Formalizing the SAFECode Type System

Daniel Huang and Greg Morrisett

Harvard University, Cambridge MA 02138, USA,
dehuang@fas.harvard.edu,
greg@eecs.harvard.edu

Abstract. The Secure Virtual Architecture (SVA) provides an object-level integrity policy, similar to type-safety, for languages such as C and C++, and thus rules out a wide range of common vulnerabilities. SVA uses an enhanced version of the Low-Level Virtual Machine (LLVM) compiler called SAFECode to enforce the policy through a combination of static and dynamic type-checks. However, this results in a relatively large trusted computing base (TCB). SVA reduces the TCB with an unverified type-checker that relies upon a paper-and-pencil proof of type-soundness for a core-language. As a further step towards increasing the assurance of the compiler, we present a mechanized proof of soundness and a verified type-checker for a realistic subset of the SAFECode type system developed using the Coq Proof Assistant.

Keywords: verification, SAFECode, LLVM, memory safety

1 Introduction

Most of our computing infrastructure is coded using low-level languages such as C/C++. Unsurprisingly, it is easy to make simple mistakes in these languages that lead to well-known vulnerabilities. In principle, recoding the infrastructure in a type-safe language would eliminate many of these vulnerabilities, but the costs of doing so seem to outweigh the benefits.

An attractive alternative is to bring the benefits of type-safety to legacy code by combining static analyses and run-time checks to automatically enforce a type-safety policy. There are many challenges in doing this effectively, as static analyses are generally too weak to reason effectively about real C/C++ programs, resulting in many false positives. The cost of inserting run-time checks and maintaining the meta-data needed to support those checks can also be prohibitively expensive. Recently, a number of systems have successfully combined the benefits of static analysis, dynamic checks, program optimization, and clever run-time representations to produce viable solutions [3, 10, 11, 5, 4].

One such system, SAFECode [5], uses sophisticated analyses and optimizations to eliminate run-time checks. However, this adds the SAFECode compiler to the trusted computing base (TCB). We could try to prove that the analyses and transformations (and subsequent optimizations) are correct as in the CompCert project [8]. Perhaps more easily, we can build a verified checker that

Type	τ ::=	`int` | `char` | `unknown` | $\tau * \rho$ | `handle`(ρ, τ)
Statements	S ::=	ϵ | S;S | $x =$ E | `store E, E` | `storeToU x, E, E`
		| `storec E, E` | `storecToU E, E` | `poolfree(E, E)`
		| `poolinit`$(\rho, \tau)x${S} | `pool`{S}`pop`(ρ)
Expressions	E ::=	`var` | V | E *op* E | `load E` | `loadFromU x, E` | `loadc E`
		| `loadcFromU E` | `cast E to` τ | `poolalloc(x, E)`
		| `(x, &E[E])` | `castint2pointer x, E to` τ
Value	V ::=	*uninit* | `Int` | `region`(ρ)

Fig. 1. Core-language presented in original SAFECode paper.

attempts to prove that rewritten and optimized code respects the SAFECode security policy. The goal of this paper is to increase the assurance of the SAFECode compiler by formalizing a realistic subset of the language and its type system, presenting a mechanically-checked proof of soundness, and building a verified checker that can be used to check code emitted by SAFECode.

2 Overview of SAFECode

Our work builds upon a previous paper describing the SAFECode system [5], which enforces an object-level integrity policy similar to, but weaker than traditional notions of type-safety. Conceptually, SAFECode instruments all dangerous operations such as loads and stores with dynamic checks. To justify the elimination of unnecessary run-time checks, the paper formalized a core-language, type system and gave a paper-pencil proof of soundness for the typing rules.

We have reproduced their core-language in Figure 1. SAFECode uses regions, similar to the approach pioneered by Tofte and Talpin [12] and later refined in Cyclone [6]. Like these previous systems, a pointer type $\tau * \rho$ is indexed by a region variable ρ indicating the region of memory it references. However, SAFECode only places objects of the same type in a given region, allowing region metadata to support efficient run-time casts. Objects whose type cannot be statically determined are put in untyped regions. The type system tracks which regions are accessible, and hence, which pointers can be safely dereferenced.

More concretely, SAFECode provides a lexically scoped construct of the form `poolinit(`ρ`, `τ`)`$\{\cdots\}$, which allocates a new region (or pool) to exclusively hold values of type τ and binds the region to ρ. Regions are typically represented as lists of pages that can be dynamically grown as new objects are allocated. Initially, the pages are zero-filled and zero is assumed to be a valid value for any type. In particular, dereferencing address zero will result in a trapped error (segmentation fault). Within the scope of the `poolinit`, programs can allocate objects of type τ in ρ using `poolalloc`, which returns a pointer of type $\tau * \rho$. Such pointers can be dereferenced (via `load`), updated (via `store`), or used to deallocate the object (via `poolfree`) while in the scope of ρ. Memory reclaimed in a region can be recycled for use at the same type. At the end of `poolinit`'s

$\Delta : \rho \rightharpoonup \tau$	$\Gamma : x \rightharpoonup \tau$
r1: i32	
r2: i32*r1	
	x: i32*r1
	y: i32*r1*r2
	z: i32*r1
	p:i32*r1
	w1: i32
	w2: i32

```
1.  poolinit(r1, i32) {
2.    poolinit(r2, i32*r1) {
3.      x = poolalloc(r1, 4);
4.      y = poolalloc(r2, 4);
5.      store x, y;
6.      z = load y;
7.      poolfree(z);
8.      p = (i32*r1) 42;
9.      w1 = load z;
10.     w2 = load p;
11.   }
12. }
```

Fig. 2. Example program and typing derivation.

scope, the entire region is deallocated and its memory can be safely recycled for use in other regions to hold values of potentially different types. SAFECode also supports checked operations `loadU(`ρ`,`q`)`, `storeU(`ρ`,`v`,`q`)`, and checked casts. These operations do not require that q is ascribed the static type $\tau * \rho$. Rather, a run-time check is performed to see if q is a valid, τ-aligned pointer into ρ. If the check fails, then the program is terminated.

Figure 2 illustrates a SAFECode program in the core-language. In the example, `r1` is a statically-typed region that holds integer values and region `r2` is a statically-typed region holding `i32*r1` values. The typing context tracks the set of region variables in scope and their types (Δ) as well as the set of variables and their types (Γ). The table to the left of the code summarizes the context used to check each line. For example, after line 1, region `r1` is in scope and is assumed to hold values of type `i32`. By line 5, we assume variable `x` has type `i32*r1` and `y` has type `i32*r1*r2`. The store instruction type-checks because `y` is a pointer value into region `r2`, `x` is of the appropriate type `i32*r1`, and `r2` holds values of type `i32*r1`. The following load instruction type-checks similarly.

Lines 7 through 10 illustrate how type-homogenous regions enable SAFECode to relax traditional notions of type-safety. For instance, the dangling pointer dereference on line 9 type-checks because the region `r1` is still live and type-homogeneity guarantees it will produce a value of type `int`. SAFECode also allows arbitrary integers to be casted to pointers. Doing so may necessitate a run-time check that can fail, but if the integer actually is an address of the appropriate type in the appropriate region, the cast will succeed. SAFECode accepts the above program as well-typed and guarantees for all possible executions that `x` and `z` always point into region `r1`, and that `y` always points into region `r2`. Furthermore, dereferencing `x` and `z` guarantees an integer, and dereferencing `y` guarantees a pointer into region `r1` (or else zero values).

Currently, there is a huge gap between the original presentation and the actual SAFECode implementation, which is a large amount of C/C++ code

implementing a LLVM bitcode transformation. At a first approximation, there is a mismatch between the C-like core-language presented in the paper and the implementation which is at the level of the LLVM IR. Moreover, the lack of any real language features (e.g. structs, control flow, procedure calls, etc.) in the core-language makes the argument of type soundness less compelling in the context of the implementation. For example, the lack of control flow and ability to express interesting data structures drastically reduces the complexity of reasoning about a region's liftime. However, the actual system supports almost the entirety of LLVM, including extra features such as region-polymorphic functions that induce a LIFO-ordering of regions not expressible in the original model.

In the rest of this paper, we describe a verified checker that we have constructed for the SAFECode compiler using the Coq Proof Assistant [1]. In particular, we describe (a) our formalization of the syntax and semantics of the SAFECode variant of the LLVM [7] intermediate language, (b) a new declarative type system that formalizes the SAFECode policy, (c) a proof that the type system is sound with respect to our semantics, (d) an executable type-checker for SAFECode, (e) a proof that the type-checker is correct with respect to the declarative typing rules, and (f) our experiences in type-checking code generated by the SAFECode compiler.

In addition to scaling the language to the actual implementation, we also hope that our reformulation of SAFECode is cleaner than the one originally presented. For example, the original small-step semantics contains 40 operational transitions (even though it is not very expressive), exposes the details of region and memory meta-data in the operational semantics leading to a less modular proof, has memory leaks, and contains many constructs with duplicate functionality such as `load`, `loadc`, `loadFromU` and `loadcFromU`. While none of these prove fatal to the soundness of the type system, we believe a cleaner model reduces clutter and provides a better intuition for how the proof of soundness is related to the actual system implementation.

3 Language and Operational Semantics

Here, we describe the formal language and semantic model that we have constructed in Coq, but take the liberty of using conventional notation to describe the ideas instead of the details of the Coq code.[1]

3.1 Language

The language is derived from the LLVM IR to mirror the SAFECode implementation as closely as possible and is summarized in Figure 3. Types τ include arbitrary width integers ($\texttt{i}n$), single/double precision floats (`d32` and `d64`), typed-pointers ($\tau * \rho$), untyped-pointers ($U(b) * \rho$) to b bytes, arrays ($[\tau \times n]$), region-polymorphic named types, and region-polymorphic functions. Named types are

[1] `http://people.fas.harvard.edu/~dehuang/projects/sc-formalism.zip`

x: local variable, ρ: region variable, f: function, ℓ: block label, i_n: n-bit integer constant, n: integer size, d: single/double precision float

type	τ ::=	$\texttt{i}n \mid \texttt{d32} \mid \texttt{d64} \mid \tau * \rho \mid U(b) * \rho \mid name\langle\overline{\rho}\rangle \mid [\tau \times n] \mid \forall\overline{\rho}.\overline{\tau}_{\overline{\rho}} \to \tau_{\overline{\rho}}$
name env	Υ ::=	$name \rightharpoonup \langle\overline{\rho}\rangle\{\tau_{1\overline{\rho}}; \cdots ; \tau_{n\overline{\rho}}\}$
operand	o ::=	$x \mid i_n \mid f \mid d \mid \texttt{undef}(\tau) \mid \texttt{blockaddr}(f, \ell) \mid \texttt{null}$
insn	ι ::=	$x = o_1\ \texttt{binop}\ o_2 \mid x = o_1\ \texttt{icmp}\ o_2 \mid x = o_1\ \texttt{fbinop}\ o_2 \mid$
		$x = o_1\ \texttt{fcmp}\ o_2 \mid x = \texttt{iconv}\ \tau_1\ o\ \texttt{to}\ \tau_2 \mid x = \texttt{fconv}\ \tau_1\ o\ \texttt{to}\ \tau_2$
		$x = \texttt{ptrtoint}\ \tau_1\ o\ \texttt{to}\ \tau_2 \mid x = \texttt{inttoptr}\ \tau_1\ o\ \texttt{to}\ \tau_2 \mid$
		$x = \texttt{extractvalue}\ \tau_1\ o,\ \tau_2\ i_n \mid x = \texttt{insertvalue}\ \tau_1\ o_1,\ \tau_2\ o_2\ i_n \mid$
		$x = \texttt{bitcast}\ \tau_1\ o\ \texttt{to}\ \tau_2 \mid x = \texttt{select}\ \tau_1\ o_1, \tau_2\ o_2, \tau_3\ o_3 \mid \texttt{exit} \mid$
		$x = \texttt{getelementptr}\ \tau_1\ o_1,\ \tau_2\ o_2 \mid x = \texttt{getelementptrU}\ \tau\ o_1\ o_2 \mid$
		$x = \texttt{load}\ \tau\ o \mid \texttt{store}\ \tau_1\ o_1,\ \tau_2\ o_2 \mid$
		$x = \texttt{poolcheck}\ \rho\ \tau\ o \mid x = \texttt{poolcheckU}\ \rho\ c\ o \mid \texttt{poolfree}\ \tau\ o$
terminator	tm ::=	$\texttt{return}\ \tau\ x \mid \texttt{br}\ \ell \mid \texttt{br}\ o\ \ell_1\ \ell_2 \mid \texttt{switch}\ \tau\ o,\ \ell\ \overline{\tau * o * \ell} \mid$
		$\texttt{indirectbr}\ \tau\ o,\ \overline{\ell} \mid x = \texttt{poolalloc}\ \tau,\ \rho,\ i,\ \ell \mid$
		$x = \tau\ \texttt{call}\ o\ \langle\overline{\rho}\rangle(\overline{o}),\ \ell \mid x = \tau\ \texttt{unsafecall}\ o\ \langle\overline{\rho}\rangle(\overline{o}),\ \ell \mid$
ϕ *node*	Φ ::=	$x = \phi(\overline{o})_{\overline{\ell}} : \tau$
block	blk ::=	$[\Phi_1; \ldots \Phi_n; \iota_1; \ldots \iota_m; tm]$
function	fn ::=	$\tau\ f\langle\overline{\rho}_p\rangle(\overline{x} : \overline{\tau}_{\overline{\rho}_p})\{$
		$\quad \overline{\rho}_l = \texttt{poolinit}\ \overline{\tau}_{\overline{\rho}_p \cup \overline{\rho}_l};$
		$\quad$ body : $\ell \to$ blk
		$\}$
fn table	F :	$f \rightharpoonup$ fn

Fig. 3. Abstract syntax for our SAFECode language. We use $\overline{x}$ to denote a list of x's. The subscripts on types indicate which regions the types can mention.

used to support aggregate and (iso-)recursive data structures. For example, to define a linked-list node parameterized over a region ρ, we can write the type $\texttt{node} = \langle\rho\rangle\{\texttt{i32};\ \texttt{node} * \rho\}$. The notation $name\langle\overline{\rho}\rangle$ indicates that a named type is instantiated with the regions specified by $\overline{\rho}$. A global environment Υ is used to associate names with their definitions. Union types can be represented using LLVM's encoding as a byte struct whose size is the maximum size of all the types in the union.

The syntax supports nested aggregates (i.e., structs) through names, but internally, we only manipulate flattened *primitive* types. For example, given named types $\texttt{Foo} = \{\texttt{i32};\ \texttt{i32*}\}$ and $\texttt{Bar} = \{\texttt{Foo};\ \texttt{i32}\}$, the flattened representation of Bar is $\{\texttt{i32};\ \texttt{i32*};\ \texttt{i32}\}$. Flattening aggregates in this fashion supports more (type-safe) projections, and avoids the need for complicated path expressions to calculate offsets. Given a name environment Υ, we define a partial function $\flat(\tau)$ that flattens τ into a vector of primitive (non-aggregate) types.

$$\flat(\texttt{i}n) = [\texttt{i}n] \qquad \flat(\texttt{d32}) = [\texttt{d32}] \qquad \flat(\texttt{d64}) = [\texttt{d64}] \qquad \flat(\tau * \rho) = [\tau * \rho]$$

$$\flat(U(b) * \rho) = [U(b) * \rho] \qquad \flat([\tau \times n]) = \flat(\tau)\ \texttt{++}\ \cdots\ \texttt{++}\ \flat(\tau)$$

$$\flat(name\langle\overline{\rho}\rangle) \ = \ (\flat(\tau_1)\ \texttt{++}\ \cdots\ \texttt{++}\ \flat(\tau_n))\{\overline{\rho}/\overline{\rho}'\} \ \text{ when } \Upsilon(name) = \langle\overline{\rho}'\rangle\{\tau_1; \cdots ; \tau_n\}$$

Note that $\flat(-)$ stops when it encounters a pointer, and is thus well-founded when recursive uses of names are limited to positions under a pointer (as in C). Here, we have omitted details of padding and alignment, which are covered in our Coq development.

Basic blocks consist of a sequence of ϕ-nodes followed by a sequence of deterministic instructions, and end with a terminator instruction. Instructions manipulate operands which are either variables or constants, and are mostly derived from the LLVM IR. The instructions are organized into two categories: The first contains instructions modeled deterministically (i.e., functionally). The second category contains LLVM terminator instructions (i.e., control-flow operators) and instructions modeled non-deterministically (i.e., axiomatically) such as `poolalloc`. Unlike LLVM, we consider a `call` to terminate a block, as a `call` will generally have non-deterministic behavior. Another difference with LLVM is that our `getelementptr` instruction does not perform multistep indexing because the type environment for aggregate data structures is already flattened.

Functions abstract over a set of caller-provided regions, and begin by defining a set of local regions which are scoped over the lifetime of the function invocation. The syntax prevents new regions from being allocated in the function body. While the actual SAFECode system allows regions to be allocated anywhere, we opt for this design because in practice, the compiler usually emits `poolinit` instructions at the beginning of a function. We encode `alloca`s (stack allocations) as a `poolalloc` and `poolfree`.

One difference between our language and the SAFECode implementation is that we introduce a dedicated untyped pointer $U(b) * \rho$ to make it easier to express that we can safely dereference b bytes, but do not know what those bytes are. As a result, the syntax provides two versions of `poolcheck/U` and `getelementptr/U` to work with types in the typed-region case and bytes in the untyped-region case.

3.2 Representation of run-time values

All run-time values are represented as a list of bit strings:

$$\begin{array}{rcl} val & ::= & [v_1; v_2; \ldots; v_n] \\ v & ::= & \mathrm{bit}_n(i) \quad (i \in [0..2^n)) \end{array}$$

For example, the 16-bit integer `0xF00D` is represented as $[\mathrm{bit}_{16}(\texttt{0xF00D})]$, whereas a struct containing a 16-bit integer and 32-bit pointer $\{\texttt{0xBEEF}; \texttt{0x0BADF00D}\}$ is represented as $[\mathrm{bit}_{16}(\texttt{0xBEEF}), \mathrm{bit}_{32}(\texttt{0x0BADF00D})]$. Note that pointers and integers (of the appropriate size) have the same representation.

There are other possible representations. For instance, CompCert [8, 9] treats a pointer as a symbolic block number and an offset within that block, (b, o). This "Swiss cheese" model effectively enforces object isolation: we simply cannot get to an object at address a through pointer arithmetic on the base address of a different object b. Such a model makes sense when trying to define the formal semantics of C which makes actions, such as treating an integer as a pointer undefined.

Local environment	$env : \text{var} \rightharpoonup \text{val}$	Primitive types	$\mathcal{K} := \{\texttt{i}n, \texttt{d32}, \texttt{d64}, \tau * \rho,$
Region instantiation	$\sigma : \mathcal{P} \rightharpoonup \mathcal{R}$		$U(b) * \rho, \forall\rho.\overline{\tau} \rightarrow \tau\}$
Memory	M : `mem_t`	Region names	$\mathcal{P} := \{\rho_1, \ldots, \rho_n\}$
Live region set	$\mathcal{L} : \text{set } \mathcal{R}$	Run-time regions	$\mathcal{R} := \{r_1, \ldots, r_n\}$
Heap typing	Σ : `heap_t`	Heap type	`heap_t` $:= \mathbb{Z} \rightharpoonup (\mathcal{K}, \mathcal{R})$
	Execution context	E	$::= (\text{fn}, \text{blk}, env, \Sigma, \sigma, \mathcal{L})$
Execution stack	$S : \text{list } E$	Machine state	$ms ::= (M, E, S)$

Fig. 4. Components of the abstract machine.

In our case, SAFECode allows integers to be used as pointers provided that the compiler can statically prove that it is safe or it is guarded with a dynamic check. To support this behavior, we found the simplest approach for our model was to treat all run-time values as bit strings.

3.3 Abstract Machine

The components of our abstract machine are presented in Figure 4. A machine state is represented by the tuple (M, E, S), where M is the current memory, E is the current execution context, and S is the control stack of execution contexts. An execution context contains information relevant to the computation in the current stack frame, including the function definition (f), a currently executing basic block (b), and an environment (env) mapping variables to run-time values. The last three components of E are used to support regions. The first of these is the heap typing Σ, which maps addresses to a pair of a primitive type and region. Conceptually, the heap typing holds the run-time meta-data, allowing the system to perform a run-time check. Next, σ contains a mapping of the region variables written in code to their actual run-time regions. The component $\mathcal{L}$ represents the set of live regions.

3.4 Memory Model and Memory Management

At the level of our abstract machine, we do not want to specify how regions are represented, nor the meta-data that is needed to manage memory. Rather, we parameterized our development over an abstract memory management interface specified axiomatically. To ensure that the axioms are consistent, we implemented a simple allocation strategy and proved that the strategy satisfied the axioms.

We treat memory as a partial map from integers to bytes, paired with some allocator-specific meta-data of abstract type:

$$\texttt{mem\_t} : (\mathbb{Z} \rightharpoonup \texttt{byte}) \times \texttt{metadata\_t}$$

Conceptually, this meta-data can encode information such as the size of memory or a specific allocation strategy. In our instance of the memory model, we use

$$
\begin{array}{ll}
\texttt{mload}: & \texttt{mem\_t} \times \mathbb{Z} \times \mathcal{K} \to \text{val}_\perp \\
\texttt{mstore}: & \texttt{mem\_t} \times \mathbb{Z} \times \mathcal{K} \times \text{val} \to \texttt{mem\_t}_\perp \\
\texttt{mpoolalloc}: & \texttt{mem\_t} \times \text{set}\ \mathcal{R} \times \texttt{heap\_t} \times \overline{\mathcal{K}} \times \mathbb{N} \times \mathcal{R} \to (\mathbb{Z} * \texttt{mem\_t})_\perp \\
\texttt{mpoolfree}: & \texttt{mem\_t} \times \mathbb{Z} \to \texttt{mem\_t}_\perp \\
\texttt{mpoolinit}: & \texttt{mem\_t} \times \sigma \times \text{set}\ \mathcal{R} \times \texttt{heap\_t} \times (\mathcal{R} * \mathcal{K}) \to (\texttt{mem\_t} * \texttt{heap\_t} * \mathcal{R})_\perp \\
\texttt{mpooldel}: & \texttt{mem\_t} \times \text{set}\ \mathcal{R} \times \texttt{heap\_t} \times \text{set}\ \mathcal{R} \to (\texttt{mem\_t} * \texttt{heap\_t} * \text{set}\ \mathcal{R})_\perp \\
\texttt{mcheck}: & \texttt{heap\_t} \times \mathbb{Z} \times \mathcal{R} \times \overline{\mathcal{K}} \to \text{bool}
\end{array}
$$

Fig. 5. Memory signature. $\overline{X}$ indicates a list of values drawn from the domain X.

$$
\begin{array}{l}
\texttt{mload}\ M\ 0\ \tau = \perp \qquad\qquad \texttt{mstore}\ M\ 0\ \tau\ v = \perp \\
\texttt{mload}\ (\texttt{mstore}\ M\ a\ \tau\ v)\ a\ \tau = v \\
\texttt{mload}\ (\texttt{mstore}\ M\ a_1\ \tau_1\ v)\ a_2\ \tau_2 = \texttt{mload}\ M\ a_2\ \tau_2 \text{ when} \\
\quad [a_1, a_1 + \texttt{sizeof}(\tau_1)) \text{ disjoint } [a_2, a_2 + \texttt{sizeof}(\tau_2)) \\
\texttt{mload}\ (\#2\ \texttt{mpoolalloc}\ M\ \mathcal{L}\ \Sigma\ \overline{\tau}')\ a\ \tau = v \implies \texttt{mload}\ M\ a\ \tau = v \\
\texttt{mload}\ (\texttt{mpoolfree}\ M\ a)\ a'\ \tau = \texttt{mload}\ M\ a'\ \tau
\end{array}
$$

Fig. 6. Selected memory operation equations. We write #2 for second projection.

the meta-data to ensure that all addresses in use fall within the range of 64-bit machine integers to model a finite memory with 2^{64} addresses.

We summarize the memory operations and sketch the pre- and post-conditions in Figures 5, 6 and 7. The `mload` function reads $\texttt{sizeof}(\tau)$ bytes from the specified address and returns an optional value. The `mstore` function writes $\texttt{sizeof}(\tau)$ bytes coming from the specified value to the specified address and optionally returns the updated memory. Loading or storing to the null pointer 0 results in failure. We note the types are used strictly for size information.

The operations `mpoolalloc` and `mpoolfree` are used to allocate and free memory within a specified region. Unlike conventional allocation, `mpoolalloc` does not generate fresh locations, i.e., the heap typing remains invariant. Instead, it checks to see if there is a block of memory mapped in the heap typing at the specified type and region. If successful, `mpoolalloc` returns a pointer to that memory and can update its internal meta-data. The specification allows repeated calls to `mpoolalloc` for the same type to return the same pointer. In practice, one would use the internal meta-data for a freshness guarantee. Similarly, `mpoolfree` does not change the heap typing, but reflects that its meta-data might have changed to reclaim a set of addresses. This allows reclaimed addresses to be recycled for allocations of the same type.

The operation `mpoolinit` allocates fresh locations for a specified region. It first generates a region name fresh from $\mathcal{L}$. It then allocates fresh locations for that region, zeroes out the allocated memory, and updates the heap typing to

$$\{\ M(a) = b_1 \wedge \ldots \wedge M(a + \texttt{sizeof}(\tau)) = b_n\ \}$$
$$\texttt{mload}\ M\ a\ \tau = v$$
$$\{\ v = \texttt{endian}\ [b_1, \ldots, b_n]\ \}$$

$$\{\ \}$$
$$\texttt{poolalloc}\ M\ \mathcal{L}\ \Sigma\ \overline{\tau}\ n\ r = (a, M')$$
$$\{\ \texttt{mcheck}\ \Sigma\ a\ r\ \overline{\tau} = \text{true}\ \}$$

$$\{\ \exists v_2, \texttt{mload}\ M\ a\ \tau = v_2\ \}$$
$$\texttt{mstore}\ M\ a\ \tau\ v_1 = M'$$
$$\{\ M(a) = b_1 \wedge \ldots \wedge M(a + \texttt{sizeof}(\tau)) = b_n \quad \text{where } v_1 = \texttt{endian}(b_1, \ldots, b_n)\ \}$$

$$\{\ \}$$
$$\texttt{mcheck}\ \Sigma\ a\ r\ [\tau_1, \ldots, \tau_n] = \text{true}$$
$$\{\Sigma(a) = (\tau_1, r),\ \Sigma(a + \texttt{sizeof}(\tau_1)) = (\tau_2, r), \ldots\}$$

$$\{\ \}$$
$$\texttt{poolinit}\ M\ \sigma\ \mathcal{L}\ \Sigma\ (\rho, \tau) = (M', \Sigma', r)$$
$$\{\ a_1, \ldots, a_n \text{ fresh } \wedge\ r \notin \mathcal{L}\ \wedge\ \flat(\tau) = [\tau_1, \ldots, \tau_m]\ \wedge$$
$$\Sigma' = \{a_i \mapsto (\tau_1[\sigma], r)\} \uplus \{a_i + \texttt{sizeof}(\tau_1) \mapsto (\tau_2[\sigma], r)\} \uplus \ldots \uplus \Sigma\ \wedge$$
$$\texttt{mload}\ M\ a_i\ \tau = \bot \text{ for } i = 1, \ldots, n\ \wedge \texttt{mload}\ M'\ a_i\ \tau = 0 \text{ for } i = 1, \ldots, n\ \}$$

$$\{r \in \mathcal{L}\}$$
$$\texttt{mpooldel}\ M\ \mathcal{L}\ \Sigma\ r = (M', \Sigma', \mathcal{L}')$$
$$\{\ \mathcal{L} = r \cup \mathcal{L}'\ \wedge \Sigma = (a_1 \mapsto (\tau, r)) \uplus \ldots \uplus (a_n \mapsto (\tau, r)) \uplus \Sigma'\ \wedge$$
$$\texttt{mload}\ M'\ a_1\ \tau = \bot\ \wedge \ldots \wedge\ \texttt{mload}\ M'\ a_n\ \tau = \bot\}$$

Fig. 7. Axiomatic specification of memory operations. We use $M(a)$ to abbreviate the first projection of M at a, and $\tau[\sigma]$ to instantiate τ with regions specified in σ.

map the appropriate addresses to the appropriate types and region.[2] The operation `mpooldel` is the inverse of `mpoolinit`. It runs through the input heap typing and frees up all addresses mentioning the specified regions. Lastly, the operation `mcheck` models a run-time check. Given a heap typing, it verifies whether an address belongs to a region at the correct type. Note that `mcheck` works at the level of primitive types, so if the check returns true for a specified address, region and list of primitive types, it will also return true for any truncation of the sequence, holding the other parameters constant.

3.5 Operational Semantics

We structure the semantics as an evaluation function (Coq function) and a small-step relation (inductive predicate) on abstract machine states.

Figure 8 shows selected definitions from our `evalblock` function, which computes the resulting memory and local environment after evaluating all deterministic instructions in a basic block. The parts of the abstract machine not mentioned such as the control stack remain invariant when executing a basic block. There are two distinguished failure states `Err` and `Halt`. The `Err` state denotes a stuck state that our soundness theorem will rule out. A transition to the `Err` state can occur if we look up a variable that is not bound in the environment. A transition to the `Halt` state indicates that the run-time has prevented

[2] In our Coq development, we break this operation into two parts, one for fresh region creation, and one for region allocation to support mutually recursive regions.

```
evalblock nil M env =
  Ok(M, env)
evalblock (ι :: ῑ) M env =
  match eval ι M env with
    | Ok(M′, env′) =>
      evalblock ῑ M′ env′
    | ans => ans
```

```
eval [x = poolcheck ρ τ o] M env =
  match eval_op o env with
    | Ok(bit(a)) =>
      if mcheck Σ a σ(ρ) τ[σ]
      then Ok(M, env[x ↦ ptr(a)])
      else Halt
    | _ => Err
```

Fig. 8. Selected evaluation rules. We use the notation $\tau[\sigma]$ to indicate applying a region instantiation map σ to τ.

an unsafe memory operation. For example, a call to `poolcheck` may fail, halting the system. The functions describing instruction evaluation are straightforward, and make use of the definitions from our memory model in the case of memory instructions, or machine arithmetic in the case of binary operations.

Figure 9 shows almost all of our operational rules, omitting just the failure transitions for `call` and `poolalloc`, and the `unsafecall` and branch cases. A function call may fail if we run out of memory to hold local regions, while a poolalloc may fail if a specific region runs out of memory. The small-step semantics is compact because it pushes most of the work into the evaluation function. This is beneficial for model validation since the evaluation function is already executable. We only need to write an interpreter for the terminator instructions such as branch and return (which we can't do in Coq since it might diverge), and prove that the driver for terminator instructions respects the relation. Furthermore, if we grow a language by adding additional instructions, these instructions are unlikely to be terminator instructions, so validation should continue to scale.

4 Type System

Our goal to support real SAFECode programs means that the type system is quite complex. For the sake of brevity, we present a subset of the rules here and elide many details that are present in our Coq-development, such as sub-typing relations, primitive type manipulations, and contexts related to ϕ-node typing.

4.1 Typing rules

At its core, the type system tracks region lifetimes and ensures that type-homogeneity is preserved for typed-regions. The key idea is that pointers into live regions can always be safely dereferenced at the appropriate type. The rules used to define the typing judgments are sketched in Figure 10. Throughout, we assume a context including the function table F mapping function names to their definitions, and a type environment Υ mapping named types to their

$$\text{EVAL}\quad \frac{\texttt{evalblock}\ E.b.\bar{\iota}\ M\ E.env = (M', env')}{(M, E, S) \to (M', E[env := env'], S)} \qquad \text{EVAL-HALT}\quad \frac{\texttt{evalblock}\ E.b.\bar{\iota}\ M\ E.env = \texttt{Halt}}{(M, E, S) \to \texttt{Halt}}$$

$$\text{RETURN}\quad \frac{\texttt{mpooldel}(M, \mathcal{L}, \Sigma) = (M', \Sigma'', \mathcal{L}'') \qquad env(x_1) = v}{\begin{array}{c}(M, (f, [b.\iota = \texttt{return}\ \tau\ x_1], env, \Sigma, \sigma\mathcal{L}),\\ (f', [b'.nd = x_2 = \texttt{call}\ \tau'\ o'\langle\overline{\rho}\rangle(\overline{x}), l')], env', \Sigma', \sigma', \mathcal{L}') :: S) \to\\ (M', (f', [b'.tm = \texttt{br}\ l'], (x_2 \mapsto v) \cup env', \Sigma'', \sigma', \mathcal{L}''), S)\end{array}}$$

$$\text{CALL}\quad \frac{\begin{array}{c}F(o) = \tau\ f'\langle\overline{\rho}_p\rangle(\overline{y} : \overline{\tau}_{\overline{\rho}_p})\{\overline{\rho}_l = \texttt{poolinit}\ \overline{\tau}_{\overline{\rho}_p \cup \overline{\rho}_l}; \text{body} : \ell \to \text{blk}\}\\ b' = f'.entry \qquad env' = env[\{\overline{y} \mapsto env(\overline{x})\} \qquad \{\overline{\rho}_p \mapsto \sigma(\overline{\rho})\} = \sigma'_e\\ \{\overline{\rho}_l \mapsto rgns'\} \cup \sigma'_e = \sigma' \qquad \texttt{mpoolinit}(M, \sigma', \mathcal{L}, \Sigma, \overline{\rho}_l * \overline{\tau}_{\overline{\rho}_p \cup \overline{\rho}_l}) = (M', \Sigma', rgns')\end{array}}{\begin{array}{c}(M, (f, [b.\iota = (x_1 = \texttt{call}\ \tau'\ o'\langle\overline{\rho}\rangle(\overline{x}), l')].env, \Sigma, \sigma, \mathcal{L}), S) \to\\ (M, (f', b', env', \Sigma', \sigma'_b, rgns' \cup \mathcal{L}), (f, [b.\iota = (x_1 = \texttt{call}\ \tau'\ o'\langle\overline{\rho}\rangle(\overline{x}), l')], env, \Sigma, \sigma', \mathcal{L}) :: S)\end{array}}$$

$$\text{POOLALLOC}\quad \frac{\flat(\tau) = \overline{\tau} \qquad \texttt{mpoolalloc}(M, lo, \mathcal{L}, \Sigma, \overline{\tau}[\sigma], n, \sigma(r)) = (a, M')}{\begin{array}{c}(M, (f, [b.nd = (x = \texttt{poolalloc}\ \tau, \rho, n, \ell)], env, \Sigma, \sigma, \mathcal{L}), S) \to\\ (M, (f, [b.tm = \texttt{br}\ \ell], env[(x \mapsto \text{ptr}(a))], \Sigma, \sigma, \mathcal{L}), S)\end{array}}$$

Fig. 9. Selected operational rules. The function table F remains constant throughout operation and is not shown in the rules. The notation $a.b$ projects b from a.

definitions. In addition, the judgments use region contexts Δ to determine the region variables in scope, variable contexts Γ mapping variables to their types, and a label environment Ψ mapping block labels to their preconditions.

The well-formedness rules for types and instruction operands are straightforward. A type is well-formed ($\Delta \vdash \tau$) if all the regions that appear free in the type are in the region context Δ. Function types are required to be locally closed. That is, the argument and return types may only depend upon the region variables bound by the function. The second judgement ($\Delta \vdash o : \tau$) determines when operands are well-formed at a type.

The judgment for typing instructions has the form $\Delta; \Gamma \vdash \iota : \Gamma'$. For example, the load rule checks that the operand has pointer type $\tau * \rho$ and that the pointer type is well-formed. The postcondition guarantees that a τ value has been loaded and passes that context forward to the next instruction. The postcondition for `poolcheck` says that no matter what the operand is, we can add the checked type into the context. There is a run-time check to ensure that this rule is sound.

The `poolalloc` rule is conceptually similar to a malloc rule where the result is a pointer of the correct type. The rule additionally checks that the type requested corresponds to the region's type. Thus, we cannot allocate an `i32` in a region that holds {`i32`; `i32`}, although we can load an `i32` out. Note that a request for n objects of type τ only reveals that the pointer to the front of that object is valid. This captures the essence of SAFECode that type-safety is guaranteed

$\boxed{\Delta \vdash \tau}$ Well-formed type

$$\frac{}{\Delta \vdash \texttt{i}n} \qquad \frac{}{\Delta \vdash \texttt{d32}} \qquad \frac{}{\Delta \vdash \texttt{d64}} \qquad \frac{\Delta \vdash \tau \qquad \rho \in \mathrm{dom}(\Delta)}{\Delta \vdash \tau * \rho} \qquad \frac{\rho \in \mathrm{dom}(\Delta)}{\Delta \vdash U(b) * \rho}$$

$$\frac{\Upsilon(name) = \overline{\tau} \qquad \forall \rho \in \overline{\rho}, \rho \in dom(\Delta)}{\Delta \vdash name\langle\overline{\rho}\rangle} \qquad \frac{\overline{\rho} \vdash \overline{\tau} \to \tau}{\Delta \vdash \forall\overline{\rho}.\tau \to \tau} \qquad \frac{\Delta \vdash \tau \qquad n \neq 0}{\Delta \vdash [\tau \times n]}$$

$\boxed{\Gamma \vdash o : \tau}$ Well-formed operand (selected rules)

$$\frac{\{\tau' * \rho, U(b) * \rho\} \notin \tau}{\Gamma \vdash \texttt{undef}(\tau) : \tau} \qquad \frac{F(f) = f\langle\overline{\rho}\rangle(\overline{x} : \overline{\tau})\{body\} \to \tau}{\Gamma \vdash f : \forall\overline{\rho}.\overline{\tau} \to \tau} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash \mathrm{reg}\ x : \tau}$$

$\boxed{\Delta; \Gamma \vdash \iota : \Gamma}$ Well-formed instruction (selected rules)

$$\frac{\Delta; \Gamma \vdash o : \tau * \rho \qquad \Delta \vdash \tau * \rho}{\Delta; \Gamma \vdash x = \texttt{load}\ (\tau * \rho)\ o : \Gamma[x \mapsto \tau]} \qquad \frac{\Delta; \Gamma \vdash o : \tau' \qquad \Delta \vdash \tau * \rho}{\Delta; \Gamma \vdash x = \texttt{poolcheck}\ \rho\ \tau\ o : \Gamma[x \mapsto \tau * \rho]}$$

$\boxed{\Delta; \Psi; \Gamma \vdash tm}$ Well-formed terminator instruction (selected rules)

$$\frac{\Gamma \vdash o : \tau \qquad \Delta \vdash \tau}{\Delta; \Psi; \Gamma \vdash \texttt{return}\ \tau\ o} \qquad \frac{\Delta; \Psi; \Gamma[x \mapsto \tau * \rho] \vdash \texttt{br}\ \ell \qquad \Delta(\rho) = \tau}{\Delta; \Psi; \Gamma \vdash x = \texttt{poolalloc}\ \tau, \rho, n, \ell}$$

$$\frac{\begin{array}{c}\Gamma \vdash o : \forall\rho_p.\ \overline{\tau}_{\rho_p} \to \tau'_{\rho_p} \qquad \Gamma \vdash \overline{o} : \overline{\tau}[\overline{\rho}/\overline{\rho_p}] \\ \forall\rho \in \overline{\rho}.\ \rho \in dom(\Delta) \qquad \tau = \tau'[\overline{\rho}/\overline{\rho_p}] \qquad \flat(\tau) = \overline{\tau} \qquad \Delta; \Psi; \Gamma[x \mapsto \tau] \vdash \texttt{br}\ \ell\end{array}}{\Delta; \Psi; \Gamma \vdash x = \tau\ \texttt{call}\ o\langle\overline{\rho}\rangle(\overline{o}), \ell}$$

$\boxed{\Psi; \Delta_p; \Delta_l \vdash \mathrm{func}}$ Well-formed function

$$\frac{\begin{array}{c}\overline{\rho}_p \cap \overline{\rho}_l = \emptyset \qquad \Delta_p \vdash \overline{\rho}_p \qquad \Delta_l \vdash \overline{\rho}_l \qquad \Delta_p \subseteq \Delta_l \\ \forall\tau \in \overline{\tau}_l.\ \Delta_l \vdash \tau \qquad \Delta_p \vdash \tau \qquad \forall\tau \in \overline{\tau}_1.\ \Delta_1 \vdash \tau \qquad \forall\ell \in dom(\mathrm{body}).\ \Psi; \Delta_l \vdash \mathrm{body}(\ell)\end{array}}{\Psi; \Delta_p; \Delta_l \vdash \tau\ f\langle\overline{\rho}_p\rangle(\overline{x} : \overline{\tau}_1)\{\overline{\rho}_l = \texttt{poolinit}\ \overline{\tau}_2; \mathrm{body} : \ell \to \mathrm{blk}\}}$$

$\boxed{F_\Psi; F_\Delta \vdash \mathrm{prog}}$ Well-formed program

$$\frac{F_\Psi(f_i) = \Psi_f \qquad F_\Delta(f_i) = (\Delta_p, \Delta_l) \qquad \Psi_f; \Delta_p; \Delta_l \vdash f_i,\ \text{for } i = 1, \ldots, n}{F_\Psi; F_\Delta \vdash \{f_1, \ldots, f_n\}}$$

Fig. 10. Selected typing rules.

at the region-level, not for individual pointers. The call rule fully instantiates the function's polymorphic regions and then checks that the arguments have the appropriate types.

The typing rule for a function declaration (Figure 10) imposes a LIFO ordering on region lifetimes. The rule uses two region contexts Δ_p and Δ_l to accomplish this. Δ_p mentions only the regions the function is polymorphic in, while Δ_l extends Δ_p with locally allocated regions. We type the function signature and return type under Δ_p and the function body under context Δ_l. This ensures that regions never escape from callees to callers and that the only regions in scope of a function body are live. Typing for a function body (ommited) is done in a straightforward manner by typing all basic blocks in the body using the rules for deterministic and terminator instructions. The top-level typing rule ensures that all functions are well-formed. It introduces two new contexts $F_\Psi : f \rightharpoonup \Psi$ and $F_\Delta : f \rightharpoonup (\Delta_p, \Delta_l)$. The former maps a function to its basic block preconditions. The latter maps a function to its appropriate region contexts. The top-level mapping ensures that mutually recursive functions use consistent contexts.

4.2 Type Soundness

The most difficult part of the proof reduces to arguing about the LIFO structure of region lifetimes to ensure that pointers only point into live regions. As functions can only be declared at the top-level in LLVM, we do not need to worry about regions escaping through closures. The key invariant to highlight is that the heap typing stack is well-formed.

Definition (Well-formed stack: heap typing) For any two adjacent execution contexts E_1 and E_2, where E_1 is the callee frame and E_2 is the caller frame,

$$
\begin{array}{ll}
\text{(a)} & E_2.\Sigma \subseteq E_1.\Sigma \\
\text{(b)} & E_2.\mathcal{L} \cup \{r_1 \ldots r_n\} = E_1.\mathcal{L} \ \wedge E_2.\mathcal{L} \cap \{r_1 \ldots r_n\} = \emptyset \\
\text{(c)} & E_2.\mathcal{L} \vdash E_2.\Sigma \\
\text{(d)} & E_1.\mathcal{L} \vdash E_1.\Sigma
\end{array}
$$

In words, the heap typing increases monotonically (in terms of addresses mapped) as we move from caller to callee frames in the execution stack. Similarly, the live region set grows monotonically. Lastly, the regions mentioned in an execution context's heap typing are closed under its respective live region set.

We now state the main lemmas that are used in the proof of type soundness.

Lemma 1. (Progress and Preservation of basic block evaluation) If $F_\Psi; F_\Delta \vdash \{f_1, \ldots, f_n\} \wedge \vdash (M, E, S)$, then either `evalblk` $E.b.\bar{\imath}\ M\ E.env = \mathrm{Ok}(M', env') \wedge \vdash (M', E[env := env'], S)$ or `evalblk` $E.b.\bar{\imath}\ M\ E.env = \mathrm{Halt}$.

The proof is mechanized in Coq, but the interesting bit is that our mixed semantics allows us to prove progress and preservation simultaneously as proving that our evaluation function `evalblk` preserves the invariants also implies that we must be able to take a step. One nice property of this structure is if we extend

the language by adding additional instructions to `evalblk`, we only need to modify our proof of type soundness in one place. This was particularly useful when we were scaling our language out to handle real programs, as many instructions that we added later (e.g. `insertvalue`) required minimal proof changes.

Lemma 2. (Preservation) If $F_\Psi; F_\Delta \vdash \{f_1, \ldots, f_n\} \wedge \vdash (M, E, S) \wedge (M, E, S) \rightarrow (M', E', S')$, then $\vdash (M', E', S')$.

Recall that the core of the proof reduces to arguing about LIFO region lifetimes. As our abstract memory interface specifies that `mpoolinit` (used on a function call) and `mpooldel` (used on a return) are inverses of each other with respect to the heap typing, the proof reduces to invoking this fact to argue that a callee returns the heap-typing to the state expected by the caller.

Lemma 3. (Progress) If $F_\Psi; F_\Delta \vdash \{f_1, \ldots, f_n\} \wedge \vdash (M, E, S)$, then either $(M, E, S) \rightarrow (M', E', S') \wedge \vdash (M', E', S')$ or $(M, E, S) \rightarrow$ Halt.

This lemma has few cases because of the small operational semantics and is straightforward to prove. Our soundness result implies that a pointer with type $\tau * \rho$ in a well-typed program always points into region ρ and references a τ.

5 Evaluation

The previous section described a declarative type system and argued that it is sound. We have also built an algorithmic type system `tc` (i.e., a type-checker as a function) and proved that it respects the declarative typing rules:

Lemma 4. (Type-checker sound) If $\texttt{tc}(F_\Psi, F_\Delta, \{f_1, \ldots, f_n\}) = \text{true}$, then $F_\Psi; F_\Delta \vdash \{f_1, \ldots, f_n\}$.

The type-checker is straightforward to write and prove sound. It can be extracted as an executable OCaml program to serve as a verified type checker for SAFECode. Unfortunately, the SAFECode compiler emits code that does not adhere strictly to the SAFECode type system. For ease of code generation, the compiler erases almost all region and type information from the LLVM bitcode. This significantly increases the difficulty of applying our type-checker as we now need to perform type-inference.

We had to write two pieces of code to close this gap. First, we wrote an LLVM pass in C++ that crawls SAFECode's internal structures that annotates the resulting code with region information. Second, we wrote an OCaml pass that performs type-inference and translates the input LLVM bitcode into our representation. The bulk of the OCaml pass is dedicated to reconstructing the types that SAFECode erases. In principle, we should not need to write this pass because conceptually, the SAFECode compiler produces this typing derivation when instrumenting code. Our type-checker checks the output of these two pieces of code. The Coq formalization is about 12000 lines of code, while our OCaml translation and inference pass is about 4800.

5.1 Experimental Results

In addition to the two pieces of code, we had to make a few simplifications that possibly introduce unsoundness into the system to type-check real code. First, we translate calls to library functions where we do not have the code or are not instrumented by SAFECode into `unsafecall`s. We also cannot type variable argument functions and certain LLVM intrinsic functions and translate those to `unsafecall`s as well.[3] Lastly, to keep our type system from becoming too unwieldy, we choose not to add a typing context to handle aliases. In practice, SAFECode sometimes calls `poolcheck` on a variable x, and later uses an alias of x without a check. As a workaround, every time a `poolcheck` is encountered during translation, we emit an extra `poolcheck` for x's aliases.

We ran our type-checker on micro-benchmarks included with the SAFECode distribution and on the Olden benchmarks [2], a pointer-intensive test suite on which the original SAFECode system was evaluated. We discovered some bugs with the current SAFECode[4] instrumentation of a few programs in the Olden benchmarks, mostly with region instantiations. In the program *bh*, we found that a call to a region-polymorphic function was not instantiated with a region. In the program *em3d*, we found that a pointer to a function-local pool was allocated and returned to the caller, violating the LIFO region invariant. We also discovered some false-positives. For example, in the program *perimeter*, SAFECode performs an interval-analysis over multiple program paths to determine that an array index variable is statically in-bounds. This is a limitation of our type-checker, as it cannot reason about the above analysis. Although our type-checker is still a prototype, it is already effective at finding bugs.

6 Related and Future Work

Zhao et al. [13] presented a semantics for the LLVM IR formalized in Coq and used it to prove the correctness of a closely related, but alternative technique for enforcing spacial memory safety on C code called Softbound [10]. On the one hand, their proof is more impressive because it shows the correctness of the transformation. On the other hand, their model for LLVM's IR cannot handle idioms that arise in real C/C++ programs, (e.g., casting a pointer to an integer and then back) because their treatment of memory is too high-level. Furthermore, our type-checker can be used not only to validate the initial transformation of the code, but also the code that comes out of subsequent optimizations.

In addition to SAFECode, there is a rich history of region-based type systems, first pioneered by Tofte and Talpin [12] and later refined in Cyclone [6] that our type system draws inspiration from. In many regards, our type system is much simpler because regions can only be passed "downwards" to functions and never returned. Furthermore, SAFECode does not support lexically scoped

[3] Consequently, preservation holds only on code that does not contain `unsafecall`s.

[4] The system we evaluated our type-checker on is the most current implementation and not the one presented in the original paper.

closures or existential types, so there is no need for type-and-effects systems. Languages such as Cyclone had many more cases in which region names could escape function scope, and thus required a much more complicated type system. In contrast, the regions in SAFECode and our type system are type-homogenous and allow explicit deallocation of memory in these regions, operations which were not permitted except in restricted cases in those languages.

In our future work, we hope to thoroughly test our semantics to make sure that it is compatible with the actual semantics implemented by the LLVM compiler and SAFECode run-time system. However, we structured our semantics to lighten the burden of validation as explained in Section 3.

Acknowledgements We thank Gregory Malecha, John Criswell, Joseph Tassarotti, Stephen Chong and our reviewers for their helpful discussions.

References

1. Coq Proof Assistant. http://coq.inria.fr/.
2. M. C. Carlisle. Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines. *PhD thesis*, 1996.
3. Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast Byte-Granularity Software Fault Isolation. In *Proc.*, SOSP '09, 2009.
4. John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proc.*, SOSP '07, 2007.
5. Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. In *Proc.*, PLDI '06, 2006.
6. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-Based Memory Management in Cyclone. In *Proc.*, PLDI '02, 2002.
7. Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc.*, International Symposium on Code Generation and Optimization (CGO'04), Mar 2004.
8. Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7), July 2009.
9. Xavier Leroy and Sandrine Blazy. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *J. Autom. Reason.*, 41(1), July 2008.
10. Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proc.*, PLDI '09, 2009.
11. George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proc.*, POPL '02, 2002.
12. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2), February 1997.
13. Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proc.*, POPL '12, 2012.