Dsolve: Safety Verification via Liquid Types*

Ming Kawaguchi, Patrick M. Rondon, and Ranjit Jhala

University of California, San Diego
{mwookawa,prondon,jhala}@cs.ucsd.edu

Abstract. We present DSOLVE, a verification tool for OCAML. DSOLVE automates verification by inferring "Liquid" refinement types that are expressive enough to verify a variety of complex safety properties.

1 Overview

Refinement types are a means of expressing rich program invariants by combining classical types with logical predicates. For example, using refinement types, one can express the fact that `x` is an array of positive integers by stating that `x` has the type $\{\nu : \texttt{int} \mid 0 < \nu\}$ `array`. While refinement types have been shown to be a powerful technique for verifying higher-order functional programs [1–4], refinement type systems have previously been difficult to use because of a high programmer annotation burden.

We present DSOLVE, a tool that automates the verification of safety properties of OCAML programs by inferring refinement types. Using DSOLVE, we were able to verify properties of real-world OCAML programs as diverse as array bounds safety and correctness of sorting and tree-balancing algorithms while incurring a modest overhead in terms of the annotations and hints required for verification. Further, we were able to use the refinement types inferred by DSOLVE on buggy programs to diagnose and correct the problems, demonstrating its value as a tool for program understanding.

DSOLVE works by inferring *Liquid Types*, which are refinement types whose refinements are conjunctions of predicates taken from a user-provided finite set of *logical qualifiers*. Each logical qualifier is a predicate over the program variables and the special value variable ν, which is used to refer to values of the refined type. The Liquid Type restriction makes inference tractable while still retaining enough expressiveness to verify safety properties of real-world OCAML programs.

2 Example

In this section, we illustrate Liquid Types and show how DSOLVE is able to verify a polymorphic, higher-order, array-manipulating program, shown in Figure 1. We will show how DSOLVE statically verifies the safety of the program's array

* This work was supported by NSF grants CCF-0644361, CNS-0720802, CCF-0702603, and a gift from Microsoft Research.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 123–126, 2010.

```
1  let rec foldn m n b g =
2    if m < n then foldn (m+1) n (g m b) g else b
3
4  let weighted_avg x w =
5    if Array.length x > 0 && Array.length x = Array.length w then
6      let b = x.(0) * w.(0), w.(0) in
7      let f = fun i (sum, n) -> sum + x.(i) * w.(i), n + w.(i) in
8      let sum, n = foldn 1 (Array.length x) b f in
9        sum / n
10   else
11     assert false
12
13 let _ = weighted_avg [|10; 15; 20|] [|1; 1; 1|]
```

Fig. 1. An Example OCaml Program

accesses and division operation (i.e., that the array indices are within bounds on lines 5, 6, 7 and that the denominator is non-zero on line 9).

Qualifiers. DSOLVE takes a set of logical qualifiers as input from the user, which it uses to construct refinement types. Assume that the user has supplied the following qualifiers: $\{0 < \nu,\ \star \leq \nu,\ \nu < \star,\ \nu < \texttt{len}\ \star\}$, where the uninterpreted function symbol `len` is an abbreviation for `Array.length` and $\star$ denotes a “wildcard” that is instantiated with program variables.

The higher-order function `foldn` folds over the integers from `m` to `n`. DSOLVE infers that `foldn` calls `g` with values between `m` and `n`, that is, `foldn` has type

$$\texttt{m}{:}\texttt{int} \rightarrow \texttt{n}{:}\texttt{int} \rightarrow\ \alpha \rightarrow (\texttt{g}{:}\{\nu : \texttt{int} \mid \texttt{m} \leq \nu \wedge \nu < \texttt{n}\} \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha.$$

This is a Liquid Type since the refinement for the input of `g` is the conjunction of $\star \leq \nu$ and $\nu < \star$, where the wildcards are instantiated with `m` and `n`, respectively.

The function `weighted_avg` uses `foldn` to compute the weighted average of the array `x`’s values using the corresponding weights in array `w`. From the call on line 13, DSOLVE infers that `x` and `w` have the same positive length and that `w` contains only positive entries. DSOLVE then determines that the condition on line 5 is always true, so the assertion on line 11 never executes, and that the array accesses on line 6 are within bounds. Using the types of `foldn` and `x`, DSOLVE also determines that function `f` on line 7 has type

$$\texttt{f} :: \{\nu : \texttt{int} \mid 0 < \nu \wedge \nu < \texttt{len x} \wedge \nu < \texttt{len w}\} \rightarrow \texttt{int} * \texttt{pos} \rightarrow \texttt{int} * \texttt{pos}.$$

where `pos` abbreviates $\{\nu : \texttt{int} \mid 0 < \nu\}$. Thus, DSOLVE determines that all accesses to arrays `x` and `w` within `f` are safe. Finally, DSOLVE determines from `f`’s type that `n` is always positive, and so the division on line 9 is safe.

Modular Verification. DSOLVE verified the safety of this program using whole-program analysis, i.e., by analyzing the call to `weighted_avg` on line 13. The programmer could also verify the above program by writing the following interface specification (or “contract”) for `weighted_avg`:

$$\texttt{x}{:}\{\nu : \texttt{int array} \mid \texttt{len}\ \nu > 0\} \rightarrow \{\nu : \texttt{pos array} \mid \texttt{len}\ \nu = \texttt{len x}\} \rightarrow \texttt{int}$$

DSOLVE can use these specifications to verify modules without driver code and also to verify a module’s clients.

3 Tool

Architecture. DSOLVE is divided into the following three phases, described in detail in [5, 6]. First, the OCAML compiler's parser and typechecker are used to translate the input program to a typed AST; this phase also parses the module's refinement type specification. Second, the typed AST is traversed to generate a set of subtyping constraints over templates that represent the potentially unknown refinement types of the program expressions. Third, the constraints are solved using predicate abstraction over a finite set of predicates generated from user-provided logical qualifiers. This pass uses the Z3 SMT solver [7] to discharge logical implications corresponding to the subtyping constraints. If the constraints can be satisfied, the program is deemed safe. Otherwise, DSOLVE reports a type error and the lines in the original source program that yielded the unsatisfiable constraints.

DSOLVE is conservative. If an error is reported, it may be because the program is unsafe, or because the set of qualifiers provided was insufficient, or because the invariants needed to prove safety cannot be expressed within our refinement type system.

Input. DSOLVE takes as input a source (`.ml`) file containing an OCAML program, an interface (`.mlq`) file containing a refinement type specification for the interface functions of the `.ml` file, and a qualifier (`.hquals`) file containing a set of logical qualifiers. DSOLVE combines the qualifiers from the `.hquals` file with some scraped from the specification `.mlq` file and a standard qualifier library to obtain the set of logical qualifiers used to infer liquid types.

Output. DSOLVE produces as output a refinement type for each program expression in a standard OCAML type annotation (`.annot`) file. The user can view the inferred refinement types using standard tools like EMACS, VIM, and CAML2HTML. If all the constraints are satisfied, the program is reported as safe. Otherwise, DSOLVE outputs warnings indicating the potentially unsafe expressions in the program.

Modular Checking. DSOLVE verifies one module at a time. If a module depends on another module, it can be checked against that module's `.mlq` file; the other module's source code is not required.

Abstract Modules. It is possible to create a `.mlq` file which defines types, axioms (background predicates), and uninterpreted functions without a `.ml` file. Such "abstract modules" allow the user to extend DSOLVE with reasoning about mathematical structures which do not appear directly in the program. For example, an abstract module `Set.mlq` might contain a type which represents a polymorphic set collection, along with an appropriate refined interface and axioms which build a set theory. This set theory can be used in another module's type refinements; for example, it may be used in a sorting module to verify that the sets of elements in the input and output lists of a sorting function are equal.

Availability. The DSOLVE source distribution is available, along with benchmarks and an online demo, at `http://pho.ucsd.edu/liquid/`.

4 Experiments

We report the results of applying Dsolve to real-world OCaml programs.

Static Array Bounds Checking. We have previously used benchmarks from the DML project [1, 8] to show that Dsolve significantly reduces annotation overhead burden in the static verification of array safety [5]. In our study, we automatically generated qualifiers of the form $\nu \bowtie X$, where $\bowtie \in \{<, \leq, =, \neq, >, \geq\}$ and $X \in \{0, \star, \texttt{len}\ \star\}$. This allowed us to reduce annotation overhead from 17% of LOC using DML to under 1% of LOC using Dsolve. Runtimes ranged from 1 to 64 seconds, the longest being for bitv [9], a 426-line bit vector library.

Data Structures. We have also used Dsolve to verify data structure invariants in production OCaml libraries [6], including that OCaml's `List.stablesort` outputs a sorted list, that OCaml's `Map` module implements an AVL tree and that `Map`'s keys form a set. Runtimes in this study ranged from 1 to 103 seconds, the longest being for vec [10], a 343-line OCaml extensible array library.

Program Understanding. Dsolve also helped us find and fix a subtle bug in vec. A vec extensible array is represented by a balanced tree with a balance factor of at most 2. As originally released, vec contained a flawed recursive balancing routine, `recbal`, which was meant to efficiently merge two balanced trees of arbitrarily different heights into a single balanced tree. When run on this code, the strongest invariant Dsolve could infer was that the output tree would have a balance factor of at most 4. By changing `recbal` and re-inferring types, we were able to isolate the faulty code paths and find test inputs with output balance factor of 4. Dsolve verified the fix, which the author adopted.

References

1. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: PLDI (1998)
2. Cui, S., Donnelly, K., Xi, H.: ATS: A language that combines programming with theorem proving. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 310–320. Springer, Heidelberg (2005)
3. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. In: CSF (2008)
4. Dunfield, J.: A Unified System of Type Refinements. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (2007)
5. Rondon, P., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI (2008)
6. Kawaguchi, M., Rondon, P., Jhala, R.: Type-based data structure verification. In: PLDI, pp. 304–315 (2009)
7. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. Xi, H.: DML code examples, `http://www.cs.bu.edu/fac/hwxi/DML/`
9. Filliátre, J.C.: Bitv, `http://www.lri.fr/~filliatr/software.en.html`
10. de Alfaro, L.: Vec, `http://www.dealfaro.com/~luca/vec.html`