Filling a Niche: Using Spare Bits to Optimize Data Representations

NOAH LEV BARTELL-MANGEL[*], College of Marin, USA

1 INTRODUCTION

Many programming languages strive to provide both high developer productivity and efficient execution. One way to improve program performance without sacrificing developer productivity is by optimizing *data representation*. Algebraic datatypes (ADTs) provide an especially good opportunity to improve data representations, since they give compilers significant leeway in how the abstract type description can be represented.

A naïve representation of an ADT is as a pair of two fields: a tag, tracking which variant a particular value is, and an untagged union, holding the chosen variant's field. Of course, there is no need to track a tag when there is only one variant, so some compilers perform a simple data representation optimization that skips the tag when only one variant exists.

Still, much more powerful optimizations are possible. One of these more powerful optimizations is the use of "niches", or "spare bits", in types to store ADT tags. Niches[1] are ranges of invalid (*i.e.,* unused) values in a type's representation that can be used to store other data. Booleans are an example of a type with a niche; on many CPUs, the smallest unit of memory is a byte, while booleans only use the values `0` and `1`. As a result, the `bool` type possesses a niche in the range $[2, 255]$.

To illustrate the niche optimization, consider the Rust type `Option<bool>`. It has three possible values: `None`, `Some(false)`, and `Some(true)`. Thus, it should be possible to represent it using only one byte. With the naïve representation of ADTs, though, `Option<bool>` takes up two bytes of memory: one for `Option`'s tag and one for the boolean.

To enable compact data representations, Rust's compiler—and other compilers—record the niches in a type's representation so they can be used to encode tags of containing types. However, the algorithm has several subtleties (described in detail in §2), and to our knowledge, it has not before been formally described or proven correct.

Because of these subtleties, our goals with this research have been the following:

(1) implement the algorithm in a standalone artifact (available here[2]),
(2) formally describe the key parts of the algorithm (§2), and
(3) establish a proof of correctness (we sketch a proof in §3).

2 KEY IDEAS

The niche optimization algorithm has two main parts: layout and translation. The layout algorithm takes an abstract, high-level type and computes a *layout*, which describes how the type is to be represented. The translation step then uses computed layouts to generate low-level code, close to the level of LLVM IR, that takes advantage of niche representations.

For some high-level types, layout is trivial; *e.g.,* 64-bit integers get a layout that describes them as 64-bit integers. The interesting part of layout is when an ADT is analyzed.

[*]Undergraduate Student; Advisor: Dimitri Racordon
[1]The term "niche" comes from the terminology used in the Rust compiler.
[2]https://github.com/camelid/type-layout

Author's address: Noah Lev Bartell-Mangel, College of Marin, Kentfield, USA, noahlev.cs@gmail.com.

The first step in computing an ADT's layout is checking how many variants it has. If it has zero variants, we represent it as an empty record since there are no possible values. If it has one variant, we represent it "transparently" as its field; in that case, the type is a newtype wrapper. If it has more than one variant, we need to check if the type is the right shape for storing its tag in a niche. We call this attribute of a type "nicheability".

2.1 Nicheability

The conditions for nicheability are as follows:

(1) There are n nullary variants (fieldless variants).
(2) There is exactly one non-nullary variant (variant with a field).
(3) The non-nullary variant's field has a niche large enough to represent n distinct tag values.

Going back to our example from before, `Option<bool>` satisfies all three conditions: (1) `None` ($n = 1$), (2) `Some`, (3) `bool` only uses 0 and 1, so 2 can be used to represent the tag.

It should be possible to extend the optimization to work with types that have more than one non-nullary variant. In that case, the non-nullary variants would likely need to have niches with the same available values at the same memory offsets. Swift already performs a limited version of this extension, *e.g.,* when all of the fields are `Bool`-shaped. Rust does not perform this extension; we have chosen to start from Rust's simpler version as a basis for our work. The rules for nicheability must be considered carefully, or compilers will introduce undefined behavior into users' code via accesses of uninitialized memory.

The reader may well be wondering why only n tag values are needed, rather than $n+1$ to include the non-nullary variant. The reason is that the non-nullary variant is represented transparently as its field. So, for instance, `Some(false)` is represented as `0`. Thus, it does not require an explicit tag value to be reserved.

Once the layout algorithm has ensured conditions (1) and (2) hold, it must attempt to "extract" a niche from the non-nullary variant's field.

2.2 Niche extraction

Extracting niches proceeds as a straightforward recursive traversal of a type's layout. Niches are extracted from two places: ADT tags that do not use the niche encoding and pointers (since in Rust, references are required to be non-null)[3].

As the layout is traversed, the extraction algorithm tracks the path to the niche currently being considered. This path is later used by the translator so it knows how to find the tag in an ADT's value, which it needs to translate pattern matches. The structure of these "tag paths" is much like the paths described in [3].

If a niche large enough to hold the tag is found, the layout where the niche was found is updated to mark the used part of the niche as claimed; and the tag path and extracted niche values are returned.

3 PATHS TO CORRECTNESS

Our third goal in pursuing this research is establishing a proof of correctness for the optimization. In this section, we sketch a high-level overview of the proof. We plan to mechanize this proof in Coq in future work.

This optimization is trickier than standard optimizations to prove correct since it affects data representation. For standard optimizations, *e.g.,* constant propagation, the values resulting from running the unoptimized and optimized programs can be proven identical. However, the niche

[3]Note that our algorithm can be configured easily, so it can be used for languages that allow null pointers too.

optimization causes the unoptimized and optimized programs to produce *different values*. For example, the unoptimized runtime value for `Some(true)` is $(1, 1)$ (assuming `Some`'s tag is 1), while the optimized value is just 1. These values are not equal! Yet, they *are* isomorphic; there is a one-to-one mapping between unoptimized and optimized values.

To "witness" this isomorphism, we need to define a decoding function from unoptimized and optimized low-level values to high-level values. The decoding function is just a recursive transformation parametrized by the layout used for the value.

Now that we have a decoding function, we need to prove that the composition of layout, translation, evaluation, and decoding applied to a high-level value returns the input high-level value, regardless of whether the optimization is enabled.

4 RELATED WORK

4.1 In research

Special cases of the niche optimization, where only null-pointer niches are used, have been briefly mentioned in several papers [1, 2]. TIL appears to have supported a version of the optimization which the authors called "constructor flattening" that was somewhat more general than just null pointers [4, p. 183]. Nonetheless, it does not seem that TIL used niches present in tags (*e.g.,* `bool`'s niche), as the full niche optimization does. To our knowledge, the optimization has never been described in the literature in the formal, general version we present.

4.2 In practice

As mentioned before, several existing compilers already perform (variants of) this optimization. That is a major reason for our interest in formalizing the optimization, to ensure it is actually correct. We hope our planned correctness proof will enable expanding the optimization since compiler developers can be more confident that they did not miss hidden problems. In addition, we hope that the succinct, abstract description we provided will help with implementing this optimization in more compilers.

The Rust compiler performs the full version of the optimization described here, calling it the "niche optimization" or "niche-filling optimization". The Swift compiler implements a variation of the optimization that is less powerful in some ways and more in others. As of `swiftc 5.5`, it misses the opportunity to optimize types like `Either<(), Bool>`, which should be equivalent to `Optional<Bool>`; meanwhile, Swift is able to optimize some ADTs with multiple non-nullary variants, unlike Rust. The nomenclature used in Swift is "spare bits optimization".

5 FUTURE WORK

Our ultimate goal with this research is to ensure the correctness of the niche optimization. To that end, we plan to implement our proof sketch from §3 in the Coq proof assistant. Once we have implemented our proof, we aim to extend the optimization and its proof to multiple non-nullary variants.

ACKNOWLEDGMENTS

Thank you to Dimitri Racordon and Richard Eisenberg for their helpful advice and suggestions.

REFERENCES

[1] Andrew W. Appel and David B. MacQueen. 1987. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, Gilles Kahn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 301–324. https://doi.org/10.1007/3-540-18317-5_17

[2] Luca Cardelli. 1984. Compiling a Functional Language. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA) *(LFP '84)*. Association for Computing Machinery, New York, NY, USA, 208–217. https://doi.org/10.1145/800055.802037

[3] Kevin Scott and Norman Ramsey. 2000. *When Do Match-Compilation Heuristics Matter?* Technical Report CS-2000-13. Department of Computer Science, University of Virginia.

[4] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. 1996. TIL: A Type-Directed Optimizing Compiler for ML. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 181–192. https://doi.org/10.1145/231379.231414