Array shape inference via the $\forall^*\exists^*.(\wedge)$ fragment of the first-order free monoid theory

Justin Slepak

March 31, 2018

Abstract

The rank-polymorphic array language Remora uses restricted dependent types in the style of Dependent ML to describe the shape of array data, which in turn drives the control structure of the program. Type inference for DML-style a type system requires a decision procedure for the theory of the language of type indices. Array shapes can be described using the free monoid over the natural numbers, and the existential fragment of this theory is known to be decidable. Type inference requires a larger fragment of the theory, with universal quantifiers accounting for variables in the environment and existential quantifiers identifying type indices which must be chosen at a call site. Here, the problem of finding a witness for this larger fragment is reduced to finding a satisfying assignment in the existential fragment.

1 Dependent types

Dependent types allow program terms to appear as arguments to type constructors. The classic example is using values of type `Nat` as an argument to a `List` type, specifying the list's length. A function can be polymorphic over terms used within its input and output types. For example, the `zip` function, which builds pairs of corresponding elements in a pair of lists, has the type

```
Π n:Nat, T:Type.
   (List n T, List n T) -> List n (T, T)
```

Π serves as a universal quantifier. The dependent arguments `n` and `T` stand for the expected length and element types for the two input lists. The type enforces the rule that we can only `zip` lists of the same length. In order to call `zip` on two lists, we must first provide some `Nat`-typed argument which accurately describes their length. It is then up to the type checker to ensure that the argument given for `n` actually is equal to the length of each input list, which is itself derived from the list's type.

Unfortunately, this check can be quite costly because `zip`'s length argument can still be any arbitrary term, performing any arbitrary computation, as long as it has type `Nat`. To reduce the difficulty of checking dependent types, Xi [4] design Dependent ML, in which dependent types are indexed by terms in a restricted index language, rather than having the entire programming language available. The design of Dependent ML is somewhat agnostic as to the details of the index language, though it does require a decision procedure for that language in order for type checking to work. Even a pared down index language can still offer expressiveness. For example, an index language based on Presburger arithmetic, with just natural numbers and addition, can still use our earlier type for `zip`, as well as many other common list-processing functions such as `take`, `drop`, `append`, and `fold`. Index variables can still be introduced into the environment by a λ-like binder, just like term and type variables in System F. Checking index equality, amounts to checking the validity of an equality whose variables are the index variables bound in the current environment.

In the case of Remora, an array-oriented language with implicit rank polymorphism, the index language includes natural numbers and addition and also sequences of natural numbers with an append operator. Remora's index theory extends Presburger arithmetic to include the free monoid over $\mathbb{N}$—the algebra corresponding to finite sequences of natural numbers, with an associative "append" operator.

2 Trickier quantification

Type checking alone is insufficient for a practical programming language when polymorphism permits detailed descriptions of functions' behavior. The extra detail explained in dependent function types is difficult to leverage when the programmer must explicitly specify index arguments to each function call. Ideally, type inference should select the index arguments automatically. Type checking effectively considers a formula of the form $\forall \overrightarrow{x}.(\bigwedge_{i=1}^{n} s_i = t_i)$, asking whether indices are guaranteed to be equal no matter what value the index variables in the environment take on. Type inference has indices it must *choose* in a way that guarantees their compatibility with any possible value for the environment's index variables. The form for type inference's constraint is $\forall \overrightarrow{x}.\exists \overrightarrow{y}.(\bigwedge_{i=1}^{n} s_i = t_i)$. What we want as a solution is what we should choose as index arguments $\overrightarrow{y}$, written in terms of the index variables $\overrightarrow{x}$ we already have in the environment. This is a tighter restriction on the form of a solution than just having a function mapping variables to terms. We require a witness whose interpretation of each existential variable is built solely by appending generators and universal variables.

Furia's survey [2] notes that both the $\forall^*$ and $\exists^*$ fragments are decidable, while the $\forall^*\exists^*$ and $\exists^*\forall^*$ fragments are not. The decision procedure for the $\exists^*$ fragment is based on Makanin's algorithm [3], and we show here how to reduce a $\forall^*\exists^*.(\wedge)$ formula to an $\exists^*.(\wedge)$ formula.

The rough intuition for solving an equation in a free monoid is to find uses of concrete generators on both sides (as opposed to variables) and consider the ways they might be aligned with each other in a solution. We then turn the original equation into a conjunction of smaller equations, one for each segment between aligned generators. For example, consider the equation $x0x1y = y10x$. We might align the two 0s *or* the two 1s. If we instead align 1s, we get $x0x = y1 \wedge y = 0x$. We might instead align the 0s, or decide that the right side's 01 fits between the left side's 0 and 1 or completely off to the left or right of it. In any equation, there are finitely many possible alignments, each generating a conjunction of equations on syntactically smaller terms.

Existential variables might turn out to partially overlap, such as solving $1x0 = y10$ with $x = 011, y = 101$. In this case, x occupies positions 1 through 3, and y occupies positions 0 through 2. However, *no existential can partially overlap a generator*. This is the same rule we would need if universal variables appeared in an equation. An existential variable partially overlapping a universal variable leads to a situation where the universal variable might take on a value incompatible with the existential overlapping it. For example, with the formula $\forall a.\exists e.a0 = 1e$, e must cover all but the leftmost position of a. Even the ability to choose a value of e based on a deep inspection of a doesn't protect us from the case where a starts with 0 instead of 1. So this formula is not true. If we were able to make a complete overlap, as in $\forall a.\exists e, f.a0 = fe$, we would have a solution: $e = 0, f = a$ (there are other solutions too). Solving the equation relies on the option of making an existential variable fully overlap a.

The similar behavior of generators and universal variables is the basis for transforming $\forall^*\exists^*.(\wedge)$ formulas over a countably generated free monoid into equivalent $\exists^*.(\wedge)$ formulas over a free monoid with a few more generators. Suppose we are given a set $\mathcal{G}$ and a formula $\forall\overrightarrow{a}.\exists\overrightarrow{e}.\bigwedge(s = t)$ on the free monoid generated by $\mathcal{G}$. What we want is a witness $\mathcal{A}$ for the original formula—a mapping from the existential variables $\overrightarrow{e}$ to terms from the free monoid on $\mathcal{G}$.

To translate into the $\exists^*$ fragment, we augment $\mathcal{G}$ with a fresh elements $\overrightarrow{g}$, one g_i for each universal variable a_i. Call this new generator set $\mathcal{G}' = \mathcal{G} \cup \{\overrightarrow{g}\}$. We then turn the original formula to $\exists\overrightarrow{e}.(s = t)[\overrightarrow{a \mapsto g}]$.

Let $\mathcal{A}'$, mapping existential variables to closed terms from the free monoid on $\mathcal{G}'$, be a witness for $\exists\overrightarrow{e}.\bigwedge(s = t)[\overrightarrow{a \mapsto g}]$. Then we can use $\mathcal{A}(e) = \mathcal{A}'(e)[g \mapsto a]$ as a witness for the original formula[1]. On the existential-only side, our witness-substituted formula becomes $\bigwedge(s[\overrightarrow{a \mapsto g}, \overrightarrow{e \mapsto \mathcal{A}'(e)}] = t[\overrightarrow{a \mapsto g}, \overrightarrow{e \mapsto \mathcal{A}'(e)}])$. Consider an arbitrary equality from this post-substitution conjunction. It must be satisfied by the witness $\mathcal{A}'$, which means the two sides must be identical sequences of generators[2]. Since we have syntactically equal terms on the left and right, back-translating this by replacing each occurrence of g by a must produce syntactically equal terms too. So our constructed $\mathcal{A}$ serves as a witness for the untranslated equality. Since $\mathcal{A}'$ satisfies each of the translated equalities, $\mathcal{A}$ must satisfy each of the original equalities.

Consider the earlier example: Let $\varphi = \forall a.\exists e, f.a0 = fe$ with generators $\{0, 1\}$. We would translate this by extending the generator set to $\{0, 1, 2\}$ and

[1]As an abuse of notation, we are using "substitution" to replace a generator with a variable rather than the other way around

[2]Working in the *free* monoid means nothing else can be equal.

rewriting the formula as $\varphi' = \exists e, f.20 = fe$. Each solution for φ' corresponds to a witness for φ. If we choose $e = 0, f = 2$, we get our earlier solution by back-translating. The other solutions $e = 20, f = \varepsilon$ and $e = \varepsilon, f = 20$ correspond to the other possible φ witnesses.

Now suppose $\varphi = \forall a.\exists \overrightarrow{e}. \bigwedge(s = t)$ is true. Then a witness $\mathcal{A}$ which only appends generators and universal variables can be translated into $\mathcal{A}'$ which serves as a witness for $\varphi' = \exists \overrightarrow{e}. \bigwedge(s = t)[\overrightarrow{a \mapsto g}]$ (with fresh generators $\overrightarrow{g}$), by defining $\mathcal{A}'(e) = \mathcal{A}(e)[\overrightarrow{a \mapsto g}]$.

This strategy depends on the property that valid $\forall^*\exists^*(\wedge)$ formulas have witnesses that fit a very restricted format. Introducing the $\vee$ connective allows a formula to require a more complicated witness. Consider the formula $\forall w.\exists x.w = ww \vee w = 0x$ where $\mathcal{G} = \{0\}$. A witness for this must choose x based on the internal structure of w. If $w = \varepsilon$, then the value of x does not matter. Otherwise w must have a 0 as its leftmost component with k more 0s following it, so we choose $x = 0^k$. From an expressiveness point of view, disjunction allows us to constrain the internal structure of universal variables. This capability is used by Durnev [1] to show the undecidability of the $\forall^1\exists^3.(\wedge\vee)$ fragment. Any single equality which tries to constrain a universal like this is not valid, so neither is any such conjunction of equalities. A valid disjunction, on the other hand, can be made up of individually non-valid formulas, so our translation creates false negatives.

References

[1] Durnev, V. G. Undecidability of the positive 3-theory of a free semigroup. *Siberian Mathematical Journal 36*, 5 (1995), 917–929.

[2] Furia, C. A. Whats decidable about sequences? In *International Symposium on Automated Technology for Verification and Analysis* (2010), Springer, pp. 128–142.

[3] Makanin, G. S. The problem of solvability of equations in a free semigroup. *Sbornik: Mathematics 32*, 2 (1977), 129–198.

[4] Xi, H. *Dependent types in practical programming.* PhD thesis, Pittsburgh, PA, USA, 1998. AAI9918624.