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

Ed Robbins
University of Kent, UK
er209@kent.ac.uk

Andy King
University of Kent, UK
a.m.king@kent.ac.uk

Tom Schrijvers
KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

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.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

Keywords reverse engineering, decompilation, recursive datatypes

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 [27], 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 of 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.

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-3549-2/16/01...$15.00
http://dx.doi.org/10.1145/2837614.2837633

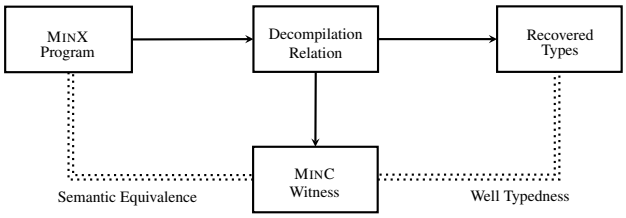

Figure 1: System overview

(b) Then we establish the *operational equivalence* of the MinC witness program and the original MinX program in the form of memory consistency.

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 [31] 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 decompilation (specification) relation is introduced in Section 5. Section 6 concerns the automation of the relation and Section 7 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, by design, abstracts 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 [29] can be avoided. Also MinX fixes a single abstract calling convention, which avoids the need for argument detection techniques and control-flow recovery [1, 7]. These restrictions can be relaxed 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 (counting from zero) up to but not including byte j of a given register r. This is defined by $R_{i:j}(r) = \vec{b}'$ 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 $H : Word \rightharpoonup Byte$ and therefore is byte addressable. To read stored objects that straddle w consecutive bytes we define a function $H^w : Word \to Byte^*$ that reads and amalgamates w 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, $w ::= 2 \mid 4$, denotes the width, in bytes, of the primitive data objects that are supported by the instruction set. The second, ι, defines the instructions themselves:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2: Structured Operational Semantics of MINX programs

categories $\oplus \in \{+, -\}$ and $\otimes \in \{+, -, *, /, \&, |, \ldots\}$, and the width w of their operands. The third category, b, defines blocks:

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

Observe that blocks are terminated by control instructions, but conditional jumps only arise within a block. The fourth category, d_x, 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

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

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

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

Figure 3: Subtyping relations of MINC programs

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 $a_0 \in dom(\lambda'_x)$.

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 the 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, each 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:

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

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 are defined (and related to MINX in Fig 7) in terms of some 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 $\pi \subseteq \{[l, u] \mid 0 \leq l \leq u \leq 2^{32} - 1\}$ that represents a set of (disjoint) memory regions, addition of a short and an address is defined as:

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

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

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

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

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

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

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

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

Figure 4: Type-correct MINC programs

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

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 err 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 associates 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 the on-line appendix, which is available at `http://kar.kent.ac.uk/51459/`.

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

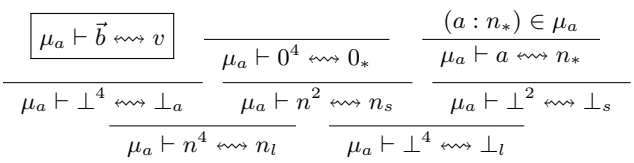

Figure 9: Value correspondence

5. 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 the three main syntactic categories of MINX.

Instruction Decompilation Figure 7 defines the decompilation judgement $\mu_\Gamma; \Gamma_c; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e$ which 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.

We do not have space to explain every rule in detail, so we highlight a number of important aspects:

The judgement is defined by syntax-directed rules, as is usually the case for compilation and elaboration relations. The main 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 ensures 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.

On 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, in MINX, the same stride depends on the size of the elements. Hence, rules like tr-$\oplus$-rc for array pointer arithmetic convert between a MINX stride of $c = m * \mathit{sizeof}(\theta)$ and a corresponding MINC stride of m.

When allocating a statically 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 the 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 their corresponding 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 and syntax-directed.

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) the witness has the same operational semantics as the original MINX program. Taken together these two properties give meaning to the statement that *the original* MINX *program inhabits the recovered types*.

5.1.1 Well-Typing

The first claim states that the recovered witness program is well-typed. This is asserted as 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, it follows 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 how 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 parameterised 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 $\vec{\mu_\Gamma}$ 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.

$$\boxed{\Sigma; \Psi; \sigma; \pi \vdash a : \tau} \qquad \text{st-comp}\frac{\Sigma; \Psi \vdash \sigma(a) : \theta}{\Sigma; \Psi; \sigma; \pi \vdash a : \theta}$$

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

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

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

Figure 6: Well-typed addresses and values

Heaps versus Stores The MINX heap H and MINC store σ are related with:

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

This relation summarises 6 different properties addressing 4 concerns. Firstly, as in the previous cases, this relation is parameterised 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, MINX addresses individual bytes and can point into the middle of a value. To bridge this gap, the address map μ_a only covers the addresses 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 first byte of the value and its width.

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

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 a consequence 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. An algorithm, however, can be derived for solving a problem by adding control to Horn

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

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

$$\text{tr-}\oplus\text{-r*}_2 \frac{(r_i : x)_w \in \mu_\Gamma \quad (r_j : y)_w \in \mu_\Gamma \quad (x : t) \in \Gamma_c \quad (y : t) \in \Gamma_c \quad \Gamma_c; \Sigma \vdash c : t}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{op}_w^\oplus\ r_i, r_j * c \overset{\iota}{\rightsquigarrow} x := x \oplus (y * c)} \qquad \text{tr-}\oplus\text{-rc} \frac{(r_i : x)_4 \in \mu_\Gamma \quad (x : \theta[]*) \in \Gamma_c \quad c = m * \mathit{sizeof}(\theta) \quad \Gamma_c; \Sigma \vdash m : t}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{op}_4^\oplus\ r_i, c \overset{\iota}{\rightsquigarrow} x := x \oplus m}$$

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

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

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

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

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

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

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

$$\text{tr-mov-ri+}_2 \frac{(r_i : x)_w, (r_j : y)_4 \in \mu_\Gamma \quad (x : \theta), (y : N*) \in \Gamma_c \quad \Sigma(N) = \langle \theta_0, \ldots, \theta_{n-1} \rangle \quad c = \sum_{k=0}^{m-1} \mathit{sizeof}(\theta_k) \quad \Sigma \vdash \theta_m <: \theta \quad \mathit{sizeof}(\theta_m) = \mathit{sizeof}(\theta) = w}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{mov}_w\ r_i, [r_j + c] \overset{\iota}{\rightsquigarrow} x := y \to m} \qquad \text{tr-mov-i+r}_2 \frac{(r_i : x))4, (r_j : y)_w \in \mu_\Gamma \quad (x : N*), (y : \theta) \in \Gamma_c \quad \Sigma(N) = \langle \theta_0, \ldots, \theta_{n-1} \rangle \quad c = \sum_{k=0}^{m-1} \mathit{sizeof}(\theta_k) \quad \Sigma \vdash \theta <: \theta_m \quad \mathit{sizeof}(\theta_m) = \mathit{sizeof}(\theta) = w}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{mov}_w\ [r_i + c], r_j \overset{\iota}{\rightsquigarrow} x \to m := y}$$

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

$$\text{tr-alloc-rc}_1 \frac{(r_i, x)_4 \in \mu_\Gamma \quad \mathit{sizeof}(\theta) = c \quad (x : \theta*) \in \Gamma_c}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{alloc}\ r_i, c \overset{\iota}{\rightsquigarrow} x := \mathsf{new}\ \theta} \qquad \text{tr-alloc-rc}_2 \frac{(r_i, x)_4 \in \mu_\Gamma \quad \mathit{sizeof}(N) = c \quad (x : N*) \in \Gamma_c}{\mu_\Gamma; \Gamma_c; \Sigma \vdash \mathsf{alloc}\ r_i, c \overset{\iota}{\rightsquigarrow} x := \mathsf{new\ struct}\ N}$$

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

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

```
iterative_sum {
        mov4 r0, 0 ;
        goto .BB1
.BB0:
        mov4 r2, [r1] ;
        add4 r0, r2 ;
        mov4 r1, [r1 + 4] ;
.BB1:
        if4 r1 goto .BB0 ;
        ret
} <(r1), r0, (r2)>
```

Figure 10: Iterative summation of a linked list in MINX

```
struct struct1 {
  long;
  struct1*;
};

iterative_sum(struct1* x) {
  long y1, y2
  0: y1 = 0;
     goto 2
  1: y2 = x->0;
     y1 = y1 + y2;
     x = x->1;
  2: if x goto 1;
     return y1
}
```

Figure 11: Iterative summation of a linked list in MINC

clauses that specify the problem [14]. Following this methodology, we have translated the decompilation relation rule-for-rule (almost verbatim) into Horn clauses, programming the control using Constraint Handling Rules (CHR) [8], which is an extension to Prolog. 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 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, derives one that is faithful to the rules of the decompilation relation.

To generate input for the solver, the self-hosting ANSI C89 compiler `ucc` [30] 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 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. 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 these two levels of checking.

The solver requires input to be pre-processed and presented in the MINX language. Figure 10 lists (pretty-printed) MINX code for summing the elements of a linked list. The arguments, return register and locals, denoted $\overrightarrow{r_{arg}}, r_{ret}, \overrightarrow{r_{loc}}$ in Section 3, correspond to `(r1)`, `r0` and `(r2)` respectively in the code listing. The mapping λ'_x is represented using the `.BB0` and `.BB1` labels to directly tag their corresponding block. Note that blocks can overlap. The label

benchmark	LoC C	LoC MINX	original structs	solns	recovered structs	time (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	2	4	6/5/6/5	77
hashquad	260	977	2	4	2/3/2/2	49
kdtree	112	891	1	1	1	1,527
leftheap	182	762	1	1	1	825
list	262	829	1	2	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	1	2	3/2	193
sets	120	355	0	1	0	276
skip	239	2,616	1	1	1	2,089
sort	364	2,339	0	1	0	2,904
splay	332	2,648	1	1	2	4,975
stackar	161	1,680	1	1	1	640
stackli	140	1,270	1	2	2/2	464
treap	288	2,580	1	1	2	3,834
tree	208	1,104	1	1	2	2,158

Figure 12: Solutions and Recovered Structures

of the entry block, a_0, is left implicit by adopting the convention that the first block is always the entry block.

Figure 11 presents the MINC witness program generated by the solver, again pretty-printed for human comprehensibility since the solver represents the witness as an abstract syntax tree. The local variables, denoted $\overrightarrow{y : \theta'}$ in Section 4, are given immediately before the entry block. The mapping λ_c is represented, again by using labels to tag the blocks. The index j, used to identify which local variable is returned in Section 4, is identified by printing each `return` statement with the variable y_j.

7. Evaluation

The solver was deployed on a suite of textbook [31] programs, chosen because of their use of data-structures. Figure 12 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 *original structs* column indicates the number of struct types defined 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; backtracking enabling all solutions to be enumerated. 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 a second on-line appendix, that is available at `http://kar.kent.ac.uk/51448/`.

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 [29, 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, 28], 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 types 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 are 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, 26] 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 works, 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.

Trusted compilation 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 [26], 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 it derives a high-level witness program in concert with the types (provided one exists). 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, which is a first step towards certified decompilation, 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.

Acknowledgments

This work was supported by grant EP/K031929/1 funded by GCHQ in association with EPSRC, and partly funded by the Flemish Fund for Scientific Research (FWO).

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. Howe, and A. King. Theory Propagation and Reification. *Science of Computer Programming*, 111:3–22, 2015.

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

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

[28] 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.

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

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

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