From MINX to MINC: Semantics-Driven Decompilation of Recursive Datatypes

Anonymous

Institute of the Unknown, The Closed University

incognito@unknown.closed.com

Abstract

Reconstructing the meaning of a program from its binary executable is known as reverse engineering; it has a wide range of applications in software security, exposing piracy, legacy systems, etc. Since *reversing* is ultimately a search for meaning, there is much interest in inferring a type (a meaning) for the elements of a binary in a consistent way. Unfortunately existing approaches do not guarantee any semantic relevance for their reconstructed types.

This paper presents a new and semantically-founded approach that provides strong guarantees for the reconstructed types. Key to our approach is the derivation of a witness program in a high-level language alongside the reconstructed types. This witness has the same semantics as the binary, is type correct by construction, and it induces a (justifiable) type assignment on the binary. Moreover, the approach effectively yields a type-directed decompiler.

We formalise and implement the approach for reversing MINX, an abstraction of x86, to MINC, a type-safe dialect of C with recursive datatypes. Our evaluation compiles a range of textbook C algorithms to MINX and then recovers the original structures.

1. Introduction

Reverse engineering is the activity of reconstructing the behaviour of a program from its binary executable. Quite apart from illicit purposes such as removing copyright protection, reversing has legitimate applications which include, but are not limited to: understanding the operation of, and threat posed by, viruses and malware; exposing flaws and vulnerabilities in commercial software, especially prior to deployment in government or industry; checking for license infringement; and interfacing with legacy software.

Reversing amounts to searching for a high-level meaning that is consistent across a binary. Typing, likewise, checks the elements of a program combine in a consistent, meaningful way, yielding a valuable program abstraction for the reverse engineer [15]. For example, datatypes can guide test-generation in fuzzing [26], help locate information in a core dump in memory-based forensics [4], and support program reconstruction [6, 21]. Unfortunately, type recovery has been more make-shift and make-do than a discipline shaped by formal principles. IDAPro, the leading commercial disassembler, applies heuristics to assign simple types to locals [9]. REWARDS [18] recovers types from a single execution trace, which sheds no light on other traces. TIE [15] badges itself as being principled, but does not relate its type judgements to the semantics of the binary (which remain unspecified). Yet a firm semantic footing for these type systems is essential: a type recovery system which derives an incorrect type can easily mislead a reverse engineer undertaking a security audit, or misdirect a fuzzer into the wrong search space. Furthermore, these existing type systems [9, 15, 18] are unable to recover recursive datatypes.

The consensus is that types assigned to the binary should correspond to the original types of the source. But, needless to say, this is unavailable. This begs the question: *what does it mean for the types to be correct, if there is no source to check correctness against?*

We answer this question from a semantic perspective by constructing a witness program in a type-safe high-level language. The witness is not an arbitrary program, but carefully constructed to semantically coincide with the binary. Then, by proving that the witness is type-correct, we unequivocally establish that the binary inhabits the recovered types. Structured operational semantics (SOS) define our exemplar low-level and high-level languages, inspired by x86 and C respectively. The centrepiece of our formalisation is a decompilation relation that defines how the witness faithfully mimics the executable, and under what conditions. Together these components add up to a semantically-justified type-based decompiler. In summary, we make the following contributions:

1. We present a novel semantics-driven approach to type recovery that validates the types inferred for a low-level (MINX) program with a high-level witness (MINC) program. Unique to our work is a rigorous connection from our MINX binary to our MINC witness, founded on three key semantic components:
   (a) *an SOS for* MINX, designed as an abstraction of x86 to elucidate crucial control-flow details, such as the argument passing convention, needed to show that the MINX and MINC memories remain truly in sync;
   (b) an *SOS and static type system for* MINC, a type-safe dialect of C designed to illustrate decompilation to pointer arithmetic and the recovery of recursive structures;
   (c) a *decompilation relation*, that conservatively specifies when a MINX program corresponds to a MINC program.
2. In two steps we formally prove that the MINX program inhabits the recovered types:
   (a) First we show that the witness program is *type-correct* for the derived types.
   (b) Then we establish the *operational equivalence* of the MINC witness program and the original MINC program in the form of memory consistency.

[Copyright notice will appear here once 'preprint' option is removed.]

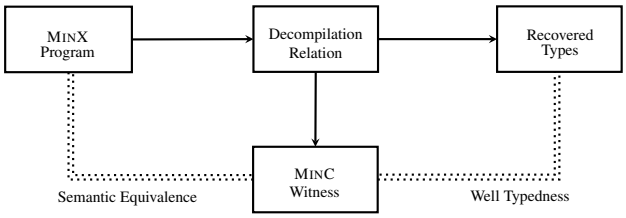


Figure 1: System overview

3. We distil a type-based decompiler from our decompiler relation and demonstrate the potential of our approach.
   (a) We show how non-deterministic choices in the relation can be replaced with constraint propagators to give a *solver* that incrementally infers the witness and the types; the resulting solver is thus solidly based on the decompilation relation.
   (b) We apply the solver to MinX binaries generated from over 21 textbook [30] C programs that manipulate splay trees, treaps, pairing heaps, etc. *All* (recursive) datatypes are successfully recovered.

2. System and Paper Overview

Figure 1 illustrates how the components of our type recovery system fit together. Solid arrows indicate flow whereas dotted lines indicate bi-directional semantic connections.

The decompilation relation sits at the heart of the diagram. It relates an input MinX program to two outputs: the recovered types, and the MinC witness program. The latter is subservient to the former, since its role (in type recovery) is to justify the recovered types. The decompilation relation is exactly that, a mathematical relation, that specifies what it means for a MinX program to be in correspondence with a MinC program. Nevertheless, a solver can be constructed, in conformance with the relation, which, given the input MinX program, computes the two outputs. Hence the annotated direction of flow.

The semantic connection on the right indicates that the MinC witness program inhabits the recovered types; the connection on the left expresses that the MinX program is semantically equivalent to the witness, in the sense that their memories remain in sync. The whole construction semantically relates the MinX program to the recovered types; a connection that follows by transitive closure.

The structure of the paper reflects the diagram; components are covered as we sweep the diagram left-to-right. Starting on the left, SOS for MinX programs are detailed in Section 3; followed by SOS for MinC programs in Section 4. With these semantic foundations in place, the decompiler compilation is introduced in Section 5. Section 6 concerns the automation of the relation and Section 7 and assesses the quality of the recovered types, on the right. We reflect on the state of related work in Section 8, before discussing the limitations of the method, along with possible remedies, in Section 9. Finally, Section 10 concludes.

3. The MinX Language

We introduce a lean instruction set, called MinX, to illustrate our semantic construction. MinX abstracts by design away from control-flow details that distract from the main goal of recovering recursive datatypes and that would otherwise make the presentation unmanageable.

Hence, the instruction set supports an unbounded number of registers. This means that register spilling does not need to be considered and SSA conversion [28] can be avoided. Also MinX fixes a single abstract calling convention, which avoids the need for argument detection techniques [1, 7]. These abstractions can be introduced in actual binaries by preprocessing steps.

3.1 Memory

An important challenge that we do not sidestep is that of differently-sized values. At the binary level, the field of a structure is accessed by a byte offset, which depends on the sizes of the data objects that precede it. This layout problem is intrinsic to type recovery and thus we assume memory is organised into bytes, and permit data objects to straddle contiguous bytes. We thus let $Bit = \{0, 1\}$ and define $Byte = Bit^8 \cup \{\bot\}$ where $\bot$ denotes a single byte of uninitialised (random) memory. A word is then a vector of bytes, that is, $Word = Byte^4$. The operator : concatenates vectors of bytes (and single bytes).

Register Bank The set of registers, each of which is of word size, is denoted $Regs = \{r_0, r_1, \ldots\}$. A function $R : Regs \to Word$ maps registers to the words they contain. To manipulate 1 byte and 2 byte objects within a register, we introduce an accessor function $R_{i:j} : Regs \to Byte^*$ which slices from byte i up to but not including byte j of a given register r. This is defined by $R_{i:j}(r) = w'$ where $R(r) = \vec{b} : \vec{b}' : \vec{b}''$ and

$$\vec{b} \in Byte^i \qquad \vec{b}' \in Byte^{j-i} \qquad \vec{b}'' \in Byte^{4-j}$$

For the register map, R, we define a notion of partial update in which only the least significant $w = |\vec{b}|$ bytes of register r are updated with the bytes $\vec{b}$ as follows:

$$R \circ_w \{r \mapsto \vec{b}\} = R \circ \{r \mapsto \vec{b} : R_{w:4}(r)\}$$

Heap Memory The heap is modelled as a (partial) function $H : Word \to Byte$ and therefore is byte addressable. To read stored objects that straddle w consecutive bytes we define a helper function $H^w : Word \to Byte^*$ that reads and amalgamates s bytes of the heap into a single vector as follows:

$$H^w(a) = \begin{cases} \{\bot\}^w & \text{if } \bot \in a \\ H(a +_4 w -_4 1) : \cdots : H(a) & \text{otherwise} \end{cases}$$

where the operations $+_4$ and $-_4$ denote addition and subtraction in 4 byte bit-vector arithmetic, and 1 denotes a 4 byte bit-vector.

3.2 Syntax

The syntax of MinX programs consists of four syntactic categories. The first, $w ::= 1 \mid 2 \mid 4$, denotes the width, in bytes, of the primitive data objects that are supported by the instruction set. The second, ι, defines the instructions themselves:

$$\begin{array}{lllll} \iota ::= & \mathsf{mov}_w & r, c & \mid \mathsf{mov}_w & r_i, r_j \\ \mid & \mathsf{mov}_w & r_i, [r_j] & \mid \mathsf{mov}_w & [r_i], r_j \\ \mid & \mathsf{mov}_w & r_i, [r_j + c] & \mid \mathsf{mov}_w & [r_i + c], r_j \\ \mid & \mathsf{eq}_w & r_i, r_j, r_k & \mid \mathsf{op}^{\oplus}_w & r_i, r_j * c \\ \mid & \mathsf{op}^{\otimes}_w & r_i, r_j & \mid \mathsf{op}^{\otimes}_w & r_i, c \\ \mid & \mathsf{alloc} & r_i, r_j & \mid \mathsf{alloc} & r_i, r_j * c \\ \mid & \mathsf{call} & r_u, a, \vec{r_v} & & \end{array}$$

where c denotes a numeric constant and $a \in Word$ is the location of the function that is to be invoked. (A Harvard architecture is assumed throughout). Square brackets indicate indirection. The instructions $\mathsf{op}^{\oplus}_w$ and $\mathsf{op}^{\otimes}_w$ are themselves parameterised by the categories $\oplus \in \{+, -\}$ and $\otimes \in \{+, -, *, /, \&, |, \ldots\}$, and the width w of their operands. The third category, b, defines blocks:

$$b ::= \iota; b \mid (\mathsf{if}_w\ r_i\ \mathsf{goto}\ a); b \mid \mathsf{goto}\ a \mid \mathsf{ret}$$

$$\boxed{\vec{R} \vdash \langle H, R, \iota\rangle \xrightarrow{\iota} \langle H', R'\rangle}$$

$$\text{ex-mov-rc}\ \frac{R' = R \circ_w \{r_i \mapsto c\}}{\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, c\rangle \xrightarrow{\iota} \langle H, R'\rangle} \qquad \text{ex-mov-rr}\ \frac{R' = R \circ_w \{r_i \mapsto R_{0:w}(r_j)\}}{\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, r_j\rangle \xrightarrow{\iota} \langle H, R'\rangle}$$

$$\text{ex-mov-ri}\ \frac{R' = R \circ_w \{r_i \mapsto H^w(R(r_j))\}}{\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, [r_j]\rangle \xrightarrow{\iota} \langle H, R'\rangle} \qquad \text{ex-mov-ir}\ \frac{\bot \notin H^4(R(r_i)) \quad H' = H \circ \{H^4(R(r_i)) +_4 n \mapsto R_{n:n+1}(r_j)\}_{n=0}^{w-1}}{\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ [r_i], r_j\rangle \xrightarrow{\iota} \langle H', R\rangle}$$

$$\text{ex-mov-r+}\ \frac{R' = R \circ_w \{r_i \mapsto H^w(R(r_j) +_4 c)\}}{\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, [r_j + c]\rangle \xrightarrow{\iota} \langle H, R'\rangle} \qquad \text{ex-mov-+r}\ \frac{\bot \notin H^4(R(r_i)) \quad H' = H \circ \{H^4(R(r_i)) +_4 c +_4 n \mapsto R_{n:n+1}(r_j)\}_{n=0}^{w-1}}{\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ [r_i + c], r_j\rangle \xrightarrow{\iota} \langle H', R\rangle}$$

$$\text{ex-eq-true}\ \frac{R_{0:w}(r_j) = R_{0:w}(r_k) \quad R' = R \circ_w \{r_i \mapsto 1\}}{\vec{R} \vdash \langle H, R, \mathsf{eq}_w\ r_i, r_j, r_k\rangle \xrightarrow{\iota} \langle H, R'\rangle} \qquad \text{ex-eq-false}\ \frac{R_{0:w}(r_j) \neq R_{0:w}(r_k) \quad R' = R \circ_w \{r_i \mapsto 0\}}{\vec{R} \vdash \langle H, R, \mathsf{eq}_w\ r_i, r_j, r_k\rangle \xrightarrow{\iota} \langle H, R'\rangle}$$

$$\text{ex-}\oplus\text{-r*}\ \frac{R' = R \circ_w \{r_i \mapsto R_{0:w}(r_i) \oplus_w (R_{0:w}(r_j) *_w c)\}}{\vec{R} \vdash \langle H, R, \mathsf{op}^{\oplus}_w\ r_i, r_j * c\rangle \xrightarrow{\iota} \langle H, R'\rangle}$$

$$\text{ex-}\otimes\text{-rc}\ \frac{R' = R \circ_w \{r_i \mapsto R_{0:w}(r_i) \otimes_w c\}}{\vec{R} \vdash \langle H, R, \mathsf{op}^{\otimes}_w\ r_i, c\rangle \xrightarrow{\iota} \langle H, R'\rangle} \qquad \text{ex-}\otimes\text{-rr}\ \frac{R' = R \circ_w \{r_i \mapsto R_{0:w}(r_i) \otimes_w R_{0:w}(r_j)\}}{\vec{R} \vdash \langle H, R, \mathsf{op}^{\otimes}_w\ r_i, r_j\rangle \xrightarrow{\iota} \langle H, R'\rangle}$$

$$\text{ex-alloc}\ \frac{\begin{array}{c}\bot \notin R(r_j) \\ \{l, l +_4 1, \ldots, l +_4 R(r_j) -_4 1\} \cap dom(H) = \emptyset \\ H' = H \circ \{l \mapsto \{\bot\}^4, l +_4 1 \mapsto \{\bot\}^4, \ldots, l +_4 R(r_j) -_4 1 \mapsto \{\bot\}^4\} \\ R' = R \circ \{r_i \mapsto l\}\end{array}}{\vec{R} \vdash \langle H, R, \mathsf{alloc}\ r_i, r_j\rangle \xrightarrow{\iota} \langle H', R'\rangle} \qquad \text{ex-alloc-bot}\ \frac{\begin{array}{c}\bot \in R(r_j) \\ R' = R \circ \{r_i \mapsto \{\bot\}^4\}\end{array}}{\vec{R} \vdash \langle H, R, \mathsf{alloc}\ r_i, r_j\rangle \xrightarrow{\iota} \langle H, R'\rangle}$$

$$\text{ex-alloc-*}\ \frac{\begin{array}{c}\bot \notin R(r_j) \\ \{l, l +_4 1, \ldots, l +_4 R(r_j) *_4 -_4 1\} \cap dom(H) = \emptyset \\ H' = H \circ \{l \mapsto \{\bot\}^4, l +_4 1 \mapsto \{\bot\}^4, \ldots, l +_4 R(r_j) *_4 -_4 1 \mapsto \{\bot\}^4\} \\ R' = R \circ \{r_i \mapsto l\}\end{array}}{\vec{R} \vdash \langle H, R, \mathsf{alloc}\ r_i, r_j * c\rangle \xrightarrow{\iota} \langle H', R'\rangle} \qquad \text{ex-alloc-*-bot}\ \frac{\begin{array}{c}\bot \in R(r_j) \\ R' = R \circ \{r_i \mapsto \{\bot\}^4\}\end{array}}{\vec{R} \vdash \langle H, R, \mathsf{alloc}\ r_i, r_j * c\rangle \xrightarrow{\iota} \langle H, R'\rangle}$$

$$\text{ex-call}\ \frac{\begin{array}{c}\phi_x(a) = \langle \overrightarrow{r_{arg}}, r_{ret}, \overrightarrow{r_{loc}}, \lambda'_x, a_0\rangle \\ R' = \{\overrightarrow{r_{arg}} \mapsto \overrightarrow{r_j}, r_{ret} \mapsto \{\bot\}^4, \overrightarrow{r_{loc}} \mapsto \{\bot\}^4\} \\ \lambda'_x; \vec{R}, R \vdash \langle H, R', \lambda'_x(a_0)\rangle \xrightarrow{b}{}^{*} \langle H', R'', \mathsf{ret}\rangle\end{array}}{\vec{R} \vdash \langle H, R, \mathsf{call}\ r_i, a, \overrightarrow{r_j}\rangle \xrightarrow{\iota} \langle H', R \circ \{r_i \mapsto R''(r_{ret})\}\rangle}$$

$$\boxed{\lambda_x; \vec{R} \vdash \langle H, R, b\rangle \xrightarrow{b} \langle H', R', b'\rangle}$$

$$\text{ex-seq}\ \frac{\vec{R} \vdash \langle H, R, \iota\rangle \xrightarrow{\iota} \langle H', R'\rangle}{\lambda_x; \vec{R} \vdash \langle H, R, \iota; b\rangle \xrightarrow{b} \langle H', R', b\rangle} \qquad \text{ex-goto}\ \frac{}{\lambda_x; \vec{R} \vdash \langle H, R, \mathsf{goto}\ a\rangle \xrightarrow{b} \langle H, R, \lambda_x(a)\rangle}$$

$$\text{ex-if-true}\ \frac{\bot \notin R_{0:w}(r_i) \quad R_{0:w}(r_i) \neq 0}{\lambda_x; \vec{R} \vdash \langle H, R, (\mathsf{if}_w\ r_i\ \mathsf{goto}\ a); b\rangle \xrightarrow{b} \langle H, R, \lambda_x(a)\rangle} \qquad \text{ex-if-false}\ \frac{\bot \notin R_{0:w}(r_i) \quad R_{0:w}(r_i) = 0}{\lambda_x; \vec{R} \vdash \langle H, R, (\mathsf{if}_w\ r_i\ \mathsf{goto}\ a); b\rangle \xrightarrow{b} \langle H, R, b\rangle}$$

Figure 2: Structured Operational Semantics of MINX programs

Observe that blocks are terminated by control instructions, but conditional jumps only arise within a block. The fourth category, d, defines how functions are declared:

$$d_x ::= \langle \overrightarrow{r_{arg}}, r_{ret}, \overrightarrow{r_{loc}}, \lambda'_x, a_0\rangle$$

where $\lambda'_x : \mathit{Word} \rightharpoonup b$ is a partial mapping from addresses to blocks. Moreover, if

$$\mathsf{labels}_x(b) = \begin{cases} \{a\} & \text{if } s = \mathsf{labels}_x(\mathsf{goto}\ a) \\ \{a\} \cup \mathsf{labels}_x(b') & \text{else if } b = (\mathsf{if}_w\ r_i\ \mathsf{goto}\ a); b' \\ \mathsf{labels}_x(b') & \text{else if } b = \iota; b' \\ \emptyset & \text{otherwise} \end{cases}$$

we require $\bigcup\{\mathsf{labels}_x(b) \mid a \mapsto b \in \lambda'_x\} \subseteq dom(\lambda'_x)$ so that jump targets are contained within λ'_x. Finally, $\overrightarrow{r_{arg}}$ and $\overrightarrow{r_{loc}}$ denote vectors of (distinct) registers and $a_0 \in dom(\lambda'_x)$.

$$\boxed{\Sigma \vdash \theta_1 <: \theta_2} \quad \text{sub-refl}\frac{}{\Sigma \vdash \theta <: \theta} \quad \text{sub-trans}\frac{\Sigma \vdash \theta_1 <: \theta_2 \quad \Sigma \vdash \theta_2 <: \theta_3}{\Sigma \vdash \theta_1 <: \theta_3} \quad \text{sub-ptr}\frac{\Sigma \vdash \theta_1 <: \theta_2}{\Sigma \vdash \theta_1* <: \theta_2*}$$

$$\text{sub-arr}\frac{\Sigma \vdash \theta_1 <: \theta_2}{\Sigma \vdash \theta_1[]* <: \theta_2[]*} \quad \text{sub-elm}\frac{}{\Sigma \vdash \theta[]* <: \theta*} \quad \text{sub-fld}\frac{\Sigma(N) = \langle \theta_0, \ldots, \theta_{n-1} \rangle}{\Sigma \vdash N* <: \theta_0*}$$

Figure 3: Subtyping relations of MINC programs

3.3 Structured Operational Semantics

Figure 2 presents our (mostly small-step) SOS for MINX as two judgements: $\vec{R} \vdash \langle H, R, \iota \rangle \xrightarrow{\iota} \langle H', R' \rangle$ and $\lambda_x; \vec{R} \vdash \langle H, R, b \rangle \xrightarrow{b} \langle H', R', b' \rangle$. The former details the behaviour of single instructions and the latter of blocks. Both are parameterised by a vector $\vec{R}$ of register assignments (needed solely in the proofs) to state that data accessible from these (shadowed) registers is not mutated by a call. For reasons of space, we comment only on noteworthy details.

For brevity, in ex-mov-rc, we write c for a vector of w bytes that represents it. Notice in ex-mov-rr how the least significant w bytes of the register r_i are mutated while the remaining $4 - w$ bytes are left intact. In the ex-mov-ri rule, the r_i register is set to 4 uninitialised values if r_j contains one uninitialised value. The rule ex-mov-ir requires the 4 bytes of r_i to be initialised (otherwise computation gets stuck). Note how the w low bytes of register r_j are copied to w consecutive bytes of the heap. Rule ex-$\oplus$-r* is included to support pointer arithmetic, where the data objects are c bytes in size. Rules ex-$\otimes$-rc and ex-$\otimes$-rr get stuck on, for instance, division by zero.

Observe how ex-alloc initialises allocated memory (to indicate garbage). If the register r_j itself contains an uninitialised byte, then rule ex-alloc-bot sets r_i to be uninitialised. The rule ex-alloc-* is employed to allocate an array of objects of size c.

The ex-call is worthy of note, as it is particularly unusual. Four bytes of each of the registers $\overrightarrow{r_j}$ are copied to $\overrightarrow{r_{arg}}$, which is abbreviated to $\overrightarrow{r_{arg}} \mapsto \overrightarrow{r_j}$ in the judgement. Local registers $\overrightarrow{r_{loc}}$ and the return register, r_{ret}, are set to uninitialised values. The entry block $\lambda'_x(a_0)$ is executed, and any subsequent block that leads on from it, until `ret` is encountered, whereupon the register values are restored and register r_j updated with the return value.

Observe how ex-if-true and ex-if-false get stuck if the decision register contains an uninitialised byte.

4. The MINC Language

MINC (Minimal C) is the language of our witness programs and the target of our decompilation. We do not use C directly because, even though C is expressive enough to capture the semantics of machine instructions, it lacks one crucial ingredient: type safety. In contrast, we designed MINC to be type-safe, and still be close enough to C to recover MINX programs.

4.1 Syntax

MINC features three different kinds of types:

$$t ::= \mathsf{short} | \mathsf{long} \qquad \theta ::= t | \tau* \qquad \tau ::= \theta | \theta[] | N$$

A *primitive type* t is either a short or a long integer. A *compact type* θ is a type that can be assigned to a program variable, it is either a primitive type t or a pointer type $\tau*$. Finally, a *general type* τ is either a compact type θ, an array type $\theta[]$ or a struct type identified by its name N.

MINC programs themselves are defined in terms of the categories of statements s, expressions e and lvalues ℓ as follows:

$$\begin{array}{lll} s ::= (\ell := e); s & \ell ::= x & e ::= \ell \mid c \mid \&x \mid f(\vec{e}) \\ \quad \mid (\mathsf{if}\ e\ \mathsf{goto}\ l); s & \quad \mid *x & \quad \mid \mathsf{new}\ \theta \mid \mathsf{new}\ \theta[e] \\ \quad \mid \mathsf{goto}\ l & \quad \mid x \to c & \quad \mid \mathsf{new\ struct}\ N \\ \quad \mid \mathsf{return} & \quad \mid x[e] & \quad \mid (e_1 \oplus e_2) \mid (e_1 \otimes e_2) \end{array}$$

where $\oplus$ and $\otimes$ are defined as in MINX. Function declarations,

$$d_c ::= f(\overrightarrow{x : \theta}) \langle \overrightarrow{y : \theta'}, l, \lambda_c, j \rangle$$

include a vector of arguments and their types $\overrightarrow{x : \theta}$, a vector of locals and their types $\overrightarrow{y : \theta'}$, an entry label $l \in \mathit{Label}$, a partial mapping of labels to statements, $\lambda_c : \mathit{Label} \rightharpoonup b$, and an index j into y indicating which local variable holds the return value. Labels must be contained within $dom(\lambda_c)$, analogous to what was previously defined.

4.2 Type System

Figure 4 defines the MINC type system as well-typing judgements $\Sigma \vdash d, \Gamma_c; \Sigma \vdash s, \Gamma_c; \Sigma \vdash \ell : \theta$ and $\Gamma_c; \Sigma \vdash e : \theta$ for the four syntactic sorts. As well as the variable type environment Γ_c, the judgements make use of Σ, which maps struct names N to vectors $\vec{\theta}$ of their field types. We also assume a global environment ϕ_c that contains all function definitions.

Because it admits type safety, the type system is stricter than that of C. For example, C allows any integer to be added to the address of a primitive type, which is not permitted in MINC. Unlike C, arrays are first-class types, which means that it is possible to pass them as arguments to functions and return them, without demoting them to simple pointers. The upshot of this is that in MINC it is possible to distinguish between $\theta*$ and $\theta[]*$.

To regain some of C's flexibility, without compromising on type safety, we have equipped MINC's type system with subtyping (see Figure 3). For instance, an array $\theta[]*$ is a subtype of $\theta*$, which allows the assignment of an array to a compatible pointer.

As evident in the rules t-s and t-l, MINC literals are tagged: short values are tagged c_s and long values c_l. Although in general pointer literals do not exist in MINC, an exception is the special value 0_*, equivalent to `NULL` in C.

4.3 Semantics

Figure 5 presents the semantics of MINC, which is defined in terms of auxiliary functions: $\mathit{sizeof}(\theta)$ gives the size of an object of type θ in bytes:

$$\mathit{sizeof}(\mathsf{short}) = 2 \quad \mathit{sizeof}(\mathsf{long}) = 4 \quad \mathit{sizeof}(\tau*) = 4$$

The functions $\mathsf{untag}_t(n_t) = n$ and $\mathsf{tag}_t(n) = n_t$ remove and add a tag t. Furthermore, given a set of non-empty ranges $\pi \subseteq \{[l, u] \mid 0 \le l \le u \le 2^{32} - 1\}$ that represents a set of (disjoint) memory regions, addition of a short and an address is defined as:

$$u +_\pi v = \begin{cases} \bot & \text{if } u = \bot \vee v = \bot \\ \mathsf{tag}_*(s) & \text{else if } \exists r \in \pi. \{m, s\} \subseteq r \\ \mathsf{err} & \text{otherwise} \end{cases}$$

where $s = \mathsf{untag}_s(u) +_4 m$ and $m = \mathsf{untag}_*(v)$. Addition of address/short, long/address, and address/long are analogously

$$\boxed{\Sigma \vdash d} \quad \text{t-def}\frac{\Gamma_c = \{\overrightarrow{x : \theta}, \overrightarrow{y : \theta'}\} \quad l \in dom(\lambda_c) \quad y_j \in \vec{y} \quad \forall l' \in dom(\lambda_c) : \ \Gamma_c; \Sigma \vdash \lambda_c(l')}{\Sigma \vdash f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta'}, l, \lambda_c, j\rangle}$$

$$\boxed{\Gamma_c; \Sigma \vdash s} \quad \text{t-assn}\frac{\Gamma_c; \Sigma \vdash \ell : \theta_1 \quad \Gamma_c; \Sigma \vdash e : \theta_2 \quad \Sigma \vdash \theta_2 <: \theta_1 \quad \Gamma_c; \Sigma \vdash s}{\Gamma_c; \Sigma \vdash (\ell := e); s} \quad \text{t-if}\frac{\Gamma_c; \Sigma \vdash e : \theta \quad \Gamma_c; \Sigma \vdash s}{\Gamma_c; \Sigma \vdash (\mathsf{if}\ e\ \mathsf{goto}\ l); s} \quad \text{t-goto}\frac{}{\Gamma_c; \Sigma \vdash \mathsf{goto}\ l} \quad \text{t-ret}\frac{}{\Gamma_c; \Sigma \vdash \mathsf{return}}$$

$$\boxed{\Gamma_c; \Sigma \vdash \ell : \theta} \quad \text{t-var}\frac{x : \theta \in \Gamma_c}{\Gamma_c; \Sigma \vdash x : \theta} \quad \text{t-ptr}\frac{\Gamma_c; \Sigma \vdash x : \tau*}{\Gamma_c; \Sigma \vdash *x : \tau} \quad \text{t-fld}\frac{\Gamma_c; \Sigma \vdash x : N* \quad \Sigma(N) = \langle \theta_0, \ldots, \theta_{n-1}\rangle}{\Gamma_c; \Sigma \vdash x \to i : \theta_i} \quad \text{t-ar}\frac{\Gamma_c; \Sigma \vdash x : \theta[]* \quad \Gamma_c; \Sigma \vdash e : t}{\Gamma_c; \Sigma \vdash x[e] : \theta}$$

$$\boxed{\Gamma_c; \Sigma \vdash e : \theta} \quad \text{t-amp}\frac{\Gamma_c; \Sigma \vdash y : \tau}{\Gamma_c; \Sigma \vdash \&y : \tau*} \quad \text{t-l}\frac{}{\Gamma_c; \Sigma \vdash c_l : \mathsf{long}} \quad \text{t-s}\frac{}{\Gamma_c; \Sigma \vdash c_s : \mathsf{short}} \quad \text{t-null}\frac{}{\Gamma_c; \Sigma \vdash 0_* : \tau*}$$

$$\text{t-new}\frac{}{\Gamma_c; \Sigma \vdash \mathsf{new}\ \theta : \theta*} \quad \text{t-new-str}\frac{}{\Gamma_c; \Sigma \vdash \mathsf{new\ struct}\ N : N*} \quad \text{t-new-ar}\frac{\Gamma_c; \Sigma \vdash e : t}{\Gamma_c; \Sigma \vdash \mathsf{new}\ \theta[e] : \theta[]*}$$

$$\text{t-}\otimes\frac{\Gamma_c; \Sigma \vdash e_1 : t \quad \Gamma_c; \Sigma \vdash e_2 : t}{\Gamma_c; \Sigma \vdash (e_1 \otimes e_2) : t} \quad \text{t-ptr-}\oplus\frac{\Gamma_c; \Sigma \vdash e_1 : \theta[]* \quad \Gamma_c; \Sigma \vdash e_2 : t}{\Gamma_c; \Sigma \vdash (e_1 \oplus e_2) : \theta[]*} \quad \text{t-}\oplus\text{-ptr}\frac{\Gamma_c; \Sigma \vdash e_1 : t \quad \Gamma_c; \Sigma \vdash e_2 : \theta[]*}{\Gamma_c; \Sigma \vdash (e_1 \oplus e_2) : \theta[]*}$$

$$\text{t-call}\frac{\phi_c(f) = f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta''}, l, \lambda_c, j\rangle, \quad \forall e_i \in \vec{e}, \theta'_i \in \vec{\theta'} : \ \Gamma_c; \Sigma \vdash e_i : \theta'_i \quad \Sigma \vdash \vec{\theta'} <: \vec{\theta} \quad \Sigma \vdash \theta''_j <: \theta_j}{\Gamma_c; \Sigma \vdash f(\vec{e}) : \theta_j}$$

Figure 4: Type-correct MINC programs

defined, so that the sum of an integer and an address n_* must fall within the same range of π as n_*. The sum of two longs is simply defined:

$$u +_\pi v = \begin{cases} \bot & \text{if } u = \bot \vee v = \bot \\ \mathsf{tag}_l(\mathsf{untag}_l(u) +_4 \mathsf{untag}_l(v)) & \text{otherwise} \end{cases}$$

The sum of two shorts is defined likewise, to cumulatively give the partial map $+_\pi : Val^2 \rightharpoonup Val$, where Val is the set of all tagged values. Subtraction $-_\pi : Val^2 \rightharpoonup Val$ is defined likewise in a piecewise manner.

The environment and store maps have signatures $\rho : Var \mapsto \mathbb{N}$ and $\sigma : \mathbb{N}_\perp \rightharpoonup Val_\perp$ where Var is a set of program variables and $\mathbb{N}$ is assumed to exclude 0. We also assume $\sigma(\perp) = \perp$. The SOS is structured according to the three categories ℓ, e and s. For brevity, we only provide commentary on noteworthy rules.

The rules l-ptr, l-fld and l-ar check $\rho(x) \neq 0$ since location 0 is reserved as a sentinel. If these checks fail then auxiliary rules, l-ptr-err, l-fld-err and l-ar-err, trigger an err state. For instance,

$$\text{l-ptr-err}\frac{\rho(x) = 0}{\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, *x\rangle \xrightarrow{\ell} \mathsf{err}}$$

Rules l-fld-err and l-ar-err are defined analogously. Moreover, the rules l-fld and l-ar check $a \in \cup\pi'$ to ensure the computed address stays a stays within allocated memory. Further complementary rules trigger err if this does not hold, indeed many rules have many error modes. The e-call rule represents an extreme case: it has one error mode for the evaluation of each e_i and execution of the block labelled $\lambda_c(l)$ can itself trigger an error.

Of particular note is e-op: if v_1 is tagged as a pointer and v_2 as an integer, the binary operation $v_1 \oplus_{\pi''} v_2$ itself will error if the resulting pointer falls outside the region enclosing v_1. This is trapped by a rule that complements e-op so as to propagate err in the expected way.

Finally note how freshly allocate memory is marked as uninitialised in rules e-new, e-ar and e-str.

4.4 Type Safety

The MINC language is a type-safe variant of C. This means that well-typed MINC programs do not get stuck. We can be formally precise about this property in the usual way, stating preservation and progress properties for the syntactic sorts of MINC.

Type safety depends on the notion of a store typing Ψ that associated a type θ with every address a in the store σ. A store σ is well-typed, denoted $\Sigma; \Psi \vdash \sigma; \pi$, iff

$$\forall (a : \theta) \in \Psi \ . \ \Sigma; \Psi; \sigma; \pi \vdash a : \theta$$

Figure 6 defines the auxiliary judgements for well-typed addresses and values.

Similar to a store, an environment ρ is well-typed, denoted $\Gamma_c; \Sigma; \Psi \vdash \rho$, iff

$$\forall (x : \theta) \in \Gamma_c \ . \ \Sigma; \Psi \vdash \rho(x) : \theta * \wedge \rho(x) \neq 0$$

For the sake of brevity, we state the two properties here only for expressions. The other propositions and all proofs can be found in appendix.

Proposition 1 (Preservation of MINC Expressions)**.** If $\Gamma_c; \Sigma; \Psi \vdash \rho, \Sigma; \Psi \vdash \sigma; \pi, \Gamma_c; \Sigma \vdash e : \theta$ and $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e\rangle \xrightarrow{e} \langle \sigma', \pi', v\rangle$ then for some $\Psi' \supseteq \Psi$:

$$\Gamma_c; \Sigma; \Psi' \vdash \rho \quad \wedge \quad \Sigma; \Psi' \vdash \sigma'; \pi' \quad \wedge \quad \Sigma; \Psi' \vdash v : \theta$$

Proposition 2 (Progress of MINC Expressions)**.** If $\Gamma_c; \Sigma; \Psi \vdash \rho, \Sigma; \Psi \vdash \sigma; \pi$ and $\Gamma_c; \Sigma \vdash e : \theta$ then: $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e\rangle \xrightarrow{e} \langle \sigma', \pi', v\rangle$ or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e\rangle \xrightarrow{e} \mathsf{err}$.

$$\boxed{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,\ell\rangle \xrightarrow{\ell} \langle \sigma',\pi',a\rangle + \mathsf{err}}$$

$$\text{l-var}\frac{}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,x\rangle \xrightarrow{\ell} \langle \sigma,\pi,\rho(x)\rangle} \qquad \text{l-ptr}\frac{\rho(x) \neq 0}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,*x\rangle \xrightarrow{\ell} \langle \sigma,\pi,\sigma(\rho(x))\rangle}$$

$$\text{l-fld}\frac{\rho(x) \neq 0 \quad a = \sigma(\rho(x)) +_\perp c \quad a \in \cup\pi}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,x \to c\rangle \xrightarrow{\ell} \langle \sigma,\pi,a\rangle} \qquad \text{l-ar}\frac{\begin{array}{c}\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,e\rangle \xrightarrow{e} \langle \sigma',\pi',v\rangle \\ \rho(x) \neq 0 \quad a = \sigma(\rho(x)) +_\perp v \quad a \in \cup\pi'\end{array}}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,x[e]\rangle \xrightarrow{\ell} \langle \sigma',\pi',a\rangle}$$

$$\boxed{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,e\rangle \xrightarrow{e} \langle \sigma',\pi',v\rangle + \mathsf{err}} \qquad \text{e-const}\frac{}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,c\rangle \xrightarrow{e} \langle \sigma,\pi,c\rangle}$$

$$\text{e-lval}\frac{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,\ell\rangle \xrightarrow{\ell} \langle \sigma',\pi',a\rangle}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,\ell\rangle \xrightarrow{e} \langle \sigma',\pi',\sigma'(a)\rangle} \qquad \text{e-amp}\frac{}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,\&x\rangle \xrightarrow{e} \langle \sigma,\pi,\rho(x)\rangle}$$

$$\text{e-new}\frac{\begin{array}{c}a \notin dom(\sigma) \cup \{0\} \\ \sigma' = \sigma \circ \{a \mapsto \perp\}\end{array}}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,\mathsf{new}\ \theta\rangle \xrightarrow{e} \langle \sigma',\pi,a\rangle} \qquad \text{e-ar}\frac{\begin{array}{c}\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,e\rangle \xrightarrow{e} \langle \sigma',\pi',v\rangle \\ \{a,\ldots,a+v-1\} \cap (dom(\sigma') \cup \{0\}) = \emptyset \\ \sigma'' = \sigma' \circ \{a \mapsto \perp,\ldots,a+v-1 \mapsto \perp\} \\ \pi'' = \pi' \cup \{[a,a+v-1]\}\end{array}}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,\mathsf{new}\ \theta[e]\rangle \xrightarrow{e} \langle \sigma'',\pi'',a\rangle}$$

$$\text{e-str}\frac{\begin{array}{c}\{a,\ldots,a+|\Sigma(N)|-1\} \cap (dom(\sigma) \cup \{0\}) = \emptyset \\ \pi' = \pi \cup \{[a,a+|\Sigma(N)|-1]\} \\ \sigma' = \sigma \circ \{a \mapsto \perp,\ldots,a+|\Sigma(N)|-1 \mapsto \perp\}\end{array}}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,\mathsf{new\ struct}\ N\rangle \xrightarrow{e} \langle \sigma',\pi`,a\rangle} \qquad \text{e-op}\frac{\begin{array}{c}\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,e_1\rangle \xrightarrow{e} \langle \sigma',\pi',v_1\rangle \\ \Sigma;\vec{\rho};\rho \vdash \langle \sigma',\pi',e_2\rangle \xrightarrow{e} \langle \sigma'',\pi'',v_2\rangle \\ v_1 \otimes_{\pi''} v_2 = v\end{array}}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,(e_1 \otimes e_2)\rangle \xrightarrow{e} \langle \sigma'',\pi'',v\rangle}$$

$$\text{e-call}\frac{\begin{array}{c}\phi_c(f) = f(\overrightarrow{x:\theta})\langle \overrightarrow{y:\theta'}, l, \lambda_c, j\rangle \qquad \forall e_i \in \vec{e}:\ \Sigma;\vec{\rho};\rho \vdash \langle \sigma_{i-1},\pi_{i-1},e_i\rangle \xrightarrow{e} \langle \sigma_i,\pi_i,v_i\rangle \\ |\ \vec{x}\ | = n \quad \{\vec{a},\vec{a'}\} \cap (dom(\sigma_n) \cup \{0\}) = \emptyset \quad \rho' = \{\overrightarrow{x \mapsto a}, \overrightarrow{y \mapsto a'}\} \quad \sigma' = \sigma_n \circ \{\overrightarrow{a \mapsto v}, \overrightarrow{a' \mapsto \perp}\} \\ \Sigma;\lambda_c;\vec{\rho},\rho;\rho' \vdash \langle \sigma',\pi_n,\lambda_c(l)\rangle \xrightarrow{s}{}^{*} \langle \sigma'',\pi',\mathsf{return}\rangle\end{array}}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma_0,\pi_0,f(\vec{e})\rangle \xrightarrow{e} \langle \sigma'',\pi',\sigma''(\rho'(y_j))\rangle}$$

$$\boxed{\Sigma;\lambda_c;\vec{\rho};\rho \vdash \langle \sigma,\pi,s\rangle \xrightarrow{s} \langle \sigma',\pi',s'\rangle + \mathsf{err}} \qquad \text{s-assn}\frac{\begin{array}{c}\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,\ell\rangle \xrightarrow{\ell} \langle \sigma',\pi',a\rangle \\ \Sigma;\vec{\rho};\rho \vdash \langle \sigma',\pi',e\rangle \xrightarrow{e} \langle \sigma'',\pi'',v\rangle \\ \sigma''' = \sigma'' \circ \{a \mapsto v\}\end{array}}{\Sigma;\lambda_c;\vec{\rho};\rho \vdash \langle \sigma,\pi,(\ell := e);s\rangle \xrightarrow{s} \langle \sigma''',\pi'',s\rangle}$$

$$\text{s-goto}\frac{l \in dom(\lambda_c)}{\Sigma;\lambda_c;\vec{\rho};\rho \vdash \langle \sigma,\pi,\mathsf{goto}\ l\rangle \xrightarrow{s} \langle \sigma,\pi,\lambda_c(l)\rangle}$$

$$\text{s-if-true}\frac{\begin{array}{c}l \in dom(\lambda_c) \quad v \neq \perp \quad v \neq 0 \\ \Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,e\rangle \xrightarrow{e} \langle \sigma',\pi',v\rangle\end{array}}{\Sigma;\lambda_c;\vec{\rho};\rho \vdash \langle \sigma,\pi,(\mathsf{if}\ e\ \mathsf{goto}\ l);s\rangle \xrightarrow{s} \langle \sigma',\pi',\lambda_c(l)\rangle} \qquad \text{s-if-false}\frac{\begin{array}{c}l \in dom(\lambda_c) \quad v \neq \top \quad v = 0 \\ \Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,e\rangle \xrightarrow{e} \langle \sigma',\pi',v\rangle\end{array}}{\Sigma;\lambda_c;\vec{\rho};\rho \vdash \langle \sigma,\pi,(\mathsf{if}\ e\ \mathsf{goto}\ l);s\rangle \xrightarrow{s} \langle \sigma',\pi',s\rangle}$$

Figure 5: Structured Operational Semantics of MINC programs

5. The Decompilation Relation

This section relates MINX and MINC programs through a decompilation relation. The relation is spread out over three judgements, one for each of MINX' three main syntactic categories.

Instruction Decompilation Figure 7 defines the decompilation judgement $\mu_\Gamma; \Gamma_c; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e$ that explains how to decompile a MINX instruction ι into a MINC assignment $\ell := e$. Additional parameters to the judgement are a variable mapping μ_Γ that relates MINX registers to MINC local variables, a MINC typing environment Γ_c for those local variables and a set of MINC struct definitions Σ.

While we do not have the space to explain every rule in detail, we do wish to highlight a number of important aspects:

The judgement is defined by syntax-directed rules, as is also usually the case for compilation and elaboration relations. The big difference is that, as the latter usually define (partial) functions, they are deterministic with typically one rule per syntactic construct. Our decompilation judgement is non-deterministic and features multiple rules per syntactic construct, one for each distinct typing that can be assigned to the instruction. For instance, the three rules tr-mov-ri$_1$, tr-mov-ri$_2$ and tr-mov-ri$_3$ compile instruction $\mathsf{mov}_w\ r_i, [r_j]$ into either $x := *y$, $x := y[0]$ or $x := y \to 0$ depending on whether y has type $\theta*$, $\theta[]*$ or $N*$.

The instruction widths w play an important role in restricting the possibilities for the recovered MINC types. For instance, rule tr-$\oplus$-rc makes sure that the width w of the arithmetic operation $\mathsf{op}^{\otimes}_w$ is identical to the size of the recovered primitive type t. Hence, a width of 4 gives rise to long and a width of 2 to short.

At several occasions the rules have to make up for the difference in memory granularity between MINX and MINC. In particular, in MINC the stride between two array elements is always 1. However, when seen in MINX, the same stride depends on the size of the elements. Hence, rules like tr-$\oplus$-rc for array pointer arithmetic have to convert between a MINX stride of $c = m * sizeof(\theta)$ and a corresponding MINC stride of $m = 1 * m$.

When allocating a staticlly known amount of memory the rules tr-alloc-rc$_1$, tr-alloc-rc$_2$ and tr-alloc-rc$_3$ also exploit the sizes of types to determine whether a primitive type, a particular struct or an array is allocated. We only support the decompilation of dynamic memory allocation for instruction $\mathsf{alloc}\ r_i, r_j * c$ where we can statically verify that the amount of allocated memory is a multiple of the memory size. This, among others, makes our decompilation relation conservative and incomplete. It is a price we gladly pay in order to provide strong guarantees about the validity of the recovered types.

Basic Block Decompilation The top half of Figure 8 defines the judgement $\mu_\lambda; \mu_\Gamma; \Gamma_c; \Sigma \vdash b \overset{b}{\rightsquigarrow} s$ for decompiling MINX basic blocks b into MINC statements s. This judgement has one additional parameter compared to the judgement for instructions: the label map μ_λ relates MINX labels to MINC ones. This map is used for decompiling goto and if. As the rules preserve the basic control flow from MINX to MINC and do not affect the types directly, they are deterministic syntax-directed ones.

Function Definition Decompilation The bottom half of Figure 8 defines the judgement $\Sigma \vdash d_x \rightsquigarrow d_c$ that decompiles a MINX function definition d_x into a MINC definition d_c. The single rule of this judgement sets up the variable and label maps, and decompiles the basic blocks with respect to an appropriate typing environment.

5.1 Meta-Theoretical Properties

Our decompilation relation satisfies two strong properties that justify its relevance: 1) the produced MINC witness program is well-typed, and 2) it has the same operational semantics as the original MINX program. Taken together these two properties give meaning to the statement that *the original* MINC *program inhabits the recovered types*.

$$\frac{\boxed{\mu_a \vdash \vec{b} \leftrightsquigarrow v}}{\mu_a \vdash \bot^4 \leftrightsquigarrow \bot_a} \qquad \frac{}{\mu_a \vdash 0^4 \leftrightsquigarrow 0_*} \qquad \frac{(a : n_*) \in \mu_a}{\mu_a \vdash a \leftrightsquigarrow n_*}$$

$$\frac{\mu_a \vdash n^4 \leftrightsquigarrow n_l}{\mu_a \vdash n^2 \leftrightsquigarrow n_s} \qquad \frac{\mu_a \vdash \bot^4 \leftrightsquigarrow \bot_l}{\mu_a \vdash \bot^2 \leftrightsquigarrow \bot_s}$$

Figure 9: Value correspondence

5.1.1 Well-Typing

The first claim states that the recovered witness program is well-typed. This falls apart into three propositions, one for each of the judgements.

Proposition 3 (Well-Typed Instruction Decompilation)**.** If $\mu_\Gamma; \Gamma_c; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e$, then for some θ_1 and θ_2:

$$\Gamma_c; \Sigma \vdash \ell : \theta_1 \quad \wedge \quad \Gamma_c; \Sigma \vdash e : \theta_2 \quad \wedge \quad \Sigma \vdash \theta_2 <: \theta_1$$

Proposition 4 (Well-Typed Block Decompilation)**.** If $\mu_\lambda; \mu_\Gamma; \Gamma_c; \Sigma \vdash b \overset{b}{\rightsquigarrow} s$ then $\Gamma_c; \Sigma \vdash s$.

Proposition 5 (Well-Typed Definition Decompilation)**.** If $\Sigma \vdash d_x \rightsquigarrow d_c$ then $\Sigma \vdash d_c$.

For the proofs of these propositions, we refer to the appendix.

Due to the type safety of MINC, this means that the witness program is operationally well-behaved.

5.1.2 Memory Correspondence

The main semantic effect of both MINX and MINC programs is a transformation of program memory. However, because MINX and MINC programs act on very different memory structures, the semantic correspondence is not readily expressed. We first need to define what low-level and high-level memory structures correspond. Then we can express semantics preservation as the preservation of this correspondence.

There are three types of memory correspondence to consider: MINX versus MINC values, MINX heaps versus MINC stores, and MINX registers versus MINC local variables.

Value Correspondence Figure 9 defines the judgement $\mu_a \vdash \vec{b} \leftrightsquigarrow v$ that states the basic correspondence between a MINX byte sequence $\vec{b}$ and a MINC value v. This judgement is parameterized by an address map μ_a that relates MINX and MINC addresses. The rules are obvious, relating 0 pointers, addresses, bottoms, and numeric values of the appropriate byte sizes.

Registers versus Local Variables We denote that MINX register banks $\vec{R}$ and MINC local variables $\vec{\rho}$ are pair-wise related with:

$$\mu_a; \vec{\mu_\Gamma}; \sigma \vdash \vec{R} \leftrightsquigarrow \vec{\rho}$$

The relation is parameterized by an address map μ_a, register-variable maps $\vec{gamma}$ and a store σ.

The relation stands for:

$$\forall (r : x)_w \in \mu_{\Gamma,i} : \exists n_* : (x : n_*) \in \rho_i : \\ \exists v : (n_* : v) \in \sigma \wedge \exists \vec{b} : \vec{b} = R_{i,0:w}(r) \wedge \mu_a \vdash \vec{b} \leftrightsquigarrow v$$

The relation expresses that any related register r and local variable x have associated values $\vec{b}$ and v that are related. The main complication is that the local variables are store-mapped whereas the registers are not heap-mapped.

$$\boxed{\Sigma;\Psi;\sigma;\pi \vdash a:\tau} \quad \text{st-comp}\frac{\Sigma;\Psi \vdash \sigma(a):\theta}{\Sigma;\Psi;\sigma;\pi \vdash a:\theta} \quad \text{st-fld}\frac{\Sigma(N)=\langle\theta_0,\ldots,\theta_{n-1}\rangle \quad [a,a+n-1]\in\pi \quad \forall i\in[0,n-1].\Sigma;\Psi\vdash\sigma(a+i):\theta_i}{\Sigma;\Psi;\sigma;\pi\vdash a:N} \quad \text{st-ar}\frac{[a-n,a+m]\in\pi \quad \forall i\in[-n,m].\Sigma;\Psi\vdash\sigma(a+i):\theta}{\Sigma;\Psi;\sigma;\pi\vdash a:\theta[]}$$

$$\boxed{\Sigma;\Psi\vdash v:\theta} \quad \text{vt-bot}\frac{}{\Sigma;\Psi\vdash\bot:\theta} \quad \text{vt-s}\frac{}{\Sigma;\Psi\vdash c_s:\mathsf{short}} \quad \text{vt-l}\frac{}{\Sigma;\Psi\vdash c_l:\mathsf{long}}$$

$$\text{vt-null}\frac{}{\Sigma;\Psi\vdash 0_l:\tau*} \quad \text{vt-addr}\frac{(a:\tau)\in\Psi}{\Sigma;\Psi\vdash a:\tau*} \quad \text{vt-subt}\frac{\Sigma;\Psi\vdash v:\theta_1 \quad \Sigma\vdash\theta_1<:\theta_2}{\Sigma;\Psi\vdash v:\theta_2}$$

Figure 6: Well-typed addresses and values

Heaps versus Stores We denote that MINX heap H and MINC store σ are related with:

$$\mu_a;\nu_a;\pi;\vec{\rho}\vdash H \leftrightsquigarrow \sigma$$

This relation summarizes 6 different properties addressing 4 concerns. Firstly, as in the previous cases this relation parameterized by an address map μ_a that relates addresses in H with addresses in σ. Obviously, these related addresses point to related values.

$$\forall(a,n_*)_w\in\mu_a:\exists\vec{b},v:\\ \vec{b}=H_{0:w}(a)\wedge v=\sigma(a)\wedge\mu_a\vdash\vec{b}\leftrightsquigarrow v$$

Secondly, we have to contend with the difference in granularity between MINX and MINC: While we can only address values as a whole in MINC, MINXaddresses individual bytes and can point into the middle of a value. To overcome this gap, the address map μ_a only covers the addresse in H that point at the first byte of a value. The complementary *header* map ν_a relates each address in H (especially those pointing into the middle of a value) to the address of the value's first byte the value's width.

$$\forall a\in dom(H):\exists a',w:\nu_a(a)=\langle a',w\rangle\wedge(a'+w)\geq a$$

Obviously, these header addresses are fixpoints of ν_a:

$$\forall\langle a,w\rangle\in range(\nu_a):\nu_a(a)=\langle a,w\rangle$$

Moreover, they are covered by μ_a:

$$\forall\langle a,w\rangle\in range(\nu_a):\exists n_*:(a:n_*)_w\in\mu_a$$

Thirdly, not all addresses in σ are related to an address in H. This is an outfall of the discrepancy between registers and local variables: the store-mapped local variables (i.e., those tracked in $\vec{\rho}$ or ρ) have no counterpart in H. Hence, they need not have a counterpart in the relation.

$$\forall n_*\in(dom(\sigma)-range(\vec{\rho},\rho)):\exists a,w:(a:n_*)_w\in\mu_a$$

Finally, adjacent MINC addresses in a range tracked by π must be related to adjacent addresses in MINX(taking into account the width w of the value).

$$\forall[n_*,n_*+c]\in\pi:\forall i\in[0,c-1]:\exists a,a',w,w':\\ a+w=a'\wedge(a,n_*+i)_w\in\mu_a\wedge(a',n_*+i+1)_{w'}\in\mu_a$$

5.1.3 Semantics Preservation

With the memory relations in place we can state one important aspect of semantics preservation as: the original MINX program and the corresponding decompiled MINC program take related memories to related memories.

Proposition 6 (Preservation of Related Memory for Instructions)**.** If

- $\mu_\Gamma;\Gamma_c;\Sigma\vdash\iota\overset{\iota}{\rightsquigarrow}\ell:=e$
- $\Gamma_c;\Sigma;\Psi\vdash\rho$
- $\Sigma;\Psi\vdash\sigma;\pi$
- $\mu_a;\nu_a;\pi;\vec{\rho},\rho\vdash H\leftrightsquigarrow\sigma$
- $\mu_a;\vec{\mu_\Gamma},\mu_\Gamma;\sigma\vdash\vec{R},R\leftrightsquigarrow\vec{\rho},\rho$
- $\vec{R}\vdash\langle H,R,\iota\rangle\xrightarrow{\iota}\langle H',R'\rangle$,
- $\Sigma;\vec{\rho};\rho\vdash\langle\sigma,\pi,\ell\rangle\xrightarrow{\ell}\langle\sigma',\pi',a\rangle$, and
- $\Sigma;\vec{\rho};\rho\vdash\langle\sigma',\pi',e\rangle\xrightarrow{e}\langle\sigma'',\pi'',v\rangle$

then for some $\mu'_a\supseteq\mu_a$ and $\nu'_a\supseteq\nu_a$:

- $\mu'_a;\vec{\mu_\Gamma},\mu_\Gamma;\sigma'\circ\{a\mapsto v\}\vdash\vec{R},R\leftrightsquigarrow\vec{\rho},\rho$
- $\mu'_a;\nu'_a;\pi';\vec{\rho},\rho\vdash H'\leftrightsquigarrow\sigma'\circ\{a\mapsto v\}$

A second aspect of the semantics preservation is that, if the MINX program does not get stuck, the MINC program may get stuck only through a violation of memory that is guarded by π.

Proposition 7 (Preservation of Progress for Instructions)**.** If

- $\mu_\Gamma;\Gamma_c;\Sigma\vdash\iota\overset{\iota}{\rightsquigarrow}\ell:=e$
- $\Gamma_c;\Sigma;\Psi\vdash\rho$
- $\Sigma;\Psi\vdash\sigma;\pi$
- $\mu_a;\nu_a;\pi;\vec{\rho},\rho\vdash H\leftrightsquigarrow\sigma$
- $\mu_a;\vec{\mu_\Gamma},\mu_\Gamma;\sigma\vdash\vec{R},R\leftrightsquigarrow\vec{\rho},\rho$
- $\vec{R}\vdash\langle H,R,\iota\rangle\xrightarrow{\iota}\langle H',R'\rangle$

then

- $\Sigma;\vec{\rho};\rho\vdash\langle\sigma,\pi,\ell\rangle\xrightarrow{\ell}\mathsf{err}$ or
- $\Sigma;\vec{\rho};\rho\vdash\langle\sigma,\pi,\ell\rangle\xrightarrow{\ell}\langle\sigma',\pi',a\rangle$ and $\Sigma;\vec{\rho};\rho\vdash\langle\sigma',\pi',e\rangle\xrightarrow{e}\mathsf{err}$, or
- $\Sigma;\vec{\rho};\rho\vdash\langle\sigma,\pi,\ell\rangle\xrightarrow{\ell}\langle\sigma',\pi',a\rangle$ and $\Sigma;\vec{\rho};\rho\vdash\langle\sigma',\pi',e\rangle\xrightarrow{e}\langle\sigma'',\pi'',v\rangle$.

There are similar such propositions for basic blocks and for function definitions. Again we refer to the appendix for the proofs.

6. Implementation

The decompilation relation is a conceptual device, literally a relation, which details what it means for a MINX program to be in correspondence with a MINC program. However, it is well-known [14] that an algorithm can be derived for solving a problem by adding control to relations that specify the problem. Following this methodology, we have translated the decompilation relation rule-for-rule (almost verbatim) into Prolog clauses, using Constraint Handling Rules (CHR) [8] to program the control. Control defaults to leftmost goal selection, with the exception of predicates annotated as CHR. These are interpreted as constraints, which reside in a constraint store, and interact

$$\boxed{\mu_\Gamma; \Gamma_c; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e}$$

$$\text{tr-}\oplus\text{-r*}_1 \frac{(r_i : x)_4, (r_j : y)_4 \in \mu_\Gamma \quad (x : \theta[]*), (y : \mathsf{long}) \in \Gamma_c \quad c = m * \mathit{sizeof}(\theta) \quad \Gamma_c; \Sigma \vdash m : \mathsf{long}}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{op}_4^\oplus\ r_i, r_j * c \overset{\iota}{\rightsquigarrow} x := x \oplus (y * m)}$$

$$\text{tr-}\oplus\text{-r*}_2 \frac{(r_i : x)_w \in \mu_\Gamma \quad (r_j : y)_w \in \mu_\Gamma \quad (x : t) \in \Gamma_c \quad (y : t) \in \Gamma_c \quad \Gamma_c; \Sigma \vdash c : t}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{op}_w^\oplus\ r_i, r_j * c \overset{\iota}{\rightsquigarrow} x := x \oplus (y * c)}$$

$$\text{tr-}\oplus\text{-rc} \frac{(r_i : x)_4 \in \mu_\Gamma \quad (x : \theta[]*) \in \Gamma_c \quad c = m * \mathit{sizeof}(\theta) \quad \Gamma_c; \Sigma \vdash m : t}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{op}_4^\oplus\ r_i, c \overset{\iota}{\rightsquigarrow} x := x \oplus m}$$

$$\text{tr-}\otimes\text{-rc} \frac{(r_i : x)_w \in \mu_\Gamma \quad (x : t) \in \Gamma_c \quad \mathit{sizeof}(t) = w \quad \Gamma_c; \Sigma \vdash c : t}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{op}_w^\otimes\ r_i, c \overset{\iota}{\rightsquigarrow} x := x \otimes c}$$

$$\text{tr-}\otimes\text{-rr} \frac{(r_i : x)_w \in \mu_\Gamma \quad (r_j : y)_w \in \mu_\Gamma \quad (x : t) \in \Gamma_c \quad (y : t) \in \Gamma_c \quad \mathit{sizeof}(t) = w}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{op}_w^\otimes\ r_i, r_j \overset{\iota}{\rightsquigarrow} x := x \otimes y}$$

$$\text{tr-mov-rc} \frac{(r_i : x)_w \in \mu_\Gamma \quad (x : t) \in \Gamma_c \quad \Gamma_c; \Sigma \vdash c : t \quad \mathit{sizeof}(t) = w}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{mov}_w\ r_i, c \overset{\iota}{\rightsquigarrow} x := c}$$

$$\text{tr-mov-r0} \frac{(r_i : x)_4 \in \mu_\Gamma \quad (x : \tau*) \in \Gamma_c}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{mov}_4\ r_i, 0 \overset{\iota}{\rightsquigarrow} x := 0}$$

$$\text{tr-mov-rr} \frac{(r_i : x)_w \in \mu_\Gamma \quad (r_j : y)_w \in \mu_\Gamma \quad (x : \theta_1) \in \Gamma_c \quad (y : \theta_2) \in \Gamma_c \quad \mathit{sizeof}(\theta_1) = \mathit{sizeof}(\theta_2) = w \quad \Sigma \vdash \theta_2 <: \theta_1}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{mov}_w\ r_i, r_j \overset{\iota}{\rightsquigarrow} x := y}$$

$$\text{tr-mov-ri}_1 \frac{(r_i : x)_w, (r_j : y)_4 \in \mu_\Gamma \quad (x : \theta_1), (y : \theta_2*) \in \Gamma_c \quad \Sigma \vdash \theta_2 <: \theta_1 \quad \mathit{sizeof}(\theta_1) = \mathit{sizeof}(\theta_2) = w}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{mov}_w\ r_i, [r_j] \overset{\iota}{\rightsquigarrow} x := *y}$$

$$\text{tr-mov-ir}_1 \frac{(r_i : x)_4, (r_j : y)_w \in \mu_\Gamma \quad (x : \theta_1*), (y : \theta_2) \in \Gamma_c \quad \Sigma \vdash \theta_2 <: \theta_1 \quad \mathit{sizeof}(\theta_1) = \mathit{sizeof}(\theta_2) = w}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{mov}_w\ [r_i], r_j \overset{\iota}{\rightsquigarrow} *x := y}$$

$$\text{tr-mov-ri}_2 \frac{(r_i : x)_w, (r_j : y)_4 \in \mu_\Gamma \quad (x : \theta_1), (y : \theta_2[]*) \in \Gamma_c \quad \Sigma \vdash \theta_2 <: \theta_1 \quad \mathit{sizeof}(\theta_1) = \mathit{sizeof}(\theta_2) = w}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{mov}_w\ r_i, [r_j] \overset{\iota}{\rightsquigarrow} x := y[0]}$$

$$\text{tr-mov-ir}_2 \frac{(r_i : x)_4, (r_j : y)_w \in \mu_\Gamma \quad (x : \theta_1[]*), (y : \theta_2) \in \Gamma_c \quad \Sigma \vdash \theta_2 <: \theta_1 \quad \mathit{sizeof}(\theta_1) = \mathit{sizeof}(\theta_2) = w}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{mov}_w\ [r_i], r_j \overset{\iota}{\rightsquigarrow} x[0] := y}$$

$$\text{tr-mov-ri}_3 \frac{(r_i : x)_w, (r_j : y)_4 \in \mu_\Gamma \quad (x : \theta), (y : N*) \in \Gamma_c \quad \Sigma(N) = \langle \theta_0, \ldots, \theta_{n-1} \rangle \quad \Sigma \vdash \theta_0 <: \theta \quad \mathit{sizeof}(\theta_0) = \mathit{sizeof}(\theta) = w}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{mov}_w\ r_i, [r_j] \overset{\iota}{\rightsquigarrow} x := y \to 0}$$

$$\text{tr-mov-ir}_3 \frac{(r_i : x)_4, (r_j : y)_w \in \mu_\Gamma) \quad (x : N*), (y : \theta) \in \Gamma_c \quad \Sigma(N) = \langle \theta_0, \ldots, \theta_{n-1} \rangle \quad \Sigma \vdash \theta <: \theta_0 \quad \mathit{sizeof}(\theta_0) = \mathit{sizeof}(\theta) = w}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{mov}_w\ [r_i], r_j \overset{\iota}{\rightsquigarrow} x \to 0 := y}$$

$$\text{tr-mov-ri+}_1 \frac{(r_i : x)_w, (r_j : y)_4 \in \mu_\Gamma \quad (x : \theta_1), (y : \theta_2[]*) \in \Gamma_c \quad \mathit{sizeof}(\theta_1) = \mathit{sizeof}(\theta_2) = w \quad \Sigma \vdash \theta_2 <: \theta_1 \quad c = m * \mathit{sizeof}(\theta) \quad \Gamma_c; \Sigma \vdash m : t}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{mov}_w\ r_i, [r_j + c] \overset{\iota}{\rightsquigarrow} x := y[m]}$$

$$\text{tr-mov-i+r}_1 \frac{(r_i : x)_4, (r_j : y)_w \in \mu_\Gamma \quad (x : \theta_1[]*), (y : \theta_2) \in \Gamma_c \quad \mathit{sizeof}(\theta_1) = \mathit{sizeof}(\theta_2) = w \quad \Sigma \vdash \theta_2 <: \theta_1 \quad c = m * \mathit{sizeof}(\theta) \quad \Gamma_c; \Sigma \vdash m : t}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{mov}_w\ [r_i + c], r_j \overset{\iota}{\rightsquigarrow} x[m] := y}$$

$$\text{tr-mov-ri+}_2 \frac{(r_i : x)_w, (r_j : y)_4 \in \mu_\Gamma \quad (x : \theta), (y : N*) \in \Gamma_c \quad \Sigma(N) = \langle \theta_0, \ldots, \theta_{n-1} \rangle \quad c = \sum_{k=0}^{m-1} \mathit{sizeof}(\theta_k) \quad \Sigma \vdash \theta_m <: \theta \quad \mathit{sizeof}(\theta_m) = \mathit{sizeof}(\theta) = w}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{mov}_w\ r_i, [r_j + c] \overset{\iota}{\rightsquigarrow} x := y \to m}$$

$$\text{tr-mov-i+r}_2 \frac{(r_i : x))4, (r_j : y)_w \in \mu_\Gamma \quad (x : N*), (y : \theta) \in \Gamma_c \quad \Sigma(N) = \langle \theta_0, \ldots, \theta_{n-1} \rangle \quad c = \sum_{k=0}^{m-1} \mathit{sizeof}(\theta_k) \quad \Sigma \vdash \theta <: \theta_m \quad \mathit{sizeof}(\theta_m) = \mathit{sizeof}(\theta) = w}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{mov}_w\ [r_i + c], r_j \overset{\iota}{\rightsquigarrow} x \to m := y}$$

$$\text{tr-alloc-r*} \frac{(r_i, x)_4 \in \mu_\Gamma \quad (r_j, y)_{\mathit{sizeof}(t)} \in \mu_\Gamma \quad \Gamma_c; \Sigma \vdash m : t \quad (x : \theta[]*) \in \Gamma_c \quad (y : t) \in \Gamma_c \quad c = \mathit{sizeof}(\theta) * m}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{alloc}\ r_i, r_j * c \overset{\iota}{\rightsquigarrow} x := \mathsf{new}\ \theta[y * m]}$$

$$\text{tr-alloc-rc}_3 \frac{(r_i, x)_4 \in \mu_\Gamma \quad (x : \theta[]*) \in \Gamma_c \quad c = m * \mathit{sizeof}(\theta) \quad \Gamma_c; \Sigma \vdash m : t}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{alloc}\ r_i, c \overset{\iota}{\rightsquigarrow} x := \mathsf{new}\ \theta[m]}$$

$$\text{tr-alloc-rc}_1 \frac{(r_i, x)_4 \in \mu_\Gamma \quad \mathit{sizeof}(\theta) = c \quad (x : \theta*) \in \Gamma_c}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{alloc}\ r_i, c \overset{\iota}{\rightsquigarrow} x := \mathsf{new}\ \theta}$$

$$\text{tr-alloc-rc}_2 \frac{(r_i, x)_4 \in \mu_\Gamma \quad \mathit{sizeof}(N) = c \quad (x : N*) \in \Gamma_c}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{alloc}\ r_i, c \overset{\iota}{\rightsquigarrow} x := \mathsf{new\ struct}\ N}$$

$$\text{tr-call} \frac{\phi_c(f) = f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta'}, l, \lambda_c, j \rangle \quad (r_u : u)_{\mathit{sizeof}(\theta_u)} \in \mu_\Gamma \quad \overrightarrow{(r_v : v)_{\mathit{sizeof}(\theta_v)}} \in \mu_\Gamma \quad (u : \theta_u) \in \Gamma_c \quad \overrightarrow{(v : \theta_v)} \in \Gamma_c \quad \Sigma \vdash \theta'_j <: \theta_u \quad \Sigma \vdash \vec{\theta}_v <: \vec{\theta}}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{call}\ r_u, f, \overrightarrow{r_v} \overset{\iota}{\rightsquigarrow} u := f(\vec{v})}$$

Figure 7: Decompilation of instructions

$$\boxed{\mu_\lambda;\mu_\Gamma;\Gamma_c;\Sigma \vdash b \overset{b}{\rightsquigarrow} s}$$

$$\text{tr-ret}\ \frac{}{\mu_\lambda;\mu_\Gamma;\Gamma_c;\Sigma \vdash \mathsf{ret} \overset{b}{\rightsquigarrow} \mathsf{return}} \qquad \text{tr-goto}\ \frac{\mu_\lambda(a) = l}{\mu_\lambda;\mu_\Gamma;\Gamma_c;\Sigma \vdash \mathsf{goto}\ a \overset{b}{\rightsquigarrow} \mathsf{goto}\ l}$$

$$\text{tr-if}\ \frac{\begin{array}{c}\mu_\lambda(a) = l \quad \mu_\Gamma(r_i) = x \quad (x:\theta) \in \Gamma_c \\ sizeof(\theta) = w \quad \mu_\lambda;\mu_\Gamma;\Gamma_c;\Sigma \vdash b \overset{b}{\rightsquigarrow} s\end{array}}{\mu_\lambda;\mu_\Gamma;\Gamma_c;\Sigma \vdash (\mathsf{if}_w\ r_i\ \mathsf{goto}\ a); b \overset{b}{\rightsquigarrow} (\mathsf{if}\ x\ \mathsf{goto}\ l); s} \qquad \text{tr-instr}\ \frac{\begin{array}{c}\mu_\Gamma;\Gamma_c;\Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e \\ \mu_\lambda;\mu_\Gamma;\Gamma_c;\Sigma \vdash b \overset{b}{\rightsquigarrow} s\end{array}}{\mu_\lambda;\mu_\Gamma;\Gamma_c;\Sigma \vdash \iota; b \overset{b}{\rightsquigarrow} \ell := e; s}$$

$$\boxed{\Sigma \vdash d_x \rightsquigarrow d_c}$$

$$\text{tr-def}\ \frac{\begin{array}{c}\mu_\Gamma = \{\overrightarrow{r_x \mapsto x}, \overrightarrow{r_y \mapsto y}\} \qquad \Gamma_c = \{\overrightarrow{x:\theta}, \overrightarrow{y:\theta'}\} \qquad r_{y_j} \in \overrightarrow{r_y} \qquad a \in dom(\lambda_x) \\ \mu_\lambda = \{dom(\lambda_x) \mapsto dom(\lambda_c)\} \quad \mu_\lambda(a) = l \quad \forall (a \mapsto l) \in \mu_\lambda :\ \mu_\lambda;\mu_\Gamma;\Gamma_c;\Sigma \vdash \lambda_x(a) \overset{b}{\rightsquigarrow} \lambda_c(l)\end{array}}{\Sigma \vdash \langle f, \overrightarrow{r_x}, \overrightarrow{r_y}, a, \lambda_x, j\rangle \rightsquigarrow f(\overrightarrow{x:\theta})\langle \overrightarrow{y:\theta'}, l, \lambda_c, j\rangle}$$

Figure 8: Decompilation of basic blocks and function definitions

with one another to realise propagation, delay non-deterministic choice, and thereby avoid needless backtracking. As an illustration, consider the `struct(`N`,`c`,`m`,`θ`)` constraint which holds iff $\Sigma(N) = \langle\theta_0, \ldots, \theta_{n-1}\rangle$, $c = \sum_{i=0}^{m-1} \mathsf{sizeof}(\theta_i)$ and $\theta = \theta_m$. Two such constraints in the store that share the same N can be combined into one provided they share the same byte offset c, an action that both simplifies the store and performs propagation. This can be specified in CHR as:

```
struct(N,C,M1,Ty1) \ struct(N,C,M2,Ty2) <=>
    M1 = M2, Ty1 = Ty2.
```

Furthermore, given a CHR constraint `sizeOf(`θ`,`w`)` that holds iff $w = \mathsf{sizeof}(\theta)$, and two constraints `struct(`N`,`c`,`m`,`θ`)` and `struct(`N`,`$c+w$`,`m'`,`θ'`)` it follows $m' = m + 1$. This form of propagation can be realised in CHR using:

```
struct(N,C1,M1,Ty1), struct(N,C2,M2,_Ty2) ==>
    nonvar(C1), nonvar(C2), sizeOf(Ty1,W),
    nonvar(W), C2 =:= C1 + W
    |
    M2 #= M1 + 1.
```

CHR rules are likewise used to express the subtyping relation. In all, this gives a solver for computing a witness and its type, in less than 900 LOC, but more importantly, yields one that is faithful to the rules of the decompilation relation.

To generate input for the solver, the self-hosting ANSI C89 compiler `ucc` [29] was retargeted to generate MINX. Our derivative, dubbed `minxcc`, supports the core features of C89. To stay within MINX, `minxcc` applies some rather unusual transformations. Each constant string, which is normally encoded as a global pointer literal, is converted into in a function that returns a pointer to newly allocated heap memory, that contains the string. `malloc` (and its friends) are replaced by the `alloc` instruction, while calls to `free` are removed completely. Memory thus grows as execution proceeds, exactly as specified in Fig 5. Mathematical operations such as `=<` and logical operations such as `xor` are reduced to MINX operations using equivalences taken from Hacker's Delight [11].

As a sanity check, we wrote a Haskell interpreter for MINX, following the SOS semantics of Fig 2. The results of interpreting the MINX code, on various inputs, were then checked against those obtained by compiling the benchmarks using `gcc`, which was taken as ground truth. An interpreter was written in Haskell, realising the SOS of Fig 5, to check the generated MINC witness programs. As a final check, and for the satisfaction of going full circle, a translator was written in Haskell to convert the MINC witness program into C, for testing with `gcc`. Each benchmark was subject to each of these three levels of checking.

	LoC				recovered	time
benchmark	**C**	**MINX**	**solns**	**structs**	**structs**	**(mS)**
aatree	315	2,734	1	1	1	6,543
avltree	269	2,188	1	1	2	2,041
binheap	184	2,558	2	2	2/1	109
binomial	303	3,732	2	2	5/4	249
hashsep	256	1,017	4	2	6/5/6/5	77
hashquad	260	977	4	2	2/3/2/2	49
kdtree	112	891	1	1	1	1,527
leftheap	182	762	1	1	1	825
list	262	829	2	1	2/1	1,054
mergesort	135	664	1	1	1	628
pairheap	298	3,216	1	1	2	3,316
queue	188	1,960	1	1	1	886
redblack	317	2,918	2	1	3/2	193
sets	120	355	1	0	0	276
skip	239	2,616	1	1	1	2,089
sort	364	2,339	1	0	0	2,904
splay	332	2,648	1	1	2	4,975
stackar	161	1,680	1	1	1	640
stackli	140	1,270	2	1	2/2	464
treap	288	2,580	1	1	2	3,834
tree	208	1,104	1	1	2	2,158

Figure 10: Solutions and Recovered Structures

7. Evaluation

The solver was deployed on a suite of textbook [30] programs, chosen because of their use of data-structures. Figure 10 lists the *benchmarks*, complete with *LoC* for the C and the MINX assembly files. The *solns* column records the number of type assignment and witness program pairs generated by the solver. The *structs* columns indicates the number of struct types declared in the benchmark, whereas *recovered structs* records the number of struct definitions in each of the solutions. Thus 6/5/6/5, for example, indicates that the first solution has 6 recovered structs, the second has 5, the third 6, and the fourth 5. The benchmarks were run on a single core Intel Atom Z540 at 1.86GHz with 2GB of RAM. The benchmarks, assembly files, and witnesses (which embed the recovered types) are all available in the online appendix.

Observe that some benchmarks have more than one solution and some solutions have a different number of struct definitions than the original program. This is due to several factors: Homogeneous structs, where every (accessed) field has the same type, cannot be

distinguished from arrays, and therefore can be typed either as an array or a struct. This issue is exhibited by the `binheap` benchmark.

When the same struct type is used in separate parts of a program (e.g., in functions that are never called) our decompiler generates distinct copies of the struct. In most cases the definitions are identical (as in the `tree` benchmark), however a function may not access every field of a struct, leading to an under-constrained type assignment problem and a struct definition that omits the unaccessed fields, as in the `treap` and `avltree` benchmarks. Combining these issues can result in multiple solutions that differ in their types and number of structs, which arises in the `binomial` benchmark.

The key point, however, is not visible from the table: there exists one witness program whose regenerated types are identical to those of the original benchmark. Moreover, for benchmarks with multiple solutions, every witness has types compatible with those of the original program (in the sense that arrays are compatible with homogeneous structs). In addition, all *recursive* types in every witness are present in the original, and every *recursive* type present in the original appears in every witness. Furthermore, when translated back into C, each witness behaves as the original benchmark.

8. Related Work

A self-contained introduction to type recovery is given in [28, chapter 5], which summarises the problem as "The ... problem for a decompiler is to associate each piece of data with a high-level type". The author, like others [6, 27], introduces a dataflow analysis over type lattice of primitive types, but accepted wisdom is to formulate type inference as constraint solving because dataflow analysis classically deals with unidirectional flows.

Dynamic type recovery Dynamic techniques have been suggested for type recovery [18], in which types are reconstructed from an execution trace. Each memory location accessed by the program is tagged with a timestamp because the same location can store values of different type over its lifetime. Each location and timestamp pair is then assigned a type, in either the on-line or off-line phases of the type recovery algorithm. In the on-line phase known types are propagated to the pairs, as a value which inhabits a type, is stored in a location. However, the type may remain unknown until the control encounters a system call or a library call, or some machine instruction, whose arguments or operands expose the type. The on-line phase is thus augmented with an off-line phase which propagates type assignment against the control to resolve any unassigned pair. The method requires an oracle to supervise the selection of the trace, and an examination of one trace will fail to infer types that hold universally across the whole program.

Somewhat surprisingly, Bayesian unsupervised learning has been applied to recognise structure in memory images [5]. The memory image is scanned, looking for all pointers, which are then used to locate the positions of objects and their size, which is bounded by the distance to the next object. Unsupervised learning is then used to classify malware according to its memory layout, a technique that could be taken further with static type recovery.

Recursive datatypes Mycroft [21] recognised that type reconstruction could rule out inconsistent decompilation steps and thereby aid program reconstruction. This link is formalised in our decompilation relation that is the centrepiece of our formalisation. His work was inspired by the desire to synthesise datatypes from register transfer language (RTL) code generated from BCPL, which itself is untyped, not distinguishing between arrays and structures. He discussed the issue of padding, which arises when some of the fields of a structure, but not all, can be inferred, as well as proposing a type unification algorithm for synthesising a recursive datatype when a type variable is unified with a term containing it.

This approach has been reexamined though the perspective of SMT [25] using the theory of rational trees (cyclic unification) [10], so that the solved form can directly encode the inferred recursive datatypes. Operations such as addition can be assigned one of three possible types, depending on whether the operands are an integer and a pointer, or vice versa, or simply two integers. This case splitting can be encoded propositionally and the theory used for datatype assembly. Neither of these work, however, formally relate the recovered types to the binary itself. They focus on solving rather than semantics.

SecondWrite [7] extends work on variable recovery [1, 2] with so-called best effort pointer analysis [7, Section 5.2] to infer some datatypes: they "dig into the points-to set to discover if it is pointing to an address which is declared as the starting point of a structure". Generic types are used for symbols for which they cannot infer types, and type casts are introduced to convert the generic type to the actual types used in an operation.

Following the idea that "well-typed programs cannot go wrong" [19], type recovery has been muted as a check for the validity of low-level code [24]. Recursive types are recovered using a rational-tree solver from the low-level typeless template code of a graph reduction machine. If the solver fails, the code is judged unsafe.

Verified decompilation and disassembly Decompilation is not always to C: Java bytecode has been decompiled into recursive functions, based on type theory [12], which is amenable to formal reasoning. Worthy of particular note, is the decompilation of machine code into the language of HOL4 [22]. With a view to proving full functional correctness, machine code is decompiled into tail-recursive functions. These functions describe the effect of the machine code, yet offer a layer of abstraction above it. Properties proved for the function are, by an automatically derived theorem, related to the original machine code, so the decompiler does not need to be proved correct. Recursive predicates could be defined in HOL4 to assert in that memory conformed to a recursive datatype, but for the purposes for engaging with the reverse engineer, it seems more natural to decompile to a type-safe dialect of C.

Disassembly, the act of decoding the bit patterns of machine instructions into a textual representation, is itself non-trivial for self-modifying code. Self-modification is used to disguise malware but also arises in JIT compilation. In general, disassembly requires indirect jump targets to be computed, which can be approximated by abstract interpretation [13]. In the case of self modifying code, each memory write needs to be checked to determine how it modifies the code base. Modifications to the code base themselves entail a form of abstract decoding in which the analyser does not recover the exact instruction, but a collection of applicable instructions. Nevertheless disassembly has been formalised [3], though we consider self-modification to be beyond the scope of our study.

Further afield, is the wide body of work on trusted compilation, most notably represented by the CompCert project that produced a fully certified optimising C compiler [16]. The CompCert compiler transforms source code into machine code incrementally through a large number of intermediate languages, each of which is designed to handle a specific compilation stage such as common subexpression elimination, register allocation or control flow linearisation. Correctness is verified at each successive level of transformation with proofs created using a proof assistant. Where CompCert aims at proving semantic correctness of these optimisations, our interest is in type preservation and any witness, no matter how closely it mirrors the binary, is sufficient for our type correctness argument.

Typed Assembly Language (TAL) [20] represents another approach to trusted compilation, where the aim is to prove that type consistency is maintained through compilation to an assembly language that can be type checked to prove safety properties of the executable code. The limitation is that no machine exists that can

execute TAL, so as a compromise the code is type checked either when assembled to machine code or by a runtime loader that recognises a special typed object format. In contrast we have provided a type inference algorithm for assembler that allows type checking without the presence of any explicit type information in the binary.

9. Discussion

System and library calls provide a rich source of type information, from both their arguments and return types [18]. In the short term, we plan to harvest these types and exploit them in our solver. We will also relax the restrictions on the MINX calling convention, using dataflow analysis to identify arguments, that is, what is written before and what is read after a function call [1, 7]. To aid readability, we will also recognise the shape of certain (reoccurring) structures to refine the auto-generated structure and field names.

We only recover types if there exists a well-typed witness that corresponds semantically to the binary. This is not a limitation; it is a deliberate design choice. While not a problem for our benchmarks, the binary could be compiled from source that includes cast conversions, which are erased in the binary, but induce type conflicts. With a view to deployment, we intend to extend MINC to union types but preserve the type safety of MINC by raising a type error if a union field value is out of range. Recent work on the automatic localisation of type errors [23] has shown how (weighted) MaxSMT [17] can be applied to compute type conflicts that are minimal, subject to a compiler-specific ranking criterion. The idea is to minimise the sum of the weights of the unsatisfied (type) constraints. Since type recovery can been formulated as SMT [25], we intend to apply error localisation to introduce a union type whenever a conflict is encountered, but do so in a way that minimises impact on the witness. This is a medium term goal.

The step beyond inferring an arbitrary well-typed witness for the binary, is to infer a witness that is comprehensible to the reverse engineer. Enriching the decompilation relation with more rules, possibly even for sequences of instructions, would increase the class of MINC programs that can be regenerated, and provide the solver with latitude to select one decompilation over another. As a long term research goal, we intend to explore how preferences [17] can steer the solver towards emitting a more intelligible witness, though it is far from clear how this can be quantified.

10. Conclusions

We have answered the fundamental question of how to derive types from a binary executable that truly have semantic meaning. Our answer is both principled and unique in that derives a high-level witness program in concert with the types. We prove that the witness inhabits the inferred types, and establish a notion of memory consistency with the binary, thereby showing that the binary conforms to the inferred types. Apart from establishing type correctness, the construction also yields a type-based decompiler. We have evaluated the decompiler on more than 20 textbook programs and, for all, have recovered a witness program in a type-safe dialect of C, complete with the original recursive datatypes.

References

[1] G. Balakrishnan and T. Reps. Analyzing Memory Accesses in x86 Executables. In *CC*, LNCS, pages 5–23. Springer, 2004.

[2] G. Balakrishnan and T. Reps. Divine: Discovering Variables in Executables. In *VMCAI*, LNCS, pages 1–28. Springer, 2007.

[3] S. Blazy, V. Laporte, and D. Pichardie. Verified Abstract Interpretation Techniques for Disassembling Low-level Self-modifying Code. In *ITP*, volume 8558 of *LNCS*, pages 128–143, 2014.

[4] E. Chan, S. Venkataraman, N. Tkach, K. Larson, A. Gutierrez, and R. H. Campbell. Characterizing Data Structures for Volatile Forensics. In *Systematic Approaches to Digital Forensic Engineering*, pages 1–9, 2011.

[5] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging For Data Structures. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 231–244. USENIX, 2008.

[6] E. Dolgova and A. Chernov. Automatic Reconstruction of Data types in the Decompilation Problem. *Programming and Computer Software*, 35(2):105–119, 2009.

[7] K. Elwazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable Variable and Data Type Detection in a Binary Rewriter. In *PLDI*, pages 51–60, 2013.

[8] T. Frühwirth. *Constraint Handling Rules*. CUP, 2009.

[9] I. Guilfanov. A Simple Type System for Program Reengineering. In *WCRE*, pages 357–. IEEE Computer Society, 2001.

[10] J. Jaffar. Efficient Unification over Infinite Terms. *New Generation Computing*, 2(3):207–219, 1984.

[11] H. S. Warren Jr. *Hacker's Delight*. Addison-Wesley, 2002.

[12] S. Katsumata and A. Ohori. Proof-Directed De-compilation of Low-Level Code. In *ESOP*, volume 2028 of *LNCS*, pages 352–366. Springer, 2001.

[13] J. Kinder, H. Veith, and F. Zuleger. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *VMCAI*, volume 5403 of *LNCS*, pages 214–228. Springer, 2009.

[14] R. Kowalski. Algorithm = Logic + Control. *CACM*, 22(7):424–436, 1979.

[15] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *NDSS*. The Internet Society, 2011.

[16] X. Leroy. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *POPL*, pages 42–54, 2006.

[17] C. M. Li and F. Manyà. MaxSAT, Hard and Soft Constraints. In *Handbook of Satisfiability*, pages 613–631. IOS Press, 2009.

[18] Z. Lin, X. Zhang, and D. Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *NDSS*. The Internet Society, 2010.

[19] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[20] G. Morrisett and D. Walker. From System F to Typed Assembly Language. *TOPLAS*, 21(3):527–568, 1999.

[21] A. Mycroft. Type-Based Decompilation (or Program Reconstruction via Type Reconstruction). In *ESOP*, volume 1576 of *LNCS*, pages 208–223. Springer, 1999.

[22] M. O. Myreen, M. J. C. Gordon, and K. Slind. Machine-Code Verification for Multiple Architectures - An Application of Decompilation into Logic. In *FMCAD*, pages 1–8, 2008.

[23] Z. Pavlinovic, T. King, and T. Wies. Finding Minimum Type Error Sources. In *OOPSLA*, pages 525–542. ACM Press, 2014.

[24] M. P. Peres Cervantes. *Static Methods to Check Low-Level Code for a Graph Reduction Machine*. PhD thesis, University of York, 2014. `http://etheses.whiterose.ac.uk/id/eprint/6248`.

[25] E. Robbins, J. M. Howe, and A. King. Theory Propagation and Rational-Trees. In *PPDP*, pages 193–204. ACM Press, 2013.

[26] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[27] K. Troshina, Y. Derevenets, and A. Chernov. Reconstruction of composite types for Decompilation. In *Working Conference on Source Code Analysis and Manipulation*, pages 179–188, 2010.

[28] M. J. Van Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, University of Queensland, 2007. `http://espace.library.uq.edu.au/view/UQ:158682`.

[29] W. Wang. Ucc, 2014. `http://ucc.sourceforge.net/`.

[30] M. A. Weiss. *Data Structures and Algorithm Analysis in C*. Addison-Wesley, 1996.

A. Proof Appendix

A.1 Type Safety

We write $\Sigma; \Psi \vdash \sigma; \pi$ to signify that

$$\forall (a : \theta) \in \Psi \,.\, \Sigma; \Psi; \sigma; \pi \vdash a : \theta$$

We also write $\Gamma_c; \Sigma; \Psi \vdash \rho$ to signify that

$$\forall (x : \theta) \in \Gamma_c \,.\, \Sigma; \Psi \vdash \rho(x) : \theta * \wedge \rho(x) \neq 0$$

Moreover, we write $\Gamma_c; \Sigma \vdash \lambda_c$ to signify that

$$\forall s \in \mathit{range}(\lambda_c).\ \Gamma_c; \Sigma \vdash s$$

Proposition 8 (safety for lvalue evaluation)**.**

1. Progress: if
   - $\Gamma_c; \Sigma; \Psi \vdash \rho$
   - $\Sigma; \Psi \vdash \sigma; \pi$
   - $\Gamma_c; \Sigma \vdash \ell : \theta$

   then
   (a) $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \langle \sigma', \pi', a \rangle$ or
   (b) $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \mathsf{err}$.
2. Preservation: if
   - $\Gamma_c; \Sigma; \Psi \vdash \rho$
   - $\Sigma; \Psi \vdash \sigma; \pi$
   - $\Gamma_c; \Sigma \vdash \ell : \theta$
   - $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \langle \sigma', \pi', a \rangle$

   then for some $\Psi' \supseteq \Psi$
   (a) $\Gamma_c; \Sigma; \Psi' \vdash \rho$
   (b) $\Sigma; \Psi' \vdash \sigma'; \pi'$
   (c) $\Sigma; \Psi' \vdash a : \theta *$

Proposition 9 (safety for expression evaluation)**.**

1. Progress: if
   - $\Gamma_c; \Sigma; \Psi \vdash \rho$
   - $\Sigma; \Psi \vdash \sigma; \pi$
   - $\Gamma_c; \Sigma \vdash e : \theta$

   then
   (a) $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e \rangle \xrightarrow{e} \langle \sigma', \pi', v \rangle$ or
   (b) $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e \rangle \xrightarrow{e} \mathsf{err}$.
2. Preservation: if
   - $\Gamma_c; \Sigma; \Psi \vdash \rho$
   - $\Sigma; \Psi \vdash \sigma; \pi$
   - $\Gamma_c; \Sigma \vdash e : \theta$
   - $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e \rangle \xrightarrow{e} \langle \sigma', \pi', v \rangle$

   then for some $\Psi' \supseteq \Psi$
   (a) $\Gamma_c; \Sigma; \Psi' \vdash \rho$
   (b) $\Sigma; \Psi' \vdash \sigma'; \pi'$
   (c) $\Sigma; \Psi' \vdash v : \theta$

Proposition 10 (safety for statement evaluation)**.**

1. Progress: if
   - $\Gamma_c; \Sigma; \Psi \vdash \rho$
   - $\Sigma; \Psi \vdash \sigma; \pi$
   - $\Gamma_c; \Sigma \vdash s$
   - $\Gamma_c; \Sigma \vdash \lambda_c$

   then
   (a) $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, s \rangle \xrightarrow{s} \langle \sigma', \pi', s' \rangle$ or
   (b) $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, s \rangle \xrightarrow{s} \mathsf{err}$ or
   (c) $s = \mathsf{return}$.
2. Preservation: if
   - $\Gamma_c; \Sigma \vdash s$
   - $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, s \rangle \xrightarrow{s} \langle \sigma', \pi', s' \rangle$
   - $\Gamma_c; \Sigma; \Psi \vdash \rho$
   - $\Sigma; \Psi \vdash \sigma; \pi$

   then for some $\Psi' \supseteq \Psi$
   (a) $\Gamma_c; \Sigma; \Psi' \vdash \rho$
   (b) $\Sigma; \Psi' \vdash \sigma'; \pi'$
   (c) $\Gamma_c; \Sigma \vdash s'$

Proposition 11 (safety for function definitions)**.**

1. Progress: if
   - $\Sigma \vdash f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta'}, l, \lambda_c, j \rangle$
   - $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda_c(l) \rangle \xrightarrow{s}{}^{*} \langle \sigma', \pi', \mathsf{return} \rangle$
   - $\Gamma_c = \{\overrightarrow{x : \theta}, \overrightarrow{y : \theta'}\}$
   - $\Gamma_c; \Sigma; \Psi \vdash \rho$
   - $\Sigma; \Psi \vdash \sigma; \pi$

   then
   (a) $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda_c(l) \rangle \xrightarrow{s}{}^{*} \langle \sigma', \pi', \mathsf{return} \rangle$ or
   (b) $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda_c(l) \rangle \xrightarrow{s}{}^{*} \mathsf{err}$ (we assume this subsumes divergence).
2. Preservation: if
   - $\Sigma \vdash f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta'}, l, \lambda_c, j \rangle$
   - $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda_c(l) \rangle \xrightarrow{s}{}^{*} \langle \sigma', \pi', \mathsf{return} \rangle$
   - $\Gamma_c = \{\overrightarrow{x : \theta}, \overrightarrow{y : \theta'}\}$
   - $\Gamma_c; \Sigma; \Psi \vdash \rho$
   - $\Sigma; \Psi \vdash \sigma; \pi$

   then for some $\Psi' \supseteq \Psi$
   (a) $\Gamma_c; \Sigma; \Psi' \vdash \rho$
   (b) $\Sigma; \Psi' \vdash \sigma'; \pi'$

Proof 1. Propositions 8, 9, 10 and 11 proved together by mutual structural induction on the typing judgements for ℓ, e, s and d_c.

- By case analysis on $\Gamma_c; \Sigma \vdash \ell : \theta$ in Fig. 4. To show 1b or conversely 1a, 2a, 2b and 2c hold for proposition 8. Observe that 2a holds if $\Psi' \supseteq \Psi$.

  1. Let $\ell = x$. By rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \rangle \xrightarrow{\ell} \langle \sigma, \pi, a \rangle$ where $a = \rho(x)$ hence 1a holds. Put $\Psi' = \Psi$. Since $\Gamma_c; \Sigma; \Psi \vdash \rho$ it follows $\Sigma; \Psi' \vdash \rho(x) : \theta *$ and 2c holds. Moreover $\Sigma; \Psi' \vdash \sigma; \pi$ and 2b holds.
  2. Let $\ell : \theta = *x : \tau$. Since $\Gamma_c; \Sigma; \Psi \vdash \rho$ it follows $a = \rho(x) \neq 0$. By rule l-ptr $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, *x \rangle \xrightarrow{\ell} \langle \sigma, \pi, \sigma(a) \rangle$ thus 1a holds. Put $\Psi' = \Psi$. By rule t-ptr $\Gamma_c; \Sigma \vdash x : \tau *$ and by $\Gamma_c; \Sigma; \Psi \vdash \rho$ it follows $\Sigma; \Psi \vdash a : \tau * *$. By rule vt-addr $(a : \tau *) \in \Psi$ and by $\Sigma; \Psi \vdash \sigma; \pi$ it follows $\Sigma; \Psi; \sigma; \pi \vdash a : \tau *$. By rule st-comp $\Sigma; \Psi \vdash \sigma(a) : \tau *$ thus $\Sigma; \Psi' \vdash \sigma(a) : \tau *$ and 2c holds. Moreover $\Sigma; \Psi' \vdash \sigma; \pi$ and 2b holds.
  3. Let $\ell : \theta = x \to c : \theta_c$. Since $\Gamma_c; \Sigma; \Psi \vdash \rho$ let $a = \rho(x) \neq 0$ and let $v = \sigma(a) +_\perp c$. If $\rho(x) = 0$ or $v \notin \cup\pi$ then 1b holds. Otherwise $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \to c \rangle \xrightarrow{\ell} \langle \sigma, \pi, v \rangle$ and 1a holds. Put $\Psi' = \Psi$. By rule t-fld $\Gamma_c; \Sigma \vdash x : N *$ and by rule t-var $(x : N *) \in \Gamma_c$ and by $\Gamma_c; \Sigma; \Psi \vdash \rho$ it follows $\Sigma; \Psi \vdash \rho(x) : N * *$. By rule vt-addr $(\rho(x) : N *) \in \Psi$ and by $\Sigma; \Psi \vdash \sigma; \pi$ it follows $\Sigma; \Psi; \sigma; \pi \vdash \rho(x) : N *$ and by rule st-comp $\Sigma; \Psi \vdash \sigma(\rho(x)) : N *$. By rule vt-addr $(\sigma(\rho(x)) : N) \in \Psi$ and by $\Gamma_c; \Sigma; \Psi \vdash \rho$ it follows $\Sigma; \Psi; \sigma; \pi \vdash \sigma(\rho(x)) : N$ and by rule st-fld $\Sigma; \Psi \vdash \sigma(\sigma(\rho(x)) + c) : \theta_c$. By rule st-comp $\Sigma; \Psi; \sigma; \pi \vdash \sigma(\rho(x)) + c : \theta_c$ and by $\Gamma_c; \Sigma; \Psi \vdash \rho$ it follows $(\sigma(\rho(x)) + c : \theta_c) \in \Psi$ and by rule vt-addr

$$\boxed{\Sigma \vdash \theta} \qquad \frac{}{\Sigma \vdash \mathsf{short}} \qquad \frac{}{\Sigma \vdash \mathsf{long}} \qquad \frac{\Sigma \vdash \tau}{\Sigma \vdash \tau*} \qquad \frac{N \in \Sigma}{\Sigma \vdash N}$$

$$\boxed{\Sigma \vdash \mathsf{decls} \xrightarrow{d} \Sigma'} \qquad \frac{}{\Sigma \vdash \epsilon \xrightarrow{d} \Sigma} \qquad \frac{\begin{array}{c}\Sigma(N) = \bot \vee N \notin dom(\Sigma) \\ \Sigma' = \Sigma \circ \{N \mapsto \vec{\theta}\} \\ \forall \theta_i \in \vec{\theta}.(\Sigma' \vdash \theta_i) \\ \Sigma' \vdash \mathsf{decls} \xrightarrow{d} \Sigma''\end{array}}{\Sigma \vdash \mathsf{struct}\ N(\vec{\theta}); \mathsf{decls} \xrightarrow{d} \Sigma''} \qquad \frac{N \notin dom(\Sigma) \quad \Sigma' = \Sigma \circ \{N \mapsto \bot\} \quad \Sigma' \vdash \mathsf{decls} \xrightarrow{d} \Sigma''}{\Sigma \vdash \mathsf{struct}\ N; \mathsf{decls} \xrightarrow{d} \Sigma''}$$

Figure 11: Well-formed type declarations of MINC programs

$\Sigma; \Psi \vdash \sigma(\rho(x)) + c : \theta_c*$ and 2c holds since $\Psi' = \Psi$. Moreover $\Sigma; \Psi' \vdash \sigma; \pi$ and 2b holds.

4. Let $\ell = x[e']$. By rule t-ar $\Gamma_c; \Sigma \vdash e' : t$ hence by mutual induction:
   - Either $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e' \rangle \xrightarrow{e} \mathsf{err}$. By rule e-lval-err $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x[e'] \rangle \xrightarrow{e} \mathsf{err}$. Hence 1b.
   - Or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e' \rangle \xrightarrow{e} \langle \sigma', \pi', v \rangle$. If $\rho(x) = 0$ then 1a holds by rule e-lval-err. Otherwise let $a = \sigma'(\rho(x)) +_\bot v$. If $a \notin \cup \pi'$ then 1a holds. Otherwise by rule l-ar $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x[e'] \rangle \xrightarrow{\ell} \langle \sigma', \pi', a \rangle$. Hence 1a holds.
     By induction there exists $\Psi' \supseteq \Psi$ such that $\Sigma; \Psi' \vdash \sigma'; \pi'$. By rule t-ar $\Gamma_c; \Sigma \vdash x : \theta[]*$ and by rule t-var $(x : \theta[]*) \in \Gamma_c$ and by $\Gamma_c; \Sigma; \Psi' \vdash \rho$ it follows $\Sigma; \Psi' \vdash \rho(x) : \theta[] **$. By rule vt-addr $(\rho(x) : \theta[]*) \in \Psi'$ and by $\Sigma; \Psi' \vdash \sigma'; \pi'$ it follows $\Sigma; \Psi'; \sigma'; \pi' \vdash \rho(x) : \theta[]*$ and by rule st-comp $\Sigma; \Psi' \vdash \sigma'(\rho(x)) : \theta[]*$. By rule vt-addr $(\sigma'(\rho(x)) : \theta[]) \in \Psi'$ and by $\Gamma_c; \Sigma; \Psi' \vdash \rho$ it follows $\Sigma; \Psi'; \sigma'; \pi' \vdash \sigma'(\rho(x)) : \theta[]$ and by rule st-ar $\Sigma; \Psi' \vdash \sigma'(\sigma'(\rho(x)) + v) : \theta$. By rule st-comp $\Sigma; \Psi'; \sigma'; \pi' \vdash \sigma'(\rho(x)) + v : \theta$ and by $\Gamma_c; \Sigma; \Psi' \vdash \rho$ it follows $(\sigma'(\rho(x)) + v : \theta) \in \Psi'$ and by rule vt-addr $\Sigma; \Psi' \vdash \sigma'(\rho(x)) + v : \theta*$ and 2c holds. Moreover $\Sigma; \Psi' \vdash \sigma; \pi$ and 2b holds.

- By case analysis on $\Gamma_c; \Sigma \vdash e : \theta$ in Fig. 4. To show that either 1b or conversely 1a, 2a 2b and 2c of Proposition 9 hold. Observe that 2a holds if $\Psi' \supseteq \Psi$.

  1. Let $e : \theta = \&x : \tau*$. By rule t-amp $\Gamma_c; \Sigma \vdash x : \tau$ thus $(x : \tau) \in \Gamma_c$ and by $\Gamma_c; \Sigma; \Psi \vdash \rho$ it follows $\Sigma; \Psi \vdash a : \tau*$ where $a = \rho(x) \neq 0$. By rule e-amp $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \&x \rangle \xrightarrow{e} \langle \sigma, \pi, a \rangle$ hence 1a holds. Put $\Psi' = \Psi$ thus $\Sigma; \Psi' \vdash a : \tau*$ and 2c holds whilst 2b is immediate.
  2. Let $e : \theta = c_l : \mathsf{long}$. By rule e-const $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, c_l \rangle \xrightarrow{e} \langle \sigma, \pi, c_l \rangle$. Hence 1a.
     Let $\Psi' = \Psi$. By rule vt-l $\Sigma; \Psi \vdash c_l : \mathsf{long}$. Hence 2c. Also 2b.
  3. Let $e : \theta = c_s : \mathsf{short}$. By rule e-const $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, c_s \rangle \xrightarrow{e} \langle \sigma, \pi, c_s \rangle$. Hence 1a.
     Let $\Psi' = \Psi$. By rule vt-s $\Sigma; \Psi \vdash c_s : \mathsf{short}$. Hence 2c. Also 2b.
  4. Let $e : \theta = 0_l : \tau*$. By rule e-const $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, 0_l \rangle \xrightarrow{e} \langle \sigma, \pi, 0_l \rangle$. Hence 1a.
     Let $\Psi' = \Psi$. By rule vt-null $\Sigma; \Psi \vdash c_s : \tau*$. Hence 2c. Also 2b.
  5. Let $e : \theta = \mathsf{new}\ \tau : \tau*$. By rule e-new $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{new}\ \tau \rangle \xrightarrow{e} \langle \sigma', \pi, a \rangle$ where $\sigma' = \sigma \circ \{a \mapsto \bot\}$. Hence 1a.
     Let $\Psi' = \Psi \circ \{a \mapsto \tau\}$. By rule vt-addr $\Sigma; \Psi \vdash a : \tau*$ hence 2c. Also by rule vt-bot $\Sigma; \Psi' \vdash \bot : \tau$ by and rule st-comp $\Sigma; \Psi'; \sigma'; \pi \vdash a : \tau$ hence $\Sigma; \Psi' \vdash \sigma'; \pi$ and 2b holds.
  6. Let $e : \theta = \mathsf{new\ struct}\ N : N*$ and $n = |\Sigma(N)|$. By rule e-str $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{new\ struct}\ N \rangle \xrightarrow{e} \langle \sigma', \pi', a \rangle$ where $\sigma' = \sigma \circ \{a \mapsto \bot, \ldots, a + n - 1 \mapsto \bot\}$ and $\pi' = \pi \cup \{[a, a + n - 1]\}$. Put $\Psi' = \Psi \cup \{a : N, a + 1 : \theta_1, \ldots, a + n - 1 : \theta_{n-1}\}$. By rule vt-addr $\Sigma; \Psi' \vdash a : N*$ hence 2c holds.
     Let $i \in [0, n-1]$. Then $\sigma'(a+i) = \bot$ hence $\Sigma; \Psi' \vdash \sigma'(a+i) : \theta_i$ by rule vt-bot therefore $\Sigma; \Psi'; \sigma'; \pi' \vdash a + i : \theta_i$. By rule st-fld $\Sigma; \Psi'; \sigma'; \pi' \vdash a : N$ hence 2b holds.
  7. Let $e : \theta = \mathsf{new}\ \theta[e] : \theta[]*$. By rule t-new-ar $\Gamma_c; \Sigma \vdash e : t$ hence by induction:
     - Either $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e \rangle \xrightarrow{e} \mathsf{err}$. By rule e-ar-err $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{new}\ \theta[e] \rangle \xrightarrow{e} \mathsf{err}$. Hence 1b.
     - Or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e \rangle \xrightarrow{e} \langle \sigma', \pi', v \rangle$. By rule e-ar $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{new}\ \theta[e] \rangle \xrightarrow{e} \langle \sigma'', \pi'', a \rangle$ where $\sigma'' = \sigma' \circ \{a \mapsto \bot, \ldots, a + v - 1 \mapsto \bot\}$. Hence 1a.
       By induction there exists $\Phi' \supseteq \Phi$ such that $\Sigma; \Psi' \vdash \sigma'; \pi'$. Put $\Psi'' = \Psi' \circ \{a \mapsto \theta[], \ldots, a + v - 1 \mapsto \theta[]\}$. By rule vt-addr it follows $\Sigma; \Psi'' \vdash a : \theta[]*$ hence 2c. By rule vt-bot it follows $\Sigma; \Psi'' \vdash \bot : \theta[]$ and by st-comp it follows $\Sigma; \Psi''; \sigma''; \pi'' \vdash a + i : \theta[]$ for all $i \in [0, v - 1]$ hence 2b.
  8. Let $e : \theta = (e_1 \oplus e_2) : t$. By rule t-$\otimes$ $\Gamma_c; \Sigma \vdash e_1 : t$ and $\Gamma_c; \Sigma \vdash e_2 : t$. Hence by induction:
     - Either $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e_1 \rangle \xrightarrow{e} \mathsf{err}$. By rule e-op-err$_1$ $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (e_1 \oplus e_2) \rangle \xrightarrow{e} \mathsf{err}$. Hence 1b.
     - Or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma', \pi', e_2 \rangle \xrightarrow{e} \mathsf{err}$. Like previous case.
     - Or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e_1 \rangle \xrightarrow{e} \langle \sigma', \pi', v_1 \rangle$ and $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma', \pi', e_2 \rangle \xrightarrow{e} \langle \sigma'', \pi'', v_2 \rangle$.
       - Either $v_1 \oplus_\pi v_2 = \mathsf{err}$. By rule e-op-err$_3$ $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (e_1 \oplus e_2) \rangle \xrightarrow{e} \mathsf{err}$. Hence 1b.
       - Or $v_1 \oplus_\pi v_2 = v$. By rule e-op $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (e_1 \oplus e_2) \rangle \xrightarrow{e} \langle \sigma', \pi, v \rangle$. Hence 1a.
         By induction $\Sigma; \Psi'' \vdash v_1 : t$ and $\Sigma; \Psi'' \vdash v_2 : t$. If $t = \mathsf{short}$ then $v = \bot$ or $v = n_s$ where $n \in [-2^{15}, 2^{15} - 1]$. If $v = \bot$ then $\Sigma; \Psi'' \vdash v : \mathsf{short}$. by rule vt-bot. Otherwise if $v = n_s$ then $\Sigma; \Psi'' \vdash v : \mathsf{short}$ by rule vt-s. An analgous argument holds if $t = \mathsf{long}$ hence 2c. Also 2b trivially by induction.
  9. Let $e : \theta = (e_1 \oplus e_2) : \tau[]*$. Similar to previous case.

10. Let $e : \theta = f(\vec{e}) : \theta_j$. By rule t-call $\Gamma_c; \Sigma \vdash e_i : \theta'_i$ where $\phi_c(f) = f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta''}, l, \lambda_c, j\rangle$ and $\Sigma \vdash \vec{\theta'} <: \vec{\theta}$. With respect to e_i there are two possibilities:
    - Either for some i: $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma_{i-1}, \pi_{i-1}, e_i\rangle \xrightarrow{e} \mathsf{err}$. Then by rule e-call-err it follows that 1b holds.
    - Or for all i: $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma_{i-1}, \pi_{i-1}, e_i\rangle \xrightarrow{e} \langle \sigma_i, \pi_i, v_i\rangle$ and by the inductive hypothesis $\Sigma; \Psi_i \vdash \theta_i : v_i$ and $\Sigma; \Psi_i \vdash \sigma_i; \pi_i$. Let $\Psi' = \Psi_n \cup \{\overrightarrow{a : \theta}, \overrightarrow{a' : \theta'}\}$. Then it is easy to verify $\Sigma; \Psi' \vdash \sigma'; \pi_n$ and $\Gamma_c; \Sigma; \Psi' \vdash \rho'$. By the progress induction hypothesis we then have for s:
        – Either $\Sigma; \lambda_c; \vec{\rho}, \rho; \rho' \vdash \langle \sigma', \pi_n, \lambda_c(l)\rangle \xrightarrow{s}{}^{*} \langle \sigma'', \pi', \mathsf{return}\rangle$. Hence $1a$.
        – Otherwise 1b.

        Preservation follows from the induction hyptheses for all e_i and s.

- By case analysis on $\Gamma_c; \Sigma \vdash s$ in Fig. 4. To show that either 1b or conversely 1a, 2a, 2b and 2c of Proposition 10 hold. Observe that 2a holds if $\Psi' \supseteq \Psi$.
    1. Let $\Gamma_c; \Sigma \vdash (\ell := e); s$. From the induction hypothesis for ℓ, either $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell\rangle \xrightarrow{\ell} \mathsf{err}$, and hence 1b, or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell\rangle \xrightarrow{\ell} \langle \sigma', \pi', a\rangle$. In the latter case, we have either $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma', \pi', e\rangle \xrightarrow{e} \mathsf{err}$, and hence 1b, or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma', \pi', e\rangle \xrightarrow{e} \langle \sigma'', \pi'', v\rangle$. By s-assn we then have $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (\ell := e); s\rangle \xrightarrow{s} \langle \sigma''', \pi'', s\rangle$ where $\sigma''' = \sigma'' \circ \{a \mapsto v\}$ and hence 1a.
       We get $\Gamma_c; \Sigma \vdash s$ from t-assn. Hence 2c. From the induction hypotheses for ℓ and e we get type preservations $\Sigma; \Psi'' \vdash a : \theta_1*$ and $\Sigma; \Psi'' \vdash v : \theta_2$ and type consistency $\Sigma; \Psi'' \vdash \sigma''; \pi''$. Hence, through rule vt-addr we know that $(a : \theta_1) \in \Psi''$. From rule t-assn we know $\Sigma \vdash \theta_2 <: \theta_1$. Hence, through rule vt-subt we have $\Sigma; \Psi'' \vdash v : \theta_1$. Since $\sigma'''(a) = v$ we have hence by rule st-comp $\Sigma; \Psi''; \sigma'''; \pi'' \vdash a : \theta_1$. Hence $\Sigma; \Psi'' \vdash \sigma'''; \pi''$. Thus 2b.
    2. Let $\Gamma_c; \Sigma \vdash (\mathsf{if}\ e\ \mathsf{goto}\ l); s$. Then
        - Either $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e\rangle \xrightarrow{e} \mathsf{err}$. Hence 1b.
        - Or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e\rangle \xrightarrow{e} \langle \sigma', \pi', v\rangle$. Then
            – Either $v = \bot$. Hence 1b.
            – Or $v = 0$. Then by rule s-if-false $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (\mathsf{if}\ e\ \mathsf{goto}\ l); s\rangle \xrightarrow{s} \langle \sigma', \pi s,'\rangle$. Hence 1a. We call this scenario 1.
            – Or $v \neq 0 \wedge v \neq \bot$. Then
                • Either $l \notin dom(\lambda_c)$. Then 1b.
                • Or $s' = \lambda_c(l)$. Then by rule s-if-true $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (\mathsf{if}\ e\ \mathsf{goto}\ l); s\rangle \xrightarrow{s} \langle \sigma', \pi s',\rangle$. Hence 1a. We call this scenario 2.

       In scenario 1 we have from t-if $\Gamma_c; \Sigma \vdash s$. Hence 2c. In scenario 2 we have that $s' \in range(\lambda_c)$. Hence $\Gamma_c; \Sigma \vdash s'$. Hence 2c. In both scenarios we have from the induction hyptheses for e that $\Sigma; \Psi' \vdash \sigma'; \pi'$. Hence 2b.
    3. Let $\Gamma_c; \Sigma \vdash \mathsf{goto}\ l$. Then either $l \notin dom(\lambda_c)$ and thus $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{goto}\ l\rangle \xrightarrow{s} \mathsf{err}$. Hence 1b. Alternatively $\lambda_c(l) = s$. Then by rule s-goto $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{goto}\ l\rangle \xrightarrow{s} \langle \sigma, \pi, s\rangle$. Hence 1a.
       From $\Gamma_c; \Sigma \vdash \lambda_c$ it follows that $\Gamma_c; \Sigma \vdash s$. Hence 2c. Let $\Psi' = \Psi$. Then 2b.
    4. Let $\Gamma_c; \Sigma \vdash \mathsf{return}$. Hence 1c. Also vacuously 2c and 2b.

- Proposition 11 follows by the repeated application of Proposition 10 combining progress and preservation at every step.
  Besides the givens of Proposition **??**, Proposition 10 also requires $\Gamma_c; \Sigma \vdash \lambda_c$. This is given by rule t-def which is the only possible way that the well-typing of the function definition could have been constructed.

A.2 Well-Typed Decompilation

Proposition 12 (well-typed instruction decompilation)**.** If $\mu_\Gamma; \Gamma_c; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e$ then for some θ_1 and θ_2

1. $\Gamma_c; \Sigma \vdash \ell : \theta_1$
2. $\Gamma_c; \Sigma \vdash e : \theta_2$
3. $\Sigma \vdash \theta_2 <: \theta_1$

Proof 2. The proof proceeds by case analysis on the inference rules of the instruction translation relation.

1. Case tr-$\oplus$-r*$_1$. Let $\theta_1 = \theta_2 = \theta[]*$. From tr-$\oplus$-r*$_1$ we have $(x : \theta[]*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \theta[]*$. Hence 1. From tr-$\oplus$-r*$_1$ we have $\Gamma_c; \Sigma \vdash m : \mathsf{long}$. From tr-$\oplus$-r*$_1$ we have $(y : \mathsf{long}) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash y : \mathsf{long}$. From both of these we get by rule t-$\otimes$ $\Gamma_c; \Sigma \vdash y * m : \mathsf{long}$. From that and the type of x we get through rule t-ptr-$\oplus$ $\Gamma_c; \Sigma \vdash x \oplus (y * m) : \theta[]*$. Hence 2. From rule sub-refl 3.
2. Case tr-$\oplus$-r*$_2$. Let $\theta_1 = \theta_2 = t$. From tr-$\oplus$-r*$_2$ we have $(x : t) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : t$. Hence 1. From tr-$\oplus$-r*$_2$ we have $\Gamma_c; \Sigma \vdash c : t$. From tr-$\oplus$-r*$_1$ we have $(y : t) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash y : t$. From both of these we get by rule t-$\otimes$ $\Gamma_c; \Sigma \vdash y * c : t$. From that and the type of x we get through rule t-$\otimes$ $\Gamma_c; \Sigma \vdash x \oplus (y * c) : t$. Hence 2. From rule sub-refl 3.
3. Case tr-$\otimes$-rc. Let $\theta_1 = \theta_2 = t$. From tr-$\otimes$-rc we have $(x : t) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : t$. Hence 1. From tr-$\otimes$-rc we have $\Gamma_c; \Sigma \vdash c : t$. From that and the previous $\Gamma_c; \Sigma \vdash x : t$ we have by rule t-$\otimes$ $\Gamma_c; \Sigma \vdash x \otimes c : t$. Hence 2. From rule sub-refl 3.
4. Case tr-$\otimes$-rr. Let $\theta_1 = \theta_2 = t$. From tr-$\otimes$-rr we have $(x : t) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : t$. Hence 1. From tr-$\otimes$-rr we have $(y : t) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash y : t$. From that and the previous $\Gamma_c; \Sigma \vdash x : t$ we have by rule t-$\otimes$ $\Gamma_c; \Sigma \vdash x \otimes y : t$. Hence 2. From rule sub-refl 3.
5. Case tr-$\oplus$-rc. Let $\theta_1 = \theta_2 = \theta[]*$. From tr-$\oplus$-rc we have $(x : \theta[]*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \theta[]*$. Hence 1. From tr-$\oplus$-rc we have $\Gamma_c; \Sigma \vdash m : t$. From that and the previous $\Gamma_c; \Sigma \vdash x : \theta[]*$ we have by rule t-ptr-$\oplus$ $\Gamma_c; \Sigma \vdash x \oplus m : \theta[]*$. Hence 2. From rule sub-refl 3.
6. Case tr-mov-rc. Let $\theta_1 = \theta_2 = t$. From tr-mov-rc we have $(x : t) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : t$. Hence 1. From tr-mov-rc we have $\Gamma_c; \Sigma \vdash c : t$. Hence 2. From rule sub-refl 3.
7. Case tr-mov-r0. Let $\theta_1 = \theta_2 = \tau*$. From tr-mov-r0 we have $(x : \tau*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \tau*$. Hence 1. From t-null we have $\Gamma_c; \Sigma \vdash 0 : \tau*$. Hence 2. From rule sub-refl 3.
8. Case tr-mov-rr. From tr-mov-rr we have $(x : \theta_1) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \theta_1$. Hence 1. From tr-mov-rr we have $(y : \theta_2) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash y : \theta_2$. Hence 2. From tr-mov-rr we have $\Sigma \vdash \theta_2 <: \theta_1$. Hence 3.
9. Case tr-mov-ri$_1$. From tr-mov-ri$_1$ we have $(x : \theta_1) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \theta_1$. Hence 1. From tr-mov-ri$_1$ we have $(y : \theta_2*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash y : \theta_2*$. Then by rule t-ptr $\Gamma_c; \Sigma \vdash *y : \theta_2$. Hence 2. From tr-mov-ri$_1$ we have $\Sigma \vdash \theta_2 <: \theta_1$. Hence 3.
10. Case tr-mov-ir$_1$. From tr-mov-ir$_1$ we have $(x : \theta_1*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \theta_1*$. Then by rule t-ptr $\Gamma_c; \Sigma \vdash *x : \theta_1$. Hence 1. From tr-mov-ir$_1$ we have $(y : \theta_2) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash y : \theta_2$. Hence 1. From tr-mov-ir$_1$ we have $\Sigma \vdash \theta_2 <: \theta_1$. Hence 3.
11. Case tr-mov-ri$_2$. From tr-mov-ri$_2$ we have $(x : \theta_1) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \theta_1$. Hence 1. From tr-mov-ri$_2$ we have $(y : \theta_2[]*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash y : \theta_2[]*$. Also by rule t-l$\Gamma_c; \Sigma \vdash 0 : \mathsf{long}$. Then by rule t-ar$\Gamma_c; \Sigma \vdash y[0] : \theta_2$. Hence 2. From tr-mov-ri$_2$ we have $\Sigma \vdash \theta_2 <: \theta_1$. Hence 3.
12. Case tr-mov-ir$_2$. From tr-mov-ir$_2$ we have $(x : \theta_1[]*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \theta_1[]*$. Also by rule t-l$\Gamma_c; \Sigma \vdash 0 : \mathsf{long}$. Then by rule t-ar $\Gamma_c; \Sigma \vdash x[0] : \theta_1$. Hence 1. From tr-mov-ir$_2$ we have $(y : \theta_2) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash y : \theta_2$. Hence 2. From tr-mov-ir$_2$ we have $\Sigma \vdash \theta_2 <: \theta_1$. Hence 3.
13. Case tr-mov-ri$_3$. From tr-mov-ri$_3$ we have $(x : \theta) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \theta$. Hence 1. From tr-mov-ri$_3$ we have $(y : N*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash y : N*$. Then by rule t-fld$\Gamma_c; \Sigma \vdash y \to 0 : \theta_0$. Hence 2. From tr-mov-ri$_3$ we have $\Sigma \vdash \theta_0 <: \theta$. Hence 3.
14. Case tr-mov-ir$_3$. From tr-mov-ir$_3$ we have $(x : N*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : N*$. Then by rule t-fld$\Gamma_c; \Sigma \vdash x \to 0 : \theta_0$. Hence 1. From tr-mov-ir$_3$ we have $(y : \theta) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash y : \theta$. Hence 2. From tr-mov-ir$_3$ we have $\Sigma \vdash \theta <: \theta_0$. Hence 3.
15. Case tr-mov-ri+$_1$. From tr-mov-ri+$_1$ we have $(x : \theta_1) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \theta_1$. Hence 1. From tr-mov-ri+$_1$ we have $(y : \theta_2[]*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash y : \theta_2[]*$. Also from tr-mov-ri+$_1$ we have $\Gamma_c; \Sigma \vdash m : t$. Then by rule t-ar$\Gamma_c; \Sigma \vdash y[m] : \theta_2$. Hence 2. From tr-mov-ri+$_1$ we have $\Sigma \vdash \theta_2 <: \theta_1$. Hence 3.
16. Case tr-mov-i+r$_1$. From tr-mov-i+r$_1$ we have $(x : \theta_1[]*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \theta_1[]*$. Also from tr-mov-i+r$_1$ we have $\Gamma_c; \Sigma \vdash m : t$. Then by rule t-ar$\Gamma_c; \Sigma \vdash x[m] : \theta_1$. Hence 1. From tr-mov-i+r$_1$ we have $(y : \theta_2) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash y : \theta_2$. Hence 2. From tr-mov-i+r$_1$ we have $\Sigma \vdash \theta_2 <: \theta_1$. Hence 3.
17. Case tr-mov-ri+$_2$. From tr-mov-ri+$_2$ we have $(x : \theta) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \theta$. Hence 1. From tr-mov-ri+$_2$ we have $(y : N*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash y : N*$. Then by rule t-fld$\Gamma_c; \Sigma \vdash y \to m : \theta_m$. Hence 2. From tr-mov-ri+$_2$ we have $\Sigma \vdash \theta_m <: \theta$. Hence 3.
18. Case tr-mov-i+r$_2$. From tr-mov-i+r$_2$ we have $(x : N*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : N*$. Then by rule t-fld$\Gamma_c; \Sigma \vdash x \to m : \theta_m$. Hence 1. From tr-mov-i+r$_2$ we have $(y : \theta_2) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash y : \theta_2$. Hence 2. From tr-mov-i+r$_2$ we have $\Sigma \vdash \theta <: \theta_m$. Hence 3.
19. Case tr-alloc-r*. From tr-alloc-r* we have $(x : \theta[]*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \theta[]*$. Hence 1. From tr-alloc-r* we have $\Gamma_c; \Sigma \vdash m : t$. From tr-alloc-r* we have $(y : t) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash y : t$. From both of these we get by rule t-$\otimes$ $\Gamma_c; \Sigma \vdash y * m : t$. Then from t-new-ar we get $\Gamma_c; \Sigma \vdash \mathsf{new}\ \theta[y * m] : \theta[]*$. Hence 2. From rule sub-refl 3.
20. Case tr-alloc-rc$_1$. From tr-alloc-rc$_1$ we have $(x : \theta*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \theta*$. Hence 1. From t-new we get $\Gamma_c; \Sigma \vdash \mathsf{new}\ \theta : \theta*$. Hence 2. From rule sub-refl 3.
21. Case tr-alloc-rc$_2$. From tr-alloc-rc$_2$ we have $(x : N*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : N*$. Hence 1. From t-new-str we get $\Gamma_c; \Sigma \vdash \mathsf{new}\ N : N*$. Hence 2. From rule sub-refl 3.
22. Case tr-alloc-rc$_3$. From tr-alloc-rc$_3$ we have $(x : \theta[]*) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \theta[]*$. Hence 1. From tr-alloc-rc$_3$ we have $\Gamma_c; \Sigma \vdash m : t$. Then from rule t-new-ar we have $\Gamma_c; \Sigma \vdash \mathsf{new}\ \theta[m] : \theta[]*$. Hence 2. From rule sub-refl 3.
23. Case tr-call. From tr-call we have $(u : \theta_u) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash u : \theta_u$. Hence 1. We have:
    - From tr-call we have $\phi_c(f) = f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta'}, l, \lambda_c, j\rangle$.
    - From tr-call we have $\overrightarrow{(v : \theta_v)} \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash \vec{v} : \vec{\theta}_v$.
    - From tr-call we have $\Sigma \vdash \vec{\theta}_v <: \vec{\theta}$.
    - By rule sub-reflwe have $\Sigma \vdash \theta'_j <: \theta'_j$.

Hence by rule t-call we have $\Gamma_c; \Sigma \vdash: \theta'_j$. Hence 2. From tr-call we have $\Sigma \vdash \theta'_j <: \theta_u$. Hence 3.

Proposition 13 (well-typed block decompilation)**.** If $\mu_\lambda; \mu_\Gamma; \Gamma_c; \Sigma \vdash b \overset{b}{\rightsquigarrow} s$ then $\Gamma_c; \Sigma \vdash s$.

Proof 3. This proof proceeds by structural induction on the block translation relation.

1. Case tr-instr. From tr-instrwe have $\mu_\Gamma; \Gamma_c; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e$. Hence, by Proposition 12 we have$\Gamma_c; \Sigma \vdash \ell : \theta_1$, $\Gamma_c; \Sigma \vdash e : \theta_2$ and $\Sigma \vdash \theta_2 <: \theta_1$. Also by rule tr-instr we have $\mu_\lambda; \mu_\Gamma; \Gamma_c; \Sigma \vdash b \overset{b}{\rightsquigarrow} s$. Hence by the induction hypothesis we have $\Gamma_c; \Sigma \vdash s$. Then by rule t-assn we have $\Gamma_c; \Sigma \vdash \ell := e; s$
2. Case tr-if. From tr-if we have $(x : \theta) \in \Gamma_c$. Then by rule t-var $\Gamma_c; \Sigma \vdash x : \theta_u$. Also from tr-if we have $\mu_\lambda; \mu_\Gamma; \Gamma_c; \Sigma \vdash b \overset{b}{\rightsquigarrow} s$. Hence, from the induction hypothesis we have $\Gamma_c; \Sigma \vdash s$ Then the proposition follows from rule t-if.
3. Case tr-goto. This follows from rule t-goto.
4. Case tr-ret. This follows from rule t-ret.

Proposition 14 (well-typed definition decompilation)**.** If $\Sigma \vdash d_x \rightsquigarrow d_c$ then $\Sigma \vdash d_c$.

Proof 4. We show that the four preconditions to rule t-def are satisfied:

1. From rule tr-def we know that $\Gamma_c = \{\overrightarrow{x : \theta}, \overrightarrow{y : \theta'}\}$.
2. From rule tr-def we know that $a \in dom(\lambda_c)$ and $l = \mu_\lambda(a)$. Hence $l \in range(\mu_\lambda)$. From the rule we also know that $range(\mu_\lambda) = dom(\lambda_c)$. Hence $l \in dom(\lambda_c)$.
3. From rule tr-def we know that $r_{y_j} \in \overrightarrow{r_y}$. We also know that $y_j = \mu_\Gamma(r_{y_j})$ and that $\overrightarrow{y} = \mu_\Gamma(\overrightarrow{r_y})$. Hence $y_j \in \overrightarrow{y}$.
4. From rule tr-def we know that $\forall (a \mapsto l) \in \mu_\lambda : \ \mu_\lambda; \mu_\Gamma; \Gamma_c; \Sigma \vdash \lambda_x(a) \overset{b}{\rightsquigarrow} \lambda_c(l)$. From Proposition 13 we then know that $\forall l \in range(\mu_\lambda) : \ \Gamma_c; \Sigma \vdash \lambda_c(l)$. From rule tr-def we know that $range(\mu_\lambda) = dom(\lambda_c)$. Hence $\forall l \in dom(\lambda_c) : \ \Gamma_c; \Sigma \vdash \lambda_c(l)$.

Hence by rule t-def we conclude $\Sigma \vdash f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta'}, l, \lambda_c, j\rangle$.

A.3 Semantics Preservation

A.3.1 Semantics Preservation

Instructions We prove Propositions 7 and 6 together.

Proof 5. The proof proceeds by case analysis on the derivation of the judgement $\mu_\Gamma; \Gamma_c; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e$.

1. Case tr-$\oplus$-r*$_1$. Then $\iota = (\mathsf{op}_4^\oplus \ r_i, r_j * c)$, $\ell = x$ and $e = x \oplus (y * m)$.
   (a) This case is not possible. Rule ex-$\oplus$-r* always applies.
   (b) In this case rules ex-$\oplus$-r* is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{op}_4^\oplus \ r_i, r_j * c\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_4 \{r_i \mapsto \vec{b}_i \oplus_4 (\vec{b}_j *_4 c)\}$ where $\vec{b}_i = R_{0:4}(r_i)$ and $\vec{b}_j = R_{0:4}(r_j)$.
   Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a\rangle$ with $a = \rho(x)$. Also through rules e-op, e-lval, l-var and e-const we obtain $\Sigma; \vec{\rho}; \rho \ \vdash \ \langle \sigma, \pi, (x \oplus (y * m))\rangle \xrightarrow{e} \langle \sigma, \pi, v\rangle$ where $v = v_x \oplus_\pi (v_y *_\pi m)$, $v_x = \sigma(a)$, $a' = \rho(y)$ and $v_y = \sigma(a')$.
   From rule tr-$\oplus$-r*$_1$ we know $(r_i : x)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_i \leftrightsquigarrow v_x$. Similarly, we know $\mu_a \vdash \vec{b}_j \leftrightsquigarrow v_y$. Then from $(x : \theta[]*) \in \Gamma_c$ and the store typing of σ it follows that $v_x = n_*$ and from the success of the addition, it also follows that $[n_*, n_* \oplus (v_y * m)] \subseteq\in \pi$. Hence, also from the store typing all m values at the addresses in this range have type θ. From the related heaps it then follows with $c/m = sizeof(\theta)$ that $\mu_a \vdash (\vec{b}_i \oplus_4 (\vec{b}_j *_4 c)) \leftrightsquigarrow (v \oplus_\pi (v_y * m))$. Hence, the update registers are still related.
2. Case tr-$\oplus$-r*$_2$. Then $\iota = (\mathsf{op}_w^\oplus \ r_i, r_j * c)$, $\ell = x$ and $e = x \oplus (y * c)$.
   (a) This case is not possible. Rule ex-$\oplus$-r* always applies.
   (b) In this case rules ex-$\oplus$-r* is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{op}_w^\oplus \ r_i, r_j * c\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}_i \oplus_w (\vec{b}_j *_w c)\}$ where $\vec{b}_i = R_{0:w}(r_i)$ and $\vec{b}_j = R_{0:w}(r_j)$.
   Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a\rangle$ with $a = \rho(x)$. Also through rules e-op, e-lval, l-var and e-const we obtain $\Sigma; \vec{\rho}; \rho \ \vdash \ \langle \sigma, \pi, (x \oplus (y * m))\rangle \xrightarrow{e} \langle \sigma, \pi, v\rangle$ where $v = v_x \oplus_\pi (v_y *_\pi m)$, $v_x = \sigma(a)$, $a' = \rho(y)$ and $v_y = \sigma(a')$.
   From rule tr-$\oplus$-r*$_2$ we know $(r_i : x)_w \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_i \leftrightsquigarrow v_x$. Similarly, we know $\mu_a \vdash \vec{b}_j \leftrightsquigarrow v_y$. It then follows that $\mu_a \vdash (\vec{b}_i \oplus_w (\vec{b}_j *_w c)) \leftrightsquigarrow (v \oplus_\pi (v_y * c))$. Hence, the update registers are still related.
3. Case tr-$\otimes$-rc. Then $\iota = (\mathsf{op}_w^\otimes \ r_i, c)$, $\ell = x$ and $e = x \otimes c$.
   (a) This case is not possible. Rule ex-$\otimes$-rc always applies.
   (b) In this case rules ex-$\otimes$-rc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{op}_w^\otimes \ r_i, c\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b} \otimes_w c\}$ where $\vec{b} = R_{0:w}(r_i)$.
   Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a\rangle$ with $a = \rho(x)$. Also through rules e-op, e-lval, l-var and e-const we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (x \otimes c)\rangle \xrightarrow{e} \langle \sigma, \pi, v'\rangle$ where $v' = v \otimes_\pi c$ and $v = \sigma(a)$.
   From rule tr-$\otimes$-rc we know $(r_i : x)_w \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b} \leftrightsquigarrow v$. Then from $(x : t) \in \Gamma_c$ and $w = sizeof(t)$ it follows that $\mu_a \vdash (\vec{b} \otimes_w c) \leftrightsquigarrow (v \otimes_\pi c)$. Hence, the update registers are still related.
4. Case tr-$\oplus$-rc. Then $\iota = (\mathsf{op}_4^\oplus \ r_i, c)$, $\ell = x$ and $e = x \oplus m$.
   (a) This case is not possible. Rule ex-$\otimes$-rc always applies.
   (b) In this case rules ex-$\otimes$-rc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{op}_4^\oplus \ r_i, c\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_4 \{r_i \mapsto \vec{b} \oplus_4 c\}$ where $\vec{b} = R_{0:4}(r_i)$.
   Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a\rangle$ with $a = \rho(x)$. Also through rules e-op, e-lval, l-var and e-const we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (x \oplus m)\rangle \xrightarrow{e} \langle \sigma, \pi, v'\rangle$ where $v' = v \oplus_\pi m$ and $v = \sigma(a)$.
   From rule tr-$\oplus$-rc we know $(r_i : x)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b} \leftrightsquigarrow v$. Then from $(x : \theta[]*) \in \Gamma_c$ and the store typing of σ it follows that $v = n_*$ and from the success of the addition, it also follows that $[n_*, n_* \oplus m] \subseteq\in \pi$. Hence, also from the store typing all m values at the addresses in this range have type θ. From the related heaps it then follows with $c/m = sizeof(\theta)$ that $\mu_a \vdash (\vec{b} \oplus_4 c) \leftrightsquigarrow (v \oplus_\pi m)$. Hence, the update registers are still related.
5. Case tr-$\otimes$-rr. Then $\iota = (\mathsf{op}_w^\otimes \ r_i, r_j)$, $\ell = x$ and $e = x \otimes y$.
   (a) This case is not possible. Rule ex-$\otimes$-rr always applies.
   (b) In this case rules ex-$\otimes$-rc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{op}_w^\otimes \ r_i, r_j\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}_i \oplus_w \vec{b}_j\}$ where $\vec{b}_i = R_{0:w}(r_i)$ and $\vec{b}_j = R_{0:w}(r_j)$.
   Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a\rangle$ with $a = \rho(x)$. Also through rules e-op, e-lval and l-var

we obtain $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,(x \otimes y)\rangle \xrightarrow{e} \langle\sigma,\pi,v\rangle$ where $v = v_x \otimes_\pi v_y$, $v_x = \sigma(a)$, $a' = \rho(y)$ and $v_y = \sigma(a')$.
From rule tr-$\otimes$-rr we know $(r_i : x)_w \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_i \leftrightsquigarrow v_x$. By similar reasoning we know $\mu_a \vdash \vec{b}_j \leftrightsquigarrow v_y$. Then from $(x : t) \in \Gamma_c$, $(y : t) \in \Gamma_c$ and $w = \mathit{sizeof}(t)$ it follows that $\mu_a \vdash (\vec{b}_i \otimes_w \vec{b}_j) \leftrightsquigarrow (v_x \otimes_\pi v_y)$. Hence, the update registers are still related.

6. Case tr-mov-rc. Then $\iota = (\mathsf{mov}_w\ r_i, c)$, $\ell = x$ and $e = c$.
   (a) This case is not possible. Rule ex-mov-rc always applies.
   (b) In this case rules ex-mov-rc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, c\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_w \{r_i \mapsto c\}$.
   Similarly, through rule l-var $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,x\rangle \xrightarrow{\ell} \langle\sigma,\pi,a\rangle$ with $a = \rho(x)$. Also through rule e-const we obtain $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,c\rangle \xrightarrow{e} \langle\sigma,\pi,c\rangle$.
   We know that $\mu_a \vdash c \leftrightsquigarrow c$. Hence, the update registers are still related.
7. Case tr-mov-r0. Then $\iota = (\mathsf{mov}_4\ r_i, 0)$, $\ell = x$ and $e = 0$.
   (a) This case is not possible. Rule ex-mov-rc always applies.
   (b) In this case rules ex-mov-rc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_4\ r_i, 0\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_4 \{r_i \mapsto 0\}$.
   Similarly, through rule l-var $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,x\rangle \xrightarrow{\ell} \langle\sigma,\pi,a\rangle$ with $a = \rho(x)$. Also through rule e-const we obtain $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,0\rangle \xrightarrow{e} \langle\sigma,\pi,0\rangle$.
   We know that $\mu_a \vdash 0 \leftrightsquigarrow 0$. Hence, the update registers are still related.
8. Case tr-mov-rr. Then $\iota = (\mathsf{mov}_w\ r_i, r_j)$, $\ell = x$ and $e = y$.
   (a) This case is not possible. Rule ex-mov-rr always applies.
   (b) In this case rules ex-mov-rr is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, r_j\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}\}$ where $\vec{b} = R_{0:w}(r_j)$.
   Similarly, through rule l-var $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,x\rangle \xrightarrow{\ell} \langle\sigma,\pi,a\rangle$ with $a = \rho(x)$. Also through rules e-lval and l-var we obtain $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,y\rangle \xrightarrow{e} \langle\sigma,\pi,v\rangle$ where $v = \sigma(a')$ and $a' = \rho(y)$.
   From rule tr-mov-rr we know $(r_j : y)_w \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b} \leftrightsquigarrow v$. Also from rule tr-mov-rr we know $(r_i : x)_w \in \mu_\Gamma$. Hence, the registers are related. After the update we can see that they are still related.
9. Case tr-mov-ri$_1$. Then $\iota = (\mathsf{mov}_w\ r_i, [r_j])$, $\ell = x$ and $e = *y$.
   (a) This case is possible iff $R(r_j) = 0$ or $R(r_j) = \bot$. Because of the related registers and, from rule tr-mov-ri$_1$, $(r_j : y)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_j) \leftrightsquigarrow \sigma(\rho(y))$. In either of the cases for $R(r_j)$ we also have $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,y\rangle \xrightarrow{e} \mathsf{err}$.
   (b) In this case rules ex-mov-ri is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, [r_j]\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}_2\}$ where $\vec{b}_2 = H^w(\vec{b}_1)$ and $\vec{b}_1 = R_(r_j)$.
   Similarly, through rule l-var $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,x\rangle \xrightarrow{\ell} \langle\sigma,\pi,a\rangle$ with $a = \rho(x)$. Also through rule l-ptr and l-var we obtain $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,*y\rangle \xrightarrow{e} \langle\sigma,\pi,v_2\rangle$ where $v_2 = \sigma(v_1)$, $v_1 = \sigma(a')$ and $a' = \rho(y)$.
   From rule tr-mov-ri$_1$ we know $(r_j : y)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_1 \leftrightsquigarrow v_1$. From related stores, we also know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. Also from rule tr-mov-ri$_1$ we know $(r_i : x)_w \in \mu_\Gamma$. Hence, the registers are related. After the update we can see that they are still related.
10. Case tr-mov-ri$_2$. Then $\iota = (\mathsf{mov}_w\ r_i, [r_j])$, $\ell = x$ and $e = y[0]$.
    (a) This case is possible iff $R(r_j) = 0$ or $R(r_j) = \bot$. Because of the related registers and, from rule tr-mov-ri$_2$, $(r_j : y)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_j) \leftrightsquigarrow \sigma(\rho(y))$. In either of the cases for $R(r_j)$ we also have $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,y\rangle \xrightarrow{e} \mathsf{err}$.
    (b) In this case rules ex-mov-ri is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, [r_j]\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}_2\}$ where $\vec{b}_2 = H^w(\vec{b}_1)$ and $\vec{b}_1 = R_(r_j)$.
    Similarly, through rule l-var $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,x\rangle \xrightarrow{\ell} \langle\sigma,\pi,a\rangle$ with $a = \rho(x)$. Also through rules e-lval, l-ar and e-const we obtain $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,y[0]\rangle \xrightarrow{e} \langle\sigma,\pi,v_2\rangle$ where $v_2 = \sigma(v_1)$, $v_1 = \sigma(a')$ and $a' = \rho(y)$.
    From rule tr-mov-ri$_2$ we know $(r_j : y)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_1 \leftrightsquigarrow v_1$. From related stores, we also know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. Also from rule tr-mov-ri$_2$ we know $(r_i : x)_w \in \mu_\Gamma$. Hence, the registers are related. After the update we can see that they are still related.
11. Case tr-mov-ri$_3$. Then $\iota = (\mathsf{mov}_w\ r_i, [r_j])$, $\ell = x$ and $e = y \to 0$.
    (a) This case is possible iff $R(r_j) = 0$ or $R(r_j) = \bot$. Because of the related registers and, from rule tr-mov-ri$_3$, $(r_j : y)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_j) \leftrightsquigarrow \sigma(\rho(y))$. In either of the cases for $R(r_j)$ we also have $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,y\rangle \xrightarrow{e} \mathsf{err}$.
    (b) In this case rules ex-mov-ri is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, [r_j]\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}_2\}$ where $\vec{b}_2 = H^w(\vec{b}_1)$ and $\vec{b}_1 = R_(r_j)$.
    Similarly, through rule l-var $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,x\rangle \xrightarrow{\ell} \langle\sigma,\pi,a\rangle$ with $a = \rho(x)$. Also through rules e-lval and l-fldwe obtain $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,y \to 0\rangle \xrightarrow{e} \langle\sigma,\pi,v_2\rangle$ where $v_2 = \sigma(v_1)$, $v_1 = \sigma(a')$ and $a' = \rho(y)$.
    From rule tr-mov-ri$_3$ we know $(r_j : y)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_1 \leftrightsquigarrow v_1$. From related stores, we also know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. Also from rule tr-mov-ri$_3$ we know $(r_i : x)_w \in \mu_\Gamma$. Hence, the registers are related. After the update we can see that they are still related.
12. Case tr-mov-ir$_1$. Then $\iota = (\mathsf{mov}_w\ [r_i], r_j)$, $\ell = *x$ and $e = y$.
    (a) This case is possible iff $R(r_i) = 0$ or $R(r_i) = \bot$. Because of the related registers and, from rule tr-mov-ir$_1$, $(r_i : x)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_i) \leftrightsquigarrow \sigma(\rho(x))$. In either of the cases for $R(r_i)$ we also have $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,x\rangle \xrightarrow{\ell} \mathsf{err}$.
    (b) In this case rules ex-mov-ir is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ [r_i], r_j\rangle \xrightarrow{\iota} \langle H', R\rangle$. Here $H' = H \circ \{\vec{b}_1, \ldots, \vec{b}_1 + (w-1) \mapsto \vec{b}_2\}$ where $\vec{b}_1 = R(r_i)$ and $\vec{b} = R_{0:w}(r_j)$.
    Similarly, through rule l-ptr $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,*x\rangle \xrightarrow{\ell} \langle\sigma,\pi,v_1\rangle$ with $v_1 = \sigma(a)$ and $a = \rho(x)$. Also through rules e-lval and l-varwe obtain $\Sigma;\vec{\rho};\rho \vdash \langle\sigma,\pi,y\rangle \xrightarrow{e} \langle\sigma,\pi,v_2\rangle$ where $v_2 = \sigma(a')$ and $a' = \rho(y)$.
    From rule tr-mov-ir$_1$ we know $(r_j : y)_w \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. From related stores, we also know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. Also from rule tr-mov-ir$_1$ we know $(r_i : x)_w \in \mu_\Gamma$. Hence, $\mu_a \vdash \vec{b}_1 \leftrightsquigarrow v_1$. Since $(x : \theta_1*) \in \Gamma_c$, we know that v_1 is an address. Because of related heaps, we then know that $(\vec{b}_1, v_1) in \mu_a$. After the update we can see that they are still related.
13. Case tr-mov-ir$_2$. Then $\iota = (\mathsf{mov}_w\ [r_i], r_j)$, $\ell = x[0]$ and $e = y$.

(a) This case is possible iff $R(r_i) = 0$ or $R(r_i) = \bot$. Because of the related registers and, from rule tr-mov-ir$_2$, $(r_i : x)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_i) \leftrightsquigarrow \sigma(\rho(x))$. In either of the cases for $R(r_i)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \rangle \xrightarrow{\ell}$ err.

(b) In this case rules ex-mov-ir is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w \ [r_i], r_j \rangle \xrightarrow{\iota} \langle H', R \rangle$. Here $H' = H \circ \{\vec{b}_1, \ldots, \vec{b}_1 + (w-1) \mapsto \vec{b}_2\}$ where $\vec{b}_1 = R(r_i)$ and $\vec{b} = R_{0:w}(r_j)$.

Similarly, through rule l-ar and e-const $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x[0] \rangle \xrightarrow{\ell} \langle \sigma, \pi, v_1 \rangle$ with $v_1 = \sigma(a)$ and $a = \rho(x)$. Also through rules e-lval and l-varwe obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y \rangle \xrightarrow{e} \langle \sigma, \pi, v_2 \rangle$ where $v_2 = \sigma(a')$ and $a' = \rho(y)$.

From rule tr-mov-ir$_2$ we know $(r_j : y)_w \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. From related stores, we also know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. Also from rule tr-mov-ir$_2$ we know $(r_i : x)_w \in \mu_\Gamma$. Hence, $\mu_a \vdash \vec{b}_1 \leftrightsquigarrow v_1$. Since $(x : \theta_1[]*) \in \Gamma_c$, we know that v_1 is an address. Because of related heaps, we then know that $(\vec{b}_1, v_1) in \mu_a$. After the update we can see that they are still related.

14. Case tr-mov-ir$_3$. Then $\iota = (\mathsf{mov}_w \ [r_i], r_j)$, $\ell = x \to 0$ and $e = y$.

(a) This case is possible iff $R(r_i) = 0$ or $R(r_i) = \bot$. Because of the related registers and, from rule tr-mov-ir$_3$, $(r_i : x)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_i) \leftrightsquigarrow \sigma(\rho(x))$. In either of the cases for $R(r_i)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \rangle \xrightarrow{\ell}$ err.

(b) In this case rules ex-mov-ir is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w \ [r_i], r_j \rangle \xrightarrow{\iota} \langle H', R \rangle$. Here $H' = H \circ \{\vec{b}_1, \ldots, \vec{b}_1 + (w-1) \mapsto \vec{b}_2\}$ where $\vec{b}_1 = R(r_i)$ and $\vec{b} = R_{0:w}(r_j)$.

Similarly, through rule l-fld $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \to 0 \rangle \xrightarrow{\ell} \langle \sigma, \pi, v_1 \rangle$ with $v_1 = \sigma(a)$ and $a = \rho(x)$. Also through rules e-lval and l-varwe obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y \rangle \xrightarrow{e} \langle \sigma, \pi, v_2 \rangle$ where $v_2 = \sigma(a')$ and $a' = \rho(y)$.

From rule tr-mov-ir$_3$ we know $(r_j : y)_w \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. From related stores, we also know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. Also from rule tr-mov-ir$_3$ we know $(r_i : x)_w \in \mu_\Gamma$. Hence, $\mu_a \vdash \vec{b}_1 \leftrightsquigarrow v_1$. Since $(x : N*) \in \Gamma_c$, we know that v_1 is an address. Because of related heaps, we then know that $(\vec{b}_1, v_1) in \mu_a$. After the update we can see that they are still related.

15. Case tr-mov-ri+$_1$. Then $\iota = (\mathsf{mov}_w \ r_i, [r_j + c]$, $\ell = x$ and $e = y[m]$.

(a) This case is possible iff $R(r_j) = 0$, $R(r_j) = \bot$ or $(R(r_j) + c) \notin dom(H)$. Because of the related registers and heaps, and from rule tr-mov-ri+$_1(r_j : y)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_j) \leftrightsquigarrow \sigma(\rho(y))$. In either of the first two cases for $R(r_j)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y[m] \rangle \xrightarrow{\ell}$ err. In the last case, because of related heaps, it also has to be that $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y[m] \rangle \xrightarrow{\ell}$ err.

(b) In this case rules ex-mov-r+ is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w \ r_i, [r_j + c] \rangle \xrightarrow{\iota} \langle H, R' \rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}\}$ where $\vec{b} = H^w(\vec{b'})$ and $\vec{b} = R(r_j) +_4 c$.

Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \rangle \xrightarrow{\ell} \langle \sigma, \pi, a \rangle$ with $a = \rho(x)$. Also through rules e-lval, l-arand e-const we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y[m] \rangle \xrightarrow{e} \langle \sigma, \pi, v \rangle$ where $v = \sigma(a'' + m)$, $a'' = \sigma(a')$ and $a' = \rho(y)$.

From rule tr-mov-ri+$_1$ we know $(r_j : y)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b'} \leftrightsquigarrow a''$. From the translation rule we also have $(y : \theta[]*) \in \Gamma_c$. Because of the progress, it means that $[a'', a'' + m] \subseteq\in \pi$. Because of the related heaps and well-typed store it follows that $\mu_a \vdash \vec{b} \leftrightsquigarrow v$. Also from rule tr-mov-ri+$_1$ we know $(r_i : x)_w \in \mu_\Gamma$. After the update we can see that they are still related.

16. Case tr-mov-ri+$_2$. Then $\iota = (\mathsf{mov}_w \ r_i, [r_j + c]$, $\ell = x$ and $e = y \to m$.

(a) This case is possible iff $R(r_j) = 0$, $R(r_j) = \bot$ or $(R(r_j) + c) \notin dom(H)$. Because of the related registers and heaps, and from rule tr-mov-ri+$_2(r_j : y)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_j) \leftrightsquigarrow \sigma(\rho(y))$. In either of the first two cases for $R(r_j)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y[m] \rangle \xrightarrow{\ell}$ err. In the last case, because of related heaps, it also has to be that $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y \to m \rangle \xrightarrow{\ell}$ err.

(b) In this case rules ex-mov-r+ is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w \ r_i, [r_j + c] \rangle \xrightarrow{\iota} \langle H, R' \rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}\}$ where $\vec{b} = H^w(\vec{b'})$ and $\vec{b} = R(r_j) +_4 c$.

Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \rangle \xrightarrow{\ell} \langle \sigma, \pi, a \rangle$ with $a = \rho(x)$. Also through rules e-lval and l-fld we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y \to m \rangle \xrightarrow{e} \langle \sigma, \pi, v \rangle$ where $v = \sigma(a'' + m)$, $a'' = \sigma(a')$ and $a' = \rho(y)$.

From rule tr-mov-ri+$_2$ we know $(r_j : y)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b'} \leftrightsquigarrow a''$. From the translation rule we also have $(y : N*) \in \Gamma_c$ and $\Sigma(N) = \langle \theta_0, \ldots, \theta_n \rangle$. Because of the progress, it means that $[a'', a'' + m] \subseteq\in \pi$. Because of the related heaps and well-typed store it follows that $\mu_a \vdash \vec{b} \leftrightsquigarrow v$. Also from rule tr-mov-ri+$_1$ we know $(r_i : x)_w \in \mu_\Gamma$. After the update we can see that they are still related.

17. Case tr-mov-i+r$_1$. Then $\iota = (\mathsf{mov}_w \ [r_i + c], r_j$, $\ell = x[m]$ and $e = y$.

(a) This case is possible iff $R(r_i) = 0$, $R(r_i) = \bot$ or $(R(r_i) + c) \notin dom(H)$. Because of the related registers and heaps, and from rule tr-mov-i+r$_1(r_i : x)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_i) \leftrightsquigarrow \sigma(\rho(x))$. In either of the first two cases for $R(r_i)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x[m] \rangle \xrightarrow{\ell}$ err. In the last case, because of related heaps, it also has to be that $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x[m] \rangle \xrightarrow{\ell}$ err.

(b) In this case rules ex-mov-+r is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w \ [r_i + c], r_j \rangle \xrightarrow{\iota} \langle H', R \rangle$. Here $H' = H \circ \{H(R(r_i)) +_4 c +_4 n \mapsto R_{n:n+1}(r_j)\}_{n=0}^{w-1}$.

Similarly, through rule l-ar $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x[m] \rangle \xrightarrow{\ell} \langle \sigma, \pi, a \rangle$ with $a = a' + m$ and $a' = \rho(x)$. Also through rules e-lval and l-var we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y \rangle \xrightarrow{e} \langle \sigma, \pi, v \rangle$ where $v = \sigma(a'')$ and $a'' = \rho(y)$.

From rule tr-mov-i+r$_1$ we know $(r_i : x)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash R(r_i) \leftrightsquigarrow a'$. From the translation rule we also have $(x : \theta[]*) \in \Gamma_c$. Because of the progress, it means that $[a', a' + m] \subseteq\in \pi$. Because of the related heaps and well-typed store it follows that $(R(r_i) + c, a' + m) \in \mu_a$. Also from rule tr-mov-ri+$_1$ we know $(r_j : y)_w \in \mu_\Gamma$. Hence, $\mu_a \vdash R_{0:w}(r_j) \leftrightsquigarrow v$. After the update we can see that $(R(r_i) + c)$ and $a' + m$ are still related.

18. Case tr-mov-i+r$_2$. Then $\iota = (\mathsf{mov}_w \ [r_i + c], r_j$, $\ell = x \to m$ and $e = y$.

(a) This case is possible iff $R(r_i) = 0$, $R(r_i) = \bot$ or $(R(r_i) + c) \notin dom(H)$. Because of the related registers and heaps, and from rule tr-mov-i+r$_2(r_i : x)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_i) \leftrightsquigarrow \sigma(\rho(x))$. In either of the first two cases for $R(r_i)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \to m \rangle \xrightarrow{\ell}$ err. In

the last case, because of related heaps, it also has to be that $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \to m \rangle \xrightarrow{\ell} \mathsf{err}$.

(b) In this case rules ex-mov-+r is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ [r_i + c], r_j \rangle \xrightarrow{\iota} \langle H', R \rangle$. Here $H' = H \circ \{H(R(r_i)) +_4 c +_4 n \mapsto R_{n:n+1}(r_j)\}_{n=0}^{w-1}$.

Similarly, through rule l-ar $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \to m \rangle \xrightarrow{\ell} \langle \sigma, \pi, a \rangle$ with $a = a' + m$ and $a' = \rho(x)$. Also through rules e-lval and l-var we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y \rangle \xrightarrow{e} \langle \sigma, \pi, v \rangle$ where $v = \sigma(a'')$ and $a'' = \rho(y)$.

From rule tr-mov-i+r$_2$ we know $(r_i : x)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash R(r_i) \leftrightsquigarrow a'$. From the translation rule we also have $(x : N*) \in \Gamma_c$. Because of the progress, it means that $[a', a' + m] \subseteq\in \pi$. Because of the related heaps and well-typed store it follows that $(R(r_i) + c, a' + m) \in \mu_a$. Also from rule tr-mov-ri+$_1$ we know $(r_j : y)_w \in \mu_\Gamma$. Hence, $\mu_a \vdash R_{0:w}(r_j) \leftrightsquigarrow v$. After the update we can see that $(R(r_i) + c)$ and $a' + m$ are still related.

19. Case tr-alloc-r*. Then $\iota = (\mathsf{alloc}\ r_i, r_j * c$, $\ell = x$ and $e = \mathsf{new}\ \theta[y * m]$.
    (a) Rule ex-alloc-* only fails iff $R(r_j) = \bot$. Similarly, while rules l-var, e-const and e-op do not fail, rule e-ar fails iff $\sigma(\rho(y)) = \bot$. Since $(r_j : y) \in \mu_\Gamma$, both failures coincide.
    (b) This case is similar to that of tr-alloc-rc$_2$.
20. Case tr-alloc-rc$_1$. Then $\iota = (\mathsf{alloc}\ r_i, c$, $\ell = x$ and $e = \mathsf{new}\ \theta$.
    (a) Rule ex-alloc cannot fail. Similarly, rules l-var and e-new do not fail.
    (b) In this case rules ex-alloc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{alloc}\ r_i, c \rangle \xrightarrow{\iota} \langle H', R' \rangle$. Here $R' = R \circ_4 r_i \mapsto a$. Also $H' = H \circ \{a + i \mapsto \bot\}_{i=0}^{c-1}$.

    Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \rangle \xrightarrow{\ell} \langle \sigma, \pi, a' \rangle$ where $a' = \rho(x)$. Also through rule e-new we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{new}\ \theta \rangle \xrightarrow{e} \langle \sigma', \pi, a'' \rangle$ where $\sigma' = \sigma \circ \{a'' \mapsto \bot\}$.

    Then choose $\mu'_a = \mu_a \circ \{(a : a'')_c\}$. Since $\mu_a \vdash \bot \leftrightsquigarrow \bot$ these fresh addresses are related. Also pick $\nu'_a = \nu_a \circ \{a + i \mapsto (a, c)\}_{i=0}^{c-1}$.
21. Case tr-alloc-rc$_2$. Then $\iota = (\mathsf{alloc}\ r_i, c$, $\ell = x$ and $e = \mathsf{new\ struct}\ N$.
    (a) Rule ex-alloc cannot fail. Similarly, rules l-var and e-str do not fail.
    (b) In this case rules ex-alloc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{alloc}\ r_i, c \rangle \xrightarrow{\iota} \langle H', R' \rangle$. Here $R' = R \circ_4 r_i \mapsto a$. Also $H' = H \circ \{a + i \mapsto \bot\}_{i=0}^{c-1}$.

    Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \rangle \xrightarrow{\ell} \langle \sigma, \pi, a' \rangle$ where $a' = \rho(x)$. Also through rule e-str we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{new\ struct}\ \theta \rangle \xrightarrow{e} \langle \sigma', \pi, a'' \rangle$ where $\sigma' = \sigma \circ \{a'' + i \mapsto \bot\}_{i=0}^{n-1}$ with n is the number of fields in the struct.

    The new memory relations are straightforward.
22. Case tr-alloc-rc$_3$. Then $\iota = (\mathsf{alloc}\ r_i, c$, $\ell = x$ and $e = \mathsf{new}\ \theta[m]$.
    (a) Rule ex-alloc cannot fail. Similarly, rules l-var,e-str and e-const do not fail.
    (b) In this case rules ex-alloc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{alloc}\ r_i, c \rangle \xrightarrow{\iota} \langle H', R' \rangle$. Here $R' = R \circ_4 r_i \mapsto a$. Also $H' = H \circ \{a + i \mapsto \bot\}_{i=0}^{c-1}$.

    Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \rangle \xrightarrow{\ell} \langle \sigma, \pi, a' \rangle$ where $a' = \rho(x)$. Also through rule e-ar we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{new}\ \theta[m] \rangle \xrightarrow{e} \langle \sigma', \pi, a'' \rangle$ where $\sigma' = \sigma \circ \{a'' + i \mapsto \bot\}_{i=0}^{m-1}$.

    The new memory relations are straightforward.
23. Case tr-call. This case follows coinductively.

Basic Blocks The two propositions for basic blocks are the following.

Proposition 15 (Preservation of Progress for Basic Blocks)**.** If

- $\mu_\lambda; \mu_\Gamma; \Gamma_c; \Sigma \vdash b \overset{b}{\rightsquigarrow} s$
- $\forall (a : l) \in \mu_\lambda : \mu_\lambda; \mu_\Gamma; \Gamma_c; \Sigma \vdash \lambda_x(a) \overset{b}{\rightsquigarrow} \lambda_c(l)$
- $\Gamma_c; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$
- $\mu_a; \nu_a; \pi; \vec{\rho}, \rho \vdash H \leftrightsquigarrow \sigma$
- $\mu_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\lambda_x; \vec{R} \vdash \langle H, R, b \rangle \xrightarrow{b} \langle H', R', b' \rangle$

then

- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, s \rangle \xrightarrow{s} \mathsf{err}$ or
- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, s \rangle \xrightarrow{s} \langle \sigma', \pi', s' \rangle$.

Proposition 16 (Preservation of Related Memory for Basic Blocks)**.** If

- $\mu_\lambda; \mu_\Gamma; \Gamma_c; \Sigma \vdash b \overset{b}{\rightsquigarrow} s$
- $\forall (a : l) \in \mu_\lambda : \mu_\lambda; \mu_\Gamma; \Gamma_c; \Sigma \vdash \lambda_x(a) \overset{b}{\rightsquigarrow} \lambda_c(l)$
- $\Gamma_c; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$
- $\mu_a; \nu_a; \pi; \vec{\rho}, \rho \vdash H \leftrightsquigarrow \sigma$
- $\mu_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\lambda_x; \vec{R} \vdash \langle H, R, b \rangle \xrightarrow{b} \langle H', R', b' \rangle$
- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, s \rangle \xrightarrow{s} \langle \sigma', \pi', s' \rangle$

then for some $\mu'_a \supseteq \mu_a$ and $\nu'_a \supseteq \nu_a$:

- $\mu'_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma' \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\mu'_a; \nu'_a; \pi'; \vec{\rho}, \rho \vdash H' \leftrightsquigarrow \sigma'$

Proof 6. The proof is straightforward.

Function Definitions The two propositions for function definitions are the following.

Proposition 17 (Preservation of Progress for Function Definitions)**.** If

- $\Sigma \vdash \langle f, \overrightarrow{r_x}, \overrightarrow{r_y}, a, \lambda_x, j \rangle \rightsquigarrow f(\overrightarrow{x : \theta}) \langle \overrightarrow{y : \theta'}, l, \lambda_c, j \rangle$
- $\mu_\Gamma = \{\overrightarrow{r_x \mapsto x}, \overrightarrow{r_y \mapsto y}\}$
- $\Gamma_c = \{\overrightarrow{x : \theta}, \overrightarrow{y : \theta'}\}$
- $\Gamma_c; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$
- $\mu_a; \nu_a; \pi; \vec{\rho}, \rho \vdash H \leftrightsquigarrow \sigma$
- $\mu_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\lambda_x; \vec{R} \vdash \langle H, R, \lambda_x(a) \rangle \xrightarrow{b} \langle H', R', b' \rangle$

then

- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda_c(l) \rangle \xrightarrow{s} \mathsf{err}$ or
- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda(l) \rangle \xrightarrow{s} \langle \sigma', \pi', s' \rangle$.

Proposition 18 (Preservation of Related Memory for Function Definitions)**.** If

- $\mu_\lambda; \mu_\Gamma; \Gamma_c; \Sigma \vdash b \overset{b}{\rightsquigarrow} s$
- $\mu_\Gamma = \{\overrightarrow{r_x \mapsto x}, \overrightarrow{r_y \mapsto y}\}$
- $\Gamma_c = \{\overrightarrow{x : \theta}, \overrightarrow{y : \theta'}\}$
- $\Gamma_c; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$

- $\mu_a; \nu_a; \pi; \vec{\rho}, \rho \vdash H \leftrightsquigarrow \sigma$
- $\mu_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\lambda_x; \vec{R} \vdash \langle H, R, \lambda_x(a) \rangle \xrightarrow{b} \langle H', R', b' \rangle$
- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda(l) \rangle \xrightarrow{s} \langle \sigma', \pi', s' \rangle$.

then for some $\mu'_a \supseteq \mu_a$ and $\nu'_a \supseteq \nu_a$:

- $\mu'_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma' \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\mu'_a; \nu'_a; \pi'; \vec{\rho}, \rho \vdash H' \leftrightsquigarrow \sigma'$

Proof 7. The proof is straightforward.