \text{dom}(\gamma)$ will never be used again anyway, otherwise a separation logic proof could not have been constructed. If both set differences are non-empty, a combination of both arguments applies. In every practical execution using our lifted operational semantics, $\text{dom}(s_\text{s}) == \text{dom}(\gamma)$ will hold, but we do not need this restriction for the proof.

6.5 Simulation proofs

Now we show that this relation R is a valid simulation relation. This comes down to proving that it has the following two properties (based on the definition in Chlipala's FRAP handbook [Chlipala 2017]):

Theorem 2 (R is a simulation relation)**.**
Given a relation R. If the following 2 properties hold, then R is a simulation relation:

(1) *If* $(\langle s_s, h_s\rangle \mid \{P\}\ \text{return}\ \{Q\})\ R\ (\langle s_t, h_t\rangle \mid t)$*, then* $t = \text{return}$*. This rule is used to guarantee equi-termination in the proof, when we know that the source program terminates.*

(2) *Whenever* $s_1\ R\ s_2$ *and* $s_1 \hookrightarrow s_1'$ *(using the shorthand notation) , there exists an* s_2' *such that* $s_2 \hookrightarrow^+ s_2'$ *and* $s_1'\ R\ s_2'$*. Note that this condition requires* $\hookrightarrow$ *to perform multiple steps at once, denoted* $\hookrightarrow^*$ *and* $\hookrightarrow^+$ *if at least one step is taken. Also note that at the source level, not the code itself is executed by* $\hookrightarrow$*, but a separation-logic derivation of the code. We will thus need to lift* $\hookrightarrow$ *from the semantics of regular source code to semantics for the execution of separation logic proven code, hence manipulating separation logic triples instead of regular source expressions. This* lifted *version of the source-level operational semantics is defined in the* preservation *theorem below.*

If we were to use $\rightsquigarrow$ instead of $\rightsquigarrow_{\text{NoStubs}}$ in the definition of R when proving the second property for the simulation relation, namely that there are n steps in the target language for each step in the source, we run into problems with outcall stubs, as the outcall stubs also perform checks *after* the wrapped function call, and there is no proper $1 : n$ mapping of steps anymore. This is exactly the reason for defining the alternative form of compilation $\rightsquigarrow_{\text{NoStubs}}$ above. The thus defined simulation relation R can be linked back to the original compilation using the following theorem (assuming that we have already proven that the described R is a simulation relation, which we will do below):

Theorem 3.
If $s \rightsquigarrow t$ *and* $(\langle s_s, h_s\rangle \mid s)\ R\ (\langle s_t, h_t\rangle \mid t')$*, then* t *equi-terminates with* t' *for the given heap and stack.*

Proof.

A sufficient condition for the above theorem to hold is that none of the guards encountered in stubs during the execution of t ever return false, as stubs take only finitely many steps to execute and do nothing else than performing guards and passing on return values or function arguments. This fact has to be proven for both incall and outcall stubs, which results in a proof in two parts.

1. Incall stubs
Two types of precondition checks are performed before the function call; checks cs_{pre} of the chunks lengths (and non-null checks) and checks of the pure heap cp_{pre}.

- cs_{pre}: holds almost trivially, because of the relation between s_t and P_δ, which dictates that the linear capabilities that are provided to the function f have to have the same length as the separation logic chunks they correspond to (and these capabilities cannot be null either).
- cp_{pre}: From the fact that $\vdash \phi_\delta$ holds in the definition of the simulation relation and the fact that ϕ is the pathcondition of the called function in this case, we have that $\vdash [PRE_p]_\delta$. The guards in cp_{pre} on the other hand check if $\vdash PRE_p$. In other words, if δ maps each symbolic variable x to the value stored in the incall stub, then the incall stub will never fail. This is indeed the case, as the definition of δ requires that $\forall x \in ID_{\text{prog}}.\ s_s(x) = [\gamma(x)]_\delta$, where γ is the identity function (over the argument names) at the start of function execution and x is equal to $s_t(x)$ (which is equal to $s_s(x)$ up to the representation of pointers).

2. Outcall stubs
Analogous.

□

Corollary.
It suffices to use the previously defined relation R to prove equi-termination for the original compilation

as well. The reason is that if we know that $(\langle s_s, h_s\rangle \mid s)\ R\ (\langle s_t, h_t\rangle \mid t')$ *and if we prove that the simulation relation implies equi-termination (one of the proven inference rules below), the previous theorem implies that* s *and* t *also equi-terminate.*

In order to be able to prove that R is indeed a valid simulation relation, we need the following *preservation theorem* for separation logic derivations with respect to the operational semantics.

Theorem 4 (Preservation)**.**
Given a sequence $\overline{s_i}$ *of function bodies being executed, where* $\forall\ i.\, s_i = \{P\}_{\gamma_1}\ s_i^p\ \{Q\}_{\gamma_2}$ *(or special case:* $\overline{s_i} = \vdash$ *sprog) and the fact that* $\langle \overline{s}, h\rangle \mid \overline{s_i^p} \hookrightarrow \langle \overline{s'}, h'\rangle \mid \overline{s_i'^p}$ *by a specific rule in the operational semantics. There exists a separation logic derivation* $\overline{s_i'}$*, with* $\forall\ i.\, s_i' = \{P'\}_{\gamma_1'}\ s_i'^p\ \{Q'\}_{\gamma_2'}$*, obtained by reducing the proof tree of* $\overline{s_i}$*, corresponding to the taken operational semantics step (the form of this proof tree is given in the proof to this theorem). For each such separation logic derivation, we write* $\langle \overline{s}, h\rangle \mid \overline{s_i} \hookrightarrow \langle \overline{s'}, h'\rangle \mid \overline{s_i'}$ *to denote the corresponding transformation between the two separation logic derivations, and call the semantics defined in this way the lifted operational semantics, as they lift the previously defined operational semantics to include separation logic derivations as well.*

Proof.

This proof proceeds as a case-based analysis, splitting on the specific operational semantics rule used in the source language and then applying inversion on the root of the separation logic derivation. Some attention has to be paid to the Conseq and Frame separation logic axioms, because these can interfere with the proof in many different locations. In general, they can be considered as nothing more than 'proof glue' and are deleted as soon as the statement that they're glue towards (= the lower statement, closer to the root in the proof tree) is deleted. They are hence not explicitly mentioned in the proof below.

Because most rules in the operational semantics (except for function application FApp and Return) only affect the *last* function execution in the current sequence of function body executions, we can leave the other, unchanged function executions implicit in most rules, simplifying the proof notation and avoiding clutter. The same goes for Seq statements within the current function execution. These Seq statements, corresponding to the rest of the function body, have to be skipped as well in the separation logic proof tree, since this part of the tree remains unaltered. This operation is also kept implicit, and the proof uses the currently executing source statement as a starting point, omitting the unaltered parts of the proof tree. This discussion boils down to the fact that we omit the separation logic proof of the execution context C in the proof below and show only the proofs of the contents of its holes, for convenience of notation. If we require the contents of the last 2 holes of C, as is the case for the FApp and Return rules below, we use the notation $h_1 :: h_2$ with h_1 and h_2 the proofs of the before-last and last holes of C, respectively. In the below notation of the lifted operational semantics $\hookrightarrow$, the stack and heap are left out as well, as they are identical to the stack and heap in the standard operational semantics. For clarity, $\hookrightarrow$ is subscripted with the operational semantics rule that is applied.

As discussed with the introduction of the source-level operational semantics, the FApp and Return rules used here are the *lifted* versions presented earlier. Without the extra hole annotations, the proof would not be completable.

Skip
By inversion, we have that the proof tree of skip; c must have a Seq rule applied to a Skip rule and the proof of c, $Proof_c$, as root. $Proof_c$ is -by definition- the proof of the resulting source program c.

$$\dfrac{\dfrac{}{\{\overline{m} : P\}_{\gamma_1}\ \text{skip}\ \{\overline{m} : P\}_{\gamma_1}}\ (\textsc{Skip}) \qquad \mathit{Proof}_c}{\{\overline{m} : P\}_{\gamma_1}\ \text{skip}; c\ \{\overline{n} : Q\}_{\gamma_2}}\ (\textsc{Seq}) \hookrightarrow_{(\textsc{Skip})} \mathit{Proof}_c$$

Malloc
The malloc postcondition shifts to the precondition, so that the implicit Seq statements are still valid.

$$\dfrac{}{\{\overline{m} : P\}_{\gamma_1}\ id_{\text{prog}} = \text{malloc}(sexp * \text{sizeof}(\tau))\ \{\overline{n} : Q\}_{\gamma_2}}\ (\textsc{Malloc}) \hookrightarrow_{(\textsc{Malloc})} \dfrac{}{\{\overline{n} : Q\}_{\gamma_2}\ \text{skip}\ \{\overline{n} : Q\}_{\gamma_2}}\ (\textsc{Skip})$$

For
Since the ForUnroll rule in the source-level operational semantics simply unrolls a foreach loop when executing it, we have to make sure the separation logic proof of the For rule can be used to prove each consecutive iteration. This is easily seen to be true, since the triple $\{sexp_{\gamma_{\text{pre}}} \leq i_s < sexp'_{\gamma_{\text{pre}}} * Inv[i_s]\}_{\gamma[i_s]}\ sstm\ \{Inv[i_s + 1]\}_{\gamma[i_s+1][i:\_]}$ in the For axiom can be instantiated with each value $sexp_{\gamma_{\text{pre}}} \leq i_s < sexp'_{\gamma_{\text{pre}}}$ of i_s to prove each individual For iteration.

The only caveat is that axioms such as Split in principle always create fresh resources, which will not be the case if we use the same proof for each iteration of the for loop. Since this is a simple renaming issue, we ignore this for the rest of the proof, and implicitly assume that every iteration of the unrolled for loop uses fresh variables, that were introduced when the loop was unrolled. Analogously, the compiled target code will not require hoisting anymore (we hence ignore for-hoisting for the rest of this proof), since all generated declarations will be fresh as well. This alternative mode of execution allows us to more easily construct the proof and is quite obviously equivalent to the hoisting and normal loop unrolling we defined in the compilation and normal semantics, respectively. We could alternatively have allowed the Split axiom to use non-fresh resources too, which would be a cleaner solution, but this would have complicated compilation.

Split/SplitRange (depending on the form of the resource),Join/JoinRange (depending on the form of the resource), Flatten ,Collect
Analogous to Malloc.

IfTrue
The If compilation-rule also uses hoisting to avoid mismatch between the sets of variables declared in different branches. Since hoisting itself clearly does not influence equi-termination, we ignore it in If-statements for the rest of this proof.

$$\dfrac{\mathit{Proof}_1 \qquad \mathit{Proof}_2}{\{\overline{m} : P\}_{\gamma_1}\ \text{if}\ sexp\ \text{then}\ p_1\ \text{else}\ p_2\ \{\overline{n} : Q\}_{\gamma_2}}\ (\textsc{If}) \hookrightarrow_{(\textsc{IfTrue})} \mathit{Proof}_1$$

IfFalse
Symmetric to IfTrue.

VarDecl
Analogous to Malloc.

VarAsgn
Analogous to Malloc.

FApp
A new function execution is created in the FApp rule. The proof tree for the current execution is replaced by a proof tree for a hole. A proof tree is added for the new execution. This proof tree corresponds to the Hoare triple appearing in the ImplVerif axiom for the called function. The root axiom X of the new separation logic derivation can either be Return, Conseq or Frame. In the last 2 cases, as mentioned, the top-level rules up to the Return rule are dropped for the following steps of the lifted semantics execution.

$$\frac{Cond}{\{\overline{m} : P\}_{\gamma_1}\ \overline{id_{\text{prog}}} = f(\overline{sexp})\ \{\overline{n} : Q\}_{\gamma_2}}\ (\text{FApp})$$

$$\hookrightarrow_{(\text{FApp})}\ \frac{Cond'}{\{\}_{\gamma_1}\ \overline{id_{\text{prog}}} = \bullet^{\overline{id_{\text{arg}}}=\overline{sexp}}_{\text{POST}(f),\text{PRE}(f)}\ \{\overline{n} : Q\}_{\gamma_2}}\ (\text{FApp})$$

$$::\ \frac{Proof'}{\{\overline{p} : PRE\}_{[\overline{id_{\text{arg}}:id_{\text{arg}}}]}\ BODY; \text{return } sexp\ \{\overline{q} : POST\}_{\gamma}}\ (\text{X})$$

Return

$$\frac{Cond}{\{\}_{\gamma_1}\ \overline{id_{\text{prog}}} = \bullet^{\overline{id_{\text{arg}}}=\overline{sexp}}_{\text{POST}(f)}\ \{\overline{n} : Q\}_{\gamma_2}}\ (\text{FApp})$$

$$::\ \overline{\{\overline{p} : R\}_{\gamma}\ \text{return } sexp\ \{\overline{p} : R \wedge\ result == sexp_{\gamma}\}_{\gamma}}\ (\text{Return})$$

$$\hookrightarrow_{(\text{Return})}\ \overline{\{\overline{n} : Q\}_{\gamma_2}\ \text{skip}\ \{\overline{n} : Q\}_{\gamma_2}}\ (\text{Skip})$$

ArrayMut
Analogous to Malloc.

ArrayLkup
Analogous to Malloc.

ProgExec
The specific form of the FApp statement below is achievable because the program well-formedness requires the main function to have such a compatible contract in the ProgWF axiom.

$$\frac{Cond}{\vdash (C_1\ \ldots C_k\ //@\text{main} = id\ )}\ (\text{ProgVerif}) \hookrightarrow_{(\text{ProgExec})}$$

$$\dfrac{\dfrac{Cond'}{\{\text{true}\}_{\cdot}\ \overline{\tau}\ \overline{x}\ \{\text{true}\}_{\cdot[\overline{x:v}]}}\ (\text{VarDecl}) \qquad \dfrac{Cond''}{\{\text{true}\}_{\cdot[\overline{x:v}]}\ \overline{x} = id()\ \{\text{true}\}_{\gamma}}\ (\text{FApp}) \qquad \dfrac{Cond'''}{\{\text{true}\}_{\gamma}\ \text{return}\ \overline{x}\ \{\text{true}\}_{\gamma}}\ (\text{Return})}{\{\text{true}\}_{\cdot}\ (\overline{\tau}\ \overline{x}; \overline{x} = id(); \text{return}\ \overline{x})\ \{\text{true}\}_{\gamma}}\ (\text{Seq})$$

□

The specific form of the new reduced proof for $\overline{s_i'^p}$ above, the relation between the proof tree reduction and the applied source-level operational semantics step, together with the relation between the source-level and target-level operational semantics steps are used to prove that R is a simulation relation. This proof follows below:

Proof of theorem 2 property 1.

Follows trivially from the fact that $s \rightsquigarrow t$, because R holds, and inversion, leading to the Return compilation rule as the only option. □

Proof of theorem 2 property 2.

The proof is again structured in a case-based fashion, in the same way as the proof of the preservation theorem. The trees for $\overline{s_i'^p}$ from the preservation theorem are reused here, as is the notation. This means that consequence and frame rules are again left out in the proof, but we will talk about Conseq below and the Frame does not fundamentally alter anything, as is the context in the case of the operational semantics. The source program $\overline{s_i}$ is denoted as s and the target program $\overline{t_i}$ as t.

The rule Conseq can be conceptually split up into three parts:

$$\{P_{\text{pre}}\}_{\gamma_{\text{pre}}}$$
$$ghostStep_1$$
$$\{P\}_{\gamma}$$
$$c$$
$$\{Q\}_{\gamma'}$$
$$ghostStep_2$$
$$\{Q_{\text{post}}\}_{\gamma_{\text{post}}}$$

The second step is then conceptually justified by the underlying proof $\{P\}_\gamma\ c\ \{Q\}_{\gamma'}$ and reified in the compilation rule by the compilation p of this underlying proof. The first and third step then share a common structure and form, and are compiled in the same way too, so we will only look at $ghostStep_1$, which corresponds to the compiled statements $rename_{\text{pre}}; \overline{\tau_{\text{pre}}\ n_{\text{pre}}}$. In this case, we have the assumptions that $\forall x \in \text{dom}(\gamma).\ P \vdash \gamma_{\text{pre}}(x) == \gamma(x)$, $P_{\text{leak}} \subseteq P_{\text{pre}}$, $P_{\text{leak}} \approx_{\text{Names}} P_{\text{rename}}$, $\text{CN}(P_{\text{leak}}), \text{CN}(P_{\text{rename}}) \rightsquigarrow_{\text{RenameDecl}} rename_{\text{pre}}, \vdash P_{\text{rename}} \Rightarrow P, \overline{n_{\text{pre}}}$ fresh, $\text{CN}(P) \setminus \text{CN}(P_{\text{pre}}) = \overline{n_{\text{pre}}}$ and $\text{ReifiesToType}(\overline{n_{\text{pre}}}) = \overline{\tau_{\text{pre}}}$. It suffices to prove that if the program states are related before the start of the Conseq axiom, then the target statements $rename_{\text{pre}}; \overline{\tau_{\text{pre}}\ n_{\text{pre}}}$ will take a finite number of steps and end up in a state that is still related to the original source state (with the new precondition P). From the definition of R, it is clear that dropping the assertions that are not in P_{leak} preserves the relation (since the judgement $h_s \vdash^{\text{src}}_{\delta} P$ is monotonous in the heap). $rename_{\text{pre}}$ will preserve relatedness when simultaneously changing the precondition to P_{rename}. Declaring the new variables n_{pre} (and initializing them to nulls) will preserve relatedness when simultaneously changing the precondition to P, because the separation logic implication does not allow manipulating impure chunks, except for implications of the following form:

$$exp == \text{true} \vdash assert \Leftrightarrow exp\ ?\ assert$$
$$exp == \text{false} \vdash \text{true} \Leftrightarrow exp\ ?\ assert$$

For such implications, simply declaring the fresh resource names is enough to restore relatedness.

Every other case of the proof consists of three consecutive proof steps:

(1) First, the operational semantics rule that can be applied on t is derived from the fact that $s \rightsquigarrow t$ and inversion. This step is kept implicit, because the *core* rule in source and target language will always match (there are often some extra rules for eg. declarations that have to be executed, which we often disregard).

(2) The fact that the compiled code is applicable (the precondition of each required operational semantics rule is upheld in the current program state) to t is proven.

(3) Given the source and target level operational semantics steps that are taken, namely $\langle s_s, h_s\rangle \mid s \hookrightarrow \langle s'_s, h'_s\rangle \mid s'$ and $\langle s_t, h_t\rangle \mid t \hookrightarrow \langle s'_t, h'_t\rangle \mid t'$, it is proven that if $(\langle s_s, h_s\rangle \mid s)\ R\ (\langle s_t, h_t\rangle \mid t)$, then also $(\langle s'_s, h'_s\rangle \mid s')\ R\ (\langle s'_t, h'_t\rangle \mid t')$.

Skip
Applying the Seq compilation rule to s tells us that t has the form skip; $Proof^t_c$, where $Proof_c \rightsquigarrow Proof^t_c$. By inversion, only the Skip operational semantics rule can be applied to t to obtain $t' = Proof^t_c$.

Prove that the rules are applicable:
Trivially true, since there are no conditions for applying the Skip rule.

Prove that R still holds after the steps:
The definition of t' immediately implies that $s' \rightsquigarrow t'$. Since the heap and the stack are unaltered in both target and source language, $s'\ R\ t'$ holds trivially.

Malloc
Applying the Malloc compilation rule, t will evaluate to skip using VarDecl, Malloc, VarAsgn and 2 applications of Skip.

Prove that the rules are applicable:
Trivial.

Prove that R still holds after the steps:
The target level allocated 0-length capability id_{prog1} corresponds to the original source pointer id_{prog1} through function valmap_b, where the couple (l_{source}, l_{target}) has to be added to b. Target variable id_{prog2} corresponds to the newly malloced separation-logic chunk, because the couple $(l_a, (l_{source}, 0))$ is added to δ. Multiple steps are taken in the target language because of variable declarations, but that does not matter.

Ghost Commands
The corresponding axioms: Split/SplitRange (depending on the form of the resource),Join/JoinRange (depending on the form of the resource), Flatten ,Collect
Nothing new; it has to be shown that the newly created chunks map to variables in the target stack; the source state doesn't alter.

IfTrue
Prove that the rules are applicable:
Prove that if $[\![sexp]\!]_s$ = true in the source, that the same holds in the target, so the IfTrue rule is the only one that can be applied there. This follows directly from Theorem 1.

Prove that R still holds after the steps:
Only the symbolic heap changes. The expression $sexp_\gamma$ is added to it, and we know that $[sexp_\gamma]_\delta = [\![sexp]\!]_s = \text{true}$, which means the new symbolic heap is still satisfied.

IfFalse
Analogous to IfTrue.

VarDecl
Prove that R still holds after the steps:
This step is trivial, because the variables will receive default values, related by valmap, given how $\rightsquigarrow_{\text{CompileTypes}}$ maps source types to target types and the definition of $\rightsquigarrow_{\text{def}}$. The appropriate new associates have to be added to b and δ.

VarAsgn
Prove that R still holds after the steps:
We have to show that $[\![sexp]\!]_s$ has corresponding values in the target and source language. This follows directly from Theorem 1.

FAppLifted
This case becomes far more straightforward because R uses the $\rightsquigarrow_{\text{NoStubs}}$ compilation, rather than $\rightsquigarrow$. Because of this fact, we don't have to take stubs into account here.

By applying FAppLifted, we have that $(\langle s_\text{s}, h_\text{s}\rangle, s)$ evaluates to $(\langle s'_\text{s}, h_\text{s}\rangle, s')$ with

- $s = C :: \overline{id_{\text{prog}}} = f(\overline{sexp}); F$
- $s' = C :: \{\overline{id_{\text{prog}}}\} = \bullet^{\overline{id_{\text{arg}}}=\overline{sexp}}_{POST_\text{f}}; F :: \{PRE_\text{f}\}_{[\overline{id_{\text{arg}}}:\overline{id_{\text{arg}}}]}\ BODY_\text{s}; \text{return } \overline{sexp'}\ \{POST_\text{f}\}_\_$
- $\Sigma_{\text{op},s}(f) = \{\overline{\tau_{\text{arg}}\ id_{\text{arg}}}\ \{BODY; \text{return } \overline{sexp'}\}\}$
- $[\![\overline{sexp}]\!]_{s_\text{s}} = \overline{n_s}$
- $\overline{s} = s_\text{s} :: \overline{s_{s,r}}$
- $\overline{s'} = [\overline{id_{\text{arg}}} \mapsto \overline{n_s}] :: \overline{s}$

From the compilation rule, we know that

- $$s = C :: (\vdash \{PRE[subst_{\text{pre}}]\}_\gamma\ \overline{id} = f(\overline{sexp})\ \{POST[subst_{\text{post}}]\}_{\gamma'}); \vdash F$$
- $$t = C_\text{t} :: \overline{\tau_n\ n};\ \{\overline{id}, \overline{n}\} = f_{\text{comp}}(\overline{sexp}, \overline{m}); F_\text{t}$$
- $\vdash F \rightsquigarrow_{\text{NoStubs}} F_\text{t}$
- $\Sigma(f) = \{PRE_f, POST_f, \overline{id_{\text{arg}}}\}$
- $PRE_f \approx_{\text{Names}} PRE$
- $POST_f \approx_{\text{Names}} POST$
- $\overline{id} \in \text{dom}(\gamma)$
- $\gamma' = \gamma[\overline{id} : \overline{x}]$

- $\overline{x}, \overline{n}$ fresh
- $[subst_{\text{pre}}] = [\overline{id_{\text{arg}}} \mapsto \overline{sexp}_\gamma]$
- $[subst_{\text{post}}] = [subst_{\text{pre}}][\overline{result} \mapsto \overline{x}]$
- $\text{CN}(PRE) = \overline{m}$
- $POST \rightsquigarrow_{\text{resDecl}} \overline{\tau_n\ n}$

By applying the VarDecl, Skip and FApp rules in the target, we get that $(\langle \overline{s_{\text{t}}}, h_{\text{t}} \rangle, t)$ evaluates to $(\langle \overline{s'_{\text{t}}}, h_{\text{t}} \rangle, t')$, with

- $\Sigma_{\text{op},t}(f) = \{\overline{\tau_{\text{arg}}\ id_{\text{arg}}}, \overline{\tau_n\ m}\ \{BODY_{\text{t}}; \text{return}\ \{\overline{texp'}\}\}\}$
- $[\![\overline{texp}, \overline{m}]\!]_{s_{\text{t}}} = \overline{k_t, k_{t,m}}$
- $(\overline{texp}, \overline{m}), s_{\text{t}} \rightsquigarrow_{\text{StoreLinCap}} [\overline{env}]$
- $\overline{s_{\text{t}}} = s_t :: \overline{s_{\text{r}}}$
- $\overline{s'_{\text{t}}} = [\overline{id_{\text{arg}}}, \overline{m} \mapsto \overline{k_t}, \overline{k_{t,m}}] :: s_t[\overline{env}] :: \overline{s_{\text{r}}}$
- $t' = C_{\text{t}} :: \{\overline{id_{\text{prog}}}\} = \bullet^{\overline{id_{\text{arg}}}=\overline{texp}, \overline{m}=\overline{m}}; F_{\text{t}} :: BODY_{\text{t}}; \text{return}\ \{\overline{texp'}\}$

Prove that R still holds after the steps:
It remains to prove that the two new frames in source and target both satisfy R_{comp}, assuming that R_{comp} held for the original frames of s and t. Specifically, we prove that

- $sprog, tprog \vdash_b^{\delta'} (\langle \overline{id_{\text{arg}}} \mapsto \overline{n_s}, h_{\text{s}} \rangle \mid BODY_{\text{s}})\ R_{\text{comp}}\ (\langle \overline{id_{\text{arg}}}, \overline{m} \mapsto \overline{k_t}, \overline{k_{t,m}}, h_{\text{t}} \rangle \mid BODY_{\text{t}})$ with $\overline{id_{\text{link}}} = (ID_{\log,i} \setminus \overline{id_{\text{arg}}}) \cap \text{V}(POST_f)$ (for i the id of the current frame in the R relation), $\delta' = (\delta \mid_{id_{\text{link}}})[id_{\text{arg}} \mapsto \overline{n_{\text{s}}}]$
- $sprog, tprog \vdash_b^{\delta} (\langle s_{\text{s}}, \emptyset \rangle \mid \{\overline{id_{\text{prog}}}\} = \bullet^{\overline{id_{\text{arg}}}=\overline{sexp}}_{\text{POST}(f)}; F)\ R_{\text{comp}}\ (\langle s_t[\overline{env}], \emptyset \rangle \mid \{\overline{id_{\text{prog}}}\} = \bullet^{\overline{id_{\text{arg}}}=\overline{texp}, \overline{m}=\overline{m}}; F_{\text{t}})$

Additionally, we need to show that frame linking holds.

Notice first that

- $\{\overline{id_{\text{prog}}}\} = \bullet^{\overline{id_{\text{arg}}}=\overline{sexp}}_{\text{POST}(f)}; F \rightsquigarrow \{\overline{id_{\text{prog}}}\} = \bullet^{\overline{id_{\text{arg}}}, \overline{m}=\overline{texp}, \overline{m}}; F_{\text{t}}$ This follows from decomposing the corresponding assumption in the proof of $(\langle \overline{s_{\text{s}}}, h_{\text{s}} \rangle, s)\ R\ (\langle \overline{s_{\text{t}}}; h_{\text{t}} \rangle, t)$
- $\vdash BODY_{\text{s}} \rightsquigarrow_{\text{NoStubs}} BODY_{\text{t}}$ This follows from a global assumption about the bodies in $\Sigma_{\text{op},s}$ and $\Sigma_{\text{op},t}$. It follows from the fact that both are deduced from sprog and tprog and the assumption that $\vdash sprog \rightsquigarrow_{\text{NoStubs}} tprog$ in the definition of R.

The proof validity of the source statements follows similarly.

Next, we need to prove the different facts in R_{comp}.

- $s_{\text{t}} \approx s_{\text{s}}$. For the callee stacks, this follows from Theorem 1 and from the relatedness of the stack variables in the old stack frame. For the caller stacks, this follows from the relatedness of the old stack frames, from the fact that the GatherLinCap rules will never modify variables that

do not contain linear capabilities and the fact that such values containing linear capabilities are not in the range of valmap_b.

- $h_\text{t} \approx h_\text{s}$ For the callee heaps, the properties follow from the corresponding properties of the original frames' R_comp relation. For the caller heaps, the heaps are simply empty, and the properties are trivial.
- The property that $(id_\text{arg}, \overline{m} \mapsto \overline{k_t}, \overline{k_{t,m}}), h_{t,\text{reify}} \vdash^\text{tgt}_{\delta',b} PRE_f$ follows (for the callee frames) from the fact that $s_\text{t}, h_{t,\text{reify}} \vdash^\text{tgt}_{\delta,b} PRE[subst_\text{pre}]$, because of what we know about $subst_\text{pre}$, $\overline{k_\text{t}}$ and $\overline{n_\text{s}}$, s_t. For the caller frames, the property is trivial, since the precondition is trivial.

We also need to prove the InverseMap property for the old and new frame:

- InverseMap$(\delta', PRE_f, [id_\text{arg} \mapsto id_\text{arg}], \overline{id_\text{arg}} \mapsto \overline{n_s}, h_\text{s})$: First, $\delta'(id_\text{arg}) = \overline{n_\text{s}}$ is true by definition, so this is fine. Second, the fact that $h_\text{s} \vdash^\text{src}_{\delta'} PRE_f$ follows from the corresponding fact that $h_\text{s} \vdash^\text{src}_{\delta} PRE_f[\overline{id_\text{arg}} \mapsto \overline{sexp}]$ which we have from the InverseMap property for the old frame.
- InverseMap$(\delta, \text{true}, \gamma, s_\text{s}, \emptyset)$: In this case, $\emptyset \vdash^\text{src}_{\delta} \text{true}$ holds trivially, and the fact that $s_\text{s}(x) = [\gamma(x)]_\delta$ for all $x \in \text{dom}(\gamma)$ is known from the InverseMap property for the old frame.

Finally, we need to prove the frame linking property for the R obtained from composing the new frame relations. Concretely, we need to think about the link between

- the caller frame to its parent. In this case, the required equalities follow from linking in the previous simulation step, as the postcondition and δ have not been modified.
- the callee frame and the caller frame: in this case, equality of the postcondition follows by definition, equality of $\delta'(\overline{id_\text{arg}})$ and equality of δ' on the id_link to δ follow immediately from the choice of δ'

ReturnLifted

From ReturnLifted, we know that $(\langle s_\text{s}, h_\text{s}\rangle, s)$ evaluates to $(\langle s'_\text{s}, h_\text{s}\rangle, s')$ with

- $s = C :: \{\overline{id_\text{prog}}\} = \bullet^{\overline{id_\text{arg}}=\overline{sexp}}_{\text{POST}(f)}; F :: \text{return } \{\overline{sexp'}\}$
- $s' = C :: \text{skip}; F$
- $s_\text{s} = s_{\text{s},0} :: s_{\text{s},1} :: \overline{s_{\text{s,r}}}$
- $\overline{s'_\text{s}} = s_{\text{s},1}[\overline{id_\text{prog}} \rightarrow \overline{v_\text{ret}}] :: \overline{s_{\text{s,r}}}$
- $[\![\overline{sexp'}]\!]_{s_{s,0}} = \overline{v_\text{ret}}$

From the compilation rule, we know that

- $t = C_\text{t} :: \{\overline{id_\text{prog}}, \overline{n}\} = \bullet^{\overline{id_\text{arg}}=\overline{sexp}}; F_t :: \text{return } \{sexp', \overline{m}\}$
- $\vdash F \rightsquigarrow_\text{NoStubs} F_\text{t}$

By applying the Return rule in the target, we know that $(\langle \overline{s_\text{t}}, h_\text{t}\rangle, t)$ evaluates to $(\langle \overline{s'_\text{t}}, h_\text{t}\rangle, t')$, with

- $\overline{s_\text{t}} = s_{\text{t},0} :: s_{\text{t},1} :: \overline{s_{\text{t,r}}}$

- $\overline{s'_{\mathrm{t}}} = s_{\mathrm{t},1}[\overline{id_{\mathrm{prog}}} \mapsto \overline{v_{\mathrm{ret,t}}}][\overline{n} \mapsto \overline{v'_{\mathrm{ret,t}}}] :: \overline{s_{\mathrm{t,r}}}$
- $[\![\overline{sexp'}]\!]_{s_{\mathrm{t},0}} = \overline{v_{\mathrm{ret,t}}}$
- $[\![\overline{m}]\!]_{s_{\mathrm{t},0}} = \overline{v'_{\mathrm{ret,t}}}$
- $(\overline{sexp'}, \overline{m}), s_{\mathrm{t},0} \rightsquigarrow_{\mathrm{StoreLinCap}} -$
- $t' = \mathrm{skip}; F_{\mathrm{t}}$

Note that by the lifted operational semantics and the axiom FApp, the new precondition of skip; F is $POST_f[\overline{id_{\mathrm{arg}}} \mapsto \overline{sexp_{\gamma_1}}]$ except for renaming of the chunks from $\overline{m}$ to $\overline{n}$. The new γ'_1 in the precondition is defined as $\gamma_1[\overline{id_{\mathrm{prog}}} \mapsto \overline{id_{\mathrm{res}}}]$.

Prove that R still holds after the steps:
It remains to prove that the new frame in source and target both satisfy R_{comp}, assuming that R_{comp} held for the two original frames of s and t. Specifically, we assume that

- $sprog, tprog \vdash_b^{\delta_1} (\langle s_{\mathrm{s},1}, h_{\mathrm{s},1}\rangle \mid \{\overline{id_{\mathrm{prog}}}\} = \bullet^{\overline{id_{\mathrm{arg}}}=\overline{sexp}}_{POST_f}; F)\ R_{\mathrm{comp}}\ (\langle s_{\mathrm{t},1}, h_{\mathrm{t},1}\rangle \mid \{\overline{id_{\mathrm{prog}}}, \overline{n}\} = \bullet^{\overline{id_{\mathrm{arg}}}=\overline{sexp}}; F_{\mathrm{t}})$
- $sprog, tprog \vdash_b^{\delta_0} (\langle s_{\mathrm{s},0}, h_{\mathrm{s},0}\rangle \mid \mathrm{return}\ \{\overline{sexp'}\})\ R_{\mathrm{comp}}\ (\langle s_{\mathrm{t},0}, h_{\mathrm{t},0}\rangle \mid \mathrm{return}\ \{sexp', \overline{m}\})$

and we will prove that $sprog, tprog \vdash_b^{\delta_1} (\langle s_{\mathrm{s},1}[\overline{id_{\mathrm{prog}}} \to \overline{v_{\mathrm{ret}}}], h'_{\mathrm{s},1}\rangle \mid \mathrm{skip}; F)\ R_{\mathrm{comp}}\ (\langle s_{\mathrm{t},1}[\overline{id_{\mathrm{prog}}} \mapsto \overline{v_{\mathrm{ret,t}}}][\overline{n} \mapsto \overline{v'_{\mathrm{ret,t}}}], h'_{\mathrm{t},1}\rangle \mid \mathrm{skip}; F_{\mathrm{t}})$ where

- $h'_{\mathrm{s},1} = h_{\mathrm{s},0} \uplus h_{\mathrm{s},1}$
- $h'_{\mathrm{t},1} = h_{\mathrm{t},0} \uplus h_{\mathrm{t},1}$
- $\delta'_1 = \delta_1[\overline{id_{\mathrm{res}}} \mapsto \overline{v_{\mathrm{ret}}}]$

Additionally, we will show that frame linking holds.

Considering the requirements in the definition of R_{comp}, the requirements that skip; $F \rightsquigarrow_{\mathrm{NoStubs}}$ skip; F_{t}, that skip; F is of the form $\Sigma^{ax} \vdash \{P\}_\gamma\ sstm; \mathrm{return}\ \overline{sexp}\ \{Q\}_{\gamma'}$ for $\Sigma^{ax} \in \Sigma^{ax}_{sprog}$ and $CN(P) = \overline{m}$ follow easily from the corresponding requirements for the caller stack frame.

We need to show that $s'_{\mathrm{t}} \approx s'_{\mathrm{s}}$, i.e. that

- $\mathrm{dom}(s'_{\mathrm{t},1}) = \mathrm{dom}(s'_{\mathrm{s},1}) \uplus \overline{n} \uplus \overline{n_{\mathrm{null}}} \uplus \overline{id_{\mathrm{aux}}}$
- $\forall id \in \mathrm{dom}(s'_{\mathrm{s},1}).\ s'_{\mathrm{t},1}(id) = \mathrm{valmap}_b(s'_{\mathrm{s},1}(id))$

Both follow from the definition of s'_{s} and s'_{t}, the corresponding properties about $s_{\mathrm{s},1}$ and $s_{\mathrm{t},1}$, and Theorem 1 with the facts about $s_{\mathrm{s},0}$ and $s_{\mathrm{t},0}$.

We define $h'_{\mathrm{t},1,\mathrm{real}} = h_{\mathrm{t},0,\mathrm{real}} \uplus h_{\mathrm{t},1,\mathrm{real}}$ and similarly $h'_{\mathrm{t},1,\mathrm{reify}} = h_{\mathrm{t},0,\mathrm{reify}} \uplus h_{\mathrm{t},1,\mathrm{reify}}$. The properties about $h_{\mathrm{t},1,\mathrm{real}}$ follow directly from the corresponding properties about the old frames, and likewise for the fact that $\mathrm{dom}(h'_{\mathrm{t},1,\mathrm{reify}}) \cap \mathrm{range}(b) == \emptyset$.

It remains to prove that $s'_{\mathrm{t},1}, h'_{\mathrm{t},1,\mathrm{reify}} \vdash^{\mathrm{tgt}}_{\delta'_1,b} POST_f[\overline{id_{\mathrm{arg}}} \mapsto \overline{sexp_{\gamma_1}}][\overline{result} \mapsto \overline{id_{\mathrm{res}}}]\gamma'_1$. We know that $POST_f$ does not contain program variables except for id_{arg} and the special variables $\overline{result}$, so we can drop the last γ'_1. From frame linking, we know that $POST_f$ is also the postcondition of the

callee frame, that $[\overline{sexp}_{\gamma_1}]_{\delta_1} == \delta_0(\overline{id_{\text{arg}}})$ and that $\delta_0(\overline{id_{\text{link}}}) = \delta_1(\overline{id_{\text{link}}})$ for $\overline{id_{\text{link}}} = (ID_{\log,1} \setminus \overline{id_{\text{arg}}}) \cap \text{V}(POST_f)$. We also know that $\delta_1'(\overline{id_{\text{res}}}) = \overline{v_{\text{ret}}}$.

Because of this, it suffices to prove that $s_{\text{t},1}', h_{\text{t},1,\text{reify}}' \vdash^{\text{tgt}}_{\delta_0[\overline{id_{\text{res}}} \mapsto \overline{v_{\text{ret}}}], b} POST_f$. From the R_{comp} relation for the callee frame, we know that $s_{\text{t},0}, h_{\text{t},0,\text{reify}} \vdash^{\text{tgt}}_{\delta_0, b} POST_f$. From that last judgement and the facts that $\overline{s_{\text{t}}'} = s_{\text{t},1}[\overline{id_{\text{prog}}} \mapsto \overline{v_{\text{ret,t}}}][\overline{n} \mapsto \overline{v'_{\text{ret,t}}}] :: \overline{s_{\text{t,r}}}$, $[\![\overline{m}]\!]_{s_{\text{t},0}} = \overline{v'_{\text{ret,t}}}$, it follows that $(\overline{sexp}, \overline{m}), s_{\text{t},0} \rightsquigarrow_{\text{StoreLinCap}} -$ and that $s'_{\text{t},1}(\overline{n}) = s_{\text{t},0}(\overline{m})$. Additionally, we have that $h'_{\text{t},1,\text{reify}} \supseteq h_{\text{t},0,\text{reify}}$, so it follows easily that $s_{\text{t},1}', h_{\text{t},1,\text{reify}}' \vdash^{\text{tgt}}_{\delta_0, b} POST_f$.

We still need to prove that the InverseMap property holds for the new frame, i.e. $\text{InverseMap}(\delta_1', POST_f[\overline{id_{\text{arg}}} \mapsto \overline{sexp_{\gamma_1}}][\overline{result} \mapsto \overline{id_{\text{res}}}], \gamma_1', s_{\text{s},1}', h_{\text{s},1}')$. The fact that $s'_{\text{s},1}(x) = [\gamma_1'(x)]_{\delta_1'}$ for all $x \in \text{dom}(\gamma_1')$ follows from the same fact for the old caller stack frame, as well as the definitions of $s'_{\text{s},1}$, γ_1' and δ_1'. The fact that $h'_{\text{s},1} \vdash^{\text{src}}_{\delta_0[\overline{id_{\text{res}}} \mapsto \overline{v_{\text{ret}}}], b} POST_f[id_{\text{arg}} \mapsto sexp_{\gamma_1}][\overline{result} \mapsto \overline{id_{\text{res}}}]$ follows from the corresponding fact in the InverseMap property of the old callee frame, using what we know about the shape of $POST_f$ from the axiom Return and the definition of $\overline{v_{\text{ret}}}$.

Frame linking follows easily because the relevant components of the old caller frame have not been modified.

ArrayMut
Again uses the correspondence, caused by compositionality, between the source and target values of evaluated expressions to prove that the resulting heaps will still be related.

ArrayLkup
Uses the correspondence between the source and target heaps to prove that the new stack frames will correspond as well.

ForUnroll
The operational semantics rule ForUnroll consists of nothing more than unrolling the for loop, in both the source and target languages.

Remember that we mentioned in the definition of the lifted operational semantics that we will be ignoring hoisting for the duration of this proof, because we assume resources in the source language and reified variables in the target language to be implicitly renamed in the operational semantics when a for-loop is unrolled.

Prove that the rules are applicable:
Since the expressions *texp* and *texp′* in the target ForUnroll rule are equal to *sexp* and *sexp′* in the source, since valmap holds over the source and target stack frames due to R_{comp} and given the compositionality of valmap, it holds that both for loops will either be unrolled or neither one will be.

Prove that R still holds after the steps:
This is trivial, since simply unrolling the for loop does not change any state.

GuardTrue Trivial.

ProgExec
Trivial, because stack and heap are empty in the produced source and target code, and so are the symbolic heap and the arguments, since main functions are without arguments.

□

Now that the relation R has been properly defined and it has been proven to be a simulation relation, we split up the CompilerEquiTermination inference rule we wanted to prove into 3 consecutive inference rules and prove these

First off, we prove that any source program is related to its compilation in the empty starting state (empty stack and heap).

$$\frac{\vdash s \rightsquigarrow t}{\vdash \mathfrak{C}_s[s]//@\text{main} = id \;\rightsquigarrow\; \mathfrak{C}_t[t]//@\text{main} = id} \qquad \text{(CompilIsSim)}$$
$$(\langle \bullet, \epsilon \rangle \mid\vdash \mathfrak{C}_s[s]//@\text{main} = id\ )\ R\ (\langle \bullet, \epsilon \rangle \mid \mathfrak{C}_t[t]//@\text{main} = id\ )$$

Proof.

The correctness of this inference rule follows directly from the base case of the definition of the simulation relation. □

A second inference rule demonstrates the purpose of proving that R is a simulation relation, by linking the concepts of simulation relation and equi-termination.

$$\frac{(\langle s_s, h_s \rangle \mid s)\ R\ (\langle s_t, h_t \rangle \mid t)}{s \Downarrow \Leftrightarrow t \Downarrow} \qquad \text{(SimToEquiTermin)}$$

Proof.

Left to right direction
A case-based analysis on the compilation rules shows that the target code takes a finite number of steps for each step the source code takes. If the source code terminates, it ends in a return statement, and so does the target code in that case (by the first result of Theorem 2 and assuming it does not get stuck during execution). The target program hence terminates as well.

Right to left direction
Starting from the current (terminating) target program t, either the corresponding source program s

- is stuck. We have to prove that this case cannot occur. We have to prove that if the target program does not get stuck during its execution, then neither does the source program that it was compiled from, or, by contraposition, that if the source program gets stuck during execution, then so does the target program. This intuitively follows from the fact that all source-language checks are either reified during compilation or present in the target-language operational semantics.
- has terminated in a single return statement (the only statement a non-stuck program can naturally terminate in).
- can take a step, and hence (by the second result of Theorem 2) there is some number n of steps that t can take to t', which correspond to one step taken from s to s'. The source program takes at most as many steps as the terminating target program. Using determinacy of target language

semantics and the second result of Theorem 2, the resulting programs s' and t' are still related by R.

This case-based analysis is repeated until termination (which will happen, since we know that t terminates). □

An important caveat for the above proof is that the left-to-right and right-to-left directions of the proof assume that if the source code does not get stuck, then neither does the target code, and vice versa. Both directions combined hence assume that all small programming errors (eg. index out bounds errors, runtime typing errors - the input program is assumed type-checked, ...) either appear simultaneously in both the source and target language, or do not appear at all. It is important for the sanity of our compiler, that stuck code is compiled to stuck code, and never to diverging code (or properly terminating code), as stated in the assumptions at the start of this section. We do not explicitly prove this, although it would be reasonably easy, but assume this to be a reasonable property of the compilation and the source- and target language operational semantics, as eg. linearity of separation logic chunks is translated to linearity of linear capabilities, array lengths are kept equal, etc. We will make similar assumptions on the presence of these types of program-integrity errors for all future relations we define.

Note that no explicit inductive proof is required here, in contrast to analogous proofs in a trace semantics setting [Chlipala 2017], where equality of the constructed traces has to be checked, and not just equality of the execution outcome.

Equi-termination from the empty state is implied by the above inference rule. Full source and target programs related by R hence equi-terminate given the empty heap and stack at the start.

Note how the SimToEquiTerm rule states its results in terms of the separation-logic verified source code, and not in terms of the regular source code without separation logic proof.

The last inference rule we need to prove CompilerEquitermination is a form of *coherence* to prove that no matter the proof used, the code in the lifted semantics and the code in the regular operational semantics equi-terminate. The following Coherence statement proves exactly that; we can strip away separation logic proofs without influencing termination.

$$\frac{}{(\vdash \mathit{sprog}) \Downarrow \Leftrightarrow\ \mathit{sprog} \Downarrow} \quad \text{(Coherence)}$$

Proof.

Follows from the definition of the lifted semantics in the preservation proof for the right-to-left direction, and simple separation-logic proof erasure in the left-to-right direction. The lifted semantics uses FAppLifted and ReturnLifted instead of FApp and Return, respectively, but the extra annotations do not influence equi-termination. □

The CompilerEquiTermination rule follows immediately from the sequential combination of the CompilIsSim, SimToEquiTermin and Coherence rules.

7 COMPILER SECURITY

7.1 Definitions

This section is structured similarly to section 6, since the security proof has the same building blocks as the correctness proof, but just occurs in the opposite direction.

Proving compiler security comes down to proving the following inference rule, which is the reverse direction of compiler correctness:

$$\frac{s \simeq_{\text{ctx}} s' \qquad \vdash s \rightsquigarrow t \qquad \vdash s' \rightsquigarrow t'}{t \simeq_{\text{ctx}} t'} \tag{CompilerSecurity}$$

To prove the CompilerSecurity rule, it suffices to show that for any target context $(\mathfrak{C}_t, id)$ and given the concrete proof $\vdash s$ of s used in CompilerSecurity, the following inference rule holds:

$$\frac{\vdash s \rightsquigarrow t \qquad \vdash s, (\mathfrak{C}_t, id) \rightsquigarrow_b \vdash \mathfrak{C}_s[s]//@\text{main} = id}{\mathfrak{C}_s[s]//@\text{main} = id \Downarrow \Leftrightarrow \mathfrak{C}_t[t]//@\text{main} = id \Downarrow} \tag{BTEquiTermination}$$

This rule suffices because it allows using the given source-level equi-termination result by linking equi-termination of source and target programs. In this rule, the notation $\rightsquigarrow_b$ is used to denote a so-called *back-translation*, which is a type of opposite direction compilation performed on the context $\mathfrak{C}_t$'s functions, effectively giving them provable separation-logic contracts. The back-translation is constructed in such a way that the equi-termination result of BTEquiTermination holds. This back-compilation $\rightsquigarrow_b$ will be defined in detail below.

We will again prove the above BTEquiTermination rule using a simulation argument. This argument employs a simulation relation (analogous to R in the previous section) to relate source to target *states*, denoted as follows: $(\langle s_s, h_s\rangle \mid s)\ S\ (\langle s_t, h_t\rangle \mid t)$, s is the current separation logic derivation (which also implicitly contains the source code) in the source language and t is the corresponding target level code. The variable s is a sequence of the currently executing function bodies c_i and their separation logic proofs. It is hence a sequence of proof trees. The variables s_x and h_x denote the stack and heap, where $x = s$ for the source language and $x = t$ for the target. The full notation for the relation S is in fact $sprog, tprog \vdash (\langle s_s, h_s\rangle \mid s)\ R\ (\langle s_t, h_t\rangle \mid t)$ with $sprog$ the source program that is being executed and $tprog$ the target program. We often write $s_1\ S\ s_2$ as a shorthand notation, where s_1 denotes the source state consisting of the current stack, heap and proven code and s_2, analogously, denotes the target state.

The above CompilerSecurity rule has to hold for any separation logic proof $\vdash$ of the source components s and s'. We have to, however, perform a back-translation of target context functions to the most general (hence simplest) separation logic contract possible, since this is the only way to construct separation logic proofs for the contracts of back-translated target functions. We call this general separation logic contract the *universal contract*. If we were to back-translate target level functions to functions having concrete contracts and not the universal contract, then application of the FApp rule to construct the separation logic proofs would get stuck on having to fulfill the concrete preconditions of the called functions' contract. Using the *universal contract* as basis for the back-translation also makes it easier to formalize, as will become clear in the next subsection.

A problem that arises because of these universal contracts is that universal contracts in the back-translated context do not combine well with the given non-universal contracts in the components s and s'. The proofs $\vdash$ for the functions in s and s' assume non-universal contracts for the external (imported) functions they outcall, disallowing a direct universal back-translation. The other way around, the back-translated context functions expect s and s''s functions to follow their universal contract calling convention. There is hence a mismatch between the minimal contracts in the back-translated target context and the concrete contracts in s and s'. To solve this problem, we use a back-translated version of the verified component's in- and outcall stubs to adapt from universal to non-universal contracts and back.

Subsection 7.2 defines a couple new concepts and notations we will need to define the back-translation $\rightsquigarrow_b$. Subsection 7.3 will define the back-translation $\rightsquigarrow_b$ itself, which includes a discussion of universal contracts and the role of stubs in the matching of universal to concrete contracts. This subsection also proves that the back-translated context $\mathfrak{C}_s$ indeed constructs a valid source program $\vdash \mathfrak{C}_s//@\text{main} = id$. This entails proving that the different back-translated target statements combine into a proper function proof and that the back-translated stubs perform proper matching of universal to concrete contracts. With all definitions out of the way, subection 7.4 finally proves the BTEquiTermination rule states above, in a fashion very similar to the proof in section 6.

7.2 Auxiliary concepts for the back-translation

7.2.1 Back-translating types and values. Linear capabilities $l^{[a,b]}$ in the target language incorporate an interval $[a, b]$ they can be dereferenced on (they contain no index because they always point to the first interval index a). If we were to back-translate these linear capabilities to regular source-level pointers (l, a), then the information that b provides, in other words the interval length $b - a + 1$, would be lost. We could never define a proper back-translation this way, since there would be no way to back-translate the target statement length(n) with n a linear capability if the length of a was not given in the separation logic contract, as there is no length function in the source language. A first, naive, solution to this problem is to back-translate capabilities n to both a source level pointer n and an integer n^{len} storing n's length. However, this solution does not suffice for nested pointers. Consider the following sample target-level program: int$**$ $n = \ldots$; int$*$ $a = n[k]$. If we were to introduce a *len* variable for each target-level variable, we would back-translate the first line to int $* * n = \ldots$; int $n^{\text{len}} = \ldots$; and the second line to int $* a = n[k]$; int $a^{\text{len}} = \ldots$, but now there is no way for us to possibly know the value of a^{len}, as only the length of n itself was back-translated and not the length of the contents of n. For deeper nestings of arrays, we need to store the lengths of all levels of arrays as well in the back-translation, in case we need these. Taking the len for the top-level only just does not back-translate all information and can hence never lead to an equi-terminating scheme. If we back-translate a k-dimensional array, we need to back-translate lengths for all k dimensions as well. To be able to do this succinctly, we introduced the notion of length-2 tuple types or *pair types* (we do not need tuples of general length), written (τ_1, τ_2) , where τ_1 and τ_2 are any source/target-level type. This takes away the need to define separate length-variables in the back-translation.

Much like the compilation rules in section 4 defined the compilation of source level types CompileType, this section defines the back-translation of target level types InvCompileType. Three of the four cases can just be derived from the inversion of the compilation of types CompileType. The case for inversely compiling linear capabilities shows how lengths are incorporated into the back-translated type, as mentioned in the previous paragraph.

$$\frac{}{\text{int} \rightsquigarrow_{\text{InvCompileType}} \text{int}}\ (\textsc{InvCompileInt}) \qquad \frac{\tau' \rightsquigarrow_{\text{InvCompileType}} \tau}{\tau'*_0 \rightsquigarrow_{\text{InvCompileType}} \tau*}\ (\textsc{InvCompileSrcPtr})$$

$$\frac{\tau_1 \rightsquigarrow_{\text{InvCompileType}} \tau_1' \quad \dots \quad \tau_k \rightsquigarrow_{\text{InvCompileType}} \tau_k'}{(\tau_1, \dots, \tau_k) \rightsquigarrow_{\text{InvCompileType}} (\tau_1', \dots, \tau_k')}\ (\textsc{InvCompileTuple}) \qquad \frac{\tau \rightsquigarrow_{\text{InvCompileType}} \tau'}{\tau* \rightsquigarrow_{\text{InvCompileType}} (\tau'*, \text{int})}\ (\textsc{InvertCapability})$$

Back-translated arrays are hence represented as tuples, where the first part is an array containing all arrays of the level below and their lengths, and the second part contains the length of the array on the current level itself. The expression operations .1 and .2 are respectively the left and right projection of a pair and are present in source, target and separation logic expressions. Pointers of type $\tau*_0$ are back-translated to source types $\tau*$. The reason source-level pointers $\tau*$ are downcast to $\tau*_0$ and not to ints is that otherwise, no distinction could be made between integers and pointer addresses anymore at the target level, and this would result in problems during back-translation. It should be noted that adding the $\tau*_0$-type to the target language is no real concession; it could just be implemented by a 0-length capability on a capability machine like CHERI. Even though we did not allow any pointer arithmetic on regular pointers in the target language (this would make our fully abstract compilation slightly harder because we would need indexes for capabilities but would not result in any added power), we do have to allow pointer arithmetic on $\tau*_0$-type pointers, because the compilation translates pointer arithmetic on the source-level variables to pointer arithmetic on the corresponding target-level variables.

An important and useful property for types is the following:

$$\text{InvCompileType} \circ \text{CompileType} = id_{\text{f}}$$

.

7.2.2 Universal Contracts. A central part of the back-translation is the notion of universal contract, mentioned above. This section is used to define this notion.

When pointer-type function arguments are back-translated, they correspond to separation logic chunks of unknown length l in the back-translated function's separation logic precondition. For this reason, we introduced a logical list type, so that variables of this type can represent lists of possibly unknown length. Notice that these list variables cannot appear in boundary contracts, as these contracts allow only symbolic variables that have a program equivalent, and these variables cannot have program-level equivalents, as they are a pure type of list variable. These variables do not appear in boundary contracts in the back-translation, as they only appear in minimal contracts of back-translated context functions, which will not be imported or exported by components.

A specific type of disjunction has to be allowed in order to be able to work with back-translated linear capabilities in source language proofs. Back-translated linear capabilities either correspond to a heap chunk and a regular pointer in the target language *or* equal the back-translated null pointer and have no corresponding heap chunk. To represent these two possibilities, a separation logic expression of the form $a == (\text{null}_0, 0)$? *a_has_chunk* is used in the definition of universal contracts below. The disjunction expression is often used with a first disjunct of the form $id_{\text{prog}} == (\text{null}_0, 0)$. We define the shortened notation is_nullptr for this check as follows:

Definition (null pointer check)**.**

$$\text{is\_nullptr}(exp) = (exp == (\text{null}_0, 0))$$

Disjunction can be made to work together with the original disjunction-less separation logic syntax by including it in the Consequence rule. The Consequence rule can add or drop the is_nullptr(exp)$\vee$ condition depending on the need.

Having defined all the necessary concepts, we now define the concept of the universal contract for a single back-translated separation logic expression exp for target type τ_t. The function $\text{univ\_contr}_{\tau_t}(exp, [index], isOuter)$ returns a separation logic assertion representing the universal contract of source expression exp of type τ_t, where $[index]$ is a pre-existing index to be used in any logical list appearing in the universal contract and $isOuter$ denotes if a separation logic chunk name has already been used for an outer chunk or not.

The $index$ is initially [], and gets filled in with the proper variable names by traversing nested levels of resources. In the same vein, $isOuter$ is originally true, and only becomes false when the first chunk name is introduced. We define $\text{univ\_contr}_{\tau_t*}(exp) \equiv \text{univ\_contr}_{\tau_t*}(exp, [], \text{true})$ to simplify notation for the default universal contract. We also often just write $\text{univ\_contr}_{\tau}(exp)$ all the same when the context is clear and assume that the correct index $index$ and top-level naming (if necessary) is used. We also write $\text{univ\_contr}^{\overline{n}}_{\tau}(exp)$, where $\overline{n}$ is the tuple of names appearing in the universal contract.

Definition ($\text{univ\_contr}_{\tau_t}(\cdot)$)**.**

$$
\begin{aligned}
&\text{univ\_contr}_{\text{int}}(exp, \_, isOuter) = \text{true}\\
&\text{univ\_contr}_{\tau_s*_0}(exp, \_, isOuter) = \text{true}\\
&\text{univ\_contr}_{\tau_t*}(exp, [index], isOuter) =\\
&\quad !\,\text{is\_nullptr}(exp)\ ?\\
&\quad n : [exp.1 + i \mapsto_{\tau_s} l[index][i] *\\
&\quad \text{univ\_contr}_{\tau_t}(l[index][i], [index][i], \text{false}) \mid 0 \le i < \text{length}(l[index])]\\
&\quad * \text{length}(l[index]) == exp.2\\
&\quad \textit{given that } \tau_t \rightsquigarrow_{InvCompileType} \tau_s \textit{ and } l \textit{ fresh}\\
&\quad n \textit{ only present if } isOuter == \text{true}\\
&\text{univ\_contr}_{(\tau_1,\ldots,\tau_k)}(exp, [index], isOuter) =\\
&\quad \text{univ\_contr}_{\tau_1}(exp.1, [index], isOuter) *\\
&\quad \ldots * \text{univ\_contr}_{\tau_k}(exp.k, [index], isOuter)
\end{aligned}
$$

Notice that the defined universal contracts are not linear, whereas we did require all boundary function contracts to be linear in section 3. This is no problem, as the above universal contract notation will never appear in import or export boundary function contracts and hence never show up in stub guards, as all back-translated target functions will be non-boundary functions in the back-translation. These types of contracts will hence never be compiled into stubs. The same holds

true for the fact that the universal contracts contain list variables and quantification: these will never appear in stubs either and do not require program-level reified expressions.

The universal quantification used in the definition is a finite form of universal quantification, which is hence decidable and poses no computational problems for the verification tool. The definition can entirely be unrolled once the lengths are known.

Extending the above universal contract definition for a single variable to a set of variables result in the following general definition of the *universal contract* of a piece of executing code:

Definition (Universal Contract)**.**
Given that we want to back-translate target-level statement c, which mentions free target-level variables (ie. variables not declared in c) $\mathrm{FV}(c)$*. The universal precondition contract for c's back-translation is the following:* $\mathrm{univ\_pre} = *\{\mathrm{univ\_contr}_\tau(\gamma_{\mathrm{pre}}(id)) \mid id \in \mathrm{FV}(c) \wedge \mathrm{TypeOfVar}(id) = \tau\}$*. The universal postcondition contract has the same form, but now the set of variables also includes the variables declared in c. The set of all free and declared variables is denoted* $\mathrm{V}(c)$ *(variables only defined in one branch of any* If*-statement in c are not in* $\mathrm{V}(c)$*). The universal postcondition is: (note that no variables can be erased entirely once they're declared, so even nulled variables appear in the minimal contract)* $\mathrm{univ\_post} = *\{\mathrm{univ\_contr}_\tau(\gamma_{\mathrm{post}}(id)) \mid id \in \mathrm{V}(c) \wedge \mathrm{TypeOfVar}(id) = \tau\}$.

The previous discussion only captures the symbolic heap part of a separation logic triple. To define the environment, we also require the set of auxiliary variables $\mathrm{AUX}(c)$ *generated in the back-translated code* c_b*. These auxiliary variables are one-use, hence being reset to the* null*-value for their type after each back-translated block of code, never to be used again. Any auxiliary variables from previous pieces of code are framed off using the* Frame *rule. For the environment, we have* $\gamma_{\mathrm{pre}} = [\mathrm{FV}(c) : \overline{v_{\mathrm{univ}}}]$ *with* $\overline{v_{\mathrm{univ}}}$ *a set of symbolic variables for which* $\mathrm{UniqueId}(\overline{v_{\mathrm{univ}}})$ *holds for the precondition environment, and analogously* $\gamma_{\mathrm{post}} = [\mathrm{V}(c) : \overline{v'_{\mathrm{univ}}}][\mathrm{AUX}(c) : \_]$ *with* $\mathrm{UniqueId}(\overline{v'_{\mathrm{univ}}})$ *for the postcondition environment.*

The universal contract definition is used both to actually construct the universal contract of a back-translated target function f (where care has to be taken with return *statements, as will be discussed later) and to construct extended separation logic triples of code c to prove the back-translated function contracts. For code c for which* $c \rightsquigarrow_b c_{\mathrm{b}}$*, the proof triple is* $\{\mathrm{univ\_pre}\}_{\gamma_{\mathrm{pre}}}\ c_{\mathrm{b}}\ \{\mathrm{univ\_post}\}_{\gamma_{\mathrm{post}}}$*, with all four components as defined above. If all variables are well-scoped, the different universal contract pieces will automatically link together (possibly requiring extra applications of the* Frame *rule below) and form a sound source program proof of the enveloping function's universal contract.*

7.2.3 Back-compiling expressions. This subsection describes how expressions are translated from the source to the target language during compilation and how they are translated from the target to the source language during back-compilation.

For compilation, target expressions have the length function as an extra function compared to source expressions. Other than that, the basic structures of types match when compiling from source to target. Any *sexp* could hence be kept identical when compiled to a corresponding *texp*.

Back-compiling *texp* cannot be done using the identity function. Difficulties arise because $\tau*$ type-expressions produce pair type-expressions in the back-compilation, which fundamentally change the structure of the expression and hence require some measures when back-compiling expressions containing pointers. No pointer arithmetic is allowed on linear capabilities in the target language, which makes the back-translation somewhat easier. A second type of difficulty is caused by the fact that length-functions are not present in the source language and should be back-translated

somehow and neither are addr functions. For the mapping, we again have the identity for most cases, except for the two difficulties previously mentioned. We get the following non-identity rules because of the type and statement mismatch:

These first two rules below are the identity when null has a $\tau*_0$ type (ie. null_0), but not when null has a pointer type.

$$\frac{}{\text{null} \rightsquigarrow_{\text{b}} (\text{null}_0, 0)} \quad \text{(BackCompNull)}$$

$$\frac{}{\text{null}_0 \rightsquigarrow_{\text{b}} \text{null}_0} \quad \text{(BackCompNull)}$$

$$\frac{sexp \rightsquigarrow_{\text{b}} texp}{\text{addr}(sexp) \rightsquigarrow_{\text{b}} texp.1} \quad \text{(BackCompAddr)}$$

$$\frac{sexp \rightsquigarrow_{\text{b}} texp}{\text{length}(sexp) \rightsquigarrow_{\text{b}} texp.2} \quad \text{(BackCompLen)}$$

We introduce the notation $sexp_{\text{b}}$ to denote the back-translated version of *sexp*, ie. *texp* in $sexp \rightsquigarrow_{\text{b}} texp$.

7.2.4 Erasure in the source language. For back-translated source language expressions, we will need to erase source program variables to parallel the implicit effect of capability erasure in the target language (parallel behavior in order to prove equi-termination between the target code and its back-translation). Rigorously defining these source-level erasure rules once will make the back-translation easier and more coherent. We emulate the erasure effect the linearity of the capabilities in the target language causes, by explicitly creating assignments in the source language. The rules for this are very similar to the target-level erasure rules presented earlier, with assignments instead of stack alteration, and are presented below.

In order for the below rules to work, we need to be able to replace stack values in source-level variables, while keeping them well-typed. If the *sexp* we want to emulate erasure on contains an integer k, we do not know whether it is a length and part of a back-translated pointer pair, or it is a regular integer. Erasure should hence be performed based on the original target expression's type.

The judgment $sexp, \tau_{\text{s}} \rightsquigarrow_{\text{EraseIDProg}} sexp$ erases all linear components (corresponding to capabilities in the target) present in a given source expression based on its *target* type, whereas the judgment $sexp, \tau_{\text{s}} \rightsquigarrow_{\text{EmulateNulling}} stuckstm, sstm$ generates the actual assignments to perform the erasure in the source language. The variable *stuckstm* adds a guard(false) on faulty input, to make sure the back-translated code gets stuck before doing any assignments. On non-faulty input, *stuckstm* is just skip. The variable *sstm* contains the assignments themselves.

The way we define the $\rightsquigarrow_{\text{EmulateNulling}}$ judgment is very analogous to the $\rightsquigarrow_{\text{StoreLinCap}}$ judgment, using $\rightsquigarrow_{\text{EraseIDProg}}$ instead of $\rightsquigarrow_{\text{ValErase}}$, $\rightsquigarrow_{\text{GatherNull}}$ instead of $\rightsquigarrow_{\text{GatherLinCap}}$ and finally $\rightsquigarrow_{\text{TupleToPAsgn}}$ instead of $\rightsquigarrow_{\text{TupleToSAsgn}}$.

All non-linear types stay the same, whereas all linear types are reset to their default value.

$$\frac{}{exp, \text{int} \rightsquigarrow_{\text{EraseIDProg}} exp} \quad \text{(EraseRebuildInt)}$$

$$\frac{}{exp, \tau*_0 \rightsquigarrow_{\text{EraseIDProg}} exp} \quad \text{(EraseRebuildSrcPtr)}$$

$$\frac{\begin{array}{c} exp.1, \tau_1 \rightsquigarrow_{\text{EraseIDProg}} exp'_1 \\ \dots \\ exp.k, \tau_k \rightsquigarrow_{\text{EraseIDProg}} exp'_k \end{array}}{exp, (\tau_1, \dots, \tau_k) \rightsquigarrow_{\text{EraseIDProg}} (exp'_1, \dots, exp'_k)} \quad \text{(EraseRebuildPair)}$$

$$\frac{}{exp, \tau* \rightsquigarrow_{\text{EraseIDProg}} (\text{null}_0, 0)} \quad \text{(EraseRebuildPtr)}$$

Now, the rules for GatherNull are exactly the ones given for GatherLinCap before, with a few very minor differences (eg. no more stack frame as input). Only a few rules are notable (ie. do not output •). We discuss these below.

The first one is IdProgStore. The new corresponding rule looks as follows:

$$\frac{id_{\text{prog}}, \text{TypeOfVar}_{\text{t}}(id_{\text{prog}}) \rightsquigarrow_{\text{EraseIDProg}} erase}{id_{\text{prog}} \rightsquigarrow_{\text{GatherNull}} (id_{\text{prog}}, erase)} \quad \text{(IDProgEmulate)}$$

The second one is TupleStore.

$$\frac{\begin{array}{c} exp_1 \rightsquigarrow_{\text{GatherNull}} v_1 \\ \dots \\ exp_k \rightsquigarrow_{\text{GatherNull}} v_k \end{array}}{(exp_1, \dots, exp_k) \rightsquigarrow_{\text{GatherNull}} v_1 \dots v_k} \quad \text{(TupleEmulate)}$$

Finally, the 2 rules for projection.

$$\frac{exp = (exp_1, \dots, exp_i, \dots, exp_k) \qquad exp_i, s \rightsquigarrow_{\text{GatherNull}} v}{exp.i, s \rightsquigarrow_{\text{GatherNull}} v} \quad \text{(ProjectTupleEmulate)}$$

$$\frac{exp = id_{\text{prog}} \qquad id_{\text{prog}}, s \rightsquigarrow_{\text{GatherNull}} (id_{\text{prog}}, (erase_1, \dots, erase_{\text{k}}))}{exp.i, s \rightsquigarrow_{\text{GatherNull}} (id_{\text{prog}}.i, erase_{\text{i}})} \quad \text{(ProjectIDProgEmulate)}$$

For the auxiliary function $\rightsquigarrow_{\text{TupleToPAsgn}}$, we follow $\rightsquigarrow_{\text{TupleToPAsgn}}$ as follows (reusing $\rightsquigarrow_{\text{IDFilter}}$):

$$\frac{}{\epsilon \rightsquigarrow_{\text{TupleToPAsgn}} \epsilon} \quad \text{(TupleToPAsgnEps)}$$

$$\frac{\overline{v_{\text{rest}}} \rightsquigarrow_{\text{TupleToPAsgn}} asgn}{(id_{\text{prog}}, v)\ \overline{v_{\text{rest}}} \rightsquigarrow_{\text{TupleToPAsgn}} asgn; id_{\text{prog}} = v} \quad \text{(TupleToPAsgnID)}$$

$$\frac{\begin{array}{c} v' \rightsquigarrow_{\text{TupleToPAsgn}} asgn \\ \overline{v_{\text{rest}}} \rightsquigarrow_{\text{IDFilter}} (id_{\text{prog}}.i_2, v_2) \ \ldots \ (id_{\text{prog}}.i_k, v_k), v' \\ \text{TypeOfVar}(id_{\text{prog}}) = (\tau_1, \ldots, \tau_k) \qquad v = (v'_1, \ldots v'_k) \\ \forall i_j \in \{i_1, \ldots i_k\}.\, v'_{i_j} = v_j \\ \forall i \notin \{i_1, \ldots i_k\}.\, v'_i = id_{\text{prog}}.i \end{array}}{(id_{\text{prog}}.i_1, v_1)\ \overline{v_{\text{rest}}} \rightsquigarrow_{\text{TupleToPAsgn}} asgn; id_{\text{prog}} = v} \quad \text{(TupleToPAsgnIDIndex)}$$

We can reuse the predicate CheckStuck from before, but we cannot afford to create no rule for the case where ¬ CheckStuck. This is because we cannot have the creation of the back-translation get stuck; we want the *execution* to get stuck. The solution is to add guard(false) if there is overlap between variables, so that the source program gets stuck at the same time as the target program. We hence redefine CheckStuck, but now as a function, that outputs skip where the predicate version would have been true and guard(false) where it would have been false.

$$\frac{\begin{array}{c} exp, s \rightsquigarrow_{\text{GatherNull}} tuple \\ tuple, s \rightsquigarrow_{\text{TupleToPAsgn}} asgn \\ \text{CheckStuck}(tuple) = stuckstm \end{array}}{exp, s \rightsquigarrow_{\text{EmulateNulling}} stuckstm, asgn} \quad \text{(EmulateNulling)}$$

An important observation is that if a source-level pointer id_{prog} (or a pair containing pointers) is assigned an expression containing id_{prog} and we null source-level variables, we will erase id_{prog} itself (including the newly assigned value) given the current source-level erasure rules. This was no problem in the target, because erasure happens before assignment in the operational semantics, which is no option now, as we would have no value to assign anymore. We could fix this problem by disallowing -both in target and source- pointer program variables to be assigned expressions causing themselves to be erased, which is only a very minor restriction, as there is no pointer arithmetic in the target anyway, so this kind of assignment does not have any effect.

A more rigorous alternative solution is to check what variable name is being assigned to and to not null the variable corresponding to this variable name. For this reason, we could define a new judgment $sexp, \overline{id} \rightsquigarrow_{\text{EmulateNullingNotID}} stuckstm, sstm$ that works exactly like EmulateNulling, but does not generate assignments for any variables $\overline{id}$ in its version of the TupleEmulateStuck and TupleEmulateOK rules. The problem here, is that nulling would not correspond to how nulling in the target language works during a function call (even though the end result is the same), hence breaking the simulation relations we will try to build later on.

Our final solution is to use auxiliary, throw-away variables to store the expressions we want to null in and perform the nulling on the original expressions *before* the function call, handing the auxiliary variables to the function at call time. This ensures behavior parallel to the target language, without jeopardizing the integrity of the $\overline{id}$-values.

7.2.5 Converting between list and array resources. An important aspect of back-translating stubs and some target statements is the fact that back-translated code assumes that all resources are range-expressions, whereas some source code and statements such as Malloc in the source language do not, ie. they operate on array resources, not on range resources. A conversion between the two representations is hence often necessary, and defined here.

Array resource to range expression. We call the first procedure ArrayToRange($id^{\mathrm{p}}_{\mathrm{addr}}$, $id^{\mathrm{p}}_{\mathrm{len}}$) and it converts the array chunk corresponding to these parameters into a range chunk in the following way (where $\gamma(id^{\mathrm{p}}_{\mathrm{addr}}) = id_{\mathrm{addr}}, \gamma(id^{\mathrm{p}}_{\mathrm{len}}) = id_{\mathrm{len}}$):

$\{n : id_{\mathrm{addr}} \mapsto id_{\mathrm{cont}} * \mathrm{length}(id_{\mathrm{cont}}) == id_{\mathrm{len}}\}_\gamma$
if $id^{\mathrm{p}}_{\mathrm{len}} == 1$ then
  $\{\sim * id_{\mathrm{len}} == 1\}_\gamma$ $\mapsto l[i]$ is a shorthand for $\mapsto [l[i]]$
  $\{n : id_{\mathrm{addr}} + 0 \mapsto id_{\mathrm{cont}}[0] * \mathrm{length}(id_{\mathrm{cont}}) == id_{\mathrm{len}} * id_{\mathrm{len}} == 1\}$
  //@collect n
  $\{n' : [id_{\mathrm{addr}} + i \mapsto id_{\mathrm{cont}}[i] \mid 0 \leq i < id_{\mathrm{len}}]\}$
else

$\{\sim * id_{\text{len}} \mathrel{!}= 1\}_\gamma \qquad 0 < 1 < id_{\text{len}}$
//@split $n[1]$
$\{n_1 : id_{\text{addr}} + 0 \mapsto id_{\text{cont}}[0] * n_2 : id_{\text{addr}} + 1 \mapsto \text{take}(id_{\text{cont}}, 1, id_{\text{len}})\}$
//@collect n_1
$\{n_2 : id_{\text{addr}} + 1 \mapsto \text{take}(id_{\text{cont}}, 1, id_{\text{len}})$
$\quad * n' : [id_{\text{addr}} + i_s \mapsto id_{\text{cont}}[i_s] \mid 0 \leq i < 1]\}$
$Inv[i_s] = n : id_{\text{addr}} + i_s \mapsto \text{take}(id_{\text{cont}}, i_s, id_{\text{len}})$
$\quad * n' : [id_{\text{addr}} + i \mapsto id_{\text{cont}}[i] \mid 0 \leq i < i_s]$
$\gamma[i_s] = \gamma[j : i_s]$
$\{Inv[1]\}_\gamma$
foreach$(1 \leq j < id^{\text{p}}_{\text{len}} - 1)\{$
$\{1 \leq i_s < id_{\text{len}} - 1 * Inv[i_s]\}$
//@split $n[1]$
$\{n_1 : id_{\text{addr}} + i_s \mapsto id_{\text{cont}}[i_s]$
$\quad * n_2 : id_{\text{addr}} + i_s + 1 \mapsto \text{take}(id_{\text{cont}}, i_s + 1, id_{\text{len}})$
$\quad * n' : [id_{\text{addr}} + i \mapsto id_{\text{cont}}[i] \mid 0 \leq i < i_s]\}$
//@collect n_1
$\{n'_1 : [id_{\text{addr}} + i \mapsto id_{\text{cont}}[i] \mid i_s \leq i < i_s + 1] * \ldots\}$
//@join $n'\ n'_1$
$\{n_2 : id_{\text{addr}} + i_s + 1 \mapsto \text{take}(id_{\text{cont}}, i_s + 1, id_{\text{len}})$
$\quad * n'' : [id_{\text{addr}} + i \mapsto id_{\text{cont}}[i] \mid 0 \leq i < i_s + 1]\}$ (Conseq: rename)
$\{n : id_{\text{addr}} + i_s + 1 \mapsto \text{take}(id_{\text{cont}}, i_s + 1, id_{\text{len}})$
$\quad * n' : [id_{\text{addr}} + i \mapsto id_{\text{cont}}[i] \mid 0 \leq i < i_s + 1]\}$
$\{Inv[i_s + 1]\}$
$\}$
$\{Inv[id_{\text{len}} - 1]\}_{\gamma[id_{\text{len}}-1]}$
//@collect n
$\{n'_1 : [id_{\text{addr}} + i \mapsto id_{\text{cont}}[i] \mid id_{\text{len}} - 1 \leq i < id_{\text{len}}]$
$\quad * n' : [id_{\text{addr}} + i \mapsto id_{\text{cont}}[i] \mid 0 \leq i < id_{\text{len}} - 1]\}$
//@join $n'\ n'_1$
$\{n' : [id_{\text{addr}} + i \mapsto id_{\text{cont}}[i] \mid 0 \leq i < id_{\text{len}}]\}$
$\{n' : [id_{\text{addr}} + i \mapsto id_{\text{cont}}[i] \mid 0 \leq i < id_{\text{len}}]\}$

The variable j is an auxiliary variable and is never used again.

Range expression to array resource. A very analogous procedure RangeToArray($id^{\text{p}}_{\text{addr}}$, $id^{\text{p}}_{\text{len}}$) for the opposite direction can be written out as well. It converts the range resource corresponding to

these parameters into an array chunk (again, $\gamma(id^{\mathrm{p}}_{\mathrm{addr}}) = id_{\mathrm{addr}}, \gamma(id^{\mathrm{p}}_{\mathrm{len}}) = id_{\mathrm{len}}$). This procedure is implemented in the following way:

$\{n' : [id_{\mathrm{addr}} + i \mapsto id_{\mathrm{cont}}[i] \mid 0 \leq i < id_{\mathrm{len}}] * \mathrm{length}(id_{\mathrm{cont}}) == id_{\mathrm{len}}\}$
if $id^{\mathrm{p}}_{\mathrm{len}} == 1$ then
  //@flatten n'
else
  $\{\sim * id_{\mathrm{len}} != 1\}_{\gamma}$ $0 < 1 < id_{\mathrm{len}}$
  //@split $n[1]$
  $\{n_1 : [id_{\mathrm{addr}} + i_s \mapsto id_{\mathrm{cont}}[i_s] \mid 0 \leq i < 1]$
    $n_2 : [id_{\mathrm{addr}} + i_s \mapsto id_{\mathrm{cont}}[i_s] \mid 1 \leq i < id_{\mathrm{len}}]\}$
  //@flatten n_1 ·
  $Inv[i_s] = n : id_{\mathrm{addr}} + i_s \mapsto \mathrm{take}(id_{\mathrm{cont}}, 0, i_s)$
    $* n' : [id_{\mathrm{addr}} + i \mapsto id_{\mathrm{cont}}[i] \mid i_s \leq i < id_{\mathrm{len}}]$
  $\gamma[i_s] = \gamma[j : i_s]$
  $\{Inv[1]\}_{\gamma}$
  foreach$(1 \leq j < id^{\mathrm{p}}_{\mathrm{len}} - 1)\{$
    $\{1 \leq i_s < id_{\mathrm{len}} - 1 * Inv[i_s]\}$
    //@split $n'[1]$
    $\{n : id_{\mathrm{addr}} + i_s \mapsto \mathrm{take}(id_{\mathrm{cont}}, 0, i_s)$
      $* n'_1 : [id_{\mathrm{addr}} + i \mapsto id_{\mathrm{cont}}[i] \mid i_s \leq i < i_s + 1]$
      $* n'_2 : [id_{\mathrm{addr}} + i \mapsto id_{\mathrm{cont}}[i] \mid i_s + 1 \leq i < id_{\mathrm{len}}]\}$
    //@flatten n'_1 ·
    $\{n_1 : id_{\mathrm{addr}} + i \mapsto id_{\mathrm{cont}}[i_s] * \ldots\}$
    //@join n n_1
    $\{n : id_{\mathrm{addr}} + i_s \mapsto \mathrm{take}(id_{\mathrm{cont}}, 0, i_s + 1)$
      $* n' : [id_{\mathrm{addr}} + i \mapsto id_{\mathrm{cont}}[i] \mid i_s + 1 \leq i < id_{\mathrm{len}}]\}$
    $\{Inv[i_s + 1]\}$
  $\}$
  $\{Inv[id_{\mathrm{len}} - 1]\}_{\gamma[id_{\mathrm{len}}-1]}$
  //@flatten n ·
  //@join n n_1
  $\{n : id_{\mathrm{addr}} \mapsto id_{\mathrm{cont}}\}_{\gamma}$
$\{n : id_{\mathrm{addr}} \mapsto id_{\mathrm{cont}}\}_{\gamma}$

The variable j is an auxiliary variable and is never used again.

This procedure differs from the procedure in the previous section, only where //@collect n has been replaced by //@flatten n.

7.3 Back-translation rules

This section describes all the rules used in the back-translation $\rightsquigarrow_{\mathrm{b}}$ to Hoare triples.

The guard statements that appear in all of the following back-translation rules are created to make the universal contract at the start of the back-translated proof triple more concrete, so that the preconditions of the separation logic axiom for the back-translated statement are met. After the application of the concrete back-translated statement, the consequence rule is used to go back to the universal contract in the postcondition of the back-translated triple. For each back-translation rule that produces code c_{b}, it has to be proven that the universal contract triple from the previous section holds. Analogously, in the case of functions, components and programs, the existence of a proof $\vdash$ has to be shown.

In order to be able to prove equi-termination in the security proof later on, the back-translated code has to parallel the target-level erasure of linear capabilities. The correct erasure statements are generated by the EmulateNulling compilation rule in the source language (and the EraseIDProg rule if the erasure has to happen within an array).

We first define the back-translation rules while ignoring the back-translation of stubs and using universal contracts for the original source functions as well, and then in a second set of rules define the back-translation of stubs to convert to and from universal contracts.

Recall that, for every piece of code c we back-translate in the below rules, we have to prove the triple $\{\text{univ\_pre}\}_{\gamma_{\text{pre}}}\ c_{\mathrm{b}}\ \{\text{univ\_post}\}_{\gamma_{\text{post}}}$, as defined in the definition in section 7.2.2. The only exception to this pattern is made by the Frame rule below, because it forms the glue between different levels of back-translated code. For back-translated functions, components and entire programs, we have to prove that the back-translation forms a proper proof $\vdash$.

We do not repeat the environment γ if it does not alter. We also do not explicitly mention auxiliary variables in the environment once they have been used, as there is no universal contract for them anyway and they can map to any symbolic expression in the environment γ. We often write γ as a shorthand fork γ_{pre} when applying the environment to an expression, if the concrete environment is clear from context, to avoid needless clutter.

In the back-translation, the back-translated core statement will often be preceded by a host of guard() and variable declarations. This happens because the universal contract must be made more precise in order for the separation logic axiom of the back-translated core statement to be applicable. We will mention this explicitly a couple times, by eg. "Meeting Malloc precondition" in the below Malloc statement, but will not repeat this statement each time. The same goes for obvious applications of the Consequence rule.

7.3.1 *Basic statement rules.*

$$\frac{}{\begin{array}{l}\text{skip} \rightsquigarrow_{\text{b}} \\ \{\text{univ\_pre}\}_{\gamma_{\text{pre}}} \\ \text{skip} \\ \{\text{univ\_pre}\}_{\gamma_{\text{pre}}} \qquad \text{FV}(c) == \text{V}(c) == \emptyset \\ \{\text{univ\_post}\}_{\gamma_{\text{post}}}\end{array}} \quad (\textsc{Skip})$$

Note that the expression $exp == \text{NULL}(\tau)$ that implicitly appears in the below proof can be made into a valid univ_contr for any expressions exp of type τ by application of the Consequence rule, because the universal contract always allows for null values. This fact will henceforth be referred to as 'NULL(τ) univ'.

$$\frac{\tau \rightsquigarrow_{\text{InvCompileType}} \tau' \qquad n_{\text{aux}} \text{ fresh}}{\begin{array}{l} n = \text{malloc}(texp * \text{sizeof}(\tau)) \rightsquigarrow_{\text{b}} \\ \{\text{univ\_pre}\}_{\gamma_{\text{pre}}} \\ \text{guard}(texp_{\text{b}} > 0); \tau' * n_{\text{aux}}; \qquad \text{Meeting Malloc precondition} \\ n_{\text{aux}} = \text{malloc}(texp_{\text{b}} * \text{sizeof}(\tau')); \\ \{\text{univ\_pre} * n_{\text{c}} : id_{\text{log}} \mapsto \text{repeat}(texp_{\text{b},\gamma_{\text{pre}}}, \text{NULL}(\tau'))\}_{\gamma_{\text{pre}}[n_{\text{aux}}:id_{\text{log}}]} \\ n = (n_{\text{aux}}, texp_{\text{b}}) \\ \{\sim\}_{\gamma_{\text{pre}}[n:(id_{\text{log}}, texp_{\text{b},\gamma})]} \\ \{\text{univ\_pre} * n_{\text{c}} : id_{\text{log}} \mapsto \text{repeat}(texp_{\text{b},\gamma_{\text{pre}}}, \text{NULL}(\tau')) \\ \quad * id_{\text{fresh}} = (id_{\text{log}}, texp_{\text{b},\gamma_{\text{pre}}})\}_{\gamma_{\text{pre}}[n:id_{\text{fresh}}]} \\ \text{ArrayToRange}(id^{\text{p}}_{\text{addr}}, id^{\text{p}}_{\text{len}}) \text{ with } n == (id^{\text{p}}_{\text{addr}}, id^{\text{p}}_{\text{len}}) \\ \{\text{univ\_pre} * n'_{\text{c}} : [id_{\text{log}} \mapsto \text{NULL}(\tau') \mid 0 \le i < texp_{\text{b},\gamma_{\text{pre}}}] \\ \quad * id_{\text{fresh}} = (id_{\text{log}}, texp_{\text{b},\gamma_{\text{pre}}})\}_{\gamma_{\text{pre}}[n:id_{\text{fresh}}]} \\ \{\text{univ\_pre} * n'_{\text{c}} : [id_{\text{fresh}}.1 + i \mapsto l[i] \\ \quad * l[i] == \text{NULL}(\tau') \mid 0 \le i < id_{\text{fresh}}.2] \qquad (\textsc{Conseq}), \text{NULL}(\tau') \text{ univ} \\ \quad * \text{length}(l) == id_{\text{fresh}}.2\}_{\gamma_{\text{pre}}[n:id_{\text{fresh}}]} \qquad \text{FV}(c) == \text{V}(c) \\ \{\text{univ\_post}\}_{\gamma_{\text{post}}} \end{array}} \quad (\textsc{Malloc})$$

As was the case for compilation, the back-translation might produce variable declarations that are not present in the target code, breaking the restriction on If- and For-statements not containing declarations. Hoisting of variable declarations will hence be required again for these two types of statements. This never results in any problems with the proof of the back-translated code, as a VarDecl statement does not alter any existing universal contracts, and just expands γ with the declared variable identifier. Expediting variable declaration can never break a sound proof.

The fact that $\{P\}_{\gamma}$ *sstm* $\{Q\}_{\gamma'}$ in the below code, implies that $\{Q\}_{\gamma'}$ *nondecl* $\{Q\}_{\gamma'}$; since 'NULL(τ) univ' and Q has the form of a universal contract. This separation logic state $\{Q\}_{\gamma'}$ is the one used to proof the For statement using the For separation logic axiom.

$$\frac{\begin{array}{c} tstm \rightsquigarrow_{\mathrm{b}} \{P\}_{\gamma}\ sstm\ \{Q\}_{\gamma'} \\ sstm \rightsquigarrow_{\mathrm{ExtractDecl}} decl, nondecl \end{array}}{\begin{array}{l} \mathrm{foreach}(texp \le i < texp')\{tstm\} \rightsquigarrow_{\mathrm{b}} \\ \{\mathrm{univ\_pre}\}_{\gamma_{\mathrm{pre}}} \\ decl; \qquad \mathrm{NULL}(\tau)\ \mathrm{univ} \\ \{Q\}_{\gamma' \setminus [i:\_]} \\ \mathrm{foreach}(texp_{\mathrm{b}} \le i < texp'_{\mathrm{b}})\{nondecl\} \qquad \mathrm{V}(c) == \mathrm{FV}(c) \cup i \\ \{\mathrm{univ\_post}\}_{\gamma_{\mathrm{post}}} \end{array}} \quad (\textsc{For})$$

We first introduce an auxiliary property we will be needing below. To prove the axiom for variable assignment and some other axioms below, we need the following property of universal contracts:

Theorem 5 (Compositionality of universal contracts)**.**
Assume $\{\mathrm{univ\_pre}\}_{\gamma_{\mathrm{pre}}}$ *holds, ie. the universal contract holds for any variable* id_{prog} *in* $\mathrm{FV}(c)$*. Then take a back-translated expression* $texp_{\mathrm{b}}$ *where the type of the original texp is* τ *and that only uses variables from* $\mathrm{FV}(c)$*. The separation logic statement* $\mathrm{univ\_contr}_{\tau}(texp_{\mathrm{b},\gamma_{\mathrm{pre}}})$ *can now be derived to hold, IF texp does not store the same target pointer twice (ie. it would have stuckstm* $=$ skip *in the judgment* $texp_{\mathrm{b}} \rightsquigarrow_{EmulateNulling} stuckstm, asgn$*), as is eg. the case in* (a, a) *if a is of type* $\mathrm{int}*$ *or in* $((a, b).1, a)$ *with a of the same type. On the other hand, eg.* $a == a$ *and* $(a, a, b).3$ *are fine.*

Proof.

Simple inductive case-based analysis on the target-type τ.

- If τ is int or $\tau'_{\mathrm{s}}*_0$, then the universal contract is true, which automatically holds.
- If τ is $\tau'_{\mathrm{t}}*$, then $texp_{\mathrm{b}}$ is either a back-translated pointer variable id_{prog} (given that universal contract holds), a back-translated null-pointer $(\mathrm{null}_0, 0)$ (null-pointer case in universal contract definition for pointers) or a projection of a tuple $texp'_{\mathrm{b}}.i$ (take the ith case in the universal contract of the tuple $texp'_{\mathrm{b}}$, which we know holds by induction).
- If τ is $(\tau_1, \ldots, \tau_k)$ then $texp_{\mathrm{b}}$ is either a back-translated tuple variable id_{prog} (given that universal contract holds), a projection of a nested tuple $texp'_{\mathrm{b}}.i$ (same as before) or a tuple $(texp'_{\mathrm{b},1}, \ldots, texp'_{\mathrm{b},k})$ (the universal contract holds for each $texp'_{\mathrm{b},i}$ by induction, and we combine these by using the tuple case in the definition for universal contracts). This bullet is the reason why *texp* cannot return the same target pointer twice in the final value, as we would have to use the same non-duplicable universal contract multiple times here.

□

This compositionality property will be useful at any point where an arbitrary expression is associated to a symbolic variable, ie. also when proving the axiom for the Return statement below.

Analogous to $exp == \mathrm{NULL}(\tau)$ above, again note that the assignment created by $n \rightsquigarrow_{\mathrm{EmulateNulling}} stuck, c$ below enforces the universal contract to hold for the source variable n, after executing the assignment command c. This fact will henceforth be referred to as 'EmulateNulling univ' and made more concrete in the theorem 6. Intuitively, the reason this works is that EmulateNulling sets all back-translated pointers that correspond to target-level capabilities and that are present in the source-value n to $(\mathrm{null}_0, 0)$, the null-pointer value required by their universal contract. Note that

we place the *stuck* statements at the start of the back-translated code, allowing the guard(false) statement to prove any post-condition in the case of faulty target code. We will always show the proof for the case where *stuck* ≡ skip, leaving this other, trivial, case implicit.

Theorem 6.
[EmulateNulling Univ] Given the contract $\{\text{univ\_pre}\}_{\gamma_{\text{pre}}}$*, containing contracts for, among others, all variables* $\overline{x}$ *present in an expression* $texp_{\text{b}}$ *(in the trivial variable erasure case, we have* $\overline{x} = texp_{\text{b}} = id_{\text{prog}}$*). By compositionality, we can now derive* $\text{univ\_contr}(texp_{\text{b},\gamma_{\text{pre}}})$*, hereby reducing* $\text{univ\_contr}_{\text{pre}}$ *to* $\text{univ\_contr}'_{\text{pre}}$ *by consuming non-duplicable resources corresponding to back-translated pointers. The property EmulateNulling Univ then states that applying the assignment asgn created by* $texp_{\text{b}} \rightsquigarrow_{EmulateNulling} stuckstm, asgn$ *(under the assumption that* $stuckstm = \text{skip}$*) will restore the universal contract* $\text{univ\_contr}'_{\text{pre}}$ *to* $\text{univ\_contr}_{\text{pre}}$*. It has been intuitively stated above why this works.*

$$\frac{n \rightsquigarrow_{\text{EmulateNulling}} _, c \qquad \text{TypeOfVar}_{\text{t}}(n) = \tau*}{\begin{array}{l}
\{n', n''\} = \text{split}(n, texp) \rightsquigarrow_{\text{b}} \\
\{\text{univ_pre}\}_{\gamma_{\text{pre}}} \\
\{\text{univ_pre}' * \text{univ_contr}_{\tau*}(n_{\text{s}})\} \qquad \gamma_{\text{pre}}(n) == n_{\text{s}} \\
\text{guard}(!\,\text{is_nullptr}(n)); \text{guard}(0 < texp_{\text{b}} < n.2); \\
\{\text{univ_pre}' * n_{\text{c}} : [n_{\text{s}}.1 + i \mapsto l[i] * \text{univ_contr}_{\tau}(l[i]) \mid \\
\quad 0 \le i < \text{length}(l)] * \text{length}(l) == n_{\text{s}}.2 * 0 < texp_{\text{b},\gamma} < n_{\text{s}}.2\} \\
//@\text{split } n_{\text{c}}[texp_{\text{b}}]; \\
\{\text{univ_pre}' * \text{length}(l) == n_{\text{s}}.2 * 0 < texp_{\text{b},\gamma} < n_{\text{s}}.2 \\
\quad * n'_{\text{c}} : [n_{\text{s}}.1 + i \mapsto l[i] * \text{univ_contr}_{\tau}(l[i]) \mid 0 \le i < texp_{\text{b},\gamma}] \\
\quad * n''_{\text{c}} : [n_{\text{s}}.1 + i \mapsto l[i] * \text{univ_contr}_{\tau}(l[i]) \mid texp_{\text{b},\gamma} \le i < \text{length}(l)]\} \\
n' = (n.1, texp_{\text{b}}); n'' = (n.1 + texp_{\text{b}}, n.2 - texp_{\text{b}}); \\
\quad \text{shift } n''_{\text{c}} \text{ over } texp_{\text{b},\gamma}, l' = \text{take}(l, 0, texp_{\text{b},\gamma}), l'' = \text{take}(l, texp_{\text{b},\gamma}, n_{\text{s}}.2) \\
\{\text{univ_pre}' * \text{length}(l') == texp_{\text{b},\gamma} * \text{length}(l'') == n_{\text{s}}.2 - texp_{\text{b},\gamma} \\
\quad * n'_{\text{c}} : [n_{\text{s}}.1 + i \mapsto l'[i] * \text{univ_contr}_{\tau}(l'[i]) \mid 0 \le i < \text{length}(l')] \\
\quad * n''_{\text{c}} : [(n_{\text{s}}.1 + texp_{\text{b},\gamma}) + i \mapsto l''[i] * \text{univ_contr}_{\tau}(l''[i]) \mid \\
\quad\quad 0 \le i < \text{length}(l'')]\}_{\gamma_{\text{pre}}[n':(n_{\text{s}}.1, texp_{\text{b},\gamma})][n'':(n_{\text{s}}.1+texp_{\text{b},\gamma}, n_{\text{s}}.2-texp_{\text{b},\gamma})]} \\
\{\text{univ_pre}' * \text{univ_contr}_{\tau*}(\gamma(n')) * \text{univ_contr}_{\tau*}(\gamma(n''))\}_{\gamma} \\
c \qquad \text{EmulateNulling univ, V}(c) == \text{FV}(c) \\
\{\text{univ_post}\}_{\gamma_{\text{post}}}
\end{array}} \quad (\textsc{Split})$$

Note that in the above proof univ_pre′ must have also contained the original universal contracts for n' and n'', because these variables were free in c. These original contracts become obsolete and are leaked in the last step, when the contract univ_post is formed.

$$\frac{n' \rightsquigarrow_{\text{EmulateNulling}\ \_}, c' \qquad n'' \rightsquigarrow_{\text{EmulateNulling}\ \_}, c'' \qquad \text{TypeOfVar}_t(n') = \text{TypeOfVar}_t(n'') = \tau*}{\begin{array}{l} n = \text{join}(n', n'') \rightsquigarrow_b \\ \{\text{univ\_pre}\}_{\gamma_{\text{pre}}} \\ \text{guard}(!\,\text{is\_nullptr}(n')); \text{guard}(!\,\text{is\_nullptr}(n'')); \qquad \gamma_{\text{pre}}(n') == n'_s, \gamma_{\text{pre}}(n'') == n''_s \\ \text{guard}(n''.1 == n'.1 + n'.2); \\ \{\text{univ\_pre}' * \text{length}(l') == n'_s.2 * \text{length}(l'') == n''_s.2 \\ \quad * n'_c : [n'_s.1 + i \mapsto l'[i] * \text{univ\_contr}_\tau(l'[i]) \mid 0 \le i < \text{length}(l')] \\ \quad * n''_c : [n''_s.1 + i \mapsto l''[i] * \text{univ\_contr}_\tau(l''[i]) \mid 0 \le i < \text{length}(l'')] \\ \quad * n''_s.1 == n'_s.1 + n'_s.2\} \\ \qquad \text{shift } n''_c \text{ over length}(l), l = \text{append}(l', l''), \text{use } n''_s.1 == n'_s.1 + n'_s.2 \\ \{\text{univ\_pre}' * \text{length}(l) == n'_s.2 + n''_s.2 \\ \quad * n'_c : [n'_s.1 + i \mapsto l[i] * \text{univ\_contr}_\tau(l[i]) \mid 0 \le i < \text{length}(l')] \\ \quad * n''_c : [n'_s.1 + i \mapsto l[i] * \text{univ\_contr}_\tau(l[i]) \mid \text{length}(l') \le i < \text{length}(l)]\} \\ //@\text{join } n'_c\ n''_c; \\ n = (n'.1, n'.2 + n''.2); \\ \{\text{univ\_pre}' * \text{length}(l) == n'_s.2 + n''_s.2 * n_c : [n'_s.1 + i \mapsto l[i] \\ \quad * \text{univ\_contr}_\tau(l[i]) \mid 0 \le i < \text{length}(l')]\}_{\gamma_{\text{pre}}[n:(n'_s.1, n'_s.2+n''_s.2)]} \\ \{\text{univ\_pre}' * \text{univ\_contr}_{\tau*}(\gamma(n))\}_\gamma \\ c'; c'' \qquad \text{EmulateNulling univ, V}(c) == \text{FV}(c) \\ \{\text{univ\_post}\}_{\gamma_{\text{post}}} \end{array}} \quad (\textsc{Join})$$

The Frame rule described below can be used to make the 2 back-translated statements in the below rule match up.

$$\frac{p_1 \rightsquigarrow_b \{P\}_\gamma\ sstm_1\ \{Q\}_{\gamma'} \qquad p_2 \rightsquigarrow_b \{Q\}_{\gamma'}\ sstm_2\ \{R\}_{\gamma''}}{p_1; p_2 \rightsquigarrow_b \{P\}_\gamma\ sstm_1; sstm_2\ \{R\}_{\gamma''}} \quad (\textsc{Seq})$$

The If separation logic axiom requires an extra argument *sexp* or !*sexp* in the symbolic heap, but this can be added in using the Conseq separation logic axiom while constructing the below proof.

The back-translations of p_1 and p_2 in the below rule might contain different variable declarations, and might hence prove different universal contracts in their postconditions. As long as the environments correspond on the variables they have in common, this is no problem, as we can eg extend the proof of $sstm1$ by prepending it with the declarations $decl \setminus decl_1$, which is effectively what happens during hoisting. The reason this works is again because 'NULL(τ) univ'.

$$\frac{\begin{array}{c} p_1 \rightsquigarrow_{\mathrm{b}} \{P\}_\gamma \ sstm_1 \ \{Q_1\}_{\gamma_1'} \\ p_2 \rightsquigarrow_{\mathrm{b}} \{P\}_\gamma \ sstm_2 \ \{Q_2\}_{\gamma_2'} \\ D = \mathrm{dom}(\gamma_1') \cap \mathrm{dom}(\gamma_2') \qquad \gamma_1'(D) == \gamma_2'(D) \\ sstm_1 \rightsquigarrow_{\mathrm{ExtractDecl}} decl_1, nondecl_1 \\ sstm_2 \rightsquigarrow_{\mathrm{ExtractDecl}} decl_2, nondecl_2 \\ decl = decl_1 \cup decl_2 \end{array}}{\begin{array}{c} \text{if } sexp \text{ then } p_1 \text{ else } p_2 \\ \rightsquigarrow_{\mathrm{b}} \{P\}_\gamma \ decl; \text{if } sexp_{\mathrm{b}} \text{ then } nondecl_1 \text{ else } nondecl_2 \ \{Q_1 \cup Q_2\}_{\gamma_1' \cup \gamma_2'} \end{array}} \quad \text{(If)}$$

$$\frac{\begin{array}{c} \tau \rightsquigarrow_{\mathrm{InvCompileType}} \tau' \\ \gamma_{\mathrm{post}} = \gamma_{\mathrm{pre}}[id_{\mathrm{prog}} : id_{\mathrm{f}}] \qquad id_{\mathrm{f}} \text{ fresh} \\ \text{univ\_pre} * id_{\mathrm{f}} == \mathrm{NULL}(\tau') \Rightarrow \text{univ\_post because} \\ \mathrm{NULL}(\tau') \text{ univ} \end{array}}{\tau \ id_{\mathrm{prog}} \rightsquigarrow_{\mathrm{b}} \{\text{univ\_pre}\}_{\gamma_{\mathrm{pre}}} \ \tau' \ id_{\mathrm{prog}} \ \{\text{univ\_post}\}_{\gamma_{\mathrm{post}}}} \quad \text{(VarDecl)}$$

Note that the below two lines of code do indeed restore the universal contract, for the following reason. Deriving univ_contr($texp_{\mathrm{b},\gamma_{\mathrm{pre}}}$) requires 'using up' the universal contracts of all the logical variables appearing in $texp_{\mathrm{b},\gamma_{\mathrm{pre}}}$. The only non-duplicable part of a universal contract is the range expression for τ_{t}-typed universal contracts. The assignments generated by EmulateNulling replace exactly those non-duplicable parts that have been consumed to construct univ_contr($texp_{\mathrm{b},\gamma_{\mathrm{pre}}}$) by null pointer expressions, allowing the reconstruction of the consumed universal contracts.

In the case where $texp_{\mathrm{b},\gamma_{\mathrm{pre}}}$ contains the same τ_{t}-typed logical variable twice, we will not be able to construct a universal contract in the way outlined here. This forms no problem, however, as the rules for EmulateNulling add guard(false) to the code in this faulty case, which allows us to prove any contract using Conseq.

As outlined earlier, we use an auxiliary variable id_{aux} to emulate nulling properly in the case of recursive used of the variable id_{prog}.

$$\frac{\begin{array}{c} texp_{\mathrm{b}}, id \rightsquigarrow_{\mathrm{EmulateNulling}} stuck, c \\ \mathrm{TypeOfVar}(id_{\mathrm{prog}}) = \tau \qquad \tau \rightsquigarrow_{\mathrm{InvCompileType}} \tau' \end{array}}{\begin{array}{l} id_{\mathrm{prog}} = texp \rightsquigarrow_{\mathrm{b}} \\ \{\text{univ\_pre}\}_{\gamma_{\mathrm{pre}}} \\ stuck; \\ \tau'_{\mathrm{arg}} \ id_{\mathrm{aux}}; id_{\mathrm{aux}} = texp_{\mathrm{b}}; \\ c \qquad \text{EmulateNulling univ, V}(c) == \text{FV}(c) \\ id_{\mathrm{prog}} = id_{\mathrm{aux}}; \qquad \text{Compositionality: univ\_contr}(texp_{\mathrm{b},\gamma_{\mathrm{pre}}}) \text{ holds} \\ \{\text{univ\_post}\}_{\gamma_{\mathrm{post}}} \end{array}} \quad \text{(VarAsgn)}$$

We cannot use our previously written procedures ArrayToRange and RangeToArray for the next rule, as they do not work if nested range expressions are in play, which could be the case here. We hence split off the size-1 range we need, and flatten this piece individually.

When assigning the expression $texp_{2,\mathrm{b}}$, we have to use resources from univ_pre′ to be able to apply Compositionality, turning it into univ_pre′′ in the process. This is no problem, as the code c restores

the universal contracts of all variables that had their universal contracts consumed by applying Compositionality in the first place.

Given that:

$$texp_{2,\mathrm{b}} \rightsquigarrow_{\text{EmulateNulling}} stuck, c$$
$$\tau \rightsquigarrow_{\text{InvCompileType}} \tau'$$

The ArrayMut rule produces the following code:

$n[texp_1] = texp_2 \rightsquigarrow_{\mathrm{b}}$
univ_pre
stuck;
$\{\text{univ\_pre}' * \text{univ\_contr}_{\tau *}(n_{\mathrm{s}})\}$ $\gamma_{\mathrm{pre}}(n) == n_{\mathrm{s}}$
$\text{guard}(!\,\text{is\_nullptr}(n)); \text{guard}(0 \leq texp_{1,\mathrm{b}} < n.2);$
$\{\text{univ\_pre}' * n_{\mathrm{c}} : [n_{\mathrm{s}}.1 + i \mapsto l[i] * \text{univ\_contr}_{\tau}(l[i]) \mid$
$0 \leq i < \text{length}(l)] * \text{length}(l) == n_{\mathrm{s}}.2 * 0 \leq texp_{1,\mathrm{b},\gamma} < n_{\mathrm{s}}.2\}$
if $texp_{1,\mathrm{b}} == 0$
then (...) Easier version of what follows
else

$//@\text{split } n_c[texp_{1,b}];$
if $texp_{1,b} == n.2 - 1$
then (...) Easier version of what follows
else
$\{\text{univ\_pre}' * \text{length}(l) == n_s.2 * 0 < texp_{1,b,\gamma} < n_s.2 - 1 *$
$n_c' : [n_s.1 + i \mapsto l[i] * \text{univ\_contr}_\tau(l[i]) \mid 0 \le i < texp_{1,b,\gamma}] *$
$n_c'' : [n_s.1 + i \mapsto l[i] * \text{univ\_contr}_\tau(l[i]) \mid texp_{1,b,\gamma} \le i < \text{length}(l)]\}$
$//@\text{split } n_c''[1];$ Conseq: shift n_c'' over $texp_{1,b}$ and back after split
$\{n_{c,1} : [n_s.1 + i \mapsto l[i] * \text{univ\_contr}_\tau(l[i]) \mid texp_{1,b,\gamma} \le i < texp_{1,b,\gamma} + 1]$
$n_{c,2} : [n_s.1 + i \mapsto l[i] * \text{univ\_contr}_\tau(l[i]) \mid texp_{1,b,\gamma} + 1 \le i < \text{length}(l)] * ...\}$
$//@\text{flatten } n_{c,1};$
$\text{univ\_contr}_\tau(l[texp_{1,b,\gamma}])$'s range chunks get named too, if applicable, with names $\overline{n_u}$
$\{n_{\text{flat}} : n_s.1 + texp_{1,b,\gamma} \mapsto l[texp_{1,b,\gamma}] * \text{univ\_contr}_\tau^{\overline{n_u}}(l[texp_{1,b,\gamma}]) * ...\}$
$\tau' * n_{\text{aux}}; n_{\text{aux}} = n.1 + texp_{1,b};$
$\{\sim\}_{\gamma_{\text{pre}}[n_{\text{aux}}:n_s.1+texp_{1,b,\gamma}]}$
$n_{\text{aux}}[0] = texp_{2,b};$ $l' = \text{update}(l, texp_{1,b,\gamma}, texp_{2,b,\gamma})$, Compositionality
$//@\text{collect } n_{\text{flat}}, \overline{n_u};$ $\overline{n_u}$ only needed if applicable
$\{\text{univ\_pre}'' * \text{length}(l') == n_s.2 *$
$n_{c,1} : [n_s.1 + i \mapsto l'[i] * \text{univ\_contr}_\tau(l'[i]) \mid texp_{1,b,\gamma} \le i < texp_{1,b,\gamma} + 1]$
$n_{c,2} : [n_s.1 + i \mapsto l'[i] * \text{univ\_contr}_\tau(l'[i]) \mid texp_{1,b,\gamma} + 1 \le i < \text{length}(l')] *$
$n_c' : [n_s.1 + i \mapsto l'[i] * \text{univ\_contr}_\tau(l'[i]) \mid 0 \le i < texp_{1,b,\gamma}]\}$
$//@\text{join } n_{c,1}\ n_{c,2};$
$//@\text{join } n_c'\ n_c'';$
$\{\text{univ\_pre}'' * \text{length}(l') == n_s.2 *$
$n_c : [n_s.1 + i \mapsto l'[i] * \text{univ\_contr}_\tau(l'[i]) \mid 0 \le i < \text{length}(l')]\}$
c EmulateNulling univ, V(c) == FV(c)

The back-translation for array lookup is almost identical to the one for array mutation, with as only difference that a few lines at the core of the back-translation differ. The rest is boiler-plate, and the proof is identical to the proof above. Names are kept the same as in the previous proof, to allow for reuse and easier reading.

Again, some explaining with regards to the use of Compositionality in this proof is in order. The reason the three terms on the line marked with (1) recombine to univ_post on the final line is the following: the assignment $t_{\text{aux}}[0] = v$, created by EraseIdProg (which creates values upholding universal contracts), restores the universal contract of any variables that gave up non-duplicable resources during the creation of $\text{univ\_contr}_{\tau*}(texp_{1,b,\gamma})$ through compositionality. Only the resources from $\text{univ\_contr}_{\tau*}(texp_{1,b,\gamma})$ at index $texp_{2,b,\gamma}$ (ie. index 0 in t_{aux}) have been consumed

in the construction of $\text{univ\_contr}_\tau(id_{\text{fresh}})$ and need to be replaced by a universal value created through ErasedIdProg.

Given that:

$$\begin{array}{l}\text{TypeOfVar}^{\text{t}}(id_{\text{prog}}) = \tau \\ \tau \rightsquigarrow_{\text{InvCompileType}} \tau' \\ texp_{1,\text{b}}[texp_{2,\text{b}}], \tau \rightsquigarrow_{\text{EraseIDProg}} v\end{array}$$

The ArrayLkup rule produces the following code:

$id_{\text{prog}} = texp_1[texp_2] \rightsquigarrow_{\text{b}}$
$\{\text{univ\_pre}' * \text{univ\_contr}_{\tau *}(texp_{1,\text{b},\gamma})\}$ by Compositionality
$\text{guard}(!\,\text{is\_nullptr}(texp_{1,\text{b}})); \text{guard}(0 \leq texp_{2,\text{b}} < texp_{1,\text{b}}.2);$
$\{\text{univ\_pre}' * n_{\text{c}} : [n_{\text{s}}.1 + i \mapsto l[i] * \text{univ\_contr}_\tau(l[i]) \mid$ Abbreviated $texp_{1,\text{b},\gamma}$ to n_{s}
$0 \leq i < \text{length}(l)] * \text{length}(l) == n_{\text{s}}.2 * 0 \leq texp_{2,\text{b},\gamma} < n_{\text{s}}.2\}$
if $texp_{2,\text{b}} == 0$
then (...) Easier version of what follows
else

//@split $n_c[texp_{2,b}]$;
if $texp_{2,b} == texp_{1,b}.2 - 1$
then (...) Easier version of what follows
else
//@split $n_c''[1]$;
//@flatten $n_{c,1}$;
$\{n_{flat} : n_s.1 + texp_{2,b,\gamma} \mapsto l[texp_{2,b,\gamma}] * \text{univ\_contr}_\tau^{\overline{n_u}}(l[texp_{2,b,\gamma}]) * \ldots\}$
$\tau' * t_{aux}; t_{aux} = texp_{1,b}.1 + texp_{2,b}$
$\{\sim\}_{\gamma_{pre}[t_{aux}:n_s.1+texp_{2,b,\gamma}]}$
$id_{prog} = t_{aux}[0]$;
$\{\sim\}_{\gamma_{pre}[t_{aux}:n_s.1+texp_{2,b,\gamma}][id_{prog}:l[texp_{2,b,\gamma}]]}$
$t_{aux}[0] = v$ $l' = \text{update}(l, texp_{2,b,\gamma}, v_\gamma)$, EraseIDProg univ
$\{n_{flat} : n_s.1 + texp_{2,b,\gamma} \mapsto l'[texp_{2,b,\gamma}] * \text{univ\_contr}_\tau^{\overline{n'_u}}(l'[texp_{2,b,\gamma}]) *$
$\text{univ\_contr}_\tau^{\overline{n_u}}(l[texp_{2,b,\gamma}]) * \ldots\}$
//@collect $n_{flat}, \overline{n_u}$; $\overline{n'_u}$ only needed if applicable
//@join $n_{c,1}\ n_{c,2}$;
//@join $n'_c\ n''_c$;
$\{\text{univ\_pre}' * \text{length}(l') == n_s.2 *$
$n_c : [n_s.1 + i \mapsto l'[i] * \text{univ\_contr}_\tau(l'[i]) \mid 0 \le i < \text{length}(l')]$
$* \text{univ\_contr}_\tau(id_{fresh})\}_{\gamma_{pre}[id_{prog}:id_{fresh}]}$
$\{\text{univ\_pre}' * \text{univ\_contr}_{\tau*}(texp_{1,b,\gamma})$
$* \text{univ\_contr}_\tau(id_{fresh})\}_{\gamma_{pre}[id_{prog}:id_{fresh}]}$ $\text{FV}(c) == \text{V}(c)$, (1)

$$\frac{\{\text{univ\_pre}\}_{\gamma_{pre}} = \{\text{univ\_post}\}_{\gamma_{post}}}{\text{guard}(texp) \rightsquigarrow_b \{\text{univ\_pre}\}_{\gamma_{pre}}\ \text{guard}(texp_b)\ \{\text{univ\_post}\}_{\gamma_{post}}} \quad \text{(Guard)}$$

7.3.2 Function application back-translation rule. To show that the universal contract holds for the below FApp rule, we use the original separation logic FApp rule and the fact that the contract of this function f_{bt} is described in the FDecl rule below.

We using the *bt* subscript here for function names, so we do not need any substitutions afterwards when back-translating stubs: it can happen seamlessly because of the newly created names.

In the below rule, we consider EmulateNulling to be applied to the entire tuple $\overline{texp_b}$ simultaneously, so that even if different arguments use the same back-translated linear capability, execution still gets stuck because a guard(false) is inserted. This mirrors what happens in the target-level operational semantics for the FApp rule.

The below rule uses auxiliary variables id_{aux}, so that the nulling of linear capabilities can happen before the call to f_{bt}. This is necessary, because the linear capability erasure in the target language also happens while the function f is turned into a hole $\bullet$ by the operational semantics rule FApp.

$$\frac{\begin{array}{c}\overline{texp_{\text{b}}},\ \overline{id} \rightsquigarrow_{\text{EmulateNulling}} stuck, \overline{c} \\ \Sigma(f) = \{\overline{\tau_{\text{ret}}}, \overline{\tau_{\text{arg}}\ id_{\text{arg}}}\} \qquad \overline{\tau_{\text{arg}}} \rightsquigarrow_{\text{CompileType}} \overline{\tau'_{\text{arg}}}\end{array}}{\begin{array}{l}\{\overline{id}\} = f(\overline{texp}) \rightsquigarrow_{\text{b}} \\ \{\text{univ\_pre}\}_{\gamma_{\text{pre}}} \\ stuck; \\ \{\text{univ\_pre}' * \text{univ\_contr}_{\overline{\tau_{\text{arg}}}}(\overline{texp_{\text{b},\gamma}})\}_{\gamma_{\text{pre}}} \qquad \text{Compositionality} \\ \overline{\tau'_{\text{arg}}\ id_{\text{aux}}}; \overline{id_{\text{aux}}} = \overline{texp_{\text{b}}} \\ \overline{c} \qquad \text{EmulateNulling univ : univ\_pre restored} \\ \{\text{univ\_pre} * \text{univ\_contr}_{\overline{\tau_{\text{arg}}}}(\overline{texp_{\text{b},\gamma}})\}_{\gamma_{\text{pre}}[\overline{id_{\text{aux}}}:\overline{texp_{\text{b},\gamma}}]} \\ \{\overline{id}\} = f_{\text{bt}}(\overline{id_{\text{aux}}}); \\ \{\text{univ\_pre} * \text{univ\_contr}_{\overline{\tau_{\text{ret}}}}(\overline{id_{\text{fresh}}})\}_{\gamma_{\text{pre}}[\overline{id}:\overline{id_{\text{fresh}}}]} \\ \{\text{univ\_post}\}_{\gamma_{\text{post}}}\end{array}} \quad \text{(FApp)}$$

As in the compilation case, this rule includes back-translation of target-holes $\bullet^{\overline{id_{\text{arg}}}=\overline{texp}}$ to source-level holes $\bullet^{\overline{id_{\text{arg}}}=\overline{sexp}}_{POST}$. This is the reason why holes in the target language need to record their arguments as well, so that we can just set $\overline{sexp} = \overline{texp_{\text{b}}}$. The variable $POST$ then just follows straightforwardly from the generation of the universal contract for the function with declaration $\text{TypeOfVar}(\overline{id}) \bullet (\_)$ using the FDecl rule defined below, and equating $POST$ to the postcondition.

To make the above application of FDecl work, we need the assumption that it will generate the same logical variable names as the application of FDecl for the function f that was originally called, as these are the names that the callee frame has used at the time when the hole was created. We can assume this without loss of generality, as we might as well have kept the name f of the originally called function with the hole, and used this name in the call to FDecl here, resulting in the same contract (and hence logical variable names) as the actual back-translated f. Any names clashes of symbolic variables resulting from this assumption and function calls in general (eg. when calling the same f twice) can be safely ignored, as we could always introduce renaming countermeasures. Another way around this would be to not require the strict condition $POST == Q$ during frame linking, but to allow a renaming function f_{Rename} to be applied to the logical variables of Q (and then also in the application of the FApp rule). This would probably be slightly more elegant and require less assumptions than the current approach.

7.3.3 Rules for return and frame. The postcondition after the return statement is $\text{univ\_contr}(\overline{result})$, which is the universal contract of any function. This can be achieved using the Consequence rule (including chunk leaking). This rule is an exception to the general back-translation, as it does not have univ_post as the separation logic postcondition, but rather $\text{univ\_contr}(\overline{result})$.

$$\frac{tstm \rightsquigarrow_{\mathrm{b}} \{P\}_{\gamma}\ sstm\ \{Q\}_{\gamma'}}{\begin{array}{l} tstm;\ \text{return}\ \{\overline{texp}\} \rightsquigarrow_{\mathrm{b}} \\ \{\text{univ\_pre}\}_{\gamma_{\text{pre}}} \qquad \text{univ\_pre} == P * P_{\text{frame}}, \gamma_{\text{pre}} == \gamma * \gamma_{\text{frame}} \\ sstm; \\ \{Q * P_{\text{frame}}\}_{\gamma' \cup \gamma_{\text{frame}}} \\ \{\text{univ\_contr}(\overline{texp_{\mathrm{b},\gamma'}})\}_{\gamma'} \qquad \text{Compositionality} \\ \text{return}\ \{\overline{texp_{\mathrm{b}}}\} \\ \{\text{univ\_contr}(\overline{result})\}_{\gamma'} \end{array}} \quad \text{(Return)}$$

No Consequence is needed to glue together back-translated pieces of code, because all back-translated pieces fit automatically. However, a separate form of Framing is needed, to re-add any pieces we did not need for the proof we just constructed. We need to re-add variables to the environment, as well as add the universal contracts for these variables to the heap. We also need to be able to frame off auxiliary variables after they have served their purpose, since they do not correspond to target variables (and hence do not need universal contracts) and will not be used anymore anyway.

The below frame rule allows adding in variables and their universal contracts. Because this type of framing is added in, it suffices to look at the current code to construct the environment γ_{pre} for a piece of code. The Frame rule allows (temporarily) deviating from the universal contract we defined above in 7.2.2, to allow for linking of different back-translated pieces of code, and to allow describing the separation logic state during execution (which we will do later).

In the below rule γ_{frame} contains all variables that are frame off that did appear in the target language, whereas γ_{AUX} contains all auxiliary variables that are framed off (where we do not care what they map to).

$$\frac{\begin{array}{c} tstm \rightsquigarrow_{\mathrm{b}} \{P\}_{\gamma}\ c\ \{Q\}_{\gamma'} \\ R = *\{\text{univ\_contr}_{\tau}(id_{\text{prog}}) \mid \\ id_{\text{prog}} \in \gamma_{\text{frame}} \wedge \text{TypeOfVar}_{\mathrm{t}}(id_{\text{prog}}) = \tau\} \\ \gamma_{\mathrm{s}} = \gamma \uplus \gamma_{\text{frame}} \uplus \gamma_{\text{AUX}} \qquad \gamma_{\mathrm{s}}' = \gamma' \uplus \gamma_{\text{frame}} \uplus \gamma_{\text{AUX}} \end{array}}{tstm \rightsquigarrow_{\mathrm{b}} \{P * R\}_{\gamma_{\mathrm{s}}}\ c\ \{Q * R\}_{\gamma_{\mathrm{s}}'}} \quad \text{(Frame)}$$

7.3.4 Rules for functions. At first sight it seems like we cannot compile the back-translated function contract from the function declaration itself, as we are lacking all variable names for the returned values. Fortunately, we do not need the names of the return arguments, as their logical counterparts are just called *result* in the contract postcondition. These universal contracts for the *result* logical variables will automatically follow from the return statement and the Consequence axiom. In conclusion, function contexts Σ_i only need to contain the function declaration in the case of the target language, as we use universal contracts and the function's contract can be derived from its declaration.

$$\frac{\begin{array}{c}\overline{\tau_{\text{arg}}} \rightsquigarrow_{\text{InvCompileType}} \overline{\tau'_{\text{arg}}} \\ \overline{\tau_{\text{ret}}} \rightsquigarrow_{\text{InvCompileType}} \overline{\tau'_{\text{ret}}}\end{array}}{\begin{array}{c}\overline{\tau_{\text{ret}}}\ f(\overline{\tau_{\text{arg}}\ id_{\text{arg}}}) \\ \rightsquigarrow^{\text{b}}_{\text{Decl}}\ \overline{\tau'_{\text{ret}}}\ f_{\text{bt}}(\overline{\tau'_{\text{arg}}\ id_{\text{arg}}}) \\ \text{//@pre univ\_contr}_{\overline{\tau_{\text{arg}}}}(\overline{id_{\text{arg}}})\ \text{//@post univ\_contr}_{\overline{\tau_{\text{ret}}}}(\overline{result})\end{array}} \quad \text{(FDecl)}$$

The reason for writing $\vdash$ (denoting a proof) in front of the back-translated functions, components and programs in the following rules is that the result of the back-translation should be a *separation-logic proof* of the source program, and not just the source program itself. Proving that the result is indeed a proper separation logic proof of the source program proves the back-translation sound.

For the two rules below, the properness of the written $\vdash$ follows from the ImplFVerif and ContFVerif separation logic axioms. In the case of the implemented function, the existence of the triple $\{Sep_{\text{pre}}\}_{[\overline{id_{\text{arg}}:id_{\text{arg}}}]}\ sstm; \text{return } \overline{sexp}\ \{Sep_{\text{post}}\}_\gamma$ required by the ImplFVerif axiom follows directly from a combination of all previous axioms and the below Frame and Return rules, to glue everything together and to back-translate return statements, respectively.

$$\frac{\begin{array}{c}itfunc = \overline{\tau}\ f(\overline{\tau_{\text{arg}}\ id_{\text{arg}}}) \\ itfunc \rightsquigarrow^{\text{b}}_{\text{Decl}}\ \overline{\tau'_{\text{ret}}}\ f_{\text{bt}}(\overline{\tau'_{\text{arg}}\ id'_{\text{arg}}})\text{//@pre } Sep_{\text{pre}}\ \text{//@post } Sep_{\text{post}} \\ tstm; \text{return } \{\overline{texp}\} \rightsquigarrow_{\text{b}} \{Sep_{\text{pre}}\}_{[\overline{id'_{\text{arg}}:id'_{\text{arg}}}]}\ sstm; \text{return } \{\overline{sexp}\}\ \{Sep_{\text{post}}\}_\gamma\end{array}}{\begin{array}{c}itfunc\ \{tstm; \text{return } \{\overline{texp}\}\} \\ \rightsquigarrow_{\text{b}} \\ \vdash \overline{\tau'_{\text{ret}}}\ f_{\text{bt}}(\overline{\tau'_{\text{arg}}\ id'_{\text{arg}}})\text{//@pre } Sep_{\text{pre}}\ \text{//@post } Sep_{\text{post}}\{sstm; \text{return } \{\overline{sexp}\}\}\end{array}} \quad \text{(ImplFVerif)}$$

$$\frac{\begin{array}{c}csfunc = \overline{\tau}\ f(\overline{\tau_{\text{arg}}\ id_{\text{arg}}}) \\ csfunc \rightsquigarrow^{\text{b}}_{\text{Decl}}\ \overline{\tau'_{\text{ret}}}\ f_{\text{bt}}(\overline{\tau'_{\text{arg}}\ id'_{\text{arg}}})\text{//@pre } Sep_{\text{pre}}\ \text{//@post } Sep_{\text{post}}\end{array}}{csfunc \rightsquigarrow_{\text{b}} \vdash \overline{\tau'_{\text{ret}}}\ f_{\text{bt}}(\overline{\tau'_{\text{arg}}\ id'_{\text{arg}}})\text{//@pre } Sep_{\text{pre}}\ \text{//@post } Sep_{\text{post}}} \quad \text{(ContFVerif)}$$

Rules for stubs. As discussed before, we need to back-translate the target-level incall and outcall stubs that are created during the compilation of the verified component s, to form a bridging component between the universal contracts of the back-translated context and the concrete boundary contracts of the source component s. The below BackIncall and BackOutcall inference rules describe this back-translation.

We need a set of *bridging functions*, created by performing a specific kind of back-translation on the verified component's target level stubs, to bridge the two previously described gaps. Not only the contracts should be matched in the back-translated stubs, but the concrete representations of program variables require rewriting as well, as a transition to/from a source-level representation from/to a back-translation after compilation representation of the same variables is made before and after wrapped function calls in the back-translated stubs.
As a simple example, we require a transition to and from the representation $f(\text{int} * a)\text{//@pre } n : a' \mapsto_{\text{int}} [b, c, d] * a' == a$ (the source representation) from and to $f(\text{int}* a,\ (\text{int}*, \text{int})\ n)\text{//@pre } (n_{\text{chunk}} : n.1 \mapsto_{\text{int}} l * \text{length}(l) = n.2) \vee n = \text{null}_0$ (back-translation after compilation of the same source

function). The first contract is the concrete contract and the second contract the universal contract. In order to make this conversion between the arguments int $*$ a and int $*$ a, (int$*$, int) n feasible in both directions, we have to make the following assumption on the form of the contracts of boundary functions (which are the ones that cause stubs). This assumption holds on the postcondition of the function contract for export boundary functions (causing incall stubs) and on the precondition for import boundary functions (causing outcall stubs). We assume that all logical variables denoting addresses of the chunks in the contract of f are bound to a single expression in the pure heap, where this single expression contains only logical variables corresponding to arguments and result variables (in the case of the postcondition). Given this assumption, we know that we are able to easily calculate the value of each chunk variable, as it has to be represented by a separation logic expression *exp*, in which all variables are program variables. Eg. for the variable (int$*$, int) n from the above example, can hence just write $n = (exp, 3)$ in the stub to get the correct value for n at runtime, where *exp* is a.

The below rules extract the necessary data to perform the conversion between program variables. The first rule is where our condition on boundary contracts comes into play; we know that the *exp* in the below rule is also a valid *sexp*, because we enforced this restriction on contracts. No base case is needed, as we enforced there to be precisely one condition per chunk address and length, so we would never hit any base case.

$$\frac{}{id == exp \wedge \overline{cond}, id \rightsquigarrow_{\text{FindInPath}} exp} \quad \text{(FindInPathT)}$$

$$\frac{cond \neq (id == exp) \qquad \overline{cond}, id \rightsquigarrow_{\text{FindInPath}} res}{cond \wedge \overline{cond}, id \rightsquigarrow_{\text{FindInPath}} res} \quad \text{(FindInPathF)}$$

$$\frac{}{\epsilon, path \rightsquigarrow_{\text{ExtractLenAddress}} \epsilon} \quad \text{(ExtractLenAddressEmpty)}$$

$$\frac{path, n \rightsquigarrow_{\text{FindInPath}} res \qquad chunks, path \rightsquigarrow_{\text{ExtractLenAddress}} \overline{res}}{(n_{\text{chunk}} : n \mapsto [v_1, \ldots, v_k] \rightsquigarrow_{\text{ExtractLenAddress}} n_{\text{chunk}} = (res, k); \overline{res}} \quad \text{(ExtractLenAddress)}$$

Note that we cannot reuse the regular back-translation rules we used for the target-level components, as these rules use universal contracts and we are using the back-translated in- and outcall stubs to transform from/to the minimal contract to/from a non-minimal source-level contract here. We therefore need new back-translation rules for the few constructs that appear in target-level stubs. We call these new rules the *stub* back-translation rules, denoted $\rightsquigarrow_{\text{b}}^{\text{stub}}$. These rules just back-translate input target-level code to source-level code without proof. We will be constructing the proof for the back-translated stubs in a separate theorem below, after we present the back-translation rules for incall and outcall stubs. The rules back-translate to simpler code than the vanilla back-translation rules, because they will use different contracts (as will be clear from the theorem later); they do not require all the checks the original back-translation rules used, and they do not make use of Consequence to minimize contract postconditions in-between. Since the compiled stubs only contain variable declarations and assignments, array lookups, guard statements and sequencing

(and function application, but its back-translation is the identity in this case), we only need stub back-translation rules for those target-level statements.

$$\frac{\tau \rightsquigarrow_{\text{InvCompileType}} \tau'}{\tau\ id_{\text{prog}} \rightsquigarrow_{\text{b}}^{\text{stub}} \tau'\ id_{\text{prog}}} \qquad \text{(VarDeclStub)}$$

In the below rule, no erasure or stuck checking needs to happen, unlike in the regular back-translation, because all back-translated assignments are of the form $id_{\text{prog}} = \text{addr}(n)$.

$$\frac{}{id_{\text{prog}} = texp \rightsquigarrow_{\text{b}}^{\text{stub}} id_{\text{prog}} = texp_{\text{b}}} \qquad \text{(VarAsgnStub)}$$

In the below rule, again no erasure or stuck checking needs to happen, unlike in the regular back-translation, because all back-translated assignments are of the form $id_{\text{prog}} = n[k]$.

$$\frac{\text{TypeOfVar}^{\text{t}}(id_{\text{prog}}) = \tau \qquad \tau \rightsquigarrow_{\text{InvCompileType}} \tau'}{id_{\text{prog}} = texp_1[texp_2] \rightsquigarrow_{\text{b}}^{\text{stub}} \tau' * t_{\text{aux}}; t_{\text{aux}} = id_{\text{prog}}.1; t_{\text{aux}} = texp_{1,\text{b}}[texp_{2,\text{b}}]} \qquad \text{(ArrayLkupStub)}$$

$$\frac{}{\text{guard}(texp) \rightsquigarrow_{\text{b}}^{\text{stub}} \text{guard}(texp_{\text{b}})} \qquad \text{(GuardStub)}$$

$$\frac{p_1 \rightsquigarrow_{\text{b}}^{\text{stub}} sstm_1 \qquad p_2 \rightsquigarrow_{\text{b}}^{\text{stub}} sstm_2}{p_1; p_2 \rightsquigarrow_{\text{b}}^{\text{stub}} sstm_1; sstm_2} \qquad \text{(SeqStub)}$$

Σ_{C_s} in the rule below is the environment of the source-level component s, where we use the assumption that boundary functions' contracts are separable into a path and a chunk part, both for the pre-and postcondition. We just use the compiled stub notation from the compilation rules as starting point for the back-translation, as this will make our life easier. All stub variables have the same exact meaning as they did in the Incall and Outcall compilation rules. As usual, superscript t signifies a target-related variable. Erasure emulation in the source is not required, as

In the below two rules, the sets of names $\overline{n}$ and $\overline{m}$ appear both as the names of the source level array chunks, but also as the names of the back-translated reified chunks (ie. program variables). This does not result in problems as long as we consider the two name-spaces (for chunks and program variables) disjoint in the proofs. This would however result in problems during compilations, if the source chunks were reified again. We ignore these issues during recompilation, as they are solely name-related and can always be fixed by using different names and/ or renaming using the Conseq rule.

$$
\begin{array}{l}
f_{\text{sdecl}} = \{\overline{\tau_{\text{ret}}}\}\ f(\overline{\tau_{\text{arg}}\ id_{\text{arg}}}) \\
\qquad\qquad //@\text{pre}\ PRE_{\text{s}} * PRE_{\text{p}} \\
\qquad\qquad //@\text{post}\ POST_{\text{s}} * POST_{\text{p}} \\
f_{\text{tdecl}} = \{\overline{\tau^{\text{t}}_{\text{ret}}}, \overline{\tau^{\text{t}}_{\text{post}}}\}\ f(\overline{\tau^{\text{t}}_{\text{arg}}\ id_{\text{arg}}}, \overline{\tau^{\text{t}}_{\text{pre}}\ m}) \\
f^{\text{b}}_{\text{tdecl}} = \{\overline{\tau_{\text{ret}}}, \overline{\tau_{\text{post}}}\}\ f_{\text{bt}}(\overline{\tau_{\text{arg}}\ id_{\text{arg}}}, \overline{\tau_{\text{pre}}\ m}) \\
\qquad //@\text{pre}\ \text{univ\_contr}_{\overline{\tau^{\text{t}}_{\text{arg}}}, \overline{\tau^{\text{t}}_{\text{pre}}}}(\overline{id_{\text{arg}}}, \overline{m}) \\
//@\text{post}\ \text{univ\_contr}_{\overline{\tau^{\text{t}}_{\text{ret}}}, \overline{\tau^{\text{t}}_{\text{post}}}}(\overline{result}, \overline{result_n}) \\
\vdash f_{\text{sdecl}} \rightsquigarrow_{\text{Incall}} f_{\text{tdecl}}, stub_{\text{incall}} \qquad f_{\text{tdecl}} \rightsquigarrow^{\text{b}}_{\text{Decl}} f^{\text{b}}_{\text{tdecl}} \\
cs_{\text{pre}}, d_{\text{pre}}, a_{\text{pre}}, cp_{\text{pre}} \rightsquigarrow^{\text{stub}}_{\text{b}} cs^{\text{b}}_{\text{pre}}, d^{\text{b}}_{\text{pre}}, a^{\text{b}}_{\text{pre}}, cp^{\text{b}}_{\text{pre}} \\
POST_{\text{s}}, POST_{\text{p}} \rightsquigarrow_{\text{ExtractLenAddress}} a_{\text{la}}
\end{array}
$$

$$
\begin{array}{ll}
stub_{\text{incall}} = & stub_{\text{incall,b}} = \\
\left\{\begin{array}{l}
f_{\text{tdecl}}\{ \\
cs_{\text{pre}}; \\
\\
d_{\text{pre}}; a_{\text{pre}}; cp_{\text{pre}}; \\
\overline{\tau^{\text{t}}_{\text{ret}}\ result}; \overline{\tau^{\text{t}}_{\text{post}}\ n}; \\
\{\overline{result}, \overline{n}\} = f_{\text{comp}}(\overline{id_{\text{arg}}}, \overline{m}); \\
\\
\\
\text{return}\ \{\overline{result}, \overline{n}\}\}
\end{array}\right\} &
\left\{\begin{array}{l}
f^{\text{b}}_{\text{tdecl}}\{ \\
cs^{\text{b}}_{\text{pre}}; \\
\text{RangeToArray}(\overline{m.1}, \overline{m.2}) \\
d^{\text{b}}_{\text{pre}}; a^{\text{b}}_{\text{pre}}; cp^{\text{b}}_{\text{pre}}; \\
\overline{\tau_{\text{ret}}\ result}; \overline{\tau_{\text{post}}\ n}; \\
\{\overline{result}\} = f(\overline{id_{\text{arg}}}); \\
a_{\text{la}}; \\
\text{ArrayToRange}(\overline{n.1}, \overline{n.2}) \\
\text{return}\ \{\overline{result}, \overline{n}\}\}
\end{array}\right\}
\end{array}
$$

$$
\frac{}{f_{\text{sdecl}} \rightsquigarrow_{\text{BackIncall}} \vdash stub_{\text{incall,b}}}\ (\textsc{BackIncall})
$$

$$\frac{\begin{array}{c} f_{\text{sdecl}} = \{\overline{\tau_{\text{ret}}}\}\ f(\overline{\tau_{\text{arg}}\ id_{\text{arg}}}) \\ //@\text{pre}\ PRE_{\text{s}} * PRE_{\text{p}} \\ //@\text{post}\ POST_{\text{s}} * POST_{\text{p}} \\ f_{\text{tdecl}} = \{\overline{\tau^{\text{t}}_{\text{ret}}}, \overline{\tau^{\text{t}}_{\text{post}}}\}\ f(\overline{\tau^{\text{t}}_{\text{arg}}\ id_{\text{arg}}}, \overline{\tau^{\text{t}}_{\text{pre}}\ m}) \\ f^{\text{b}}_{\text{tdecl}} = \{\overline{\tau_{\text{ret}}}, \overline{\tau_{\text{post}}}\}\ f_{\text{bt}}(\overline{\tau_{\text{arg}}\ id_{\text{arg}}}, \overline{\tau_{\text{pre}}\ m}) \\ //@\text{pre}\ \text{univ\_contr}_{\overline{\tau^{\text{t}}_{\text{arg}}}, \overline{\tau^{\text{t}}_{\text{pre}}}}(\overline{id_{\text{arg}}}, \overline{m}) \\ //@\text{post}\ \text{univ\_contr}_{\overline{\tau^{\text{t}}_{\text{ret}}}, \overline{\tau^{\text{t}}_{\text{post}}}}(\overline{result}, \overline{result_n}) \\ \vdash f_{\text{sdecl}} \rightsquigarrow_{\text{Outcall}} f_{\text{tdecl}}, stub_{\text{outcall}} \qquad f_{\text{tdecl}} \rightsquigarrow^{\text{b}}_{\text{Decl}} f^{\text{b}}_{\text{tdecl}} \\ d_{\text{pre}}, a_{\text{pre}}, cs_{\text{post}}, d_{\text{post}}, a_{\text{post}}, cp_{\text{pre}} \rightsquigarrow^{\text{stub}}_{\text{b}} d^{\text{b}}_{\text{pre}}, a^{\text{b}}_{\text{pre}}, cs^{\text{b}}_{\text{post}}, d^{\text{b}}_{\text{post}}, a^{\text{b}}_{\text{post}}, cp^{\text{b}}_{\text{pre}} \\ PRE_{\text{s}}, PRE_{\text{p}} \rightsquigarrow_{\text{ExtractLenAddress}} a_{\text{la}} \\ \left\{\begin{array}{l} stub_{\text{outcall}} = \\ \{\overline{\tau^{\text{t}}_{\text{ret}}}, \overline{\tau^{\text{t}}_{\text{post}}}\}\ f_{\text{comp}}(\overline{\tau^{\text{t}}_{\text{arg}}\ id_{\text{arg}}}, \overline{\tau^{\text{t}}_{\text{pre}}\ m})\{ \\ \\ d_{\text{pre}}; a_{\text{pre}}; \\ \\ \overline{\tau^{\text{t}}_{\text{ret}}\ result}; \overline{\tau^{\text{t}}_{\text{post}}\ n}; \\ \{\overline{result}, \overline{n}\} = f(\overline{id_{\text{arg}}}, \overline{m}); \\ cs_{\text{post}}; \\ \\ d_{\text{post}}; a_{\text{post}}; cp_{\text{post}}; \\ \text{return}\ \{\overline{result}, \overline{n}\}\} \end{array}\right\} \quad \left\{\begin{array}{l} stub_{\text{outcall,b}} = \\ f_{\text{sdecl}}\{\overline{\tau_{\text{pre}}\ m}; \\ a_{\text{la}}; \\ d^{\text{b}}_{\text{pre}}; a^{\text{b}}_{\text{pre}}; \\ \text{ArrayToRange}(\overline{m.1}, \overline{m.2}) \\ \overline{\tau_{\text{ret}}\ result}; \overline{\tau_{\text{post}}\ n}; \\ \{\overline{result}, n\} = f_{\text{bt}}(\overline{id_{\text{arg}}}, m); \\ cs^{\text{b}}_{\text{post}}; \\ \text{RangeToArray}(\overline{n.1}, \overline{n.2}) \\ d^{\text{b}}_{\text{post}}; a^{\text{b}}_{\text{post}}; cp^{\text{b}}_{\text{post}}; \\ \text{return}\ \{\overline{result}\}\} \end{array}\right\} \end{array}}{f_{\text{sdecl}} \rightsquigarrow_{\text{BackOutcall}} \vdash stub_{\text{outcall,b}}}\ (\textsc{BackOutcall})$$

In contrast to the case for regular function back-translation, it is not obvious that $\vdash stub_{\text{incall,b}}$ and $\vdash stub_{\text{outcall,b}}$ hold in the previous two rules. In order to prove the back-translation of these two rules sound, we will need to prove this explicitly in the following theorem:

Theorem 7 (BackIncall and BackOutcalll produce sound proofs)**.**
The output proofs $\vdash stub_{\text{incall,b}}$ *and* $\vdash stub_{\text{outcall,b}}$ *appearing in* BackIncall *and* BackOutcalll*, respectively, are sound separation logic proofs (using the* ImplfVerif *axiom) of the functions* $stub_{\text{incall,b}}$ *and* $stub_{\text{outcall,b}}$*, respectively.*

Proof.

We split up the proof into the two cases mentioned in the theorem. Both cases are very similar and symmetrical; the pre-function call part for incall stubs is similar to the post-function call part for outcall stubs and the other way around.

1. BackIncall

A proof of the stub $stub_{\text{incall,b}}$'s contract is presented here. This proof allows application of the ImplfVerif separation logic axiom.

(1): There are no nested universal contracts inside the range, since each m's type is that of a reified array chunk $\tau*$ in the target language, where τ contains no further pointers types.
(2): Call the set of (necessarily fresh) logical variables that appear in PRE_s: $\mathrm{V}(PRE)$. Since $\mathrm{cond}_\mathrm{array}(\overline{m})$ has the same number of array resources with the same lengths (due to $cs^\mathrm{b}_\mathrm{pre}$) as those in PRE_s, we can rename all logical variables in $\mathrm{cond}_\mathrm{array}(\overline{m})$ and all chunknames (from $\overline{m_\mathrm{range}}$ to $\overline{m}$) to obtain the assertion PRE_s. Call the resource addresses appearing in $\mathrm{V}(PRE)$: $\mathrm{ADDR}(PRE)$. Then $[\overline{m} : \overline{m}]$ in the environment can be renamed to $[\overline{m} : (\mathrm{ADDR}(PRE), \overline{k_m})]$ after the renaming operation.
(3): $\mathrm{V}(PRE)$ is the set of *program* variables declared by $d^\mathrm{b}_\mathrm{pre}$ and assigned by $a^\mathrm{b}_\mathrm{pre}$. The expressions assigned are $m.1$ and $\forall 0 \leq i < k_m.\, m.1[i]$ for all $m \in \overline{m}$ (cfr. stub backtranslation rules). Since $[\overline{m} : (\mathrm{ADDR}(PRE), \overline{k_m})]$, the environment applied to those expressions again results in $\mathrm{V}(PRE)$. We hence get $[\mathrm{V}(PRE) : \mathrm{V}(PRE)]$ in the environment.
(4): The guard expressions in cp_pre are obtained by reifying all conditions in PRE_p. They contain no pointer-typed variables or expressions (only variables from $\overline{id_\mathrm{arg}}$ and $\mathrm{V}(PRE)$) and are hence identical to the expressions in $cp^\mathrm{b}_\mathrm{pre}$. Because $[\overline{id_\mathrm{arg}} : \overline{id_\mathrm{arg}}][\mathrm{V}(PRE) : \mathrm{V}(PRE)]$ in the environment, applying $cp^\mathrm{b}_\mathrm{pre}$ will amount to adding exactly PRE_p to the separation logic state.
(5): a_la assigns $n = (addr_n, k_n)$ for all $(n, addr_n) \in (\overline{n}, \mathrm{ADDR}(POST))$. This is true because a_la gathers the chunk address expressions and the fixed lengths from $POST_\mathrm{p}$. The address expressions can only contain variables from $\overline{result}$ and $\overline{id_\mathrm{arg}}$, as required before. In γ, $\overline{id_\mathrm{arg}}$ maps to itself and $\overline{result}$ to $\overline{x}$ (but $\overline{x}$ is also substituted for $\overline{result}$ in $POST$), resulting in the correct expressions for the addresses of the program variables $\overline{n}$.
(6): $\mathrm{univ\_contr}_{\underline{\tau}^\mathrm{t}_\mathrm{post}}(\overline{n})$ almost trivially follows from $\mathrm{cond}_\mathrm{range}(\overline{n})$, given an observation about $\overline{n}$ analogous to (1) and by adding in a conditional ? through Conseq

$\{\text{univ\_contr}_{\overline{\tau^{\text{t}}_{\text{arg}}}, \overline{\tau^{\text{t}}_{\text{pre}}}}(\overline{id_{\text{arg}}}, \overline{m})\}_{[\overline{id_{\text{arg}}}:\overline{id_{\text{arg}}}][\overline{m}:\overline{m}]}$

$cs^{\text{b}}_{\text{pre}};$ Guards m != (null, 0) and $m.2 == k_m$ for all $m \in \overline{m}$

$\text{cond}_{\text{range}}(m) = m_{\text{range}} : [m.1 + i \mapsto l_m[i] \mid 0 \leq i < \text{length}(l_m)]$
$\quad * \text{length}(l_m) == m.2 == k_m$ (1)

$\{\text{univ\_contr}_{\overline{\tau^{\text{t}}_{\text{arg}}}}(\overline{id_{\text{arg}}}) * \text{cond}_{\text{range}}(\overline{m})\}_{[\overline{id_{\text{arg}}}:\overline{id_{\text{arg}}}][\overline{m}:\overline{m}]}$

$\text{RangeToArray}(\overline{m.1}, \overline{m.2})$

$\text{cond}_{\text{array}}(m) = m_{\text{array}} : m.1 \mapsto l_m * \text{length}(l_m) == m.2 == k_m$

$\text{univ\_contr}_{\overline{\tau^{\text{t}}_{\text{arg}}}}(\overline{id_{\text{arg}}}) \equiv$ true, since the types $\overline{\tau^{\text{t}}_{\text{arg}}}$ do not contain pointers

$\{\text{cond}_{\text{array}}(\overline{m})\}_{[\overline{id_{\text{arg}}}:\overline{id_{\text{arg}}}][\overline{m}:\overline{m}]}$ (2)

$\{PRE_{\text{s}}\}_{[\overline{id_{\text{arg}}}:\overline{id_{\text{arg}}}][\overline{m}:(\text{ADDR}(PRE), \overline{k_m})]}$

$d^{\text{b}}_{\text{pre}}; a^{\text{b}}_{\text{pre}};$ (3)

$\{PRE_{\text{s}}\}_{[\overline{id_{\text{arg}}}:\overline{id_{\text{arg}}}][\overline{m}:\_][\text{V}(PRE):\text{V}(PRE)]}$

$cp^{\text{b}}_{\text{pre}};$ (4)

$\{PRE_{\text{s}} * PRE_{\text{p}}\}_{[\overline{id_{\text{arg}}}:\overline{id_{\text{arg}}}][\text{V}(PRE):\text{V}(PRE)]}$

$\overline{\tau_{\text{ret}}\ result}; \overline{\tau_{\text{post}}\ n};$

$\{PRE_{\text{s}} * PRE_{\text{p}}\}_{[\overline{id_{\text{arg}}}:\overline{id_{\text{arg}}}][\text{V}(PRE):\text{V}(PRE)][\overline{result}:\_][\overline{n}:\_]}$

$\{\overline{result}\} = f(\overline{id_{\text{arg}}});$ Apply FApp axiom

$\{POST_{\text{s}} * POST_{\text{p}}[\overline{result} \mapsto \overline{x}]\}_{[\overline{id_{\text{arg}}}:\overline{id_{\text{arg}}}][\overline{result}:\overline{x}][\overline{n}:\_]}$ $\overline{x}$ is fresh

Define V($POST$) and ADDR($POST$) for $POST_{\text{s}}$, analogous to PRE_{s} before.

$a_{\text{la}};$ (5)

$\{POST_{\text{s}} * POST_{\text{p}}[\overline{result} \mapsto \overline{x}]\}_{[\overline{result}:\overline{x}][\overline{n}:(\text{ADDR}(POST), \overline{k_n})]}$ Conseq on $POST_{\text{s}}$, leak $POST_{\text{p}}$

$\{\text{cond}_{\text{array}}(\overline{n})\}_{[\overline{result}:\overline{x}][\overline{n}:\overline{n}]}$

$\text{ArrayToRange}(\overline{n.1}, \overline{n.2})$

$\{\text{cond}_{\text{range}}(\overline{n})\}_{[\overline{result}:\overline{x}][\overline{n}:\overline{n}]}$ (6)

return $\{\overline{result}, \overline{n}\}$

$\{\text{univ\_contr}_{\overline{\tau^{\text{t}}_{\text{post}}}}(\overline{result_n})\}_{[\overline{result}:\overline{result}][\overline{n}:\overline{result_n}]}$

$\text{univ\_contr}_{\overline{\tau^{\text{t}}_{\text{ret}}}}(\overline{result}) \equiv$ true, since the types $\overline{\tau^{\text{t}}_{\text{res}}}$ do not contain target pointers

$\{\text{univ\_contr}_{\overline{\tau^{\text{t}}_{\text{ret}}}, \overline{\tau^{\text{t}}_{\text{post}}}}(\overline{result}, \overline{result_n})\}_{[\overline{result}:\overline{result}][\overline{n}:\overline{result_n}]}$

(BackIncallProof)

2. BackOutcall

A proof of the stub $stub_{\text{outcall,b}}$'s contract is presented here. This proof allows application of the ImplfVerif separation logic axiom. It is very analogous to the previous proof, and can hence be shortened in a couple places. We use the notation $(n)_{\text{In}}$ to refer to point (n) from the above Incall proof.

The definitions of $\mathrm{cond}_{\mathrm{range}}(m)$, $\mathrm{cond}_{\mathrm{array}}(m)$, $\mathrm{V}(\mathit{PRE})$, $\mathrm{ADDR}(\mathit{PRE})$, $\mathrm{V}(\mathit{POST})$ and $\mathrm{ADDR}(\mathit{POST})$ are reused from the Incall proof.

(1): Cfr. $(4)_{\mathrm{In}}$ with n, POST instead of m, PRE. One difference is that $cp^{\mathrm{b}}_{\mathrm{post}}$ now also contains variables from:

- $\mathrm{V}(\mathit{POST})$ with $[\mathrm{V}(\mathit{POST}) : \mathrm{V}(\mathit{POST})]$. This constitutes no problem.
- $\overline{\mathit{result}}$ with $[\overline{\mathit{result}} : \overline{x}]$. This is the only relevant non-identity-mapping present in the current environment. The pure heap that will be created by the guard statements in $cp^{\mathrm{b}}_{\mathrm{post}}$ is hence not $\mathit{POST}_{\mathrm{p}}$, but rather $\mathit{POST}_{\mathrm{p}}[\overline{\mathit{result}} \mapsto \overline{x}]$.

$\{PRE_\mathrm{s} * PRE_\mathrm{p}\}_{[\overline{id_\mathrm{arg}}:\overline{id_\mathrm{arg}}]}$
$\overline{\tau_\mathrm{pre}\ m};$
$a_\mathrm{la};$ cfr. $(5)_\mathrm{In}$ with *PRE* substituted for *POST* and without $\overline{result}$
$\{PRE_\mathrm{s} * PRE_\mathrm{p}\}_{[\overline{id_\mathrm{arg}}:\overline{id_\mathrm{arg}}][\overline{m}:(\mathrm{ADDR}(PRE),\overline{k_m})]}$
$d^\mathrm{b}_\mathrm{pre}; a^\mathrm{b}_\mathrm{pre};$ cfr. $(3)_\mathrm{In}$
$\{PRE_\mathrm{s} * PRE_\mathrm{p}\}_{[\overline{id_\mathrm{arg}}:\overline{id_\mathrm{arg}}][\overline{m}:(\mathrm{ADDR}(PRE),\overline{k_m})][\mathrm{V}(PRE):\mathrm{V}(PRE)]}$ Conseq on PRE_s, leak PRE_p
$\{\mathrm{cond}_\mathrm{array}(\overline{m})\}_{[\overline{id_\mathrm{arg}}:\overline{id_\mathrm{arg}}][\overline{m}:\overline{m}][\mathrm{V}(PRE):\mathrm{V}(PRE)]}$
$\mathrm{ArrayToRange}(\overline{m.1}, \overline{m.2})$
$\{\mathrm{cond}_\mathrm{range}(\overline{m})\}_{[\overline{id_\mathrm{arg}}:\overline{id_\mathrm{arg}}][\overline{m}:\overline{m}][\mathrm{V}(PRE):\mathrm{V}(PRE)]}$ cfr. $(6)_\mathrm{In}$
$\overline{\tau_\mathrm{ret}\ result}; \overline{\tau_\mathrm{post}\ n};$
$\mathrm{univ\_contr}_{\overline{\tau^\mathrm{t}_\mathrm{arg}}}(\overline{id_\mathrm{arg}})$ ≡ true, as before, so we just add it
$\{\mathrm{univ\_contr}_{\overline{\tau^\mathrm{t}_\mathrm{arg}}, \overline{\tau^\mathrm{t}_\mathrm{pre}}}(\overline{id_\mathrm{arg}}, \overline{m})\}_{[\overline{id_\mathrm{arg}}:\overline{id_\mathrm{arg}}][\overline{m}:\overline{m}][\mathrm{V}(PRE):\mathrm{V}(PRE)][\overline{result}:\_][\overline{n}:\_]}$
$\{\overline{result, n}\} = f_\mathrm{bt}(\overline{id_\mathrm{arg}, m});$ Apply FApp axiom
$\{\mathrm{univ\_contr}_{\overline{\tau^\mathrm{t}_\mathrm{ret}}, \overline{\tau^\mathrm{t}_\mathrm{post}}}(\overline{x}, \overline{n})\}_{[\overline{id_\mathrm{arg}}:\overline{id_\mathrm{arg}}][\mathrm{V}(PRE):\mathrm{V}(PRE)][\overline{result}:\overline{x}][\overline{n}:\overline{n}]}$ $\overline{x}$ and $\overline{n}$ (logical) are fresh
$\mathrm{univ\_contr}_{\overline{\tau^\mathrm{t}_\mathrm{ret}}}(\overline{x})$ ≡ true, as before, so we just leak it
$cs^\mathrm{b}_\mathrm{post};$ Guards n != (null, 0) and $n.2$ == k_n for all $n \in \overline{n}$, cfr. $(1)_\mathrm{In}$
$\{\mathrm{cond}_\mathrm{range}(\overline{n})\}_{[\overline{id_\mathrm{arg}}:\overline{id_\mathrm{arg}}][\mathrm{V}(PRE):\mathrm{V}(PRE)][\overline{result}:\overline{x}][\overline{n}:\overline{n}]}$
$\mathrm{RangeToArray}(\overline{n.1}, \overline{n.2})$
$\{\mathrm{cond}_\mathrm{array}(\overline{n})\}_{[\overline{id_\mathrm{arg}}:\overline{id_\mathrm{arg}}][\mathrm{V}(PRE):\mathrm{V}(PRE)][\overline{result}:\overline{x}][\overline{n}:\overline{n}]}$ cfr. $(2)_\mathrm{In}$ with n, *POST* instead of m, *PRE*
$\{POST_\mathrm{s}\}_{[\overline{id_\mathrm{arg}}:\overline{id_\mathrm{arg}}][\mathrm{V}(PRE):\mathrm{V}(PRE)][\overline{result}:\overline{x}][\overline{n}:(\mathrm{ADDR}(POST),\overline{k_n})]}$
$d^\mathrm{b}_\mathrm{post}; a^\mathrm{b}_\mathrm{post};$ cfr. $(3)_\mathrm{In}$ with n, *POST* instead of m, *PRE*
$\{POST_\mathrm{s}\}_{[\overline{id_\mathrm{arg}}:\overline{id_\mathrm{arg}}][\mathrm{V}(PRE):\mathrm{V}(PRE)][\overline{result}:\overline{x}][\overline{n}:\_][\mathrm{V}(POST):\mathrm{V}(POST)]}$
$cp^\mathrm{b}_\mathrm{post};$ (1)
$\{POST_\mathrm{s} * POST_\mathrm{p}[\overline{result} \mapsto \overline{x}]\}_{[\overline{id_\mathrm{arg}}:\overline{id_\mathrm{arg}}][\mathrm{V}(PRE):\mathrm{V}(PRE)][\overline{result}:\overline{x}][\mathrm{V}(POST):\mathrm{V}(POST)]}$
return $\{\overline{result}\}$
$\{POST_\mathrm{s} * POST_\mathrm{p}\}_{[\overline{result}:\overline{result}]}$

(BackOutcallProof)

□

7.3.5 Rules for components and programs. Component well-formedness is identical to well-foundedness in the source language (bar conditions on boundary contracts):

$$\frac{\mathrm{UniqueId}(\overline{itfunc}\ \overline{ctfunc_i}) \quad \mathrm{UniqueId}(\overline{ctfunc_\mathrm{e}}) \quad \mathrm{Subset}(\overline{ctfunc_\mathrm{e}}, \overline{itfunc})}{\vdash^\mathrm{b}_\mathrm{WF} \overline{itfunc}\ //@\mathrm{import}\ \overline{ctfunc_i}\ //@\mathrm{export}\ \overline{ctfunc_\mathrm{e}}} \quad \text{(CompWF)}$$

No import or export statements are present in the result of this rule, as the result of the back-translation of one component is not a full-fledged component: the different resulting component back-translations will be merged into one single back-translated component when the full program is back-translated below. The notation $\overline{\vdash pv_{\mathrm{x}}}$ here denotes that each function in the set has had its universal contract proven individually.

$$\frac{\begin{array}{c} tcomp = \overline{itfunc}\ //@\text{import}\ \overline{ctfunc_i}\ //@\text{export}\ \overline{ctfunc_{\mathrm{e}}} \qquad \vdash^{\mathrm{b}}_{\mathrm{WF}} tcomp \\ \forall x \in \overline{itfunc}.\ x \rightsquigarrow_{\mathrm{b}} \vdash pv_{\mathrm{x}} \end{array}}{tcomp \rightsquigarrow_{\mathrm{b}} \overline{\vdash pv_{\mathrm{x}}}} \quad (\textsc{CompVerif})$$

Program well-formedness is identical to the target language:

$$\frac{\begin{array}{c} \overline{C} = C_1\ \ldots\ C_j\ \ldots\ C_k \\ C_j = \overline{itfunc_j}\ //@\text{import}\ \overline{ctfunc_{\mathrm{i}j}}\ //@\text{export}\ \overline{ctfunc_{\mathrm{e}j}} \\ \forall j \in \{1..k\}.\ \forall l \in \{1..k\}.\ j \neq l \Rightarrow \mathrm{UniqueId}(\overline{itfunc_j}\ \overline{itfunc_l}) \\ \forall j \in \{1..k\}.\ \mathrm{Subset}(\overline{ctfunc_{\mathrm{i}j}}, \overline{ctfunc_{\mathrm{e}1}}\ \ldots\ \overline{ctfunc_{\mathrm{e}k}}) \\ \exists j \in \{1..k\}.\ \mathrm{Subset}((\overline{\tau}\ id()), \overline{ctfunc_{\mathrm{e}j}}) \end{array}}{\vdash^{\mathrm{b}}_{\mathrm{WF}} \overline{C}\ //@\text{main} = id} \quad (\textsc{ProgWF})$$

Program back-translation is different from the case for compilation. One component (the verified component) is now compiled as usual from source to target. On the other hand, the whole back-translation is merged into the same component, together with the back-translated stubs, so that all contracts containing lists and other hard-to-compile-to-stubs elements are only present on contracts of internal functions, and not in the contracts of import or export boundary functions.

Another difference with the case for compilation, is how the main function *id* is handled. During compilation, we were sure that *id* would still be an exported function in the target program, and no countermeasures had to be taken. During back-translation, however, this is not the case, as all components in $\overline{C}$ below are combined in the single source component $\overline{C_{\mathrm{b}}}$, losing their individual import and export statements in the process. This combination is necessary, because we would otherwise obtain boundary functions for C_{b} in the back-translation that do not respect the conditions on boundary function contracts. If *id* corresponds to an exported function of C_{t}, there is no issue, as it is an exported function of C_{s} too. If *id* corresponds to an imported function of C_{t}, it will be exported by some component in $\overline{C}$, since $\vdash^{\mathrm{b}}_{\mathrm{WF}}$ *tprog*, and hence also (as the back-translation of an outcall stub) by C_{b}. Lastly, a problematic case arises when *id* corresponds to a function that is exported by a component in $\overline{C}$, but is not an imported function of C_{t}. Because of the component combination in the back-translation, this function will no longer be exported by C_{b}, it will just be renamed to id_{bt}, causing problems calling the main function in the source language. The solution is to create an artificial incall stub $\overline{\tau}\ id()\ //@\text{pre true}\ //@\text{post true}\ \{\overline{\tau\ id_{\mathrm{prog}}}; \overline{id_{\mathrm{prog}}} = id_{\mathrm{bt}}(); \text{return}\ \overline{id_{\mathrm{prog}}}\}$ (with $\overline{\tau}$ the return type of the back-translated main function id_{bt}) and to add id to the exported functions of C_{b}. The proof of this incall stub with the ImplfVerif axiom is trivial, given one application of the Conseq axiom to prove the contract.

Notice the stub mismatch in the cases where the target main function *id* is exported by a component from $\overline{C}$ but not imported by C_{s}. In this case, an empty incall stub call occurs at the start of execution in respectively the source and target language, but not in the -respectively- target and source language. Given that any (possibly back-translated) main function has true as its precondition and

the incall stub will hence perform no checks, this extra call does not influence equi-termination and hence forms no fundamental issue for the security proof.

The above discussion amounts to the following two rules (where the first rule embodies the problematic case). The judgment $id, C_\mathrm{t}, \overline{C} \rightsquigarrow_{\mathrm{ShouldExportMain}} mainExport, mainStub$ checks if the above conditions for the problematic case are met, and if so, generates a declaration to be exported from C_b and a main function incall stub as described above.

$$\frac{\begin{array}{c}\overline{C} = C_1 \ \ldots \ C_j \ \ldots \ C_k \\ C_j = \_ \ //@\mathrm{import} \ \_ \ //@\mathrm{export} \ \overline{ctfunc_{\mathrm{e}j}} \\ C_\mathrm{t} = \_ \ //@\mathrm{import} \ \overline{ctfunc_\mathrm{it}} \ //@\mathrm{export} \ \_ \\ \exists j.\, (\overline{\tau} \ id()) \in \overline{ctfunc_{\mathrm{e}j}} \qquad \overline{\tau} \ id() \notin \overline{ctfunc_\mathrm{it}} \\ \overline{\tau} \rightsquigarrow_{\mathrm{InvCompileType}} \overline{\tau'} \\ mainExport = \overline{\tau'} \ id() \ //@\mathrm{pre} \ \mathrm{true} \ //@\mathrm{post} \ \mathrm{true} \\ mainStub = mainExport \ \{\overline{\tau' \ id_\mathrm{prog}}; \overline{id_\mathrm{prog}} = id_\mathrm{bt}(); \mathrm{return} \ \overline{id_\mathrm{prog}}\}\end{array}}{id, C_\mathrm{t}, \overline{C} \rightsquigarrow_{\mathrm{ShouldExportMain}} mainExport, mainStub} \quad (\textsc{ExportMain})$$

$$\frac{\begin{array}{c}\overline{C} = C_1 \ \ldots \ C_j \ \ldots \ C_k \\ C_j = \_ \ //@\mathrm{import} \ \_ \ //@\mathrm{export} \ \overline{ctfunc_{\mathrm{e}j}} \\ C_\mathrm{t} = \_ \ //@\mathrm{import} \ \overline{ctfunc_\mathrm{it}} \ //@\mathrm{export} \ \_ \\ \nexists j.\, (\overline{\tau} \ id()) \in \overline{ctfunc_{\mathrm{e}j}} \vee \overline{\tau} \ id() \in \overline{ctfunc_\mathrm{it}}\end{array}}{id, C_\mathrm{t}, \overline{C} \rightsquigarrow_{\mathrm{ShouldExportMain}} \epsilon, \epsilon} \quad (\textsc{NoExportMain})$$

$$\frac{\begin{array}{c}\vdash C_\mathrm{s} \rightsquigarrow C_\mathrm{t} \qquad tprog = C_\mathrm{t} \ \overline{C} \ //@\mathrm{main} = id \qquad \vdash^\mathrm{b}_\mathrm{WF} tprog \\ C_\mathrm{s} = \overline{v} \ //@\mathrm{import} \ \overline{u_\mathrm{i}} \ //@\mathrm{export} \ \overline{u_\mathrm{e}} \\ \forall X \in \overline{C}.\, X \rightsquigarrow_\mathrm{b} \vdash fun_\mathrm{X} \qquad fun_\mathrm{X} = \overline{f} \\ \overline{u_\mathrm{e}} \rightsquigarrow_{\mathrm{BackIncall}} \overline{\vdash stub_{\mathrm{in,b}}} \\ \overline{u_\mathrm{i}} \rightsquigarrow_{\mathrm{BackOutcall}} \overline{\vdash stub_{\mathrm{out,b}}} \\ id, C_\mathrm{t}, \overline{C} \rightsquigarrow_{\mathrm{ShouldExportMain}} main_\mathrm{exp}, main_\mathrm{st} \\ C_\mathrm{b} = \overline{fun_X} \ \overline{stub_{\mathrm{in,b}}} \ \overline{stub_{\mathrm{out,b}}} \ main_\mathrm{st} \ //@\mathrm{import} \ \overline{u_\mathrm{e}} \ //@\mathrm{export} \ \overline{u_\mathrm{i}} \ main_\mathrm{exp}\end{array}}{\vdash C_\mathrm{s}, (\overline{C}, id) \rightsquigarrow_\mathrm{b} \vdash C_\mathrm{s} \ C_\mathrm{b} \ //@\mathrm{main} = id} \quad (\textsc{ProgVerif})$$

The last step in proving the back-translation sound, is the proof that the above ProgVerif rule for the back-translation of a target program actually produces a sound separation logic proof of the source program $sprog = \vdash C_\mathrm{s} \ C_\mathrm{b} \ //@\mathrm{main} = id$. This entails proving that all conditions of the ProgVerif separation logic axiom are met for this output source program. The two conditions are:

- All components have a proof constructed using the CompVerif axiom, ie. (in this case) $\vdash C_\mathrm{s}$ and $\vdash C_\mathrm{b}$. The proof $\vdash C_\mathrm{s}$ is given. The proof $\vdash C_\mathrm{b}$ has to be constructed using the CompVerif separation logic axiom. The proof of every individual function in $\vdash C_\mathrm{b}$ follows immediately from the back-translation, so only $\vdash_\mathrm{WF} C_\mathrm{b}$ (created using the CompWF axiom) remains.

  The condition UniqueId($\overline{isfunc \ \overline{u_\mathrm{e}}}$) holds true because UniqueId($\overline{isfunc}$) holds (for every component C_j in $\overline{C}$ it holds that $\vdash^\mathrm{b}_\mathrm{WF} C_j$, and mutual disjointness follows from $\vdash^\mathrm{b}_\mathrm{WF} tprog$), UniqueId($\overline{u_\mathrm{e}}$)

holds (because $\vdash_{\text{WF}} C_\text{s}$ holds and $\vdash_{\text{WF}} C_\text{b}$ reuses C_s's exported functions as imported functions) and because $\overline{u_\text{e}}$ is disjoint from $\overline{isfunc}$ (the set of function names $\overline{isfunc}$ has bt-subcripts in the function names, except for back-translated outcall stubs, which have the same names as $\overline{u_\text{i}}$, and are hence disjoint from the names in $\overline{u_\text{e}}$ because $\vdash_{\text{WF}} C_\text{b}$).
The condition UniqueId($\overline{u_\text{i}}$) holds true because because $\vdash_{\text{WF}} C_\text{s}$ holds and $\vdash_{\text{WF}} C_\text{b}$ reuses C_s's imported functions as exported functions.
The condition Subset($\overline{u_\text{i}}$, $\overline{isfunc}$) holds true because the function names in $\overline{stub_{\text{out,b}}}$ are exactly $\overline{u_\text{i}}$.
The two $\vdash_{\text{WFBD}}$-conditions follow from the fact that CompWF is known to hold for C_s and the fact that C_b uses C_s's exported functions as imported functions and vice versa. This is exactly the reason why restrictions on boundary contracts were required to be symmetrical before.

- $\vdash_{\text{WF}}$ *sprog*: reasonably simple from the ProgWF axiom because there are only 2 components.
The first condition, UniqueId, holds because C_s only contains function names with bt-subscripts in the function names, apart from the back-translated outcall stubs. The outcall stub names are imported by C_s and hence not part of C_s's function names (because C_s is well-formed).
The Subset condition on the imported function immediately holds for C_b and C_s , because they import each other's exported functions and vice versa, and both components are well-formed (cfr. previous bullet).
Lastly, the back-translated main function *id* is either exported by C_s or by C_b, as explained above the ProgVerif back-translation rule.

7.4 Simulation

A new security simulation relation S has to be defined to specify a simulation between a target level component and its back-translation, just like we did for a source-level component and its compilation in the correctness part of the proof. A major difference with the correctness proof, however, is that the execution can now shift from the back-translated to the verified component or vice versa. We still use the simulation relation $R_\text{c}omp$ for the verified component and its compilation. The proofs will hence need to include a proof that stubs (incall or outcall depending on the situation) allow the shifting between the component-wise correctness relation $R_\text{c}omp$ and the component-wise security relation S_comp, while preserving simulation. This also means that we have to prove that stubs and their back-translation either both get stuck during execution of some guard, or they both continue running (we implicitly proved this for the correctness proof, but there were no source-level stubs there and all we had to prove was that the target-level stubs would never get stuck in the absence of a malicious attacker).

The security simulation relation S will now be defined, and its definition will be followed by the proof that it is indeed a simulation relation, a fact that is used to construct an equi-termination argument by simulation later on. The relation S is again indexed by the structures *sprog* and *tprog*, that non-ambiguously define the source code in source and target, as we need this source code for the FApp rule in the operational semantics. S is defined as follows: (again, like R, on components).

$$
\begin{aligned}
&sprog, tprog \vdash (\langle s_\text{s}, h_\text{s}\rangle \mid s)\ S\ (\langle s_\text{t}, h_\text{t}\rangle \mid t) \Leftrightarrow \\
&\qquad tprog = C_\text{t}\ \overline{C}\ \text{//@main} = id \\
&\qquad sprog = C_\text{s}\ C_\text{b}\ \text{//@main} = id \\
&\qquad \vdash C_\text{s} \rightsquigarrow C_\text{t}
\end{aligned}
$$

$$\vdash C_s, (\overline{C}, id) \rightsquigarrow_b \vdash sprog$$

Initial case

$$t = tprog,\ s = sprog$$
$$h_s = h_t = \bullet, s_s = s_t = \epsilon$$

Executing case

$$s = \overline{s_i},\ t = \overline{t_i},\ s_s = \overline{s_{si}},\ s_t = \overline{s_{ti}}$$
$$\biguplus \overline{h_{si}} = h_s,\ \biguplus \overline{h_{ti}} = h_t,$$
$$\mathrm{Loc}(b, \overline{s_{si}}, h_s, \overline{s_{ti}}, h_t)$$
$$\forall i \in \{1, \ldots, k\}.\ sprog, tprog \vdash_b^{x_i} (\langle s_{si}, h_{si}\rangle \mid s_i)\ \Delta_i\ (\langle s_{ti}, h_{ti}\rangle \mid t_i)$$

Frame Order

$$\overline{\Delta_i} = (\epsilon \mid Incall)\ R_{\mathrm{Frames}} \mid (\epsilon \mid Outcall)\ S_{\mathrm{Frames}}$$
$$R_{\mathrm{Frames}} = {R_{\mathrm{comp}}}^+\ (\epsilon \mid Outcall\ S_{\mathrm{Frames}})$$
$$S_{\mathrm{Frames}} = {S_{\mathrm{comp}}}^+\ (\epsilon \mid Incall\ R_{\mathrm{Frames}})$$

Inverse environment

$$\mathrm{InverseMap}(\delta_i, P_i, \gamma_i, s_{si}, h_{si}) \text{ given } s_i = \vdash \{P_i\}_{\gamma_i}\ s_{ci}\ \{Q_i\}_{\gamma_i'}$$
$$(\Delta_i = R_{\mathrm{comp}})\ ?\ x_i = \delta_i : x_i = \epsilon$$

Separation logic linking

$$i > 0 \Rightarrow \mathrm{Link}(s_{i-1}, s_i, \delta_{i-1}, \delta_i)$$

The only difference between the separation logic linking conditions in S above compared to in R is the generalization of δ_i to all kinds of stack frames. δ_i is defined in exactly the same way in all 4 types of relations as it was in R_{comp} and allows for argument linking.

7.5 Assertion Semantics

The judgment $h_s \vdash_\delta^{\mathrm{src}} P$ used for frame linking is reused from the correctness part of the proof above.

7.6 Component simulation relations

We define another auxiliary function $\mathrm{invvalmap}_b$, to map target to source values, defined as follows (with b again some bijection between locations, but now in the target-to-source direction):

$$\mathrm{invvalmap}_b : \mathrm{VAL}_t \to \mathrm{VAL}_s$$
$$\mathrm{invvalmap}_b(k) = k$$
$$\mathrm{invvalmap}_b(\mathrm{null}_0) = \mathrm{null}_0$$
$$\mathrm{invvalmap}_b(l_0^i) = (b(l), i)$$
$$\mathrm{invvalmap}_b(\mathrm{null}) = (\mathrm{null}_0, 0)$$
$$\mathrm{invvalmap}_b(l^{[0,n]}) = ((b(l), 0), n+1)$$
$$\mathrm{invvalmap}_b((v_1, \ldots, v_k)) = (\mathrm{invvalmap}_b(v_1), \ldots, \mathrm{invvalmap}_b(v_k))$$

Just like for valmap_b an important property of invvalmap_b is its compositionality:

Theorem 8 (Compositionality)**.**
The function invvalmap_b *is compositional. We define this as follows: if* $s_s(\overline{id_s}) = \text{invvalmap}_b(s_t(\overline{id_t}))$ *for some sets of source and target variables* $\overline{id_s}$ *and* $\overline{id_t}$ *, then for any parametrized target expression* $texp[\overline{id_t}]$*, it holds that* $[\![texp_b[\overline{id_s}]]\!]_{s_s} = \text{invvalmap}_b([\![texp[\overline{id_t}]]\!]_{s_t})$*. In other words, applying the same expression to variables related by valmap, keeps the results related.*

Proof.

Almost trivial case-based analysis on the form of target expressions *texp*. However, back-compiled expressions are not identical to their target counterpart, as we discussed earlier in section 7.2.3. We therefore quickly check the compositionality cases for the length and addr functions (defined on linear capabilities only) here:

- length is back-translated to the second component projection .2. Checking the linear capability cases in the definition of the invvalmap_b function (ie. the null and the non-null case), we can immediately see that the capability's length and the second coordinate of its back-translation correspond, and are hence related by the integer case of invvalmap_b.
- addr is back-translated to the projection .1. The argument is identical to the one made in the previous bullet, but now we have correspondence through the zero-length-capability case in the definition of invvalmap_b (with index $i = 0$).

□

The reason for the use of invvalmap_b instead of the earlier valmap_b here is that $\text{invvalmap}_b \circ \text{valmap}_b = id_f$, but $\text{valmap}_b \circ \text{invvalmap}_b \neq id_f$, because of $\tau*$-types in the target, which cannot be the result of compilation and hence cannot be passed from S_{comp} to R_{comp} frames. This left-inverse implies that invvalmap_b also holds for the R-frames, whereas, valmap_b doe not hold for the S frames per se.

The relation S_{comp} is noticeably simpler than R_{comp}, because the back-translated code mimics the target code as tightly as possible, whereas some semantics shifts do happen during compilation.

$sprog, tprog \vdash_{b^{-1}} (\langle s_s, h_s\rangle \mid s)\ S_{\text{comp}}\ (\langle s_t, h_t\rangle \mid t) \Leftrightarrow$

Notice that we **inverted the meaning** of b for this relation: $\vdash_{b^{-1}}$ above! This is possible because b is a bijection. This interpretation fits better with the target-to-source theme of the back-translation.

$t \rightsquigarrow_b s$

$s = \Sigma^{ax} \vdash \{P\}_\gamma\ sstm; \text{return } \overline{sexp}\ \{Q\}_{\gamma'} \wedge \Sigma^{ax} \in \Sigma^{ax}_{sprog}$

Relating target to source

$$\underline{h_s} \approx \underline{h_t}$$
$$\forall l.\, h_t(l) = [v_1, \ldots, v_m]$$
$$\Leftrightarrow h_s(b(l)) = [\text{invvalmap}_b(v_1), \ldots, \text{invvalmap}_b(v_m)]$$
$$\underline{s_s} \approx \underline{s_t}$$
$$\text{dom}(s_s) = \text{dom}(s_t) \uplus \overline{id_{\text{aux}}}$$
$$\forall id \in \text{dom}(s_t).\, s_s(id) = \text{invvalmap}_b(s_t(id))$$

Universality

Notice that P, Q, γ and γ' are of the form described in the Universal Contract definition in subsubsection 7.2.2, bar any (possibly auxiliary) variables (and their universal contracts) that are framed off by an outer Frame rule. This fact automatically follows from $t \rightsquigarrow_b s$ above. P and γ have the following form (including the framing):

$$\gamma = [\text{dom}(s_t) : \overline{id_{\text{univ}}}][\overline{id_{\text{aux}}} : \_] \quad \text{UniqueId}(\overline{id_{\text{univ}}})$$
$$P = *\{\text{univ\_contr}_\tau(\gamma(id_{\text{prog}}))|$$
$$id_{\text{prog}} \in \text{dom}(s_t) \wedge \text{TypeOfVar}_t(id_{\text{prog}}) = \tau\}$$

An important caveat is the following: since linear capabilities are the program-level embodiment of resources, ie. heap permissions, we would expect any linear capability value $l^{\lceil} a, b]$ to cause indexes $[a, b]$ to be present at location l in the heap of the current frame during the simulation. This condition is not explicitly enforced above. However, it *is* enforced indirectly! The condition $h_s \vdash^{\text{src}}_{\delta} P$ used to define δ, that has to hold over every frame, forces the universal contracts of each back-translated target variable to correspond with the source heap. Since both the source and target heap and the source and target stack are related by invvalmap in the definition of S_{comp}, the aforementioned condition on the correspondence of linear capabilities in the target-and the target-level heap must hold as well.

We now define relations *Incall* for incall stubs and *Outcall* for outcall stubs. These are the missing links that allow transitioning between simulation in S_{comp} and in R_{comp}.

We define the *Incall* relation as follows:

Note that we do not constrain the set $\overline{id_{\text{arg}}}$ to be the same set that the hole in the previous R_{comp} frame is annotated with. Having different sets of arguments is no problem, because frame linking will enforce correspondence of concrete program values for us. The same happens for the set of variables declared by d_{pre} in outcall stubs: we do not have to force these variables to have concrete values in the Outcall outcall relation, as frame linking will enforce any relevant values.

$$sprog, tprog \vdash_b (\langle s_s, h_s\rangle \mid s)\ \mathit{Incall}\ (\langle s_t, h_t\rangle \mid t) \Leftrightarrow$$
$$f_{\text{sdecl}} = \{\overline{\tau_{\text{ret}}}\}\ f(\overline{\tau_{\text{arg}}\ id_{\text{arg}}})$$
$$//@\text{pre}\ PRE_s * PRE_p\ //@\text{post}\ POST_s * POST_p$$
$$f_{\text{sdecl}} \text{ is an exported function of the verified component } C_s \text{ of } sprog$$

Applying the BackInCall back-translation rule to f_{sdecl} and reusing the names from this rule with the same values, we can construct the following two partially executed function bodies (they have the same form as a partially executed incall stub and its back-translation, but we will not enforce that relation, because it is needlessly complicated):

$$t = \left\{ \begin{array}{l} \{\overline{result}, \overline{n}\} = \bullet^{\overline{id}=(\overline{id_{\text{arg}}}, \overline{m})}; \\ \text{return } \{\overline{result}, \overline{n}\} \end{array} \right\}$$

$$s_{\text{uvf}} = \left\{ \begin{array}{l} \{\overline{result}\} = \bullet^{\overline{id}=(\overline{id_{\text{arg}}})}_{POST_{\text{s}} * POST_{\text{p}}}; \\ a_{\text{la}}; \\ \text{ArrayToRange}(\overline{n.1}, \overline{n.2}) \\ \text{return } \{\overline{result}, \overline{n}\} \end{array} \right\}$$

$\forall id \in \overline{id_{\text{arg}}}.\ s_{\text{t}}(id) = \text{valmap}_b(s_{\text{s}}(id))$

$\gamma(\overline{id_{\text{arg}}}) = \overline{id_{\text{arg}}}$

with γ the environment appearing in the precondition of the triple s. The previous constraint is not strictly needed, but simplifies the coming proofs somewhat, because the backtranslation soundness proof uses the fact that all argument variables map to logical variables of the same name, and it will allow more seamless reuse of facts from these proofs.

$\overline{result}, \overline{n} \in \text{dom}(s_{\text{s}}), \text{dom}(s_{\text{t}}), \text{dom}(\gamma)$

Reusing the BackIncall soundness proof from theorem 7 from the point of the function call to f onwards; where we replace f by the above source-level hole $\bullet$ (with its appropriate contract), we can (given the above constraints on γ and omitting any other non-interesting parts of γ) now trivially prove the following triple:

$$s = \{\text{true}\}_{[\overline{id_{\text{arg}}}:\overline{id_{\text{arg}}}][\overline{result}:\_][\overline{n}:\_]}\ s_{\text{uvf}}$$
$$\{\text{univ\_contr}_{\overline{\tau^{\text{t}}_{\text{ret}}}, \overline{\tau^{\text{t}}_{\text{post}}}}(\overline{result}, \overline{result_n})\}_{[\overline{result}:\overline{result}][\overline{n}:\overline{result_n}]}$$

Very analogously, although slightly more complicated because the values of some variables have to be stored before the function outcall and remembered, we can define the Outcall relation as follows:

$sprog, tprog \vdash_b (\langle s_{\text{s}}, h_{\text{s}}\rangle \mid s)\ Outcall\ (\langle s_{\text{t}}, h_{\text{t}}\rangle \mid t) \Leftrightarrow$

$f_{\text{sdecl}} = \{\overline{\tau_{\text{ret}}}\}\ f(\overline{\tau_{\text{arg}}\ id_{\text{arg}}})$

$//@\text{pre } PRE_{\text{s}} * PRE_{\text{p}}\ //@\text{post } POST_{\text{s}} * POST_{\text{p}}$

f_{sdecl} is an imported function of the verified component C_{s} of $sprog$

Applying the BackOutCall back-translation rule to f_{sdecl} and reusing the names from this rule with the same values, we can construct the following two partially executed function bodies (they have the same form as a partially executed outcall stub and its back-translation, but we will not enforce that relation, because it is needlessly complicated):

$$t = \left\{ \begin{array}{l} \{\overline{result}, \overline{n}\} = \bullet^{(\overline{id_{\text{arg}}}, \overline{m})}; \\ cs_{\text{post}}; \\ d_{\text{post}}; a_{\text{post}}; cp_{\text{post}}; \\ \text{return } \{\overline{result}, \overline{n}\} \end{array} \right\}$$

$$s_{\text{uvf}} = \left\{ \begin{array}{l} \{\overline{result}, n\} = \bullet^{(\overline{id_{\text{arg}}}, m)}_{\text{POST}(f_{\text{bt}})}; \\ cs^{\text{b}}_{\text{post}}; \\ \text{RangeToArray}(\overline{n.1}, \overline{n.2}) \\ d^{\text{b}}_{\text{post}}; a^{\text{b}}_{\text{post}}; cp^{\text{b}}_{\text{post}}; \\ \text{return } \{\overline{result}\} \end{array} \right\}$$

We reuse the set of all variables introduced in the precondition PRE_{s} called $\text{V}(PRE)$, from the back-translation soundness proofs. This is actually an overestimate, since we could be using the subset $\text{V}(PRE_{\text{used}}) = (\text{V}(POST_{\text{p}}) \setminus \text{V}(POST_{\text{s}})) \setminus \overline{id_{\text{arg}}}$, but it makes formalization slightly more uniform and causes no other problems.

$\forall id \in \overline{id_{\text{arg}}}, \text{V}(PRE).\ s_{\text{t}}(id) = \text{valmap}_b(s_{\text{s}}(id))$

$\gamma(\overline{id_{\text{arg}}}) = \overline{id_{\text{arg}}} \qquad \gamma(\text{V}(PRE)) = \text{V}(PRE)$

with γ the environment appearing in the precondition of the triple s. The previous constraint is not strictly needed, but simplifies the coming proofs somewhat, because the back-translation soundness proof uses the fact that all argument and precondition variables map to logical variables of the same name, and it will allow more seamless reuse of facts from these proofs.

$\overline{result}, \overline{n} \in \text{dom}(s_{\text{s}}), \text{dom}(s_{\text{t}}), \text{dom}(\gamma)$

Reusing the BackOutcall soundness proof from theorem 7 from the point of the function call to f onwards; where we replace f by the above source-level hole $\bullet$ (with its appropriate contract), we can (given the above constraints on γ and omitting any other non-interesting parts of γ) now trivially prove the following triple:

$$s = \{\text{true}\}_{[\overline{id_{\text{arg}}}:\overline{id_{\text{arg}}}][\text{V}(PRE):\text{V}(PRE)][\overline{result}:\_][\overline{n}:\_]}\ s_{\text{uvf}}\ \{POST_{\text{s}} * POST_{\text{p}}\}_{[\overline{result}:\overline{result}]}$$

7.7 Simulation proofs

Analogous to the case for correctness, this section proves the BTEquiTermination rule through the properties of the different simulation relations we previously described. We first dedicate a subsection to proving the building blocks needed for the proof of BTEquiTermination and only then proof this rule in the next subsection.

7.7.1 Auxiliary simulation proofs. Just as we did for the proof of the SimToEquiTermin inference rule during the correctness proof, we will -in this section and further on- again assume that the back-translation and the back-translated incall and outcall stubs produce code where program-integrity errors (ie. dynamic typing errors, scoping issues, array index out of bounds errors, etc.) appear simultaneously in both source- and target language or do not appear at all. This means that

if we know that either the source or target program terminates, we know that the other one can never get stuck on these types of program errors (it cán get stuck on regular guard statements!). We do not explicitly prove this assumption, but think it is a reasonable one to make, given how the back-translation combines with the source-semantics to mimic the target semantics (this is the central idea of the back-translation), including eg. emulating linearity of capabilities, keeping array lengths the same as in the target, adding all necessary guard to make the back-translated code get stuck when the target code does, etc. For any non-trivial ways to get stuck, the proofs below will of course still contain proofs of equi-failure.

Simulation for $\overline{S}_{\text{comp}}$. Just like we did for the R-relation in the previous section (a proof that we will adapt to our current setting in the next paragraph), we now prove that S where $\forall i.\ \Delta_i == S_{\text{comp}}$ is a simulation relation. We denote this specific restriction of S as $\overline{S}_{\text{comp}}$ and define the equivalent for R_{comp} frames only, ie. $\overline{R}_{\text{comp}}$. We look at the setting where we only have back-translated components and no verified components, so S_{comp} frames only, because we want to prove that both $\overline{R}_{\text{comp}}$ and $\overline{S}_{\text{comp}}$ allow simulation between source and target before we look at transitions between them through stubs. The theorem below is analogous to how we analyzed R being a simulation relation in the absence of stubs in the previous section.

Theorem 9 ($\overline{S}_{\text{comp}}$ is a simulation relation)**.**
The following 2 properties of $\overline{S}_{\text{comp}}$ *hold:*

(1) *If* $(\langle s_{\text{s}}, h_{\text{s}}\rangle \mid s)\ \overline{S}_{\text{comp}}\ (\langle s_{\text{t}}, h_{\text{t}}\rangle \mid \text{return})$, *then* $s = \{P\}\ \text{return}\ \{Q\}$. *This rule is used to garantee equi-termination in the proof, when we know that the target program terminates*

(2) *Whenever* $s_2\ \overline{S}_{\text{comp}}\ s_1$ *and* $s_1 \hookrightarrow s_1'$ *(using the shorthand state notation), there exists a* s_2' *such that* $s_2 \hookrightarrow^+ s_2'$ *and* $s_2'\ \overline{S}_{\text{comp}}\ s_1'$. *We again need a semantics to operationally manipulate separation logic triples instead of regular source expressions. This requires a kind of combined semantics, in nature just like the lifted correctness semantics, but different, as detailed below.*

For the lifted operational semantics in the previous theorem, we should use the same style of semantics as in the preservation theorem 4 in the correctness section, but now perform multiple steps at a time in $\hookrightarrow$ (because the intermediate steps do not have minimal contracts). This means that we could actually write $s_2 \hookrightarrow s_2'$ in the above theorem, since all steps will be combined into one anyway.

We still need to prove that any intermediate states are valid separation logic proofs, but this is trivial. The definition of these operational semantics is trivial, as each step just corresponds to the reduction of one back-translated block (so we know the postcondition Q is also a minimal contract) to 'skip' and transforms the minimal contract in the precondition to the one in the postcondition of the executed block the next one according to this block. Let us illustrate with the Malloc lifted semantics rule as an example:

$$\dfrac{\begin{array}{c}\gamma = [\overline{\text{dom}(s_{\text{s}})} : \overline{\text{dom}(s_{\text{s}})}] \\ c = (\text{guard}(texp_{\text{b}} > 0); \\ \tau' * n_{\text{aux}}; n_{\text{aux}} = \text{malloc}(texp_{\text{b}} * \text{sizeof}(\tau')); n = (n_{\text{aux}}, texp_{\text{b}}))\end{array}}{\{\text{univ\_contr\_pre}\}_\gamma\ c\ \{\text{univ\_contr\_post}\}_\gamma}\ (\textsc{Malloc})$$

$$\hookrightarrow_{(\textsc{Malloc})} \dfrac{}{\{\text{univ\_contr\_post}\}_\gamma\ \text{skip}\ \{\text{univ\_contr\_post}\}_\gamma}\ (\textsc{Skip})$$

The exact code c comes from the Malloc back-translation rule, and the fact that the universal contract proof exists is proven there as well. The same schema can be followed to define every rule of our lifted operational semantics. Note that we will never need a lifted operational semantics rule to execute a single malloc statement on its own, as the back-translation will never produce such a statement.

The only exceptions to the above schema are If-statements, For-statements, sequencing with skip and the ProgExec rule (but all are simple rules). These cases are handled similarly to the lifted operational semantics in the correctness proof. Just as in the correctness case, we disregard the issues of hoisting for the If and For rule. For the If rule, we just ignore the fact that different branches can now have different environments, since this clearly does not influence execution. In the For rule, we again employ the alternative (but clearly equivalent) operational semantics where fresh resource names are created in the source proof during a loop unroll, and fresh variable names are created in the target, so that there is no need for hoisting.

In conclusion, we actually made a security version of theorem 4, that introduced the operational semantics for the compilation, and proved an analogous result, ie. that the proof tree of a back-translated program can soundly be manipulated along with the back-translated program itself.

Given this definition of the lifted semantics, we can now prove that $\overline{S}_{\text{comp}}$ is indeed a valid simulation relation, ie. the equivalent of the proof of theorem 2, but now for theorem 9 above.

Proof of theorem 9 property 1.

The first case is again trivial and just follows from the definition of the back-translation and of S_{comp}. □

Proof of theorem 9 property 2.

For the second case, the proof is again structured in a case-based fashion, in the same way as the proof of theorem 2 in the correctness section. The proof will contain a lot of similarities to the latter proof. Applications of the Frame rule are again left implicit in the proof; they are stripped from the back-translated code when encountered. Unlike for correctness, applications of Conseq need not be erased, since the back-translation did not require applications of Conseq outside of back-translated code blocks.

The proof is again structured in a case-based fashion, in the same way as the proof of the preservation theorem. The trees for $\overline{s_i'^p}$ from the preservation theorem are reused here, as is the notation.

Every case of the proof consists of three consecutive proof steps:

(1) First, the lifted operational semantics rule that can be applied on s is derived from the fact that $t \rightsquigarrow_{\text{b}} s$ and inversion. This step is kept implicit, because the rule in source and target language will always match.

(2) The fact that the lifted operational semantics code is applicable (the preconditions of each back-translated statement are upheld in the current program state) to s is proven. This entails that for each rule, we will have to investigate the applicable back-translation rule, check what code is generated, and make sure this code is executable (ie. the lifted operational semantics step can be taken) in the current state given that $(\langle s_{\text{s}}, h_{\text{s}}\rangle \mid s)\ \overline{S}_{\text{comp}}\ (\langle s_{\text{t}}, h_{\text{t}}\rangle \mid t)$ holds. We can

safely ignore the introduced *stuck* code in some of the back-translation rules, since the target could not take a step if the same linear capability were used multiple times in a single statement.

(3) Given the source and target level operational semantics steps that are taken, namely $\langle s_s, h_s\rangle \mid s \hookrightarrow \langle s_s', h_s'\rangle \mid s'$ and $\langle s_t, h_t\rangle \mid t \hookrightarrow \langle s_t', h_t'\rangle \mid t'$, it is proven that if $(\langle s_s, h_s\rangle \mid s)\ \overline{S}_{\text{comp}}\ (\langle s_t, h_t\rangle \mid t)$, then also $(\langle s_s', h_s'\rangle \mid s')\ \overline{S}_{\text{comp}}\ (\langle s_t', h_t'\rangle \mid t')$.

Skip

Prove that the rules are applicable:
Trivially true, since there are no conditions for applying the Skip rule.

Prove that $\overline{S}_{\text{comp}}$ still holds after the step:
Given that $t \rightsquigarrow_b s$ holds, where $t = \text{skip}; t'$ and $s = \text{skip}; s'$, implies that $t' \rightsquigarrow_b s'$ will still hold after executing Skip in the target language. Since the heap and the stack are unaltered in both target and source language, $\overline{S}_{\text{comp}}$ obviously still holds.

Malloc

Prove that the rules are applicable:
We examine the code generated by the Malloc back-translation rule. The check guard($texp_b > 0$) will not fail, since $texp > 0$ must hold in the source (otherwise, it could not have executed Malloc), given that s_s and s_t are related by invvalmap and given compositionality of invvalmap. The fact that these checks will not fail often just derives from the fact that the target code s can execute, and some precondition in the target-level operational semantics. The other statement's preconditions are then easily met.

Prove that $\overline{S}_{\text{comp}}$ still holds after the step:
The new malloced location l_s is set to map to the target-malloced location l_t in b. The assigned variable n then clearly relates with itself through invvalmap in the source and target stack (given compositionality). The heaps are still related through invvalmap, since default values are generated in the malloced location, and these are related through invvalmap.

Ghost Commands Only Split and Join exist in the target-language operational semantics. Flatten and Collect can hence be disregarded.

Prove that the rules are applicable:
Again, all guard statements in the back-translation must hold, given that the target language can take a step (ie. the precondition of the ghost statement holds) and that source and target stack and heap are related by invvalmap.

Prove that $\overline{S}_{\text{comp}}$ still holds after the step:
It is easy to see that the back-translated assignments parallel the behavior of the target-level built-in functions join and split. $\rightsquigarrow_{\text{EmulateNulling}}$ makes sure that the old values still correspond between the source and target language as well.

IfTrue

Prove that the rules are applicable:
Prove that if $[\![texp]\!]_s$ = true in the target, then the same holds in the target because of compositionality, so the IfTrue rule is the only one that can be applied there.

Prove that $\overline{S}_{\text{comp}}$ still holds after the step:
Nothing significant changes. A Frame rule might be discarded, but this cannot make δ invalid.

IfFalse
Analogous to IfTrue.

VarDecl
Prove that $\overline{S}_{\text{comp}}$ still holds after the step:
This step is trivial, because the variables will receive default values, related by invvalmap, given how $\rightsquigarrow_{\text{InvCompileTypes}}$ maps target types to source types and the definition of $\rightsquigarrow_{\text{def}}$. The appropriate new associates have to be added to b and δ.

VarAsgn
Prove that $\overline{S}_{\text{comp}}$ still holds after the step:
We have to show that $[\![sexp]\!]_s$ and $[\![sexp_{\text{b}}]\!]_s$ have invvalmap-corresponding values in the target and source language. This follows directly from compositionality. $\rightsquigarrow_{\text{EmulateNulling}}$ makes sure that the rest of the stack is still related by invvalmap.

FApp
During an FApp call, a new frame is created in the target language. We have to prove that the same happens in the source language, that both the last already existing frame and the newly created frame are related by S_{comp} and that frame linking holds between them, so that the entire execution is still valid under $\overline{S}_{\text{comp}}$.

Prove that the rules are applicable:
Trivial, since there are no real requirements to call a function in the source (except for the function existing, which is obviously true given that the source code is back-translated from the target code).

Prove that $\overline{S}_{\text{comp}}$ still holds after the step:
First of all, we prove that S_{comp} holds in the new frame. The expressions $\overline{texp}$ and $\overline{texp_{\text{b}}}$ provided to f and f_{bt} in respectively the target and source languages are related by invvalmap, since both stack frames are and compositionality holds. Additionally, we provide the next frame with the part of the minimal heap that satisfies $h_{\text{s}} \vdash^{\text{src}}_{\delta}$ univ_pre. This heap must exist, since S_{comp} holds in the caller frame. We hand the corresponding heap h_{t} over in the target language, so that both heaps are related by invvalmap. The δ for the new frame is derived from the stack-values of the provided source expressions $\overline{texp_{\text{b}}}$, and retains the values of the previous frame for any other logical variables that were transferred. The mapping b remains unaltered, since no new locations are created.

Now we prove that S_{comp} still holds in the old frame. Related stack and heap values migrated to the new frame, so these two correspondences still hold (given that the source language also emulates linear capability erasure). The precondition and heap that were consumed cannot cause problems for the condition $h_{\text{s}} \vdash^{\text{src}}_{\delta}$ univ_pre, since they corresponded to each other. No new logical variables were declared, so δ did not change.

Frame linking is immediately satisfied from the fact that the new frame derived its δ from the previous one.

Return
During execution of a Return statement, the last frame is destroyed in the target language. We have to prove that the same happens in the source language and that the new last frames in the source and target languages are related by S_{comp}, so that the entire execution is still valid under $\overline{S}_{\text{comp}}$.

Prove that the rules are applicable:
Trivial, since there are no real requirements to return from a function in the source.

Prove that $\overline{S}_{\text{comp}}$ still holds after the step:
The expressions that are returned in source and target correspond through invvalmap, and hence so will the newly assigned return values in the previous stack frame after the function call. Since $h_s \vdash^{\text{src}}_{\delta} P$ holds in the next frame, before the return statement, and universal contracts are compositional (as proven during the back-translation), this same condition will also hold over the postcondition univ_post, ie. $h_s \vdash^{\text{src}}_{\delta}$ univ_post. The full source and target heaps from the next frame, are then just transferred to the previous one along with this produced postcondition univ_post, meaning that $h_s \vdash^{\text{src}}_{\delta}$ univ_pre will still hold for the resulting heaps and produced postcondition and that the resulting heaps will still correspond through invvalmap in the previous frame. The function δ is adapted for the new returned values, but this does not influence frame linking, and b again remains identical.

ArrayMut
We can ignore most of the back-translated code, as this is just related to the separation-logic proof (and some guard statements that obviously holds, similarly to the Malloc case) Again uses the correspondence, caused by compositionality, between the source and target values of evaluated expressions to prove that the resulting heaps will still be related.

ArrayLkup
We can ignore most of the back-translated code, as this is just related to the separation-logic proof (and some guard statements that obviously holds, similarly to the Malloc case) Uses the correspondence between the source and target heaps to prove that the new stack frames will correspond as well.

ForUnroll
The operational semantics rule ForUnroll consists of nothing more than unrolling the for loop, in both the source and target languages.

As was the case for the correctness proof, we again use the alternative schema (as mentioned above) which ignores hoisting.

Prove that the rules are applicable:
Since the expressions *texp* and *texp′* in the target ForUnroll rule are equal to *sexp* and *sexp′* in the source, since invvalmap holds over the source and target stack frames due to $\overline{S}_{\text{comp}}$ and given the compositionality of invvalmap, it holds that both for loops will either be unrolled to corresponding bodies or neither one will be.

Prove that $\overline{S}_{\text{comp}}$ still holds after the step:
This is trivial, since simply unrolling the for loop does not change any state.

GuardTrue
Trivial through compositionality.

ProgExec
Trivial, because stack and heap are empty in the produced source and target code, and so are the symbolic heap and the arguments, since main functions are without arguments.

□

The previous proof makes the following inference rule almost trivial to prove (in the pure back-translation setting):

$$\frac{(\langle s_s, h_s\rangle \mid s)\ \overline{S}_{\text{comp}}\ (\langle s_t, h_t\rangle \mid t)}{s \Downarrow \Leftrightarrow t \Downarrow} \qquad \text{(SCompToEquiTermin)}$$

The proof makes use of the fact that $\overline{S}_{\text{comp}}$ is a simulation relation and is very similar to the proof of SimToEquiTermin we made in the correctness section.

Proof.

Right to left direction
A case-based analysis on the compilation rules shows that the source code's lifted semantics, described above, takes a single lifted operational semantics step for each step the target code takes. If the target code terminates, it ends in a return statement, and so does the source code in that case (by the first result of Theorem 9 and assuming it does not get stuck during execution). The source program hence terminates as well.

Left to right direction
Starting from the current (terminating) source program s, either the corresponding target program t

- is stuck. We have to prove that this case cannot occur. We have to prove that if the source program does not get stuck during its execution, then neither does the target program that it was back-translated from, or, by contraposition, that if the target program gets stuck during execution, then so does the source program. This intuitively follows from the fact that all target-language checks are either reified during compilation or present in the source-language operational semantics.
- has terminated in a single return statement (the only statement a non-stuck program can naturally terminate in).
- can take a step, and hence (by the second result of Theorem 9) there is some number n (in fact, 1) of steps that s can take to s', which correspond to one step taken from t to t'. The target program takes at most as many steps as the terminating source program. Using determinacy of source language semantics and the second result of Theorem 9, the resulting programs s' and t' are still related by $\overline{S}_{\text{comp}}$.

This case-based analysis is repeated until termination (which will happen, since we know that t terminates). □

It is implicit in the proof of the previous rule, but again important for the sanity of our back-translation, that stuck code is back-translated to stuck code, and never to diverging code, as stated in the assumptions at the start of this section.

Simulation for $\overline{R}_{\text{comp}}$. As mentioned above, we can define a relation $\overline{R}_{\text{comp}}$, similar to $\overline{S}_{\text{comp}}$, that is a restriction on the relation S, where only R_{comp} frames occur. Notice that this relation $\overline{R}_{\text{comp}}$ in fact exactly corresponds to the relation R we defined during the correctness proof (since we also used a no-stubs version of compilation $\rightsquigarrow_{\text{NoStubs}}$ there). The following result is hence immediate from the correctness proof:

$$\frac{(\langle s_\mathrm{s}, h_\mathrm{s}\rangle \mid s)\ \overline{R}_\mathrm{comp}\ (\langle s_\mathrm{t}, h_\mathrm{t}\rangle \mid t)}{s \Downarrow \Leftrightarrow t \Downarrow} \qquad \text{(RCompToEquiTermin)}$$

Where it is again implicit that there is equi-stuckness between code and its compilation, ie. the compiler will not relate stuck code to diverging code or vice versa.

Incall and Outcall are sound. Now we prove that the transformation between S_comp and R_comp frames in the back-translated in- and outcall stubs works, ie. they perform their bridging function properly. More concretely, we prove theorems 11 and 12 below. Theorem 12 is very similar to theorem 11, but with the roles of R_comp and S_comp swapped, and each mention of Incall swapped with Outcall. Their proofs are similar as well and hence not entirely repeated.

In the proofs, we will make use of the following property, derived from the compositionality of $\mathrm{valmap}_\mathrm{b}$:

Theorem 10 (Guard equi-failing)**.**
Given a target-level guard statement $\mathrm{guard}(texp)$ *containing no variables of pointer type (ie. linear capabilities). If, for all variables* $\mathrm{V}(texp)$*, it holds in the current frame that* $s_\mathrm{s}(\mathrm{V}(texp)) == \mathrm{invvalmap}_{b^{-1}}(s_\mathrm{t}(\mathrm{V}(texp))$ *(or, equivalently using* valmap_b*, since they are each other's inverse in the absence of target-level pointers), then the guard statements* $\mathrm{guard}(texp)$ *and* $\mathrm{guard}(texp_\mathrm{b})$ *equi-fail, ie. either both succeed or both fail.*

Proof. Straightforward from the compositionality property of $\mathrm{invvalmap}_{b^{-1}}$, which gives us that $[\![texp]\!]_{s_\mathrm{s}} == \mathrm{invvalmap}_{b^{-1}}([\![texp_\mathrm{b}]\!]_{s_\mathrm{t}})$. Both expressions must be of int type to be well-typed, and hence equal to the same integer k, as seen from the integer case in the definition of $\mathrm{valmap}_\mathrm{b}$. Given the same integers in source and target, both guard statements will obviously equi-fail. □

Theorem 11 (Incall stub bridging)**.**
Bridging pre function call: *Given a program execution such that*

$$sprog, tprog \vdash (\langle s_\mathrm{s}, h_\mathrm{s}\rangle \mid s)\ S\ (\langle s_\mathrm{t}, h_\mathrm{t}\rangle \mid t)$$

consisting of n frames, and where the nth frame is an S_comp *frame*

$$sprog, tprog \vdash_{b^{-1}} (\langle s_{\mathrm{s}n}, h_{\mathrm{s}n}\rangle \mid s_n)\ S_\mathrm{comp}\ (\langle s_{\mathrm{t}n}, h_{\mathrm{t}n}\rangle \mid t_n)$$

where $t_n \equiv (\overline{id} = f(\overline{texp}); c)$ *with* f *an exported function of the verified component* C_s *of sprog (ie. an incall stub is about to be called). Applying the (lifted) operational semantics to both the target and source code then causes one of two scenarios to occur:*

(1) *Both the source code s and the target code t get stuck during the execution of the incall stub.*

(2) *Both s and t take a number of steps, ie.* $s \hookrightarrow^+ s'$ *and* $t \hookrightarrow^+ t'$*, and afterwards it holds that*

$$(\langle s'_\mathrm{s}, h'_\mathrm{s}\rangle \mid s')\ S_\mathrm{comp}\ (\langle s'_\mathrm{t}, h'_\mathrm{t}\rangle \mid t')$$

consisting of $n + 2$ *frames, where an Incall frame and an* R_comp *frame have been added compared to before.*

Bridging post function call: *Given a program execution such that*

$$sprog, tprog \vdash (\langle s_\mathrm{s}, h_\mathrm{s}\rangle \mid s)\ S\ (\langle s_\mathrm{t}, h_\mathrm{t}\rangle \mid t)$$

consisting of n frames, and where the $n-1$st and nth frames are respectively an Incall *and an* R_{comp} *frame*

$$sprog, tprog \vdash_b (\langle s_{s(n-1)}, h_{s(n-1)}\rangle \mid s_{n-1})\ \mathit{Incall}\ (\langle s_{t(n-1)}, h_{t(n-1)}\rangle \mid t_{n-1})$$
$$sprog, tprog \vdash_b (\langle s_{sn}, h_{sn}\rangle \mid s_n)\ R_{\text{comp}}\ (\langle s_{tn}, h_{tn}\rangle \mid t_n)$$

where $t_n \equiv \text{return } \overline{texp}$ *(ie. a return to an incall stub is about to happen). Applying the (lifted) operational semantics to both the target and source code then causes the following scenario to occur:*

Both s and t take a number of steps, ie. $s \hookrightarrow^+ s'$ *and* $t \hookrightarrow^+ t'$*, and afterwards it holds that*

$$(\langle s'_s, h'_s\rangle \mid s')\ S\ (\langle s'_t, h'_t\rangle \mid t')$$

consisting of $n-2$ *frames, where the* Incall *frame and an* R_{comp} *frame have been removed compared to before. If* $n-2 \leq 0$*, however, execution successfully terminates in a* return *statement (the return statement of the incall stub) for both s and t.*

Proof.

Bridging pre function call: It is clear that the next 2 newly created frames will be an Incall frame, when the function f is called, and an R_{comp} frame, when the incall stub calls the function f. We now have to prove that, following the operational semantics, the relations we expect to hold actually hold for the 2 new frames, and that the relation S_{comp} still holds for the original callee frame.

We first make some more general observations that will contain most of the proof's complexity:

(1) The location association mapping b stays the same before and after the two new frames are created, since no locations are created during this incall.

(2) Given the function call relationship between the different frames, we have the following links between their δ functions:

$$[\overline{texp_b}]_{\delta_n} = \delta_{n+1}(\overline{id_{\text{arg}}}, \overline{m})$$

,

$$\delta_n(\text{V}(\text{PRE}(f_{\text{bt}})) \setminus \overline{id_{\text{arg}}}) = \delta_{n+1}(\text{V}(\text{PRE}(f_{\text{bt}})) \setminus \overline{id_{\text{arg}}})$$

and

$$\delta_{n+2} = \delta_{n+1} \mid_{\text{V}(\text{PRE}_c * \text{PRE}_p) \cup \overline{id_{\text{arg}}}}$$

. This means (as usual swiping some logical variable renaming-related issues under the rug) that frame linking, using the Link predicate, for the new frames is immediately satisfied.

(3) By the compositionality of $\text{invvalmap}_{b^{-1}}$ in the nth frame, combined with the fact that argument values are passed from frame n to $n+1$, it holds that $s_{s(n+1)}(\overline{id_{\text{arg}}}, \overline{m}) == \text{invvalmap}_{b^{-1}}(s_{t(n+1)}(\overline{id_{\text{arg}}}, \overline{m}))$ and hence, for the arguments, also that $\text{valmap}_b(s_{s(n+1)}(\overline{id_{\text{arg}}})) == s_{t(n+1)}(\overline{id_{\text{arg}}})$, because none of the $\overline{id_{\text{arg}}}$ are reified resources, and without the cases for target linear capabilities, invvalmap and valmap are each other's inverse. Due to frame $n+1 \leftrightarrow n+2$ argument passing with the same arguments $\overline{id_{\text{arg}}}$, $\text{valmap}_b(s_{s(n+2)}(\overline{id_{\text{arg}}})) == s_{t(n+2)}(\overline{id_{\text{arg}}})$ will hold as well. Frame linking trivially holds over the argument variables $\overline{id_{\text{arg}}}$ in both frame transitions.

(4) After the call to f in the target (and to f_{bt} in the source) in frame n, a hole $\bullet^{\overline{id_{\text{arg}}}=\overline{texp}}$ ($\bullet^{\overline{id_{\text{arg}}}=\overline{texp_b}}_{\text{POST}(f_{\text{bt}})}$ in the source) remains. The universal precondition $\text{PRE}(f_{\text{bt}})$ is consumed in the source, and part of the nth frame heap $h_{sn} = h_{s,\text{PRE}} \uplus h'_{s,n}$ is transfered to frame $n+1$: the heap $h_{s,\text{PRE}}$, the

smallest heap such that $h_{\mathrm{s,PRE}} \vdash^{\mathrm{src}}_{\delta_n} \mathrm{PRE}(f_{\mathrm{bt}})$ holds (it must exist, since S_{comp} holds for frame n), is transfered. In the target, we transfer the smallest heap $h_{\mathrm{t,PRE}}$, subheap of $h_{\mathrm{t}n} = h_{\mathrm{t,PRE}} \uplus h'_{\mathrm{s},n}$, such that $\forall l.\, h_{\mathrm{s,PRE}}(l) = [v_0, \ldots, v_k] \Leftrightarrow h_{\mathrm{t,PRE}}(b(l)) = [\mathrm{valmap}_b(v_0), \ldots, \mathrm{valmap}_b(v_k)]$ holds. This target heap must exist, because S_{comp} contains a linking condition $\forall l.\, h_{\mathrm{t}n}(l) = [v_1, \ldots, v_m] \Leftrightarrow h_{\mathrm{s}n}(b^{-1}(l)) = [\mathrm{invvalmap}_{b^{-1}}(v_1), \ldots, \mathrm{invvalmap}_{b^{-1}}(v_m)]$, and reified resources for boundary functions such as f never contain nested resources (ie. nested pointers) (boundary function contracts cannot contain range expressions), hence $\mathrm{invvalmap}_{b^{-1}}$ and valmap_b are eachother's proper inverse in this scenario.

(5) The declared and assigned sets of variables d_{pre} and $d^{\mathrm{b}}_{\mathrm{pre}}$ will be related by valmap_b, since the variables $\overline{m}$ (and hence also their addresses) are related by $\mathrm{invvalmap}_{b^{-1}}$ and so are the passed-in heaps $h_{\mathrm{s,PRE}}$ and $h_{\mathrm{t,PRE}}$, as mentioned above. The assignments a_{pre} and $a^{\mathrm{b}}_{\mathrm{pre}}$ consist solely of taking addresses of the variables $\overline{m}$ and accessing these heaps.

(6) Given the relations between the variables $\overline{m}$ in both target and source, and between the variables d_{pre} and $d^{\mathrm{b}}_{\mathrm{pre}}$ (points 3 and 5), *Guard equi-failing* gives us that the checks cs_{pre} and cp_{pre} equi-fail with their respective back-translations, which is precisely what we want.

(7) After the InCall frame has performed its call to f (f_{comp} in the target), the remaining code in the InCall frame is exactly equal to the code for t and s presented in the InCall relation.

(8) We need the following auxiliary result: $h_{\mathrm{s,PRE}} \vdash^{\mathrm{src}}_{\delta_{n+1}} \mathrm{PRE_c} * \mathrm{PRE_p}$ holds, if $h_{\mathrm{s,PRE}} \vdash^{\mathrm{src}}_{\delta_{n+1}} \mathrm{PRE}(f_{\mathrm{bt}})$ holds and execution does not get stuck (which is the case, given point 4 and the correspondence between δ_n and δ_{n+1} highlighted in point 2 above). Assume this first judgment does *not* hold but the second one does, ie. δ_{n+1} applied to either $\mathrm{PRE_c}$ or $\mathrm{PRE_p}$ is inconsistent or inconsistent with $h_{\mathrm{s,PRE}}$. Since the checks $cs^{\mathrm{b}}_{\mathrm{pre}}$ and $cp^{\mathrm{b}}_{\mathrm{pre}}$ check exactly the information present in $\mathrm{PRE_c}$ and $\mathrm{PRE_p}$ (cfr. theorem 7), the execution would have gotten stuck in this case. The auxiliary result hence holds for the case of non-equi-stuck execution.

(9) The $n+1$st frame will just pass the acquired heaps $h_{\mathrm{s,PRE}}$ and $h_{\mathrm{t,PRE}}$ to the $n+2$nd frame. We now have to prove (in order to prove the R_{comp} relation) that $h_{\mathrm{s,PRE}} \vdash^{\mathrm{src}}_{\delta_{n+2}} \mathrm{PRE_c} * \mathrm{PRE_p}$. Since $\delta_{n+2} = \delta_{n+1} \mid_{\mathrm{V}(\mathrm{PRE_c} * \mathrm{PRE_p}) \cup \overline{id_{\mathrm{arg}}}}$, this is immediate from the previous point.

(10) Perhaps the most technical part of the proof is proving the condition $s_{\mathrm{t}(n+2)}, h_{\mathrm{t}(n+2),\mathrm{reify}} \vdash^{\mathrm{tgt}}_{\delta, b} \mathit{PRE}_{\mathrm{s}} * \mathit{PRE}_{\mathrm{p}}$ ($h_{\mathrm{t}(n+2),\mathrm{reify}}$ is irrelevant, since there are no range resources in boundary contracts). Range assertions cannot be used in boundary contracts, and we hence only need to check the correct reification of array resources. The case LogTgtCorrArrName for this judgment is indeed satisfied for every chunk m in $\mathit{PRE}_{\mathrm{s}}$. This set $\overline{m}$ is the same set that appears in the incall stub, which is related with itself through $\mathrm{invvalmap}_{b^{-1}}$, as shown in point 3. Name $\overline{\mathit{sexp}_m}$ the expressions passed from frame n as arguments for the variables $\overline{m}$. The fact that $\gamma_{n+1}(\overline{m}) = \overline{m}$ then gives us that $\delta_{n+1}(\overline{m}) = [\![\overline{m}]\!]_{s_{\mathrm{s}(n+1)}} = (\overline{m_{\mathrm{addr}}}, \overline{m_{\mathrm{len}}})$. We know that each logical m in $\overline{m}$ corresponds to a universal contract of pointer type in the precondition $\mathrm{PRE}(f_{\mathrm{bt}})$ of the source incall stub, where $m.1$ is the address and $m.2$ the length (cfr. definition universal contract). Since the chunks in $\mathit{PRE}_{\mathrm{s}}$ are simply derived from the chunks in $\mathrm{PRE}(f_{\mathrm{bt}})$ by applying cs_{pre}, $\overline{m_{\mathrm{addr}}}$ and $\overline{m_{\mathrm{len}}}$ are the correct addresses for all chunks $\overline{m}$ in $\mathit{PRE}_{\mathrm{s}}$ as well, ie. for each chunk $m : \mathit{exp} \mapsto \mathit{exp}'$; it must hold that $\mathit{exp}_{\delta_{n+1}} == m_{\mathrm{addr}}$ and $\mathit{exp}'_{\delta_{n+1}} == [v_0, \ldots, v_k] \Rightarrow k + 1 == m_{\mathrm{len}}$. Combining this with the $\mathrm{invvalmap}_{b^{-1}}$ relation with the target $\overline{m}$ from above, we get exactly the result from LogTgtCorrArrName.

Given these observations, we can now summarize why the relation S still holds over the new frames. Point 6 handles the equi-failing case, so we can assume success for the rest of this proof.

Frame linking is immediate from point 2. For al 3 frames n,$n+1$ and $n+2$, the condition InverseMap$(\delta_i, P_i, \gamma_i, s_{si}, h_{si})$ holds for the δ's described in point 2. For frame n, this is trivial, as we keep the same δ as before the call, with exceptions for the erased variables, but their universal conditions have been consumed anyway, and the non-consumed separation logic state P' clearly still corresponds with the remaining heap h'_{sn} through this δ. Frame $n+1$ simply transferred its separation logic state and heap to the next frame and hence trivially satisfies the InverseMap predicate (given that all variables map to themselves in γ). Point 9 proves the heap-logic correspondence for frame $n+2$ and the $s_s \leftrightarrow \delta$-correspondence follow easily, since $s_{s(n+2)}$ consists of argument values $\overline{id_{\text{arg}}}$ only, right after the function call.

The S_{comp} relation in frame n still holds, because capability nulling in the source language is emulated wrt. the target (cfr. the FApp back-translation rule) and we hence retain the required invvalmap$_{b^{-1}}$ relation between the stack frames s_{sn} and s_{tn} , because the transfered heaps $h_{s,\text{PRE}}$ and $h_{t,\text{PRE}}$ were minimal and corresponded with each other (ie. the remaining heaps will too) and because universality of the contract is trivially retained.

For frame $n+1$, the Incall relation holds because of the matching code, as mentioned in point 6 and because of the valmap$_b$-correspondence for $\overline{id_{\text{arg}}}$, as mentioned in point 3.

For frame $n+2$, the R_{comp} relation holds because of the $valmap_b$ relation between the stack frames s_{sn+2} and s_{tn+2} mentioned in point 2 and the source and target heap correspondence mentioned in point 4. The condition $s_{t(n+2)}, h_{t(n+2),\text{reify}} \vdash^{\text{tgt}}_{\delta,b} PRE_s * PRE_p$ is proven in point 10 above.

Bridging post function call: Now we have to prove that once execution returns from the nth R_{comp} and $n-1$st Incall frames, that the S_{comp} relation will still hold over the $n-2$nd frame (if it exists). This is noticeably easier, since frames are discarded and not created.

We first make some more general observations that will contain most of the proof's complexity:

(11) The location association mapping b stays the same before and after the two new frames are discarded, since no locations are created during execution.

(12) The $\overline{result}$ variables in target and source in the Incall frame are related by valmap$_b$, since valmap$_b$ hold over s_{sn} and s_{tn} in the R_{comp} frame and given the compositionality of valmap$_b$. Since none of these $\overline{result}$ variables are of pointer type in the target, the target and source are also inversely related by invvalmap$_{b^{-1}}$

(13) Conditions on boundary functions dictate that the postcondition pure heap $POST_p$ of each boundary function f has to contain expressions $\overline{exp_n}$, using only arguments from $\overline{result}$ and $\overline{id_{\text{arg}}}$ and defining the address $addr_n$ of each chunk n (with fixed length k_n). From the R_{comp} relation, we know that $s_{tn}, h_{tn,\text{reify}} \vdash^{\text{tgt}}_{\delta,b} PRE_s * PRE_p$ must hold, and hence, due to the case LogTgtCorrArrName, that, for each n, $s_{tn}(n) = b(l)^{i,i+k}$, with k the fixed chunk length and $\delta_{n+1}(exp_n) = (l,i)$. Frame linking between frames $n-1$ and n for $\overline{id_{\text{arg}}}$, and the fact that the expressions for the variables $\overline{result}$ are returned from frame n to frame $n-1$, gives us that both sets of variables will have the same δ-value in frames $n-1$ and n, ie. $\delta_{n-1}(\overline{id_{\text{arg}}}) = \delta_n(\overline{id_{\text{arg}}})$ and $\delta_{n-1}(\overline{x}) = \delta_n(\overline{result})$ (the logical variables $\overline{result}$ only exists in frame n after the return has been executed and we use $\overline{x}$ for $\gamma_{n-1}(\overline{return})$, like in theorem 7). The source stack values for $\overline{n}$ and

$\overline{return}$ in frame $n-1$ will be equal to these δ-values, because γ has identity maps for all these variables and given the definition of δ. The assignment a_{la} in the source uses variables from the sets $\overline{result}$ and $\overline{id_{\text{arg}}}$ only (cfr. the earlier made restrictions on boundary contracts) and assigns expressions (corresponding to resource lengths and addresses) over them to the back-translated resources $\overline{n}$. Given the correspondence in the values of these variables between frames $n-1$ and n, it now holds for each n that $s_{\text{s}n}(n) = ((l, i), k)$, which corresponds with the target value $s_{\text{t}n}(n) = b(l)^{i,i+k}$ above through $\text{invvalmap}_{b^{-1}}$, as expected.

(14) There is no case for equi-failing programs in the theorem, since neither the target nor the source code can fail in a non-trivial way after returning (and we ignore simple errors, as these will cause equi-failure in target and source).

(15) At the end of execution in frame n, it holds (from the definition of δ) that $h_{\text{s}n} \vdash^{\text{src}}_{\delta_{\text{n}}} \text{POST}_{\text{c}} * \text{POST}_{\text{p}}$. The returned heaps $h_{\text{s}n}$ and $h_{\text{t}n}$ (at least the 'real' part $h_{\text{t}n,\text{real}}$) also relate through valmap_b. In the incall stub proof, dropping all concrete restrictions allows deriving $\text{POST}(f_{\text{bt}})$ from $\text{PRE}_{\text{c}} * \text{PRE}_{\text{p}}$. Since this first assertion is less concrete and we have frame linking, δ_{n-1} can be chosen such that it is still satisfied in the previous frame, ie $h_{\text{s}n} \vdash^{\text{src}}_{\delta_{\text{n}-1}} \text{POST}(f_{\text{bt}})$. An analogous argument allows concluding that $h_{\text{s}n} \vdash^{\text{src}}_{\delta_{\text{n}-2}} \text{POST}(f_{\text{bt}})$ after the return from Incall.

Given these observations, we can now summarize why the relation S still holds after the destruction of the last two frames. If $n \leq 2$, then execution will just terminate on the Incall frame's return statement, as mentioned in the theorem statement.

All we need to prove, is that the S_{comp} relation still holds with the assignment of the returned values $\overline{result}$ and $\overline{n}$, with the added postcondition $\text{POST}(f_{\text{bt}})$ and the returned heaps $h_{\text{s}n}$ and $h_{\text{t}n}$. As mentioned earlier in points 12 and 13, respectively , both sets of variables $\overline{result}$ and $\overline{n}$ are related through $\text{invvalmap}_{b^{-1}}$, which will ensure that the stack frames $s_{\text{s}(n-2)}$ and $s_{\text{t}(n-2)}$ are still related through $\text{invvalmap}_{b^{-1}}$ as required. The heaps $h_{\text{s}n}$ and $h_{\text{t}n}$ are related through valmap_b, since they originated from an R_{comp} frame. They are (after throwing away any auxiliary heap locations that the R_{comp} relation might have used) hence also related by $\text{invvalmap}_{b^{-1}}$, since $POST_{\text{s}}$ contains no range resources and its reification will hence not store any pointer-typed values in the target heap either, making valmap_b and $\text{invvalmap}_{b^{-1}}$ each other's proper inverse. Since $\text{POST}(f_{\text{bt}})$ has a universal contract form over all returned values, the result will remain universal in form. Point 15 proves that we can extend δ_{n-2} with new symbolic variable mappings for all new variables in $\text{POST}(f_{\text{bt}})$ so that $h_{\text{s}n} \vdash^{\text{src}}_{\delta_{\text{n}-2}} \text{POST}(f_{\text{bt}})$ remains satisfied, completing this proof. □

Theorem 12 (Outcall stub bridging)**.**
Bridging pre function call: *Given a program execution such that*

$$sprog, tprog \vdash (\langle s_{\text{s}}, h_{\text{s}} \rangle \mid s)\ S\ (\langle s_{\text{t}}, h_{\text{t}} \rangle \mid t)$$

consisting of n frames, and where the nth frame is an R_{comp} frame

$$sprog, tprog \vdash_b (\langle s_{\text{s}n}, h_{\text{s}n} \rangle \mid s_n)\ R_{\text{comp}}\ (\langle s_{\text{t}n}, h_{\text{t}n} \rangle \mid t_n)$$

where $s_n \equiv \{P\}_\gamma\ (\overline{id} = f(\overline{texp}); c)\ \{Q\}_{\gamma'}$ with f an imported function of the verified component C_{s} of sprog (ie. an outcall stub is about to be called). Applying the (lifted) operational semantics to both the target and source code then causes the following scenario to occur:

Both s and t take a number of steps, ie. $s \hookrightarrow^+ s'$ and $t \hookrightarrow^+ t'$, and afterwards it holds that

$$(\langle s'_{\text{s}}, h'_{\text{s}} \rangle \mid s')\ S\ (\langle s'_{\text{t}}, h'_{\text{t}} \rangle \mid t')$$

consisting of $n+2$ frames, where an Outcall *frame and an S_{comp} frame have been added compared to before.*

Bridging post function call: *Given a program execution such that*

$$sprog, tprog \vdash (\langle s_s, h_s\rangle \mid s)\ S\ (\langle s_t, h_t\rangle \mid t)$$

consisting of n frames, and where the $n-1$st and nth frames are respectively an Outcall *and an S_{comp} frame*

$$sprog, tprog \vdash_b (\langle s_{s(n-1)}, h_{s(n-1)}\rangle \mid s_{n-1})\ Outcall\ (\langle s_{t(n-1)}, h_{t(n-1)}\rangle \mid t_{n-1})$$
$$sprog, tprog \vdash_{b^{-1}} (\langle s_{sn}, h_{sn}\rangle \mid s_n)\ S_{\text{comp}}\ (\langle s_{tn}, h_{tn}\rangle \mid t_n)$$

where $t_n \equiv$ return $\overline{texp}$ (ie. a return to an outcall stub is about to happen). Applying the (lifted) operational semantics to both the target and source code then causes one of two scenarios to occur:

(1) *Both the source code s and the target code t get stuck during the execution of the outcall stub.*

(2) *Both s and t take a number of steps, ie. $s \hookrightarrow^+ s'$ and $t \hookrightarrow^+ t'$, and afterwards it holds that*

$$(\langle s'_s, h'_s\rangle \mid s')\ S\ (\langle s'_t, h'_t\rangle \mid t')$$

consisting of $n-2$ frames, where the Outcall *frame and an S_{comp} frame have been removed compared to before. If $n-2 \leq 0$, however, execution successfully terminates in a* return *statement (the return statement of the outcall stub) for both s and t.*

Proof.

The proof for outcall stubs is very similar to that for incall stubs, with the difference that the techniques used for the *pre* and the *post* function call part are largely reversed with respect to the incall proof. Of course, both pre and both post parts will still have some commonalities, because in the pre-part two extra frames are created, whereas in the post part, they are destroyed again. Given the similarities with and the length of the previous proof, we highlight the high-level pieces of the proof and how they can be mapped to proofs we have already given in the previous bridging proof for incall stubs.

Bridging pre function call: It is clear that the next 2 newly created frames will be an Outcall frame, when the function f is called, and an S_{comp} frame, when the incall stub calls the function f. We now have to prove that, following the operational semantics, the relations we expect to hold actually hold for the 2 new frames, and that the relation R_{comp} still holds for the original callee frame.

We first make some more general observations that will contain most of the proof's complexity:

(1) The location association mapping b stays the same before and after the two new frames are created, since no locations are created during this incall.

(2) There is no case for equi-failing programs in the theorem, since neither the target nor the source code can fail in a non-trivial way after returning (and we ignore simple errors, as these will cause equi-failure in target and source).

(3) Frame linking and the proper values for the different δ-mappings again immediately follow from the fact that the new frames are created through function calls.

(4) We can prove that the assignment a_{la} creates values $\overline{m}$ that relate with their target- level counterparts through invvalmap$_{b^{-1}}$. The proof is analogous to point 13 in the previous proof.

(5) Similarly to point 4 in the previous proof, minimal ($\text{invvalmap}_{b^{-1}}$-related) parts of the source and target heap of frame n are split off and passed all the way to frame $n + 2$. The fact that the heaps still satisfy the universal contract (this is the inverse of point 10 in the previous proof) is derived analogously to point 15 in the previous proof.

(6) The variables d_{pre} assigned in the assignments a_{pre} are related in target and source, cfr. point 4 in the previous proof and using point 4 from this proof.

(7) Because of compositionality, the arguments $\overline{id_{\text{arg}}}$ passed to frames $n + 1$ and $n + 2$ will again be related by $\text{invvalmap}_{b^{-1}}$.

(8) We can recycle point 10 from the previous proof, to prove that the heaps returned from frame n are related and will satisfy the R_{comp} relation in frame $n - 2$ (if this frame exists).

Combining all the previous points with the definitions of R_{comp}-, Outcall- and S_{comp}-frames gives us the desired result. The argument is very similar to the *pre function call* part of the previous proof.

Bridging post function call: Now we have to prove that once execution returns from the nth S_{comp} and $n - 1$st Outcall frames, that the R_{comp} relation will still hold over the $n - 2$nd frame (if it exists). This is noticeably easier, since frames are discarded and not created.

We first make some more general observations that will contain most of the proof's complexity:

(8) The location association mapping b stays the same before and after the two new frames are created, since no locations are created during this incall.

(9) We use an argument similar to the one made in point 12 of the previous proof to derive that the returned $\overline{result}$ and $\overline{n}$ values are related through $\text{invvalmap}_{b^{-1}}$.

(10) The variables d_{post} assigned in the assignments a_{post} are related in target and source, cfr. point 4 in the previous proof and using point 9 from this proof.

(11) Given the previously proven and given (by the definition of Outcall) relations between the sets $\overline{id_{\text{arg}}}$, $\overline{n}$, $\overline{result}$, $\overline{d_{\text{post}}}$ and $\overline{d_{\text{pre}}}$, we can repeat the reasoning from point 6 to again prove *equi-failure* of the source and target code.

Combining all the previous points with the definition of R_{comp} gives us the desired result. The argument is very similar to the *post function call* part of the previous proof.

□

7.7.2 Proving simulation for S. A central inference rule we need to prove in this section is the following:

$$\frac{(\langle \bullet, \epsilon \rangle \mid\vdash \mathit{sprog})\ S\ (\langle \bullet, \epsilon \rangle \mid \mathit{tprog})}{\mathit{sprog} \Downarrow \Leftrightarrow \mathit{tprog} \Downarrow} \qquad \text{(SecToEquiTermin)}$$

Proof.

The proof of this inference rule will transpire inductively and will need to discuss the relations between the different types of stack frames and the properties they have themselves. After the application of the ProgExec rule in both source and target language executing the given statement, we can be in one of the four types of frames.

For each type of frame, we have to formulate a simulation argument that states that the following cases can occur, given that either *sprog* (left to right) or *tprog* (right to left) terminates;

(1) Either source or target code gets stuck (should be proven impossible)

(2) Either source or target code diverge (should be proven impossible)

(3) Both source and target code terminate (in a return when there are no previous frames left)

(4) Both source and target code run until they reach a function call statement, that would create a frame of a different type when executed. Alternatively, they both run until a return statement that transfers control flow to a frame of a different type. It has to be proven that after the switch of frame type, the appropriate frame relation indeed holds.

(The difference between the first and second bullet is not strictly requires by the formulation of full abstraction, but is required for a sane comilation and back-translation, as was mentioned before on multiple occasions)

If we can prove that the previous itemization holds for each type of frame and we know that either the source or target program *sprog* or *tprog* terminates, then we can inductively combine the proofs for all four types of frames and prove the theorem. Given the properties we have proven in the previous section, it is now reasonable easy to prove this result. We consider each of the four frame types in a separate case:

(1) S_{comp}
An S_{comp} frame will only create a frame-switching call to an InCall frame or return to an OutCall frame. As long as no such transition happens, our relation is effectively of the $\overline{S}_{\text{comp}}$ type.

We can then use the SCompToEquiTermin-rule and its proof from the previous subsection. This rule and its proof discussion then immediately prove case 3 and exclude cases 1 and 2. However, if a call that creates an InCall frame or a return to an OutCall frame is encountered before termination, then SCompToEquiTermin will obviously not hold. We do know that right before the call or return *scomp* still holds, since $\overline{S}_{\text{comp}}$ was proven to be a simulation relation. The proof that this transition to the InCall or OutCall frame is proper (ie. the aforementioned relations actually hold after transition) is handled in the discussion of the InCall and OutCall frames below, respectively. This handles case 4.

(2) R_{comp}
This discussion is dual to the discussion for $\overline{S}_{\text{comp}}$

An R_{comp} frame will only create a frame-switching call to an OutCall frame or return to an InCall frame. As long as no such transition happens, our relation is effectively of the $\overline{R}_{\text{comp}}$ type.

We can then use the RCompToEquiTermin-rule and its proof from the previous subsection. This rule and its proof discussion then immediately prove case 3 and exclude cases 1 and 2. However, if a call that creates an OutCall frame or a return to an InCall frame is encountered before termination, then RCompToEquiTermin will obviously not hold. We do know that right before the call or return $\overline{R}_{\text{comp}}$ still holds, since $\overline{R}_{\text{comp}}$ was proven to be a simulation relation. The proof that this transition to the OutCall or InCall frame is proper (ie. the aforementioned

relations actually hold after transition) is handled in the discussion of the InCall and OutCall frames below, respectively. This handles case 4.

(3) *Incall*
Theorem 11, which we proved in the previous section, immediately proves all necessary conditions (which includes proving the InCall-related case 4 for both R_{comp} and *scomp*.

(4) *Outcall*
Theorem 12, which we proved in the previous section, immediately proves all necessary conditions (which includes proving the OutCall-related case 4 for both R_{comp} and *scomp*.

Now, the fact that we have shown that for each type of frame there is equi-termination within the frame and that each type of frame either terminates or spawns a new frame after a finite number of steps, indeed implies equi-termination of the entire source and target programs. □

The following inference rule proves that if one program is the back-translation of the other, then both are related through the relation S. Its proof is trivial from the definition of S.

$$\frac{\vdash s \rightsquigarrow t \qquad \vdash s, (\mathfrak{C}_t, id) \rightsquigarrow_b \vdash \mathfrak{C}_s[s]//@\text{main} = id}{(\langle \bullet, \epsilon \rangle \mid\vdash \mathfrak{C}_s[s]//@\text{main} = id\ )\ S\ (\langle \bullet, \epsilon \rangle \mid \mathfrak{C}_t[t]//@\text{main} = id\ )} \quad \text{(CompilIsSec)}$$

As was the case for correctness, the same Coherence result from before is used to finish the step in the proof where we need to go from a source program with proof to and from a source program without proof, while preserving (non-)termination.

The BTEquiTermination rule now follows immediately from the sequential combination of the CompilIsSec, SecToEquiTermin and Coherence rules.

REFERENCES

Martín Abadi. 1999. Protection in programming-language translations. In *Secure Internet programming*. Springer-Verlag, 19–34.

Pieter Agten, Bart Jacobs, and Frank Piessens. 2015. Sound Modular Verification of C Code Executing in an Unverified Context. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 581–594. https://doi.org/10.1145/2676726.2676972

Adam Chlipala. 2017. Formal Reasoning About Programs.

Dominique Devriese, Marco Patrignani, and Frank Piessens. 2016. Fully-abstract compilation by approximate back-translation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 164–177. https://doi.org/10.1145/2837614.2837618

John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*. IEEE, 55–74.

Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. 2019. Linear Capabilities for Fully Abstract Compilation of Separation-Logic-Verified Code. *Proc. ACM Program. Lang.* 3, ICFP, Article 84 (Aug. 2019), 29 pages. https://doi.org/10.1145/3341688