Formalizing Representation Transformations: A Case Study of Bit Vector Types

KATHERINE PHILIP*, Portland State University, USA

1 INTRODUCTION & MOTIVATION

Representation transformation is a process that transforms the representation of data values while preserving their original meaning. This category of transformations includes both optimization techniques and compilation strategies for high-level features. **In this ongoing work, we perform a formal study of representation transformations, using a case study of *bit vector types*.**

Suppose we are working in a language in which primitive values include bit vectors of different widths. Such values are available in widely used languages. For example, C++ offers a collection of fixed-width `intN_t` types, for some implementation-defined set of integer widths `N` [1]. Other languages like LLVM (with `iN` types) allow (almost) arbitrary choices for `N`.

Let us introduce a single parametric type `Bit n`, where `n` is a type parameter that can be instantiated with any natural number to produce bit vectors of width `n`. Allowing arbitrary choices of `n` lets us store and manipulate appropriately-sized bit vector values for use cases such as bit fields (which are commonly used in low-level systems software to conform to hardware specifications or to conserve space). Users can also leverage the polymorphic nature of this type to write a single function definition that works on bit vectors of all widths.

The added flexibility of these *bit vector types* introduces several implementation challenges. First, we have values in the language that do not map directly to standard machine register sizes or single machine instructions. Therefore, we have to devise a compilation strategy that can provide a representation for these values, and generate appropriate primitive operations that can manipulate it. Second, we need to implement polymorphism in a way that allow us to perform the aforementioned representation and operation implementation generation steps. We explore these specific implementation challenges with bit vector types, with the goal establish a formal foundation that allows us to reason about the correctness of their use and implementation.

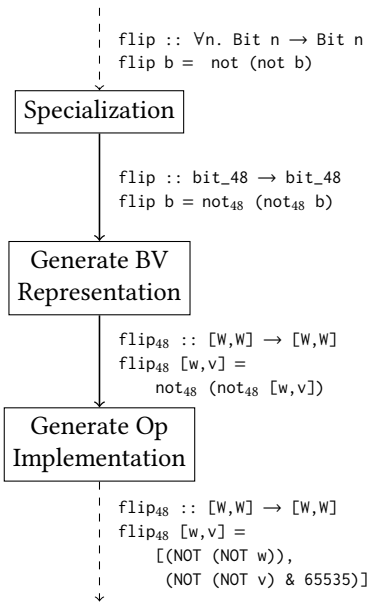

Figure 1. Pipeline for compiling bit vectors into a representation of 32-bit words $[W, \ldots, W]$

2 OUR APPROACH

We model the problem of compiling bit vector types as a series of representation transformation steps, as illustrated in Figure 1. First, we perform type-directed specialization to systematically generate monomorphic copies, each of which is specialized for a particular type. This step would specialize `Bit n` types into monomorphic bit vectors with defined widths. Then, we assign them a representation in terms of tuples of some *implementation type* (for example, in Figure 1, we use the 32-bit word type W). Once the bit vector representation is obtained, we can generate the appropriate implementations

*Graduate Student (PhD) Advised by Mark P. Jones. Email: kphilip@pdx.edu. ACM ID number: 9117643

for the primitive operations that manipulate them. These representation transformation steps modify the relation between terms and types in interesting ways which we wanted to capture in detail. To do so, we developed λ_I, a language that extends the standard STLC with let-expressions that bind identifiers to (potentially infinite) mappings between indices and terms. We provide a short overview of λ_I in Appendix A.

This indexed approach is not a new idea; We draw inspiration from prior work [4, 6, 7] that captured the details of polymorphic functions by treating them as a family of functions, indexed by type. *Type indexing* (type mapped over types) has also been utilized in the study of dependent types (for example, in Dependent ML [8]). Our approach allows the use of any index set (not just types). Additionally, we allow indices to map to any kind of terms (not just functions), even within a single map. This means, for instance, that for the same identifier, we can map some indices to a function, and map other indices to constant values. To the best of our knowledge, this combination of flexible selection of index set, and the mapping of indices to any terms is novel. We take advantage of this generality in our case study of bit vector types, and we believe that it will come useful when modeling other representation transformations as well.

3 MODELING COMPILATION OF BIT VECTOR TYPES

We model each stage that appears in Figure 1 as a family of representation transformation functions rep that takes a program in a source language and outputs an equivalent program in λ_I.

3.1 Specialization.

Figure 2 describes the core rep functions that specify how polymorphic programs in the ML-like source language L_{ML} would be represented as monomorphic programs in the target language λ_T, where index set $T = \{t \mid t \text{ is a monomorphic type in } L_{ML}\}$, given some substitution environment ρ. (The remaining functions, $\mathsf{repT}()$ for types, and $\mathsf{repE}()$ for expressions, perform a straightforward recursive translation of monomorphic types and terms.)

Representation of type schemes:

$$\begin{array}{lcl} \mathsf{repS}(\tau)\rho & = & \{\mathsf{repT}(\tau)\rho\} \\ \mathsf{repS}(\forall a.\sigma)\rho & = & \bigcup\{\mathsf{repS}(\sigma)\rho_{[\mathsf{repT}(\tau)\rho/a]} \mid \tau \in \mathrm{Type}_{ML}\} \end{array}$$

Representation of polymorphic expressions:

$$\begin{array}{lcl} \mathsf{repP}(e : \tau)\rho & = & \{\mathsf{repT}(\tau)\rho \mapsto \mathsf{repE}(e)\rho\} \\ \mathsf{repP}((\Lambda a.P) : (\forall a.\sigma))\rho & = & \bigcup \{\mathsf{repP}(P : \sigma)\rho_{[\mathsf{repT}(\tau)\rho/a]} \mid \tau \in \mathrm{Type}_{ML}\} \end{array}$$

Fig. 2. Excerpt of rep functions for modeling specialization

The translation functions for type schemes and polymorphic expressions work hand-in-hand to produce a *set* of monomorphic λ_T types and expressions to capture every possible instantiation of the polymorphic expression $P : \sigma$. We do so in a type-directed fashion, where we generate a corresponding expression for every L_{ML} type τ that applies, indexed by the λ_T representation of τ (i.e., the result of $\mathsf{repT}(\tau)\rho$). In the simple case with a monomorphic expression, where just a singleton type applies, we represent it as a singleton λ_T expression. In the case of a quantified polymorphic expression in the form of $\Lambda a.P : \forall a.\sigma$, we generate an expression for every L_{ML} type τ, instantiate all type variables a with τ, and union the resulting sets.

Generating a representation for a polymorphic definition P of type σ could be alternatively understood as the process of instantiating all possible $\tau \in [\![\sigma]\!]$ (the notation $[\![\sigma]\!]$ refers to the set of generic instances belonging to σ). So that, if $\tau \in [\![\sigma]\!]$, then $\mathsf{repT}(\tau)\rho \in \mathsf{repS}(\sigma)\rho$. We state this property as the following lemma:

Lemma 3.1. *For any ρ and σ:* $\mathsf{repS}(\sigma)\rho = \{\mathsf{repT}(\tau)\rho \mid \tau \in [\![\sigma]\!]\}$

3.2 Bit Vector Representation Generation

Figure 3 described the key rep functions to translate occurrences of monomorphic bit vector types bit_N (formatted differently to disambiguate them from the parameterized `Bit n` type) into a tuple that contains some chosen implementation type IT. The intention here is for IT to be a type that maps well into the target machine features. Whenever variables with bit vector types appear in an expression, we translate them to use the new tuple representation.

Representation of types:

$$\begin{array}{lcl} \mathsf{repT}(\chi) & = & \chi \\ \mathsf{repT}(\tau \to \tau') & = & \mathsf{repT}(\tau) \to \mathsf{repT}(\tau') \\ \mathsf{repT}(\text{bit}_N) & = & IT^{\lceil N/width(IT)\rceil} \end{array}$$

Representation of expressions (excerpt):

$$\mathsf{repE}(x_\tau) = \begin{cases} \tau = \text{bit}_N & S^{\lceil N/width(IT)\rceil} \\ \text{otherwise} & x_\tau \end{cases}$$

(where $S = genSym()$)

Fig. 3. Excerpt of rep functions for modeling bit vector representation generation

The presentation here is not the most general nor precise; Currently the representation must be computed based on a single choice of IT, as opposed to a more general formulation that could allow heterogeneous tuples that consists of a mix of ITs. In addition, this formulation currently does not capture the exact details of how should we position the bits inside an IT value, or how unused bits should be treated. We are working on an improved model that can better capture these details.

3.3 Future Work: Towards Operation Implementation Generation

Implementation generation works by replacing each use of primitive operations (that currently exists in the program as syntactic keywords) with the corresponding operation implementations, based on both the representations and bit widths (which we have preserved in the associated index of the operation, e.g., $\texttt{not}_{48}$) of the arguments. Bit vectors of different widths may map to the same representation (e.g., with IT = 32-bit W, bit vectors of widths 1 to 32 would all be mapped to *[W]*, bit vectors of widths 33 to 64 to *[W, W]*, and so on). For operations such as $\texttt{and}_N$, a single implementation may work for all bit vectors sharing the same representation. However, for other operations like $\texttt{concat}_{N,M,K}$, which concatenates two bit vectors of width N and M into a new bit vector of width $K = N + M$, each permutation of widths would require a unique implementation.

Given the large variety of primitive operations, compiler tools like mil-tools [5] or LLVM [2] typically handle implementation generation for each category of operations on a case-by-case basis. We are exploring a more systematic way to specify these implementation generation functions, ideally in a way that automatically produces a correctness result, perhaps by also generating a correctness proof alongside the implementation.

To obtain a full correctness result, we also need to reason about how these representation transformations steps transform the semantics of a program. While there are existing semantics for ML-like languages that we could use as a starting point, defining a suitable semantics for λ_I needs to be done carefully. Typical operational semantics for the regular form of a let expression let $v = e_1$ in e_2, would first evaluate e_1 down to a value, bind it to v, before proceeding to evaluate e_2. In λ_I, the e_1 position could potentially be occupied by an infinite family of functions. Naively adopting this semantics for λ_I might lead us to evaluate *all* expressions in the (potentially infinite) map, which would not match the intended meaning of the source program! To prescribe a well-behaved semantics for the potentially infinite nature of λ_I, we are exploring lazy semantics, as well as styles like Omnisemantics [3] that allows us to relate a starting state with a set of outcomes.

REFERENCES

[1] C++23 standard (ISO/IEC 14882:2023). 17.4 Integer types.

[2] Legalizer - LLVM Documentation. Legalizer - LLVM Documentation. https://llvm.org/docs/GlobalISel/Legalizer.html.

[3] Charguéraud, A., Chlipala, A., Erbsen, A., and Gruetter, S. Omnisemantics: Smooth handling of nondeterminism. *ACM Trans. Program. Lang. Syst. 45*, 1 (Mar. 2023).

[4] Jones, M. P. Partial evaluation for dictionary-free overloading. Technical Report.

[5] Jones, M. P., Bailey, J., and Cooper, T. R. MIL, a monadic intermediate language for implementing functional languages. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages* (New York, NY, USA, 2018), IFL '18, Association for Computing Machinery, p. 71–82.

[6] Ohori, A. A simple semantics for ML polymorphism. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (New York, NY, USA, 1989), FPCA '89, Association for Computing Machinery, p. 281–292.

[7] Ohori, A. Type-directed specialization of polymorphism. *Information and Computation 155*, 1 (1999), 64–107.

[8] Xi, H. Dependent ML: An approach to practical programming with dependent types. *Journal of Functional Programming 17*, 2 (2007), 215–286.

APPENDIX

A OVERVIEW OF λ_I

We give an abridged overview of λ_I. Every language in this family begins with the choice of an *index set*, I, whose members are referred to as *indices*. As a convenient shorthand, we will commonly replace the "I" in λ_I to denote the index set of the particular variant we're working with. There are two forms of program variables: λ-bound variables x_τ, that are annotated with a type, τ, and indexed variables v_i, that carry an associated index, i. The constructs available for function abstraction and application are typical as well. The **`let`** construct is more unusual because, instead of a simple let $v = e$ binding, it allows us to write an arbitrary definition, D, which is a (partial) mapping from index values to expressions $\mathrm{I} \twoheadrightarrow \mathrm{Expr}$. Partiality is important here because it allows let expressions that do not necessarily provide implementations for v at all possible types. The key typing rules that describe the behavior of indices are given below.

$$\frac{\Gamma, \Delta \vdash_D D \ : S \qquad \Gamma, \Delta \oplus \{v : S\} \vdash_{\lambda_I} e \ : \tau}{\Gamma, \Delta \vdash_{\lambda_I} \text{let } v = D \text{ in } e \ : \tau} \ [\lambda_I\text{–LetBind}] \qquad \frac{(v : S) \in \Delta \qquad i \in dom\ S}{\Gamma, \Delta \vdash_{\lambda_I} v_i \ : S(i)} \ [\lambda_I\text{–LetVar}]$$

$$\frac{dom\ D = dom\ S \qquad \forall i \in \mathrm{dom}\ D.\ \Gamma, \Delta \vdash_{\lambda_I} D(i) \ : S(i)}{\Gamma, \Delta \vdash_D D \ : S} \ [\lambda_I\text{–Defn}]$$

The [λ_I–LetBind] rule for let bindings checks every expression in D using the [λ_I–Defn] rule, where for every index in $dom\ D$, we ensure that $D(i)$ is well-typed, and we store the type in $S(i)$. Once S is constructed, we bind it into typing context Δ. In the rule for let-bound variables [λ_I–LetVar], we retrieve the partial function S associated with v from context Δ, then apply the index into S to finally obtain the expression type.