Bit Level Types for High Level Reasoning

Ranjit Jhala
UC San Diego
jhala@cs.ucsd.edu

Rupak Majumdar
UC Los Angeles
rupak@cs.ucla.edu

ABSTRACT

Bitwise operations are commonly used in low-level systems code to access multiple data fields that have been packed into a single word. Program analysis tools that reason about such programs must model the semantics of bitwise operations precisely in order to capture program control and data flow through these operations. We present a type system for subword data structures that explitictly tracks the flow of bit values in the program and identifies consecutive sections of bits as logical entities manipulated atomically by the programmer. Our type inference algorithm tags each integer value of the program with a *bitvector type* that identifies the data layout at the subword level. These types are used in a translation phase to remove bitwise operations from the program, thereby allowing verification engines to avoid the expensive low-level reasoning required for analyzing bitvector operations. We have used a software model checker to check properties of translated versions of a Linux device driver and a memory protection system. The resulting verification runs could prove many more properties than the naive model checker that did not reason about bitvectors, and could prove properties much faster than a model checker that did reason about bitvectors. We have also applied our bitvector type inference algorithm to generate program documentation for a virtual memory subsystem of an OS kernel. While we have applied the type system mainly for program understanding and verification, bitvector types also have applications to better variable ordering heuristics in boolean model checking and memory optimizations in compilers for embedded software.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification.
General Terms: Languages, Verification, Reliability.
Keywords: Bit vectors, type inference, model checking.

1. INTRODUCTION

Many programs manipulate data at subword levels where multiple data fields are packed into a single word. For example, in the virtual memory subsystem of an operating system, a 32-bit linear address can represent 20 bits of an index into a page table, 10 more bits as an offset into a page, and two permission bits. Similarly, in embedded applications where the memory footprint must be small, different fields must often be packed (sometimes automatically by a compiler) into one word. These subword records are manipulated by the programmer using bit-level operations, namely, masks and shifts. To reason about such programs manually or automatically we require techniques that accurately capture the flow of information through bitvector variables and the shifting and masking operations.

There are usually three ways of handling bitvectors in program analysis. The first is to treat bitvector operations as uninterpreted functions, ignoring the semantics of these operations. The second is to use specialized decision procedures for bitvectors [17, 2, 14]. The third is to reduce all variables to (32 or 64) individual bits ("bit-blasting") and use propositional reasoning [3, 23]. The first option, while surprisingly common [1, 8, 5, 10], especially when the tool builder does not anticipate the use of bitvectors, produces imprecisions in the analysis that show up as false positives. In our experience, the second approach of using specialized decision procedures makes the analysis much slower, just because specialized decision procedures are comparatively slower than for example, well-tuned procedures for the theory of equality and arithmetic available in decision procedures [6, 20]. The third option is attractive, due to the efficiency of SAT solvers and BDD engines, and the accuracy with which the machine semantics is reflected [3, 23]. While this option has been successfully applied in "bounded" analyses for finding bugs, the bit-blasting loses the high-level relationships between the variables that are critical for statically computing invariants, thereby making the option unsuitable for safety verification.

In this paper, we provide a fourth, type-based, option. We define a type system at the bit level that identifies consecutive sections of bits as logical subword entities manipulated by the programmer. The type system explicitly tracks bitwise operations in the program, and propagates flow of bit values across the program. The type system captures the intuition that the programmer is using the masks and shifts to identify subsections of the bits.

Consider a virtual memory system where virtual addresses are 32-bit values, organized as a 20 bit page table index, a 10 bit offset into a page (for the address), and two permission bits. The programmer finds the index and the offset using bit masks:

SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.

```
01      index =  (x & 0xFFFFF000) >> 12;
02      offset = (x & 0xFFC) >> 2;
```

and in addition, can check the permission bits:

```
03      can_read  = (x & 0x1);
04      can_write = (x & 0x2);
```

Our type system generates constraints on subfields of the 32-bit quantities defined in the program. For example, the assignment on line `01` constrains the representation of `index` to be a subtype of the rhs, and the bitmask on the rhs constrains the representation of `x` to have a 20-bit part and a 12-bit part (which may be further refined by other constraints). By generating and solving all such constraints, we infer the bit-level types for virtual addresses:

$$\langle \mathsf{index}, 20\rangle\langle \mathsf{offset}, 10\rangle\langle \mathsf{wr}, 1\rangle\langle \mathsf{rd}, 1\rangle$$

representing a structure with an index field (of 20 bits), an offset field (of 10 bits), and two permission bits.

In a second step, the types identified by our algorithm are used to "compile away" the bitvector operations, by translating bitvectors into structures, and translating masks and shifts into field accesses. The resulting program contains only assignments between integer variables and boolean tests, for which existing decision procedures [6, 20] are extremely fast. For bitwise operations that do not conform to this access pattern (e.g., in signal processing operations, security algorithms, or in condition codes where each bit independently maintains some boolean information), the type of a bitvector is just the sequence of individual bits. In this case, we are no worse than the boolean reasoning obtained by reducing the word into bits.

We have implemented the bitvector type inference and program transformation algorithm, and we have applied the transformation to check safety properties of C programs using the Blast model checker [10]. Blast uses decision procedures for equality and arithmetic. Our previous attempt to use specialized decision procedures for bitvectors did not scale to large programs. The decision procedures were unable to infer and exploit the fact that the common idiomatic use of bit-level operations is to use words as packed structures, and thus, were very inefficient. This work is the result of our attempt to build an analysis that discovered what high-level relationships were buried beneath the bitvector operations.

In our experiments, we took two implementations: one a memory management and protection framework (Mondrian [22]) and one a Linux device driver [4]. We first performed the bitvector type inference on these programs and translated them automatically to programs without bitvector operations based on the types. We then applied the Blast model checker to check safety properties on the translated programs. The Mondrian implementation was annotated with a set of assertions by the programmer. For the driver, we considered five safety properties identified in [11]. Both programs involve nontrivial bitwise operations. In a previous study [11], the operations from the driver were removed by hand. To the best of our knowledge, the Mondrian implementation was not automatically checked before. Blast was able to prove 12 of the 15 properties checked. In contrast, the bit-reduction approach implemented in Blast that applied boolean reasoning to individual bits did not finish for these examples. The remaining three properties involved reasoning about multi-dimensional arrays on the heap, which is a limitation of Blast and an important, but orthogonal, problem.

While our primary application was to lift bits to high-level structures, bitvector types are relevant even if the analysis is carried out at the boolean level, using for example SAT solvers or binary decision diagrams (BDDs). In these cases, the bitvector type determines a good variable-ordering heuristic. Precisely, variables with the same bitvector type should be interleaved in the variable order, while variables that have different bit types can be independently ordered. This can make an exponential improvement even in simple, and common, cases. Consider the following program where all variables are 32-bit integers:

```
y = (x & 0xFFFF0000)>>16;
z = (x & 0x0000FFFF);
y = y | (z << 16);
```

The bitvector types of the variables are

$$\begin{array}{ll} \texttt{x} & \langle \mathsf{a}, 16\rangle\langle \mathsf{b}, 16\rangle \\ \texttt{y} & \langle \mathsf{b}, 16\rangle\langle \mathsf{a}, 16\rangle \\ \texttt{z} & \mathbf{0}^{16}\langle \mathsf{b}, 16\rangle \end{array}$$

indicating that values from the top 16 bits of `x` and the bottom 16 bits of `y` may flow into each other, and values from the low 16 bits of `x`, the low 16 bits of `z`, and the high 16 bits of `y` may flow into each other. This suggests the variable ordering where the high 16 bits of `x` and the low 16 bits of `y` are interleaved, and the low 16 bits of `x`, the high 16 bits of `y`, and the low 16 bits of `z` are interleaved. With this ordering, the size of the BDDs is linear in the number of bits. However, with the natural ordering where all bits of `x` are ordered before all bits of `y` and all bits of `z`, the BDDs are exponential in the number of bits.

Our inferred types can also be used for memory optimizations in resource constrained systems [16, 21]: for a variable that has a bitvector type with k bits, the compiler can "pack" several allocations into one word. For variables where certain bits are unused, the compiler need not allocate those bits at all.

Related Work. Our type-based approach was directly inspired by the representation inference of Lackwit [18]. Our type system and inference algorithm is very similar to those presented by Ramalingam *et. al.* for aggregate structure inference in Cobol [19, 13]. Both the above use type inference to identify structure in weakly typed languages. Our contribution is to apply similar inference techniques at the subword level in C programs, and apply to the task of verifying code that uses low-level operations. This setting poses the additional challenge of developing a notion of subtyping that captures the idiomatic uses of bitwise operations, which is essential for not splitting the structures too finely. By producing constraints on the bit-level representations, bit-level types provide a means to identify application-level abstractions used by the programmer. The resulting types are an important program understanding tool that replaces complicated bitwise operations in the code with record structures and field accesses. By focusing on constraints derived from actual use, we are able to more accurately identify abstract structures, and high-level relationships between the subwords packed inside the integer, that are beyond the scope of the `typedef` facility provided by C at the source level and used by programmers to convey abstractions. While

Expressions	e	::=	$c \mid \mathtt{x} \mid \mathtt{x}[\mathsf{e}] \mid \mathsf{e}_1 \oplus \mathsf{e}_2$
			$\mid \mathsf{e}\&c \mid \mathsf{e}\vert c \mid \mathsf{e}{>>}n \mid \mathsf{e}{<<}n$
			where $c, n \in \mathbb{N}$, $0 \le n < N$
Predicates	p	::=	$\mathsf{e}_1 \le \mathsf{e}_2 \mid \neg\mathsf{p} \mid \mathsf{p}_1 \wedge \mathsf{p}_2 \mid \mathsf{p}_1 \vee \mathsf{p}_2$
Statements	s	::=	$\mathtt{x}{:=}\mathsf{e} \mid \mathtt{x}[\mathsf{e}]{:=}\mathsf{e}_1 \mid s; s$
			$\mid$ **if** (p) **then** s **else** s
			$\mid$ **while** (p) s

Figure 1: Program Syntax

the translation does not preserve the run-time memory layout, we have found bit-level types to be an attractive tool to examine, understand and check the usage of bit fields in unknown programs and the flow of values between different bit sections in different variables.

Our work is also similar to [9], where a dataflow analysis is provided to identify bit sections in programs. This information is used by the compiler to allocate only the required number of bits, thereby optimizing memory allocation. Our type system enables us to formalize many of the properties that were informally stated in [9], to extend uniformly to complex source level features such as pointers and functions and subtyping, to provide principal typing (or "best possible" representation) guarantees, and cleanly prove the correctness of the algorithm using standard type inference techniques. Further, we provide an explicit bound on the complexity of the algorithm (quadratic in the size of the program and in the number of bits). While quadratic in the worst case, in practice this has never been a bottleneck and all our type inference experiments ran within a few seconds.

2. BIT-LEVEL TYPES

2.1 Programs

We demonstrate our algorithm on an imperative language of `while` programs with bit-level operations. Let X be a set of program variables. For each array variable x in X, we include two variables x_idx and x_elem in X (which will be used as the "index" and "contents" of the array, respectively). Figure 1 shows the syntactic elements of our programs. For ease of exposition, we shall assume all variables and constants have N bits. Expressions are either integer constants, or N-bit bit-level constants, or N-bit variables ($\mathtt{x}$) or array accesses ($\mathtt{x}[\mathsf{e}]$, where $\mathtt{x}$ is an array variable and e its index), arithmetic operations $\mathsf{e}_1 \oplus \mathsf{e}_2$, and the bit-level operations bitwise-and with a constant ($\mathsf{e}\&c$), bitwise-or with a constant ($\mathsf{e}|c$), and right or left shift by a constant ($\mathsf{e}{>>}n$ and $\mathsf{e}{<<}n$ respectively). Predicates are built using arithmetic comparison and the boolean operators. Statements comprise assignments (to integer variables or array fields), sequential composition, conditionals, and while loops. Let $\mathsf{Exp}(X)$ denote the set of all expressions using the variables X. The operational semantics for the language is defined in the standard way, using a store mapping each variable to a bit vector and each array to an array of bitvectors and interpreting the operations in the usual way.

Example 1: [Program] On the left in Figure 2, we show a program that uses bit-level operations. We assume all values have 32 bits. The program `mget` abstracts (and simplifies) a kernel's physical memory access routine. It gets a 32-bit virtual address `p` as an input. We assume memory is organized into a one level page table, where each page is 1K bytes. We model the page table as an array `tab`. The page table is indexed using the top 20 bits of a virtual address. The next 10 bits of the virtual address is the offset into the page. The last bit of `p` is a permission bit, that must be set in order to access memory at the physical address. The second last bit of `p` is a dirty-bit that is not relevant to the current program. If the rightmost permission bit of `p` is not set, then the function `mget` returns an error condition (permission failure). The variable `p1` drops the last 12 bits from the virtual address `p`, and the variable `b1` shifts the top 20 bits to the right, this is used as an index into the page table `tab`. The page table entry returns the base address `b1` of a memory page where the address resides, the variable `base` zeroes out the last permission bit. The offset into the page table is stored in variable `off` that zeroes out all but the 10 offset bits of `p`. Finally, the address in memory is obtained by adding the offset to the base address of the memory page, and actual memory is looked up by looking at the index `a` of a global array `m` that represents physical memory. We ignore error conditions and array bounds checks for clarity. □

2.2 Types

We introduce special *bit-level types* that are refinements of the usual base types which describe how the corresponding N-bit expressions are used in the program as packed records. Consider the example of Figure 2. We want the bit-level type to specify that the virtual address `p` is used as a packed record structured as a segment with 20 bits, followed by a segment with 10 bits, followed by two one-bit segments.

Let $\mathbb{A}$ be an infinite set of *names*. A *block* is a pair $\langle \mathsf{a}, \ell \rangle$, where $\mathsf{a} \in \mathbb{A}$ and ℓ is an integer in $\{1 \ldots N\}$. We call a (resp. ℓ) the name (resp. size) of the block. We write $\mathbb{B}$ for the set of all blocks. For a block b, we write b^k for a the sequence of k repetitions of b. A *bit-level type* is a sequence of blocks $\bar{b} \equiv \langle \mathsf{a}_1, \ell_1 \rangle \ldots \langle \mathsf{a}_n, \ell_n \rangle$. The *size* $|\bar{b}|$ of a bit-level type is $\sum_i \ell_i$. A bit-level type $\bar{b}$ encodes a sequence of bits of length $|\bar{b}|$, indexed from $|\bar{b}| - 1$ (the leftmost or most significant bit) to 0 (the rightmost or least significant bit). We write $\mathsf{Typ}(\ell)$ for the set of all bit-level types of size ℓ.

Example 2: [Bit-level Types] The top row in Figure 3(a) shows a bit-level type:

$$\bar{b} \equiv \langle \mathsf{e}, 20 \rangle . \langle \mathsf{a}, 10 \rangle . \langle \mathsf{k}, 1 \rangle . \langle \mathsf{u}, 1 \rangle$$

of size 32 that encodes a virtual address, with a 20-bit page table index, a 10-bit offset, and two permission bits. Above the blocks are integers indicating the ranges of bits occupied by each block. The page table index occupies section from the 31st bit to the 12th (inclusive), the offset the section from the 11th to the 2nd, and the two permission bits are the 1st and 0th bits. □

Zero Blocks. We shall assume the set of names $\mathbb{A}$ contains a special name $\mathbf{0}$, which corresponds to blocks whose bits have value 0. Hence, the zero block $\langle \mathbf{0}, 1 \rangle$ intuitively corresponds to a block of 1 bit whose value is 0. We abbreviate the block $\langle \mathbf{0}, 1 \rangle$ to $\mathbf{0}$; thus $\mathbf{0}^k$ represents k copies of the block $\langle \mathbf{0}, 1 \rangle$, *i.e.,* k consecutive 0 bits.

Projections. For a bit-level type $\bar{b} \equiv \bar{b}' . \langle \mathsf{a}, \ell \rangle$ and an integer i we define the *block-fragment of* $\bar{b}$ *at position* i as:

$$\bar{b}[i] \equiv \begin{cases} \bar{b}'[i - \ell] & \text{if } \ell < i + 1 \\ (\mathsf{a}, i, \ell - i) & \text{o.w.} \end{cases} \qquad (1)$$

Program	Zero Constraints	Inequality Constraints
`mget(u32 p){`		
`if ((p & 0x1) == 0){`	$\tau_{\mathtt{p\&1}}[31:1] = \mathbf{0}$	$\tau_{\mathtt{p}}[0:0] \leq \tau_{\mathtt{p\&1}}[0:0]$
`error("Permission Failure");`		
`} else {`		
`p1 = p & 0xFFFFF000;`	$\tau_{\mathtt{p\&1^{20}0^{12}}}[11:0] = \mathbf{0}$	$\tau_{\mathtt{p}}[31:12] \leq \tau_{\mathtt{p\&1^{20}0^{12}}}[31:12]$
		$\tau_{\mathtt{p\&1^{20}0^{12}}} \leq \tau_{\mathtt{p1}}$
`pte = p1 >> 12;`	$\tau_{\mathtt{p1>>12}}[31:20] = \mathbf{0}$	$\tau_{\mathtt{p1}}[31:12] \leq \tau_{\mathtt{p1>>12}}[19:0]$
		$\tau_{\mathtt{p1>>12}} \leq \tau_{\mathtt{pte}}$
`b1 = tab[pte];`		$\tau_{\mathtt{pte}} \leq \tau_{\mathsf{tab\_idx}}$
		$\tau_{\mathsf{tab\_elem}} \leq \tau_{\mathtt{b1}}$
`base = b1 & 0xFFFFFFFC;`	$\tau_{\mathtt{b1\&1^{30}0^{2}}}[1:0] = \mathbf{0}$	$\tau_{\mathtt{b1}}[31:2] \leq \tau_{\mathtt{b1\&1^{30}0^{2}}}[31:2]$
		$\tau_{\mathtt{b1\&1^{30}0^{2}}} \leq \tau_{\mathtt{base}}$
`off = p & 0xFFC;`	$\tau_{\mathtt{p\&1^{10}0^{2}}}[1:0] = \mathbf{0}$	$\tau_{\mathtt{p}}[11:2] \leq \tau_{\mathtt{p\&1^{10}0^{2}}}[11:2]$
	$\tau_{\mathtt{p\&1^{10}0^{2}}}[31:12] = \mathbf{0}$	$\tau_{\mathtt{p\&1^{10}0^{2}}} \leq \tau_{\mathtt{off}}$
`a = base + off;`		$\tau_{\mathtt{base}} \leq \tau_{\mathtt{base+off}}$
		$\tau_{\mathtt{off}} \leq \tau_{\mathtt{base+off}}$
		$\tau_{\mathtt{base+off}} \leq \tau_{\mathtt{a}}$
`return m[a];`		$\tau_{\mathtt{a}} \leq \tau_{\mathtt{m\_idx}}$
`}`		$\tau_{\mathsf{m\_elem}} \leq \tau_{\mathtt{mget\_r}}$
`}`		

Figure 2: Example Program, Constraints

Intuitively, the block-fragment of $\bar{b}$ at position i is a triple $(\mathsf{a}, \ell, \ell')$ such that the ith bit of $\bar{b}$ lies in a block $\langle \mathsf{a}, \ell + \ell' \rangle$, with ℓ' bits to the left of the ith bit in $\langle \mathsf{a}, \ell + \ell' \rangle$, and ℓ bits to the right. Equivalently, $\bar{b}$ has as a suffix the sequence $\langle \mathsf{a}, \ell + \ell' \rangle.\bar{b}''$ where the size of $\bar{b}''$ is $i - \ell$. In particular, if the block-fragment at position i is $(\mathsf{a}, \ell, 0)$ then $\bar{b}$ has as a suffix the sequence $\langle \mathsf{a}, \ell \rangle.\bar{b}''$ of size i. In this case, we say that $\bar{b}$ *has a break* at position i.

For a bit-level type $\bar{b}$ and a pair of integers i, j we define the *projection of* $\bar{b}$ *to the interval* $[i : j]$ as:

$$\bar{b}[i:j] \equiv \begin{cases} \epsilon & \text{if } i < j \\ \langle \mathsf{a}, \ell \rangle.\bar{b}[i - \ell : j] & \text{if } \bar{b}[i+1] = (\mathsf{a}, \ell, 0) \\ \perp & \text{o.w.} \end{cases}$$

where $\perp$ means the value is undefined and the concatenation . is $\perp$ if either argument is $\perp$. When defined, the projection of $\bar{b}$ to $[i : j]$ is a bit-level type $\bar{b}'$ of size $i - j + 1$, such that $\bar{b}$ is of the form $\bar{b}^{+}.\bar{b}'.\bar{b}^{-}$ where the size of $\bar{b}^{+}$ (resp. $\bar{b}^{-}$) is $N - i$ (resp. j). In other words, the projection is a bit-level type corresponding to the sequence of blocks beginning at position i and ending at position j of $\bar{b}$. The projection of $\bar{b}$ to the interval $[i : j]$ is defined iff $\bar{b}$ has breaks at $i + 1$ and j.

Example 3: [Block-Fragments, Projections] The second row of Figure 3(a) shows the block-fragment of $\bar{b}$ (from the top row) at position 8. Notice that $\bar{b}$ does not have a break at that position: in particular, the block $\langle \mathsf{a}, 10 \rangle$ occupies the section from the 11th to the 2nd bit, and hence there are 6 bits of $\langle \mathsf{a}, 10 \rangle$ "remaining" on the right, at position 8, and a prefix of 4 bits on the left. Hence $\bar{b}[8]$ is $(\mathsf{a}, 6, 4)$. The third row of Figure 3(a) shows the block-fragment of $\bar{b}$ at position 12. Notice that $\bar{b}$ has a break at position 12 as there are three whole blocks to the right. Hence $\bar{b}[12]$ is $(\mathsf{a}, 10, 0)$, which in turn indicates that a block $\langle \mathsf{a}, 10 \rangle$ starts at the position 12 of $\bar{b}$. The fourth row of Figure 3(a), shows the projection of $\bar{b}$ to the interval $[11 : 1]$, which is the sequence of blocks $\langle \mathsf{a}, 10 \rangle \langle \mathsf{k}, 1 \rangle$. It is well-defined as $\bar{b}$ has a break at position $11 + 1$, and hence $\bar{b}[11 + 1] = (\mathsf{a}, 10, 0)$, and also has a break at position 1. The projection is the sequence of blocks occupying the 11^{th} through the 1^{st} bits (inclusive). $\square$

Subtyping. For two bit-level types $\bar{b}_1, \bar{b}_2$, we say that $\bar{b}_1 \leq \bar{b}_2$ if either 1) $\bar{b}_1 = \bar{b}_2$, or 2) $\bar{b}_1 \equiv \mathbf{0}^{\ell}.\langle \mathsf{a}, \ell' \rangle.\bar{b}_1'$, $\bar{b}_2 \equiv \langle \mathsf{a}, \ell + \ell' \rangle.\bar{b}_2'$, and $\bar{b}_1' \leq \bar{b}_2'$. Intuitively, $\bar{b}_1 \leq \bar{b}_2$ if for each i such that $\bar{b}_2$ has a break at position i, it is the case that $\bar{b}_1$ also has a break at that position, and moreover, if the block of $\bar{b}_2$ at that position is $\langle \mathsf{a}, \ell \rangle$ then at that position, $\bar{b}_1$ has a sequence of zero blocks followed by a block $\langle \mathsf{a}, \ell' \rangle$ such that the length of the entire sequence is ℓ.

To get an intuitive understanding of our subtyping relation, note that if $\ell \leq \ell'$ then

$$\mathbf{0}^{N-\ell} \langle \mathsf{a}, \ell \rangle \leq \mathbf{0}^{N-\ell'} \langle \mathsf{a}, \ell' \rangle$$

as the LHS corresponds to integers less than 2^{ℓ} which are a subset of integers less than $2^{\ell'}$ which is what the type on the RHS corresponds to. However, we *do not* consider $\mathbf{0}^{N-\ell-k} \langle \mathsf{a}, \ell \rangle \mathbf{0}^{k}$ to be a subtype of $\mathbf{0}^{N-\ell'} \langle \mathsf{a}, \ell' \rangle$, when $k > 0$ because in the former type, the non-zero segments are not right-aligned. This captures the idiomatic use of subtypes, wherein the programmer ensures right-alignment (via a right-shift) before assignment. Our subtyping definition generalizes this observation to bit-level types that contain multiple blocks, *i.e.* where the bitvector contains more than one packed integer integer. It is easy to check that the subtyping relation is a partial-order.

Example 4: [Zero blocks and Subtyping] The lower half of the top row of Figure 3(b) shows a bit-level type $\bar{b}_1 \equiv \mathbf{0}^{10}.\langle \mathsf{p}, 6 \rangle.\mathbf{0}^{6} \langle \mathsf{q}, 10 \rangle$ containing zero blocks. The upper half shows a type $\bar{b}_2 \equiv \langle \mathsf{p}, 16 \rangle.\langle \mathsf{q}, 16 \rangle$, such that $\bar{b}_1 \leq \bar{b}_2$. $\square$

Sound Type Assignment. A X-*type assignment* is a map $\Gamma : X \to \mathsf{Typ}(N)$ that assigns assigning a bit-level type to each element of X. For an X-type assignment Γ, we write $\Gamma \vdash e : \bar{b}$ to denote that under the assignment Γ the

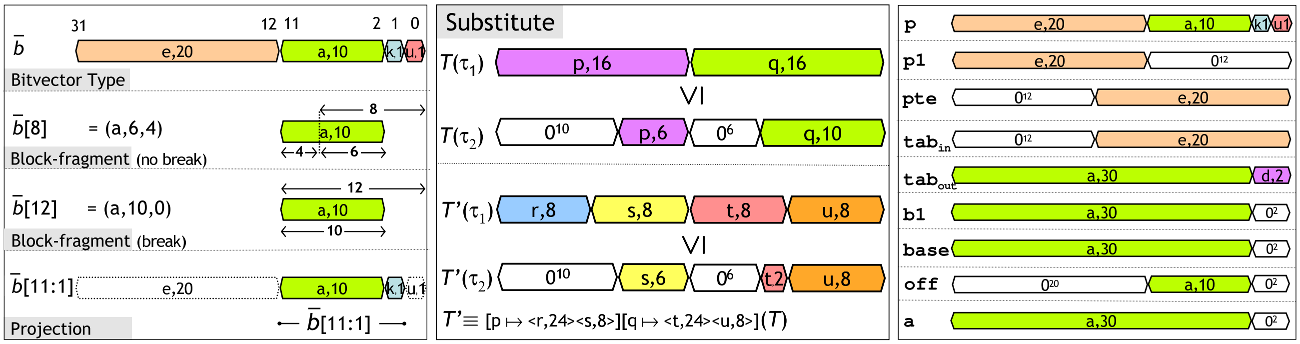

Figure 3: (a) Bit-level types and Projections (b) Substitution and Subtyping (c) Bit-level type for `mget`

$$\frac{\Gamma(x) = \bar{b}}{\Gamma \vdash x : \bar{b}}\textsc{Var} \qquad \frac{\Gamma(x\_\mathsf{idx}) = \bar{b} \qquad \Gamma(x\_\mathsf{elem}) = \bar{b}' \qquad \Gamma \vdash e : \bar{b}}{\Gamma \vdash x[e] : \bar{b}'}\textsc{Arr} \qquad \frac{\Gamma \vdash e : \bar{b} \qquad \bar{b} \leq \bar{b}'}{\Gamma \vdash e : \bar{b}'}\textsc{Sub}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau[h,l] = \tau_1[h,l] \text{ for } [h,l] \in \mathsf{Brk}(c,1)}{\Gamma \vdash (e \mid c) : \tau}\textsc{Bit-Or}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau[h,l] = \mathbf{0}^{h-l} \text{ for } [h,l] \in \mathsf{Brk}(c,0) \qquad \tau[h,l] = \tau_1[h,l] \text{ for } [h,l] \in \mathsf{Brk}(c,1)}{\Gamma \vdash (e \& c) : \tau}\textsc{Bit-And}$$

$$\frac{\Gamma \vdash e : \bar{b}}{\Gamma \vdash (e{>>}c) : \mathbf{0}^c.\bar{b}[N : N-c]}\textsc{Rshift} \qquad \frac{\Gamma \vdash e : \bar{b}}{\Gamma \vdash (e{<<}c) : \bar{b}[N-c:0].\mathbf{0}^c}\textsc{Lshift}$$

$$\frac{\Gamma \vdash e_1 : \bar{b} \qquad \Gamma \vdash e_2 : \bar{b} \qquad \bar{b} = \mathbf{0}^{N-\ell}\langle \mathsf{a}, \ell\rangle \qquad \mathsf{Bound}(e_1 \oplus e_2, \ell)}{\Gamma \vdash (e_1 \oplus e_2) : \bar{b}}\textsc{A-Op}$$

$$\frac{\Gamma \vdash e_1 : \bar{b} \qquad \Gamma \vdash e_2 : \bar{b} \qquad \bar{b} = \mathbf{0}^{N-\ell}\langle \mathsf{a}, \ell\rangle}{\Gamma \vdash (e_1 \leq e_2)}\textsc{Leq} \qquad \frac{\Gamma \vdash p_1 \qquad \Gamma \vdash p_2}{\Gamma \vdash (p_1 \wedge p_2)}\textsc{B-Op}$$

$$\frac{\Gamma \vdash l : \bar{b} \qquad \Gamma \vdash e : \bar{b}' \qquad \bar{b}' \leq \bar{b}}{\Gamma \vdash l := e : \bar{b}'}\textsc{Asgn} \qquad \frac{\Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash s_1; s_2}\textsc{Seq}$$

$$\frac{\Gamma \vdash p \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \texttt{if } p \texttt{ then } s_1 \texttt{ else } s_2}\textsc{Cond} \qquad \frac{\Gamma \vdash p \qquad \Gamma \vdash s}{\Gamma \vdash \texttt{while } (p)\ s}\textsc{While}$$

Figure 4: Typing rules

expression e has type $\bar{b}$, and we write $\Gamma \vdash s$ to denote that Γ is *sound* w.r.t. the program s. The rules that derive these judgements are shown in Figure 4.

The key rules are the assignment rule (Asgn), which requires that the lvalue assigned to be a supertype of the value to which it is assigned, the standard subtyping subsumption rule (Subs), the rules that relate the types of bit-level expressions to their sub-expressions, and the rules for arithmetic operations (A-Op).

A bit-level type $\bar{b}$ is called *single*, if it is of the form $\mathbf{0}^{\ell}\langle \mathsf{a}, \ell'\rangle$. For arithmetic operations, we require that the operands have the same type, which must be a single type, and the result expression must be within the corresponding number of bits. The requirement that the result of an arithmetic operation does not overflow the number of bits available in the block $\langle \mathsf{a}, \ell'\rangle$ can be established by either statically proving the absence of overflows or inserting run-time checks [7]. This requirement is stipulated by the predicate $\mathsf{Bound}(e, \ell)$ which states that $|e| \leq 2^{\ell}$, in the hypothesis of the rule for arithmetic operations A-Op.

For the rules Bit-And and Bit-Or, that describe the types of bitwise expressions in terms of their subexpressions we use the following notation. For a constant c, and $b \in \{0, 1\}$, let $\mathsf{Brk}(c, b)$ be the set of maximal consecutive b-intervals in c, that is, $\mathsf{Brk}(c, b) = \{[h_1 : l_1], \ldots, [h_k : l_l]\}$, where for each $i \in \{1, \ldots, k\}$, we have that the bits in the interval $[h_i : l_i]$ are all b, the $(l_i - 1)$−th and the $(h_i + 1)$−th bits (if they are in the range $[0, N-1]$) are $1 - b$, and all other bits of c are $1 - b$.

Sound typings capture the intuition of when bitvectors are correctly used in a program. It ensures the correct usage of packed fields within a bit-level (when the bit-level is used as a record), and it only allows arithmetic operations when there is effectively only one field of a record thus avoiding arithmetic overflows that go into the next fields. If the program can be soundly typed then we can translate the program into an equivalent program where the (packed) integers are replaced with records and the bitwise operations are replaced with the appropriate field selectors as described in Section 4.

Example 5: For the program `mget` from Figure 2, Figure 3(c) demonstrates a sound bit-level typing. These types correspond to the use of virtual addresses that are structured as a 20-bit index, a 10-bit offset, a dirty bit, and a permission bit. □

3. TYPE INFERENCE

The type inference problem for bitvector typings takes as input a program and generates a sound bitvector typing. We adopt a constraint based approach, where the type inference problem is solved by generating and solving a system of constraints from the program.

3.1 Constraints and Solutions

Constraints. We shall use type variables τ drawn from the set $\mathbb{T}$ to denote the (unknown) bitvector types of various program expressions. There are two kinds of constraints:

1. A *zero constraint* of the form:

$$\tau[i : j] = \mathbf{0}$$

   where $i, j \in \{0, \ldots, N-1\}$ are such that $N > i - j \geq 0$. Zero constraints stipulate that the projection of the bitvector type corresponding to τ to the interval $[i : j]$ be a sequence of $i - j + 1$ zero blocks.

2. An *inequality constraint* of the form:

$$\tau[i : j] \leq \tau'[i' : j']$$

   where $i, j, i', j' \in \{0, \ldots, N-1\}$ are integers such that $N > i-j = i'-j' \geq 0$. Inequality constraints stipulate that the values of τ, τ' be bitvector types $\bar{b}, \bar{b}'$ such that the projection of $\bar{b}$ to the interval $[i : j]$ be defined and a subtype of the projection of $\bar{b}'$ to the interval $[i' : j']$. We use $\tau \leq \tau'$ to abbreviate $\tau[N-1 : 0] \leq \tau'[N-1 : 0]$.

Solutions. A *type assignment* is a map T from $\mathbb{T}$ to $\mathsf{Typ}(N)$. An assignment T *satisfies the zero constraint* $\tau[i : j] = \mathbf{0}$ if $T(\tau)[i : j] = \mathbf{0}^{i-j+1}$. We say that an assignment T *satisfies the inequality constraint* $\tau[i : j] \leq \tau'[i' : j']$ if $T(\tau)[i : j] \leq T(\tau')[i' : j']$. We write $T \models c$ to say that T satisfies the constraint c. Notice that the assignment $\lambda\tau.\mathbf{0}^N$ trivially satisfies every constraint, but nevertheless will be useless for our purposes.

Let C be a set of zero and inequality constraints. We say that T is a *zero assignment* for C if:

1. T satisfies all zero constraints of C, and,

2. For every τ' and integer k such that $N > k \geq 0$, if $T(\tau')[k] = (\mathbf{0}, 1, 0)$ then either (a) there is a zero constraint $\tau'[i' : j'] = \mathbf{0}$ where $i' \geq k \geq j'$, or (b) there is an inequality constraint $\tau''[i'' : j''] \leq \tau'[i' : j']$ for some τ'', i'', j'' where $i' \geq k \geq j'$.

We say that T is a *principal zero assignment* for the constraints C if for all type variables τ, and all $N - 1 \geq i \geq 0$, we have $T(\tau)[i] = (\mathbf{0}, 1, 0)$ if there exists any zero assignment T' for C such that $T'(\tau)[i] = (\mathbf{0}, 1, 0)$. We say that T is a *solution* for the constraints C, written $T \models C$, if T is a principal zero assignment for C that satisfies every inequality constraint in C.

Intuitively, a principal zero assignment assigns a zero-block to an interval of a type variable if it can be inferred from the constraints that at run-time the segment will always have zeros. This may be either because of zero constraints which arise due to left shifts, right shifts or bitmasks, or because (transitively) *all* the values *assigned* to the corresponding expression are guaranteed to have zeros in that interval.

Example 6: [Constraints, Zero Assignments, Solutions] The two right columns of Figure 2 show sets of zero and inequality constraints, respectively. The first row of Figure 7 shows a zero assignment T_0 for the constraints of Figure 2. Notice that $T_0(\tau_{\mathtt{p\&1^{20}0^{12}}}) \equiv \mathbf{0}^{20}\langle \mathsf{q}, 10\rangle \mathbf{0}^2$, *i.e.*, the first twenty and last two bits (from the left) are zeros, as stipulated by the constraints $\tau_{\mathtt{p\&1^{20}0^{12}}}[31 : 12] = \mathbf{0}$ and $\tau_{\mathtt{p\&1^{10}0^2}}[1 : 0] = \mathbf{0}$. Also as there are no inequality constraints for $\tau_{\mathsf{tab\_elem}}$, we have $T_0(\tau_{\mathsf{tab\_elem}}) \equiv \langle \mathsf{m}, 32\rangle$, and similarly for $\tau_{\mathtt{b1}}$ there are no zero blocks. However, due to the bitmask, $T_0(\tau_{\mathtt{b1\&1^{30}0^2}})$, and hence $T_0(\tau_{\mathsf{base}})$ has two zero blocks, as the constraint $\tau_{\mathtt{b1\&1^{30}0^2}} \leq \tau_{\mathsf{base}}$ is the only inequality constraint with τ_{base} on the right hand side. Finally, τ_{a} only has two zero blocks in the $0-$th and $1st$ bits, as only those are "common" to τ_{base} and $\tau_{\mathtt{off}}$. The last row of Figure 7 shows a solution T w.r.t. the column variables for the constraints of Figure 2. As the assignment "agrees" with T_0 on the zero blocks, it is also a zero assignment. It is easy to check that every constraint is satisfied by the assignment. For example, notice as $\mathbf{0}^{20}\langle \mathsf{a}, 10\rangle \mathbf{0}^2 \leq \langle \mathsf{a}, 30\rangle \mathbf{0}^2$, the assignment satisfies the constraint $\tau_{\mathtt{off}} \leq \tau_{\mathsf{a}}$. □

Substitutions. A *substitution* is a map σ from $\mathbb{A} \setminus \{\mathbf{0}\}$ to $\mathsf{Typ}(N)$, such that $\sigma(\mathsf{a})$ contains no zero blocks. We write $[\mathsf{a} \mapsto \bar{b}]$ for the substitution $\lambda x.(x = \mathsf{a}\ ?\ \bar{b} : \langle x, N\rangle)$ which maps the name a to $\bar{b}$ and maps all other names x to a bitvector type with only one block $\langle x, N\rangle$ of size N. A substitution σ is extended to a map from blocks ($\mathbb{B}$) to bitvector types ($\mathbb{B}^+$) as follows:

$$\sigma(\langle \mathsf{a}, \ell\rangle) \equiv \langle \mathsf{b}, \ell'\rangle.\bar{b}[\ell - \ell' - 1 : 0]$$

where $\sigma(\mathsf{a}) = \bar{b}$ and $\bar{b}[\ell - 1] \equiv (\mathsf{b}, \ell', \cdot)$. Intuitively, the substitution $[\mathsf{a} \mapsto \bar{b}]$ maps the block $\langle \mathsf{a}, \ell\rangle$ to the sequence of blocks that is the suffix of $\bar{b}$ of size ℓ, *i.e.*, the sequence of blocks corresponding to the *rightmost* ℓ bits of $\bar{b}$.

Example 7: [Substitutions] Consider the bit-level type $\langle \mathsf{p}, 16\rangle$. When the substitution $[\mathsf{p} \mapsto \langle \mathsf{r}, 24\rangle.\langle \mathsf{s}, 8\rangle]$ is applied to it we get the bit-level type: $\langle \mathsf{r}, 8\rangle.\langle \mathsf{s}, 8\rangle$ corresponding to the rightmost 16 bits of the map image. Similarly, when the substitution $[\mathsf{q} \mapsto \langle \mathsf{t}, 24\rangle.\langle \mathsf{u}, 8\rangle]$ is applied to $\langle \mathsf{q}, 10\rangle$ we get the bit-level type $\langle \mathsf{t}, 2\rangle.\langle \mathsf{u}, 8\rangle$ corresponding to the last 8 bits of the map image. □

We extend the substitution σ to a map from bitvector types ($\mathbb{B}^+$) to bitvector types ($\mathbb{B}^+$) as:

$$\begin{array}{lcl} \sigma(\epsilon) & \equiv & \epsilon \\ \sigma(\langle \mathsf{a}, \ell\rangle.\bar{b}') & \equiv & \sigma(\langle \mathsf{a}, \ell\rangle).\sigma(\bar{b}') \end{array}$$

Finally, we extend substitutions to type assignments T as:

$$\sigma(T) \equiv \lambda\tau.\sigma(T(\tau))$$

The key property of substitutions is that they preserve constraint satisfaction.

Theorem 1. **[Substitution]** *For all sets of constraints C, type assignments T and substitutions σ, if $T \models C$ then $\sigma(T) \models C$.*

Example 8: [Substitution-Preservation] In the top half of Figure 3(b), we have an assignment T where that satisfies the constraint $\tau_1 \leq \tau_2$. The lower half shows an assignment T' which is the result of substituting q with $\bar{b}_{\mathsf{q}} \equiv \langle \mathsf{t}, 24\rangle\langle \mathsf{u}, 8\rangle$

and p with $\bar{b}_{\mathsf{p}} \equiv \langle \mathsf{r}, 24\rangle\langle \mathsf{s}, 8\rangle$. Note that $\langle \mathsf{p}, 16\rangle$ gets substituted with $\langle \mathsf{r}, 8\rangle\langle \mathsf{s}, 8\rangle$ corresponding to the suffix of size 16 of $\bar{b}_{\mathsf{p}}$. However, $\langle \mathsf{p}, 6\rangle$ is mapped to the rightmost 6 bits of $\bar{b}_{\mathsf{p}}$ *i.e.,* $\langle \mathsf{s}, 6\rangle$. Similarly, note that $\langle \mathsf{q}, 16\rangle$ gets substituted with $\langle \mathsf{t}, 8\rangle\langle \mathsf{u}, 8\rangle$ which is the suffix of size 16 of $\bar{b}_{\mathsf{q}}$. On the other hand, $\langle \mathsf{q}, 10\rangle$ gets mapped to the rightmost 10 bits of $\bar{b}_{\mathsf{q}}$, namely $\langle \mathsf{t}, 2\rangle\langle \mathsf{u}, 8\rangle$. Notice that the substituted assignment T' also satisfies the constraint $\tau_1 \leq \tau_2$. □

Principal Solutions. For two type assignments T, T', we say that $T \sqsubseteq T'$ if there exists a substitution σ such that $T' = \sigma(T)$. A solution T for a set of constraints C is called the *principal solution* if *for all* $T' \models C$, we have $T \sqsubseteq T'$.

In order to infer types, we will generate constraints on the types of each program expression, and then compute the principal solution for the constraints.

3.2 Constraint Generation

To generate constraints, we introduce for every expression e in the program, a type variable τ_e. For array variables x, we have two type variables, τ_{x_idx} and τ_{x_elem}. We then generate constraints on the type variables that capture the typing relationships between expressions and subexpressions shown in Figure 4.

Figure 5 shows the constraint generation rules from the program syntax. The constraint generation rules are given as a recursive procedure CGen that generates a set of constraints from a program. The procedure CGen uses the recursive procedures CGen_P and CGen_E as helpers to generate constraints from predicates and expressions respectively.

Theorem 2. *Let $C = \mathsf{CGen}(P)$ for a program P, and let T be a solution of C. Then the typing $\lambda e.T(\tau_e)$ is a sound bitvector typing of P.*

3.3 Inference Algorithm

We now show that every set of constraints has a principal solution by giving a two-step algorithm that computes the principal solution for a set of constraints. We give the conceptual algorithm in this section. We describe efficient graph-based data structures to implement this algorithm in the next section.

Step 1: Initial Zero Assignment. In the first step, using all the constraints, we determine which parts of the various bitvector types must be zero blocks. In particular, we compute using Algorithm T_0, a principal zero assignment T_0 such that for every $T \models C$, we have $T_0 \sqsubseteq T$. The Algorithm T_0 is a simple data-flow analysis algorithm that first processes each zero constraint $\tau[i:j]$ and for each such constraint, splitting the type for τ at positions i, j and setting the interval between the positions to $i-j+1$ zero blocks, and then propagates the zeros across inequality constraints until a fixpoint is reached. To get a principal zero assignment, a type variable τ appearing on the right hand side of an inequality constraint, gets an interval set to zeros only if *for each* inequality constraint where τ appears on the RHS, the corresponding LHS type variables have zeros in the corresponding intervals. All the remaining non-zero blocks have distinct names in the principal zero assignment. We omit a full description of the algorithm for space reasons.

Example 9: [Principal Zero Assignment] The top row of Figure 7 shows the principal solution to the constraints from Figure 2. Notice that the fourth zero constraint sets the interval $[1:0]$ for $\tau_{\mathsf{b1\&1^{30}0^2}}$ to zero, which is due to the bitmask operation. As the *only* inequality constraint with τ_{base} on the RHS is $\tau_{\mathsf{b1\&1^{30}0^2}} \leq \tau_{\mathsf{base}}$, the zeros in the interval $[1:0]$ are propagated to τ_{base} making its two least significant bits zeros. Similarly the constraints $\tau_{\mathsf{p\&1^{10}0^2}}[1:0] = \mathbf{0}$ and $\tau_{\mathsf{p\&1^{10}0^2}}[31:12] = \mathbf{0}$ cause the zero assignment $\mathbf{0}^{20}\langle \mathsf{q}, 10\rangle\mathbf{0}^2$ for $\tau_{\mathsf{p\&1^{10}0^2}}$, and the inequality constraint $\tau_{\mathsf{p\&1^{10}0^2}} \leq \tau_{\mathtt{off}}$, the only inequality constraint with $\tau_{\mathtt{off}}$ on the RHS, propagates the zeros to $\tau_{\mathtt{off}}$ giving it the type $\mathbf{0}^{20}\langle \mathsf{o}, 10\rangle\mathbf{0}^2$. □

Proposition 1. **[Principal Zero Assignment]** *For every set of zero and inequality constraints C, the procedure $\mathsf{T}_0(C)$ in time $O(|C|N)$ returns a principal zero assignment T_0 such that for every zero assignment T for C, we have $T_0 \sqsubseteq T$.*

Step 2: Refinement. In the second step, we iteratively refine the "principal" zero assignment T_0, until each of the inequality constraints are satisfied. This is done using the functions Split and Unify.

Algorithm 1 Split

Input: Type assignment T, Type Variable τ, Interval $[i:j]$
Output: A type assignment T
  if $T(\tau)[i+1] = (\mathsf{a}, \ell_2, \ell_1)$ and $\ell_1 > 0$ **then**
    $T := [\mathsf{a} \mapsto \langle \mathsf{b}_1, N-\ell_2\rangle\langle \mathsf{b}_2, \ell_2\rangle](T)$ $\{\mathsf{b}_1, \mathsf{b}_2 \text{ fresh}\}$
  if $T(\tau)[j] = (\mathsf{a}', \ell_2', \ell_1')$ and $\ell_1' > 0$ **then**
    $T := [\mathsf{a}' \mapsto \langle \mathsf{b}_1', N-\ell_2'\rangle\langle \mathsf{b}_2', \ell_2'\rangle](T)$ $\{\mathsf{b}_1', \mathsf{b}_2' \text{ fresh}\}$
  return T

$\mathsf{Split}(T, \tau, [i:j])$**.** The function Split shown in Algorithm 1 takes an assignment T, a constraint variable τ, and an interval $[i:j]$, and returns an assignment T' such that $T \sqsubseteq T'$ and $T'(\tau)[i:j]$ is defined, *i.e.,* $T'(\tau)$ has breaks at $i+1$ and j. Intuitively, function Split splits the bitvector type $T(\tau)$ at the end points of the interval. The new type assignment returns this split type for τ, and returns the type $T(\tau')$ for all other variables τ'.

Algorithm Split first finds the block fragment of $T(\tau)$ at position $i+1$. If there is already a break at position $i+1$, there is nothing to do. However, if there is no break at that position, *i.e.,* the block fragment is $(\mathsf{a}, \ell, \ell')$ and $\ell' > 0$, then a new break is introduced by substituting the block a at that position with $\langle \mathsf{b}_1, N-\ell\rangle\langle \mathsf{b}_2, \ell\rangle$ where $\mathsf{b}_1, \mathsf{b}_2$ are two fresh names. The resulting assignment is guaranteed to have a break at $i+1$. The same process is repeated (using the new assignment for τ with a break at $i+1$) for the position j. The algorithm returns the type assignment that results after the second substitution.

Example 10: [Split] The left panel of Figure 6 illustrates how $\mathsf{Split}(T, \tau_{\mathsf{p}}, [11:2])$ works on an assignment T where τ_{p} is mapped to $\langle \mathsf{p}, 32\rangle$. First, we enforce a break at position $11+1$. As $T(\tau_{\mathsf{p}})[11+1]$ is $(\mathsf{p}, 12, 20)$, we deduce there is no break and enforce it by substituting p with $\langle \mathsf{e}, 20\rangle\langle \mathsf{f}, 12\rangle$. Next, in the resulting assignment, we enforce a break at position 2. Now $T(\tau_{\mathsf{p}})[2]$ is $(\mathsf{f}, 2, 10)$ and so we substitute f with $\langle \mathsf{g}, 30\rangle\langle \mathsf{h}, 2\rangle$, and hence get an assignment T where $T(\tau)$ has breaks at $11+1$ and 2 and thus, where $T(\tau)[11:2]$ is defined. □

$\mathsf{Unify}(T, \tau[i:j] \leq \tau'[i':j'])$**.** The function Unify shown in Algorithm 2 takes a zero assignment T, and an inequality

(Expressions CGen_E)	x	$\{\}$
	$x[e]$	$\mathsf{CGen}_E(e) \cup \{\tau_e \leq \tau_{x\_\mathsf{idx}}\}$
	c	$\{\}$
	$e{<<}c$	$\mathsf{CGen}_E(e) \cup \{\tau_e[N-1-c:0] \leq \tau_{e<<c}[N-1:c], \tau_{e<<c}[c-1:0] = \mathbf{0}\}$
	$e{>>}c$	$\mathsf{CGen}_E(e) \cup \{\tau_e[N-1:c] \leq \tau_{e>>c}[N-1-c:0], \tau_{e>>c}[N-1:N-1-c] = \mathbf{0}\}$
	$e\&c$	$\mathsf{CGen}_E(e) \cup \{\tau_e[h_i:l_i] = \tau_{e\&c}[h_i:l_i] \mid [l_i,h_i] \in \mathsf{Brk}(c,1)\}$
		$\cup\{\tau_{e\&c}[h_i:l_i] = \mathbf{0} \mid [l_i, h_i \in \mathsf{Brk}(c,0)\}$
	$e \mid c$	$\mathsf{CGen}_E(e) \cup \{\tau_e[h_i:l_i] = \tau_{e\mid c}[h_i:l_i] \mid [l_i,h_i] \in \mathsf{Brk}(c,1)\}$
	$e_1 \oplus e_2$	$\mathsf{CGen}_E(e_1) \cup \mathsf{CGen}_E(e_2) \cup \{\tau_{e_1} \leq \tau_{e_1\oplus e_2}, \tau_{e_2} \leq \tau_{e_1\oplus e_2}\}$
(Predicates CGen_P)	$e_1 \leq e_2$	$\mathsf{CGen}_E(e_1) \cup \mathsf{CGen}_E(e_2) \cup \{\tau_{e_1} \leq \tau_{e_1\leq e_2}, \tau_{e_2} \leq \tau_{e_1\leq e_2}\}$
	$p_1 \wedge p_2$	$\mathsf{CGen}_P(p_1) \cup \mathsf{CGen}_P(p_2)$
	$p_1 \vee p_2$	$\mathsf{CGen}_P(p_1) \cup \mathsf{CGen}_P(p_2)$
	$\neg p$	$\mathsf{CGen}_P(p)$
(Statements CGen)	$x := e$	$\mathsf{CGen}_E(e) \cup \{\tau_e[N-1:0] \leq \tau_x[N-1:0]\}$
	$x[e_1] := e_2$	$\mathsf{CGen}_E(x[e_1]) \cup \mathsf{CGen}_E(e_2) \cup \{\tau_{e_2}[N-1:0] \leq \tau_{x\_\mathsf{elem}}[N-1:0]\}$
	$s_1; s_2$	$\mathsf{CGen}(s_1) \cup \mathsf{CGen}(s_2)$
	$if(p)thens_1elses_2$	$\mathsf{CGen}_P(p) \cup \mathsf{CGen}(s_1) \cup \mathsf{CGen}(s_2)$
	$while(p)s$	$\mathsf{CGen}_P(p) \cup \mathsf{CGen}(s)$

Figure 5: Constraint generation algorithm $\mathsf{CGen}(P)$

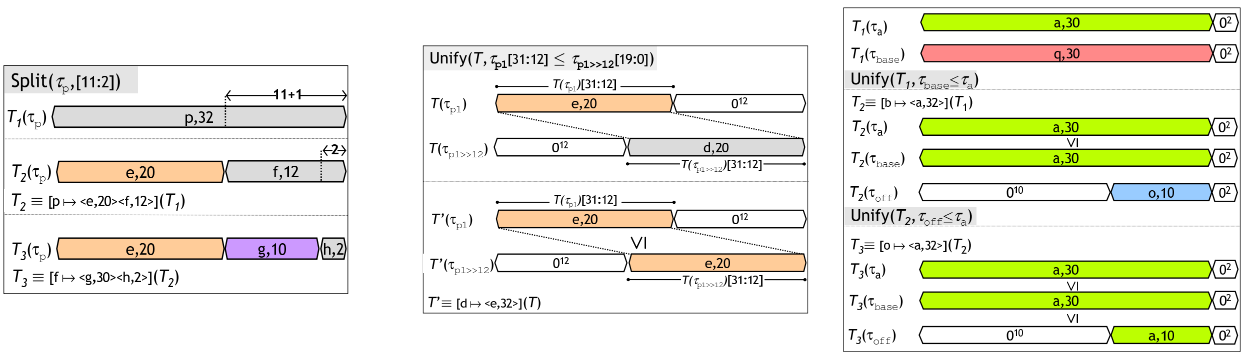

Figure 6: Split **and** Unify

constraint $c \equiv \tau[i:j] \leq \tau'[i':j']$ such that $T(\tau)[i:j]$ and $T(\tau)[i':j']$ are both defined, and returns an assignment T' such that $T \sqsubseteq T'$ and $T' \models c$. It proceeds by ensuring that the block at position i of $T(\tau)$ is the same as that of $T(\tau')$ at position i', by appropriate substitutions, and then recursing on the remaining suffixes.

EXAMPLE 11: [Unify] The middle panel of Figure 6 shows the first example of Unify. We wish to enforce the constraint $\tau_{\mathsf{p1}}[31:12] \leq \tau_{\mathsf{p1}>>12}[19:0]$, given an assignment T where the bitvectors corresponding to the LHS and RHS of the inequality are respectively $\langle \mathsf{d}, 20\rangle\langle \_, 12\rangle$ and $\langle \_, 12\rangle\langle \mathsf{e}, 20\rangle$. The block fragments of $T(\tau)$ (resp. $T(\tau')$) at position 31+1 (resp. $19+1$) are $(\mathsf{d}, 20, 0)$ and $(\mathsf{e}, 20, 0)$ and the second condition matches and the function substitutes d with $\langle \mathsf{e}, 32\rangle$ in T and recurses on $\tau[31-20:12] \leq \tau'[19-20:0]$ which trivially holds. A second example of Unify is shown in the right panel of Figure 6. Here, we wish to enforce the constraint $\tau_{\mathsf{off}}[31:0] \leq \tau_{\mathsf{a}}[31:0]$. The block fragment for the LHS are $(\mathbf{0}, 1, 0)$ (and so the first condition matches) until the recursion has shrunk the constraint's interval to $[11:0]$. At this point, the LHS block fragment is $(\mathsf{o}, 10, 0)$ and the RHS is $(\mathsf{a}, 10, 20)$. Hence, the second condition matches and we substitute o with $\langle \mathsf{a}, 32\rangle$ and thus enforce the constraint. If instead the RHS fragment had been $\mathsf{a}12\ell$, with $\ell > 10$ then the third condition would match and the function would have substituted a with some fresh $\langle \mathsf{b}, N-\ell\rangle\langle \mathsf{b}', \ell\rangle$, thus ensuring that in the subsequent recursion, the second condition matched and it sufficed to substitute b with o to enforce the constraint. $\square$

For an inequality constraint $c \equiv \tau[i:j] \leq \tau'[i':j']$ we introduce the following abbreviation:

$$\mathsf{Refine}(T, c) \equiv \mathsf{Unify}(\mathsf{Split}((\mathsf{Split}(T, \tau, [i:j]), \tau', [i':j'])), c)$$

PROPOSITION 2. **[Refine]** *For any assignment T and equality constraint c, $\mathsf{Refine}(T, c)$ returns an assignment T' such that (1) $T' \models c$, and, (2) for every $T \sqsubseteq T''$ such that $T'' \models c$, we have $T' \sqsubseteq T''$.*

Using the above, we can compute the principal solution for a set of constraints by iteratively refining the initial zero assignment. This is made precise in the algorithm shown in Figure 3.

PROPOSITION 3. **[Solve]** *For every set of inequality constraints C, and assignment T, the assignment $T' \equiv \mathsf{Solve}(C, T)$ is such that (1) $T' \models C$, and, (2) For every $T \sqsubseteq T''$ such that $T \sqsubseteq T''$, we have $T' \sqsubseteq T''$.*

Algorithm 2 Unify

Input: Type Assignment T, Constraint $c \equiv (\tau[i:j] \leq \tau'[i':j'])$
Output: A type assignment T' s.t. $T \sqsubseteq T'$ and $T' \models c$
  if $i < j$ **then**
    return T
  match $T(\tau)[i], T(\tau')[i']$ **with**
  | $(\mathbf{0}, 1, 0), \_ \longrightarrow \mathsf{Unify}(T, \tau[i-1:j] \leq \tau'[i'-1:j'])$
  | $\_, (\mathbf{0}, 1, 0) \longrightarrow$ **assert** false
  | $(\mathsf{a}, \ell, \_), (\mathsf{a}', \ell', \_)$ **when** $\ell' = \ell \longrightarrow$
    $c' := (\tau[i-\ell':j] \leq \tau[i'-\ell':j'])$
    $\mathsf{Unify}([\mathsf{a} \mapsto \langle \mathsf{a}', N\rangle](T), c')$
  | $(\mathsf{a}, \ell, \_), (\mathsf{a}', \ell', \_)$ **when** $\ell > \ell' \longrightarrow$
    $T' := [\mathsf{a} \mapsto \langle \mathsf{b}, N-\ell+\ell'\rangle\langle \mathsf{b}', \ell-\ell'\rangle](T)$ $\{\mathsf{b}, \mathsf{b}'$ fresh$\}$
    $\mathsf{Unify}(T', c)$
  | $(\mathsf{a}, \ell, \_), (\mathsf{a}', \ell', \_)$ **when** $\ell < \ell' \longrightarrow$
    $T' := [\mathsf{a}' \mapsto \langle \mathsf{b}, N-\ell'+\ell\rangle\langle \mathsf{b}', \ell'-\ell\rangle](T)$ $\{\mathsf{b}, \mathsf{b}'$ fresh$\}$
    $\mathsf{Unify}(T', c)$

Algorithm 3 Solve

Input: A Set of constraints C, Initial Assignment T_0
Output: Principal solution for C
  $T := T_0$
  while $C \neq \emptyset$ **do**
    pick and remove constraint c from C
    $T := \mathsf{Refine}(T, c)$
  return T

The theorem follows by induction on the size of C. Intuitively, the assignment after the first k iterations of the loop is the principal solution for the subset of k constraints of C that have been processed. The inductive step is proved using the fact that substitution preserves satisfaction 1 and the property of Refine given in Proposition 2. Using the above, and Proposition 1 we get the following result about principal solutions for bitvector constraints.

THEOREM 3. **[Principal Solutions]** *For every constraint set C containing the $C_{\leq}$ as inequality constraints,* $\mathsf{Solve}(C_{\leq}, \mathsf{T}_0(C))$ *is a principal solution for C and it is computed in time $O(|C|^2 N^2)$.*

EXAMPLE 12: [Solving Constraints] Figure 7 shows how the constraints C from the example of Figure 2 are solved. The top row shows the starting zero solution ($T_0 \equiv \mathsf{T}_0(C)$). Each of the following rows show how the assignment is refined so as to satisfy the inequality constraint shown in the first column of the row. The refines corresponding to the last two constraints are illustrated in the right panel of Figure 6. The solution for all the constraints is exactly the bitvector typing shown in Figure 3(c). □

3.4 Graph-based Constraint Solving

The inference algorithm in the previous section is always quadratic in the number of bits. We now present a graph data structure to represent type assignments T and which enables the efficient implementation of the primitives on which the procedures of the previous section are based: namely lookup ($T(\tau)$), finding the block-fragment at a given position ($T(\tau)[i]$), and substitution ($\sigma(T)$). In particular, the data structure allows us to perform an substitution in constant time, and thus the entire computation

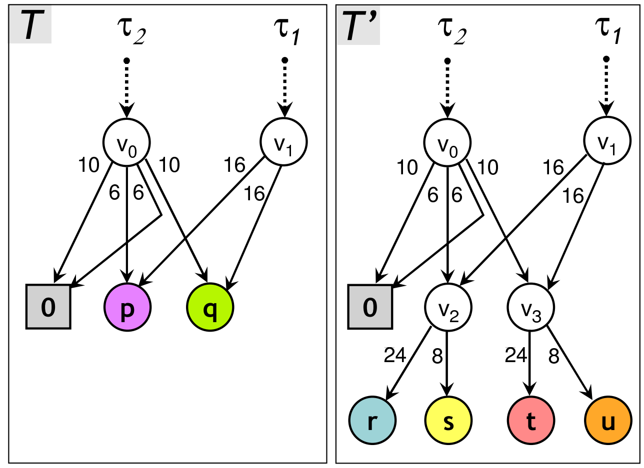

Figure 8: Assignment graphs (a) graph T, (b) graph T' obtained by expanding T

in time $O(|C|^2 \cdot W^2)$ where $W \leq N$ is the number of non-zero blocks in the final principal solution. Since for most examples, the number of non-zero blocks is much less than the total number of bits, we expect this algorithm to work better in practice, and this is the version we have implemented.

Data Structure. An *assignment graph* is a tuple $G \equiv (V, E, A, R)$ where:

- V is a set of vertices,
- E is an ordered edge relation, i.e., a map from V to sequences of successors $(V \times \mathbb{N})^*$ such that for every $v \in V$ the sequence $E(v)$ is either the empty sequence ϵ or a sequence $(v_1, \ell_1) \ldots (v_k, \ell_k)$ such that $\sum_{1 \leq j \leq k} \ell_j = N$,
- A is a map from V to $\mathbb{A} \cup \perp$, such that for every $v \in V$, we have $A(v) = \perp$ iff $E(v) \neq \epsilon$, and for every name a there is a unique vertex $A^{-1}(\mathsf{a}) \in V$ with that name, and
- R is a map from type variables τ to vertices V.

A vertex $v \in V$ is a leaf vertex if $E(v) = \epsilon$, it is an internal vertex otherwise. We shall only consider *acyclic* assignment graphs for which there do not exist vertices $v_1, \ldots, v_n$ and naturals $j_1, \ldots, j_{n-1}$ such that $(v_{i+1}, j_{i+1}) \in E(v_i)$ for all $i = 1, \ldots, n-1$, and $v_n = v_1$. For every internal vertex v, we can define the *successor-fragment of v at position i*, written $E(v)[i]$ in a manner analogous to sequences of blocks in Equation (1) by replacing names with vertices: that is, letting $E(v) = \bar{b}' \cdot (v_k, \ell_k)$, we have

$$E(v)[i] \equiv \begin{cases} \bar{b}'[i - \ell_k] & \text{if } \ell_k < i+1 \\ (v_k, i, \ell_k - i) & \text{o.w.} \end{cases}$$

Figure 8(a) shows an assignment graph for the types τ_1 and τ_2 from Figure 3. We draw the ordered edge relation as a sequence of edges labeled with numbers, where the ordering is represented left to right. We do not list leaf vertices with no incoming edges.

Computing $T(\tau)$. An internal vertex v of the assignment graph represents the bitvector type obtained by a preorder

Constraint	τ_{p}	$\tau_{\mathtt{p\&1^{10}0^2}}$	$\tau_{\mathtt{off}}$	$\tau_{\mathsf{tab\_elem}}$	$\tau_{\mathtt{b1}}$	$\tau_{\mathtt{b1\&1^{30}0^2}}$	τ_{base}	τ_{a}
T_0	$\langle \mathsf{p}, 32\rangle$	$\mathbf{0}^{20}\langle \mathsf{q}, 10\rangle\mathbf{0}^2$	$\mathbf{0}^{20}\langle \mathsf{o}, 10\rangle\mathbf{0}^2$	$\langle \mathsf{m}, 32\rangle$	$\langle \mathsf{n}, 32\rangle$	$\langle \mathsf{r}, 30\rangle\mathbf{0}^2$	$\langle \mathsf{b}, 30\rangle\mathbf{0}^2$	$\langle \mathsf{a}, 30\rangle\mathbf{0}^2$
c_1	$\langle \mathsf{e}, 20\rangle\langle \mathsf{q}, 10\rangle\langle \mathsf{c}, 2\rangle$	$\mathbf{0}^{20}\langle \mathsf{q}, 10\rangle\mathbf{0}^2$	“”	“”	“”	“”	“”	“”
c_2	$\langle \mathsf{e}, 20\rangle\langle \mathsf{q}, 10\rangle\mathbf{0}^2$	$\mathbf{0}^{20}\langle \mathsf{q}, 10\rangle\mathbf{0}^2$	$\mathbf{0}^{20}\langle \mathsf{q}, 10\rangle\mathbf{0}^2$	“”	“”	“”	“”	“”
c_3	“”	“”	“”	$\langle \mathsf{n}, 32\rangle$	$\langle \mathsf{n}, 32\rangle$	“”	“”	“”
c_4	“”	“”	“”	$\langle \mathsf{r}, 30\rangle\langle \mathsf{d}, 2\rangle$	$\langle \mathsf{r}, 30\rangle\langle \mathsf{d}, 2\rangle$	$\langle \mathsf{r}, 30\rangle\mathbf{0}^2$	“”	“”
c_5	“”	“”	“”	$\langle \mathsf{b}, 30\rangle\langle \mathsf{d}, 2\rangle$	$\langle \mathsf{b}, 30\rangle\langle \mathsf{d}, 2\rangle$	$\langle \mathsf{b}, 30\rangle\mathbf{0}^2$	$\langle \mathsf{b}, 30\rangle\mathbf{0}^2$	
c_6	“”	“”	“”	$\langle \mathsf{a}, 30\rangle\langle \mathsf{d}, 2\rangle$	$\langle \mathsf{a}, 30\rangle\langle \mathsf{d}, 2\rangle$	$\langle \mathsf{a}, 30\rangle\mathbf{0}^2$	$\langle \mathsf{a}, 30\rangle\mathbf{0}^2$	$\langle \mathsf{a}, 30\rangle\mathbf{0}^2$
c_7	$\langle \mathsf{e}, 20\rangle\langle \mathsf{a}, 10\rangle\langle \mathsf{c}, 2\rangle$	$\mathbf{0}^{20}\langle \mathsf{a}, 10\rangle\mathbf{0}^2$	$\mathbf{0}^{20}\langle \mathsf{a}, 10\rangle\mathbf{0}^2$	“”	“”	“”	“”	“”

Figure 7: Running Solve on Constraints from Figure 2. $c_1 = \tau_{\mathtt{p}}[11:2] \leq \tau_{\mathtt{p\&1^{10}0^2}}[11:2], c_2 = \tau_{\mathtt{p\&1^{10}0^2}} \leq \tau_{\mathtt{off}}, c_3 = \tau_{\mathsf{tab\_elem}} \leq \tau_{\mathtt{b1}}, c_4 = \tau_{\mathtt{b1}}[31:2] \leq \tau_{\mathtt{b1\&1^{30}0^2}}[31:2], c_5 = \tau_{\mathtt{b1\&1^{30}0^2}} \leq \tau_{\mathtt{base}}, c_6 = \tau_{\mathtt{base}} \leq \tau_{\mathsf{a}}, c_7 = \tau_{\mathtt{off}} \leq \tau_{\mathsf{a}}$.

traversal of the ordered edges in the subgraph rooted at v, and using the map A at leaf nodes. This is formalized by the function Seq defined below that takes a vertex v and a size ℓ and returns the sequence of blocks of size ℓ corresponding to the suffix of size ℓ that v represents:

$$\mathsf{Seq}(v,\ell) \equiv \begin{cases} \epsilon & \text{if } \ell = 0 \\ \langle A(v), \ell\rangle & \text{if } E(v) = \epsilon \\ \mathsf{Seq}(v',\ell') :: \mathsf{Seq}(v, \ell-\ell') & \text{if } (v', \ell', \_) = E(v)[\ell-1] \end{cases}$$

To find the sequence of blocks corresponding to a type variable τ, we compute $\mathsf{Seq}(R(\tau), N)$, i.e.,

$$T(\tau) \equiv \mathsf{Seq}(R(\tau), N)$$

For example, the node v_0 in Figure 8(a) represents the bitvector type $\mathbf{0}^{10}\langle \mathsf{p}, 6\rangle\mathbf{0}^6\langle \mathsf{q}, 10\rangle$, and the node v_1 represents the type $\langle \mathsf{p}, 16\rangle\langle \mathsf{q}, 16\rangle$. Notice that the assignment graph enables us to share the representations of the bitvector types.

Computing $T(\tau)[i]$. To find the block fragment at a position i for a type variable τ, we use the function Find defined below that takes a vertex and a size and returns the triple corresponding to the block fragment:

$$\mathsf{Find}(v,\ell) \equiv \left\{ \begin{array}{l} \text{if } E(v') = \epsilon \text{ then } (A(v), \ell', \ell'') \text{ else } \mathsf{Find}(v', \ell') \\ \text{where } (v', \ell', \ell'') = E(v)[\ell-1] \end{array} \right.$$

Now, $T(\tau)[i] \equiv$ if $E(R(\tau)) = \epsilon$ then $(A(R(\tau)), \ell, N-\ell)$ else $\mathsf{Find}(R(\tau), \ell)$.

Computing $[\mathsf{a} \mapsto \bar{b}](T)$. To compute the substitution of T using the map $[\mathsf{a} \mapsto \langle \mathsf{a}_1, \ell_1\rangle \ldots \langle \mathsf{a}_k, \ell_k\rangle]$ we find the vertex $v = A^{-1}(\mathsf{a})$, *i.e.,* corresponding to the name a, set $A(v)$ to $\perp$, and set $E(v)$ to the sequence $(A^{-1}(\mathsf{a}_1), \ell_1) \ldots (A^{-1}(\mathsf{a}_k), \ell_k)$. For example, Figure 8(b) shows a typing T' obtained from the typing T in Figure 8(a) by the substitution $[\mathsf{p} \mapsto \langle \mathsf{r}, 24\rangle\langle \mathsf{s}, 8\rangle][\mathsf{q} \mapsto \langle \mathsf{t}, 24\rangle\langle \mathsf{u}, 8\rangle]T$. Notice that $\tau_2 \leq \tau_1$ and $\sigma(\tau_2) \leq \sigma(\tau_1)$ for this substitution σ.

Combining the algorithm Solve from the previous section with the graph representation, we get the following.

THEOREM 4. *For every constraint set C, a principal solution for C can be computed in time $O(|C|^2W^2)$ where W is the number of non-zero blocks in the principal solution.*

4. TRANSLATION

Given a bitvector typing of a program, we convert integer variables on which bitwise operations are performed into structures, and syntactically replace each occurrence of bitwise operators in the program by assignment operations to appropriate fields of these structures. The result is a program where there are no bitwise operations. Specifically, for each bitvector type $\langle \mathsf{a}_1, \ell_1\rangle \ldots \langle \mathsf{a}_k, \ell_k\rangle$ assigned to an lvalue in the program, we introduce a structure with k unsigned integer fields $\mathsf{a}_1, \ldots, \mathsf{a}_k$ (we collapse the zero blocks into one field). (The conversion to unsigned integer fields is wasteful in terms of bits, however it is sufficient for our applications to verification. The translation algorithm can easily output packed structures with bit length annotations.) Each assignment x:=e in the program is then translated to a structure assignment by assigning to each field of τ_x the corresponding bit sequence from e, that is, if $\tau_x == \langle \mathsf{a}_1, \ell_1\rangle \ldots \langle \mathsf{a}_k, \ell_k\rangle$, we generate the assignments $\mathsf{x.a}_i = \mathsf{translate}(\mathsf{e}, [N-1-\sum_{j<i} \ell_j : N-1-\sum_{j\leq i} \ell_j]$ that identify the corresponding bits of e. The work of identifying the appropriate bitsection is performed by the recursive algorithm in Algorithm 9 that traverses the structure of the expression and extracts the relevant bits. It uses two support functions: the function $\mathsf{field}(x, [i:j])$ returns the field corresponding to bits $[i:j]$ by projecting the bitvector type for x to $[i:j]$. The function to_int calls a system defined function to convert bitvector structures to integers. The bitwise & and | operators are simplified away using the functions $\mathsf{simplify}_{\&}$ and $\mathsf{simplify}_{|}$ using the rules $x_i \& 0 = 0$ and $x|1 = 1$. By the typing rule, the second argument of simplify is always either all 1s or all 0s. For $\mathsf{simplify}_{\&}$, if the second argument is all 0, the function returns 0, otherwise the function returns the first argument. Dually, $\mathsf{simplify}_{|}$ returns the constant 1^{j-i} in case the second argument is all 1, and the first argument otherwise. Notice that a run-time assert is added for arithmetic operations. These asserts can be statically discharged by an auxiliary value-flow analysis.

x	$x.\mathsf{field}(x, [i:j])$
$x[e]$	$x[\mathsf{to\_int}(\tau_e, e)].\mathsf{field}(x\_\mathsf{elem}, [i:j])$
$e\&b$	$\mathsf{simplify}_{\&}(\mathsf{translate}(e, [i:j]), b[i:j])$
$e\vert b$	$\mathsf{simplify}_{\vert}(\mathsf{translate}(e, [i:j]), b[i:j])$
$e{<<}n$	$\mathsf{translate}(e, [i-n : j-n])$
$e{<<}n$	$\mathsf{translate}(e, [i+n : j+n])$
$e_1 \oplus e_2$	$\mathsf{assert}(\mathsf{Bound}(e_1 \oplus e_2, j-i))$ $\mathsf{translate}(e_1, [i:j]) \oplus \mathsf{translate}(e_2, [i:j])$

Figure 9: Translation Algorithm $\mathsf{translate}(\mathsf{e}, [i:j])$

Additionally, for array indices, and to cope with constructs not in our language (for example, bitwise operations with variables on the right hand side), we introduce conversion operations to and from unsigned integers for each structure introduced. These conversion functions perform the obvious translation from the structure representation to the corresponding integer and back. However, for our verification application, all we care about is that the functions are one-to-one. This was sufficient for proving our proper-

```
typedef u32 BOT_t;
typedef u32 k_t; typedef u32 s_t;
typedef u32 b_t; typedef u32 r_t;
struct __b0 { k_t k_1 ; BOT_t BOT_2 ; };
struct __b1 { BOT_t BOT_1 ; k_t k_2 ; };
struct __b2 { b_t b_1 ; r_t r_2 ; };
struct __b3 { b_t b_1 ; BOT_t BOT_2 ; };
struct __b4 { BOT_t BOT_1 ; b_t b_2 ; BOT_t BOT_3 ; };
struct __b5 { k_t k_1 ; b_t b_2 ; s_t s_3 ; };
u32 mget(struct __b5 p ) {
  struct __b0 p1; struct __b1 pte;
  struct __b2 b1, tab[1000];
  struct __b3 base, a; struct __b4 off;
  if (p.s_3==0) {
    error("Permission failure");
  } else {
    p1.k_1 = p.k_1; p1.BOT_2 = 0;
    pte.BOT_1 = 0; pte.k_2 = p1.k_1;
    b1.b_1 = tab[__b1_to_u32(pte)].b_1;
    b1.r_2 = tab[__b1_to_u32(pte)].r_2;
    base.b_1 = b1.b_1; base.BOT_2 = 0;
    off.BOT_1 = 0; off.b_2 = p.b_2; off.BOT_3 = 0;
    a.BOT_2 = 0; a.b_1 = base.b_1 + off.b_2;
    return m[__b3_to_u32(a)];
  }
}
```

Figure 10: Translated version

ties (but in general, one may require the exact definitions for these functions).

EXAMPLE 13: Figure 10 shows the translation of the `mget` function from Figure 2. The type inference algorithm constructs bitvector types for the variables and the translate procedure constructs C structures corresponding to these types, where each field is an unsigned 32 bit entry. The field `BOT_`n refers to a zero block in the type. For example, the structure `struct __b5` encodes the bitvector type $\langle \mathsf{k}, 20\rangle\langle \mathsf{b}, 11\rangle\langle \mathsf{s}, 1\rangle$ and `struct __b1` encodes the type $\mathbf{0}^{12}\langle \mathsf{k}, 20\rangle$. The assignments are translated to structure assignments. For array indices, we convert the bitvector structure to the corresponding integer by a system-defined function (to allow array indexing by the structures). (We have omitted the assertions for arithmetic operations.) □

5. EXPERIMENTS

5.1 Implementation and Performance

We have implemented the type inference algorithm with graph based data structures for C. We extended the algorithm of the previous sections to handle all the constructs of C, including pointers and fields. Similar to arrays, we provide a single type for a field. In addition, we handle bitwise operations with variable values (rather than constants). We implement a value set algorithm to handle the simple cases when a small set of constant values can flow into a variable. For others, we extend our translation algorithm to handle the cases outside the type system. We introduce conversion functions from integers to records and back, and cast values from one type to the other when the type system or the translation fails to handle a particular case.

We ran the tool on two sets of experiments. The first set, reported below, demonstrates that the constraint generation and type inference is efficient and scalable: in all cases, type inference took only a few seconds. The main objective of the experiments was in program understanding: we wanted to see if bitvector types produced data layout that was informally specified in the comments of the code, and to see how the types flow between program variables. The second set of experiments, described in the next section, show the applicability of the type inference to software verification.

General benchmarks. We ran the type inference on a set of Linux drivers as well as a set of programs from the Mediabench benchmark [15]. In each case, the inference took less than 5s. In several Mediabench benchmarks, bitwise operations are used for computations such as FFT. Such usage does not conform to our type system which breaks the variables involved into individual bits. However, for variables where part of the bits store data and part of the bits store permissions, the type system finds the appropriate type.

pmap. We applied our bitvector type inference algorithm to the code that sets up the page table in JOS, a small OS kernel written for education purposes [12]. The code was about 800 lines of C, with bitvector operations similar to the example code in Figure 2. The constraint generation took 0.03s and the type inference took 0.21s on a 1GHz Linux machine with 1GB RAM. There were 380 type constraints. Of the 257 lvalues, 97 had non-trivial bitvector types (the others were not involved in bitwise operations, and hence had types of the form $\langle \mathsf{p}, 32\rangle$ indicating that the 32 bits were not broken into subwords by the program). The type inference correctly identifies a physical address to have the type $\langle \mathsf{a}, 20\rangle\langle \mathsf{b}, 11\rangle\langle \mathsf{c}, 1\rangle$. Further, it identifies the variable `npage` that tracks the entries into the page table to have the type $\mathbf{0}^{12}\langle \mathsf{a}, 20\rangle$ thus showing that only the last 20 bits are relevant, and moreover, the top 20 bits from a virtual address flows into `npage` (since both have the type name a). Further, the type of the variable `cr3` (standing for the `%cr3` processor register) is also identified to be $\langle \mathsf{a}, 20\rangle\langle \mathsf{b}, 11\rangle\langle \mathsf{c}, 1\rangle$. This conformed to the hardware layer specification as well as the informal comments in the code.

5.2 Application to Verification

We ran the bitvector typing on two case studies for software verification: an implementation of the Mondrian memory protection [22] and a Linux device driver. In each case, we ran the translation algorithm on the original code (annotated with assertions), and then ran the software model checker Blast on the output. We describe each case study below.

Mondrian Memory Protection. Mondrian [22] is a fine-grained protection scheme that allows multiple protection domains to share memory. We ran the bitvector inference algorithm on an implementation of Mondrian annotated with 10 assertions provided to us by the author, E. Witchel. Mondrian extensively uses bit-packed structures to keep track of memory structures and permission bits. Since many of the assertions involved reasoning about bitvectors, Blast was able to verify only 3 of the assertions on the original code. The preprocessed code had 2687 lines. There were 775 constraints, and type inference took about 0.8s. The translated code introduced 18 different bitvector structures to represent the bitvector types inferred.

Table 1 summarizes the number of predicates found by Blast in each run, the number of predicates involving at least one bitvector field, as well as the run time for the verification. On the translated version, Blast was able to verify

Assertion	Predicates	Bit-vector	Time (s)	FP
1	6	0	9	
2	14	2	9	array
3	14	1	8	toint
4	25	5	13	
5	32	0	26	
6	64	5	29	toint
7	32	11	106	array
8	41	24	110	
9	27	10	55	
10	33	0	21	

Table 1: Mondrian experiments. Predicates is the total number of predicates found by BLAST, Bit-vector is the number of predicates that directly involved a bitvector field. FP is the reason Blast had a false positive.

6 of the 10 assertions and produced 4 false positives. We manually inspected the remaining assertions. Two of these involved data flow from (multi-level) arrays. Proving these assertions require reasoning about array data structures and more sophisticated alias analyses than Blast is currently capable of doing. Two other false positives involved loss of information because of a conversion to integers from bitvectors and back (these conversion functions were treated as uninterpreted functions by the analysis). We inspected the code manually, and found instances where bitvector operations were performed with variables, for example:

```
idx = 1 << mmpt->tab[lev];
```

However, the values that can flow into `mmpt->tab[lev]` are either 3 or 4, so we rewrote the code into a switch based on the value in the table. With this rewriting, we ran Blast again, and one of the false positives (assertion 6) went away (with 27 predicates, and 4 with bitvectors). For the second, we found a new false positive that again required reasoning about arrays.

`scull` Linux Driver. We also ran our procedure on the driver `scull` [4]. The Linux kernel uses a type `kdev_t` to hold major and minor device numbers. The type `kdev_t` is a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor number. The major and minor numbers are extracted using macros in the kernel that expand to

```
int type = (inode->i_rdev >> 20); // major
int num = (inode->i_rdev & 0xFFFFF) ; // minor
```

In a previous verification of this driver using Blast [11], these bitvector operations were translated away by hand. The type inference automatically identifies the type of the field as $\langle \mathsf{a}, 12\rangle\langle \mathsf{b}, 20\rangle$, and the translation procedure replaced the type `kdev_t` with a structure with two fields, `a` and `b`, and simplified the above code to

```
int type = (inode->i_rdev).a;
int num  = (inode->i_rdev).b;
```

This translation eliminated the bitvector operations, and Blast could check a set of five properties relating to the correctness of the driver.

Acknowledgments. We would like to thank Emmett Witchel for providing us with the Mondrian Memory Protection code and the assertions therein, and Ru-Gang Xu for attempting to use BLAST to verify the properties using the bit-reduction approach. We thank the anonymous reviewers for pointing out the closely related papers [19, 13]. This research was supported in part by the NSF grants CCF-0427202, CNS-0541606, and CCF-0546170.

6. REFERENCES

[1] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.
[2] C. Barrett, D. Dill, and J. Levitt. A decision procedure for bit-vector arithmetic. In *DAC 98: Design Automation Conference*, pages 522–527, 1998.
[3] E.M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS 04: Tools and Algorithms for the construction and analysis of systems*, LNCS 2988, pages 168–176. Springer, 2004.
[4] J. Corbet, G. Kroah-Hartman, and A. Rubini. *Linux device drivers, 3rd edition*. O'Reilly, 2005.
[5] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI 02: Programming Language Design and Implementation*, pages 57–68. ACM, 2002.
[6] D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
[7] C. Flanagan. Hybrid type checking. In *POPL 06: Principles of Programming Languages*, pages 245–256. ACM Press, 2006.
[8] J.S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI 02: Programming Language Design and Implementation*, pages 1–12. ACM, 2002.
[9] R. Gupta, E. Mehofer, and Y. Zhang. A representation for bit section based analysis and optimization. In *CC 02: Compiler Construction*, LNCS 2304, pages 62–77. Springer, 2002.
[10] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.
[11] R. Jhala and K.L. McMillan. Interpolant-based transition relation approximation. In *CAV 05: Computer-Aided Verification*, LNCS 3576, pages 39–51. Springer, 2005.
[12] JOS. Jos: An operating system kernel. http://pdos.csail.mit.edu/6.828/2005/overview.html.
[13] R. Komondoor, G. Ramalingam, J. Field, and S. Chandra. Dependent types for program understanding. In *TACAS 05: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 3440, pages 157–173. Springer, 2005.
[14] Daniel Kroening and Natasha Sharygina. Approximating predicate images for bit-vector logic. In *TACAS*, pages 242–256, 2006.
[15] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 97: IEEE/ACM Symposium on Microarchitecture*, pages 330–335. IEEE, 1997.
[16] B. Li and R. Gupta. Bit section instruction set extension of arm for embedded applications. In *CASES 02*, pages 69–78. ACM, 2002.
[17] M.O. Möller and H. Ruess. Solving bit-vector equations. In *FMCAD 98: Formal Methods in Computer-Aided Design*, LNCS 1522, pages 36–48. Springer, 1998.
[18] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *ICSE 97: International Conference on Software Engineering*, pages 338–348. ACM, 1997.
[19] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *POPL 99: Principles of Programming Languages*, pages 119–132. ACM, 1999.
[20] A. Stump, C.W. Barrett, and D.L. Dill. Cvc: A cooperating validity checker. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pages 500–504. Springer, 2002.
[21] S. Tallam and R. Gupta. Bitwidth aware global register allocation. In *POPL 03: Principles of Programming Languages*, pages 85–96. ACM, 2003.
[22] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *ASPLOS 02*. ACM, 2002.
[23] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL 05: Principles of Programming Languages*, pages 351–363. ACM, 2005.