Compositional and Lightweight Dependent Type Inference for ML

He Zhu and Suresh Jagannathan

Dept. of Computer Science
Purdue University

Abstract. We consider the problem of inferring expressive safety properties of higher-order functional programs using first-order decision procedures. Our approach encodes higher-order features into first-order logic formula whose solution can be derived using a lightweight counterexample guided refinement loop. To do so, we extract initial verification conditions from dependent typing rules derived by a syntactic scan of the program. Subsequent type-checking and type-refinement phases infer and propagate specifications of higher order functions, which are treated as *uninterpreted* first-order constructs, via subtyping chains. Our technique provides several benefits not found in existing systems: (1) it enables *compositional* verification and inference of useful safety properties for functional programs; (2) additionally provides counterexamples that serve as witnesses of unsound assertions: (3) does not entail a complex translation or encoding of the original source program into a first-order representation; and, (4) most importantly, profitably employs the large body of existing work on verification of first-order imperative programs to enable efficient analysis of higher-order ones. We have implemented the technique as part of the MLton SML compiler toolchain, where it has shown to be effective in discovering useful invariants with low annotation burden.

1 Introduction

Dependent or refinement types [20,10,28] offer a promising way to express rich invariants in functional programs that can go beyond the capabilities of traditional type systems [8] or control-flow analyses [25], albeit at the price of automatic inference. Recently, there has been substantial progress in reducing this annotation burden [15,23,14,13,16,17,19,18,22] using techniques adopted from model-checking and verification of first-order imperative programs [11,7]. These solutions, however, either (a) involve a complex reformulation of the intuitions underlying invariant detection and verification from a first-order context to a higher-order one [17,18], making it difficult to directly reuse existing tools and methodologies, (b) infer dependent types by solving a set of constraints collected by a whole-program analysis [15,23], additionally seeded with programmer-specified qualifiers, that can impact compositionality and usability, or (c) entail a non-trivial translation to a first-order setting [13], making it

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 295–314, 2013.

more complicated to relate the inferences deduced in the translated first-order representation back to the original higher-order source when there is a failure.

In this paper, we present POPEYE, a compositional verification system that integrates a first-order verification engine, unaware of higher-order control- and dataflow, into a path- and context-sensitive dependent type inference framework for Standard ML. Notably, our solution treats uses of higher-order functions as *uninterpreted* terms. In this way, we are able to directly exploit the scalability and efficiency characteristics of first-order verification tools *without* having to either consider a sophisticated translation or encoding of our functional source program into a first-order one [13], or to re-engineer these tools for a higher-order setting [17]. Our verification strategy is based on a counterexample-guided refinement loop that systematically strengthens a function's inferred dependent type based on new predicates discovered during examination of a derived counterexample path. Moreover, our strategy allows us not to only verify the validity of complex assertions, but can also be used to directly provide counterexample witnesses that disprove the validity of presumed invariants that are incorrect.

Our technique is *compositional* because it lazily propagates refinements computed at call-sites to procedures and *vise versa*, allowing procedure specifications to be strengthened incrementally. It is *lightweight* because it directly operates on source programs without the need to generate arbitrary program slices [26], translate the source to a first-order program [13], or abstract the source to a Boolean program [18]. POPEYE's design consists of two distinct parts:

1. ***Dependent Type Checking.*** Initially, we infer coarse dependent types for all local expressions within a procedure using dependent type rules that encode intraprocedural path information in terms of first-order logic formulae that range over linear arithmetic and uninterpreted functions, the latter used to abstract a program's higher-order control-flow. We build verification conditions that exploit the dependent types and which are subsequently supplied into a first-order decision procedure. Verification failure yields an intraprocedural counterexample path.
2. ***Dependent Type Refinement.*** The counterexample path can be used by existing predicate discovery algorithms to appropriately strengthen pre- and post-conditions at function calls. Newly discovered refinement predicates are propagated along subtyping chains that capture interprocedural dependencies to strengthen the dependent type signatures of the procedures used at these call-sites.

The remainder of the paper is organized as follows. In the next section, we present an informal overview of our approach. Section 3 defines a small dependently-typed higher-order core language. We formalize our verification strategy for this language in Section 4. Section 5 discusses the implementation and experimental results. Related work and conclusions are given in Sections 6 and 7.

2 Overview and Preliminaries

Dependent Types. We consider two kinds of dependent type expressions:

1. a *dependent base type* written $\{\nu : B | r\}$, where ν is a special value variable undefined in the program whose scope is limited to r, B is a base type, such as int or bool, and r is a boolean-valued expression (called a *refinement*). For instance, $\{\nu : \mathsf{int} \,|\, \nu > 0\}$ defines a dependent type that represents the set of positive integers.
2. a *dependent function type* written:

$$\{x : P_{1_x} \to P_1\} \oplus \{x : P_{2_x} \to P_2\} \oplus \ldots \oplus \{x : P_{n_x} \to P_n\}$$

abbreviated as $\oplus_i \{x : P_{ix} \to P_{io}\}$, where each $\{x : P_{i_x} \to P_i\}$ defines a function type whose argument x is constrained by dependent type P_{ix} and whose result type is specified by P_i. The different components of a dependent function type distinguish different contexts in which the function may be used. For instance,

$$\{x : \{\nu : \mathsf{int} \,|\, \nu > 0\} \to \{\nu : \mathsf{int} \,|\, \nu > x\}\} \oplus \{x : \{\nu : \mathsf{int}| \ \nu < 0\} \to \{\nu : \mathsf{int}| \ \nu < x\}\}$$

specifies the function that, in one call-site, given a positive integer returns an integer greater than x, while in another, given a negative integer returns an integer less than x. Components in a dependent function type are indexed by an implicit label, e.g., a finite call-string used in polyvariant control-flow analyses [24,12].

As shorthand, we sometimes write only the refinement predicate to represent the dependent type, omitting its type constructor. Thus, in the following, we sometimes write $\{r\}$ as shorthand for $\{\nu : B \,|\, r\}$. For example, $\{\nu > 0\}$ represents $\{\nu : \mathsf{int} \mid \nu > 0\}$. We also write B as shorthand for $\{\nu : B \,|\, \mathsf{true}\}$. For perspicuity, we use syntactic sugar to allow the $\oplus$ operator to be "pushed into" refinements:

$$\{\nu : B \mid r_1\} \oplus \{\nu : B \mid r_2\} = \{\nu : B \mid r_1 \oplus r_2\}$$
$$\{x : P_1 \to P_{r_1}\} \oplus \{x : P_2 \to P_{r_2}\} = \{x : P_1 \oplus P_2 \to P_{r_1} \oplus P_{r_2}\}$$

As a result, context-sensitive dependent types reuse the shape of ML types (Section 3.1). Additionally, we define $P.i$ to return the dependent type indexed by label i. When a function is used in a single context, we simply write $\{x : P_x \to P\}$.

Procedure Specifications. A procedure specification is given in terms of a pre- and post-condition of a procedure; we express these conditions in terms of a dependent function type where the type of the function's domain can be thought of as the function's pre-condition, and where the type of the function's range defines its post-condition.

```
fun f g x =
  if x>=0 then
    let r = g x in r
  else
    let p = f g
        q = compute x
        s = f p q
    in s
```

```
fun main h n =
  let r = f h n
  in assert (r >= 0)
```

Fig. 1. The use of higher-order procedures can make compositional dependent type inference challenging

2.1 Example

Consider the program shown in Fig. 1. This program exhibits complex dataflow (e.g., it can create an arbitrary number of closures via the partial application of `f`) and makes heavy use of higher-order procedures (e.g., the formal parameter `g` in function `f`). We wish to infer a useful specification for `f` without having to (a) supply candidate qualifiers used in the dependent types that define the specification, (b) know the possible concrete arguments that can be supplied to `g`, or (c) require details about `compute`'s definition. In spite of these constraints, our technique nonetheless associates the following non-trivial type to `f`:

$$\mathtt{f} : \{\mathtt{g} : \{\mathtt{g}_{\mathtt{arg}} : \{\nu \geq 0\}\} \rightarrow \{\nu \geq 0\}\} \rightarrow \mathtt{x} : \{\mathsf{true}\} \rightarrow \{\nu \geq 0\}\}$$

This type ascribes an invariant to `g` that asserts that `g` must take a non-negative number as an argument (as a consequence of the path constraint `(x >= 0)` within which it is applied) and returns a non-negative number as a result (as a consequence of the assertion made in `main`).

```
fun g x y = x
fun twice f x y =
  let p = f x
  in f p y
fun neg x y = -(x ())
```

```
fun main n =
  if n >= 0 then
    assert(twice neg (g n) () >= 0)
  else ()
```

Fig. 2. A function's specification can be refined based on the context in which it is used

The utility of context-sensitive dependent types arises when a function is called in different (potentially inconsistent) contexts. Consider the program shown in Fig. 2. Here, function `f` (which is supplied the argument `neg` in `main`) is called in two different contexts in the procedure `twice`. The first argument to `f` is a higher-order procedure - in the first call, this procedure is bound to the result of evaluating `g n`; in the second call, the procedure (bound to `p`) is the result of the first partial application. Since `f` negates the value yielded by applying its procedure argument to `()`, we thus infer the following specification:

$$\mathtt{f}_{\mathtt{arg}_1} : \{\{\mathsf{true} \oplus \mathsf{true}\} \rightarrow \{\nu \geq 0 \ \oplus \ \nu \leq 0\}\} \rightarrow \mathtt{f}_{\mathtt{arg}_2} : \{\mathsf{true} \oplus \mathsf{true}\} \rightarrow \{\nu \leq 0 \ \oplus \ \nu \geq 0\}$$

3 Language

We formalize our ideas in the context of a call-by-value variant of the λ-calculus with support for dependent types. The syntax of the language is shown in Fig. 3. We use $f, g, x, y, \ldots$ to range over variables; typically, f and g (as well as their subscripted variants) are only bound to abstractions, while x and y (as well as their subscripted variants) can be bound to values of any type. The special variable ν is used to denote the value of a term in its corresponding dependent type refinement predicate. The language supports a small set of base types (B), monotypes (τ), type schemas (σ) that introduce polymorphic types via type variables that are universally quantified at the outermost level, and dependent types (P) that include dependent base types and dependent function types.

Predicates (p) are Boolean expressions built from a predefined set ($\mathcal{Q}$) of first-order logical, arithmetic, and relational operators; the arguments to these operators are simple expressions - variables, constants, or function applications; to simplify the technical development, we assume function applications are A-normalized, ensuring every abstraction and function argument is associated with a program variable. A refinement expression is either a refinement variable (κ) that represents an initially unknown refinement or a concrete predicate (p). Templates (P_T) are dependent types whose refinement expressions are only refinement variables (κ). The pick or selection operator $\kappa.i$ on refinement variable allows $\oplus$ to be pushed into refinements (as described in Section 2), and hence omitted in template definitions. Instantiation of the refinement variables to concrete predicates takes place through the type refinement algorithm described in Section 4. An assert statement of the form "$\mathsf{assert}\ p; e'$" evaluates expression e' if predicate p evaluates to true and returns the special value fail otherwise.

$$f, g, x, y, \ldots \in \mathsf{Var} \quad c \in \mathsf{Constant} ::= 0, 1, \ldots, \mathsf{true}, \mathsf{false}, \ldots \quad B \in \mathsf{Base} ::= \mathsf{int} \mid \mathsf{bool}$$
$$\tau \in \mathsf{Monotype} ::= B \mid \alpha \mid \tau \to \tau \qquad \sigma \in \mathsf{PolyType} ::= \tau \mid \forall \alpha.\sigma$$
$$P \in \mathsf{DepType} ::= \{\nu : B \mid r\} \mid \oplus_i \{x : P \to P\}$$
$$r \in \mathsf{Refinement} ::= \kappa \mid p \quad p \in \mathsf{Predicate} ::= \mathcal{Q}(s, \ldots, s)$$
$$\kappa \in \mathsf{RefinementVar} ::= \kappa \mid \kappa.i \quad P_T \in \mathsf{Template} ::= \{\nu : B \mid \kappa\} \mid \{x : P_T \to P_T\}$$
$$s \in \mathsf{SimpleExp} ::= \nu \mid x \mid f\ x \quad v \in \mathsf{Value} ::= c \mid \lambda x.e$$
$$e ::= s \mid v \mid \mathsf{fix}\ e \mid \mathsf{fail} \mid \mathsf{if}\ p\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{assert}\ p;\ e \mid [\Lambda\alpha]\, e \mid [\tau]\, e$$

Fig. 3. Syntax

3.1 Dependent Type System

Type inference and checking use an ordered *type environment* Γ that consists of a sequence of dependent type bindings $x : P_x$ along with guard expressions drawn from conditional expression predicates. The use of these guard expressions makes

the type system path-sensitive since the dependent types inferred for a term are computed using the guard expressions that encode the program path taken to reach this term. We define the *shape* of a dependent type as its corresponding ML type; thus, $\mathit{Shape}(P)$ is obtained by replacing all refinements in P with true. We generalize its definition to type environments in the obvious way - hence, $\mathit{Shape}(\Gamma)$ consists only of bindings that relate variables to ML types, with all refinements replaced with true and guard expressions found in Γ removed.

$$\frac{\Gamma(x) = \{\nu \ : \ B \mid e\}}{\Gamma \vdash x \ : \ \{\nu \ : \ B \mid \nu = x\}} \ \mathsf{VarBase} \qquad \frac{\Gamma(x) \text{ not a base type}}{\Gamma \vdash x : \Gamma(x)} \ \mathsf{VarFunc}$$

$$\frac{}{\Gamma \vdash c : ty(c)} \ \mathsf{Const} \qquad \frac{\Gamma \vdash e_1 : P_1 \quad \Gamma; x : P_1 \vdash e_2 \ : \ P_2}{\Gamma \vdash \mathsf{let}\ x \ = \ e_1\ \mathsf{in}\ e_2 : P_2} \ \mathsf{Let}$$

$$\frac{\forall\, i.\ \Gamma_i; x : P_{ix} \vdash e : P_{ie} \quad \Gamma_i; x : P_{ix} \vdash P_{ie} <: P_i}{\oplus_i \Gamma_i \vdash \lambda x.e : \oplus_i \ \{x : P_{ix} \to P_i\}} \ \mathsf{Fun} \qquad \frac{\Gamma \vdash \ e \ : \ \{f : P_f \to \oplus_i \ \{x : P_{ix} \to P_i\}\}}{\Gamma; f : P_f \vdash \mathsf{fix}\ e \ : \ \oplus_i \ \{x : P_{ix} \to P_i\}} \ \mathsf{Fix}$$

$$\frac{\Gamma \vdash y \ : \ P_y \quad \Gamma \vdash P_y <: P_x \quad \Gamma \vdash f_i : (x : P_x \to P)}{\Gamma \vdash f_i(y) : [y/x]P} \ \mathsf{App}$$

$$\frac{\Gamma \vdash p : \mathsf{bool} \quad \Gamma; p \vdash e_t : P_t \quad \Gamma; \neg p \vdash e_f : P_f}{\Gamma \vdash \mathsf{if}\ p\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f \ : \ \mathcal{C}(p, P_t, P_f)} \ \mathsf{If} \qquad \frac{\Gamma \vdash f : \oplus_i \ \{x : P_{ix} \to P_i\}}{\Gamma \vdash f_j : \{x : P_{j_x} \to P_j\}} \ \mathsf{Pick}$$

$$\frac{\forall\, i.\ \Gamma_i \vdash f_i : \{x : P_{ix} \to P_i\}}{\oplus_i \Gamma_i \vdash f : \oplus_i \ \{x : P_{ix} \to P_i\}} \ \mathsf{Conc} \qquad \frac{\Gamma \vdash \{\mathsf{bool} \mid \mathsf{true}\} <: \{\mathsf{bool} \mid p\} \quad \Gamma \vdash e \ : \ P}{\Gamma \vdash \mathsf{assert}\ p;\ e \ : \ P} \ \mathsf{Assert}$$

$$\frac{\Gamma \vdash e : \forall \alpha.P \quad \Gamma \vdash P' \quad \mathit{Shape}(P') = \gamma}{\Gamma \vdash [\gamma]e : [P'/\alpha]P} \ \mathsf{Inst} \qquad \frac{\Gamma \vdash e : P \quad \alpha \text{ not free in } \Gamma}{\Gamma \vdash [\Lambda\alpha]e : \forall \alpha.P} \ \mathsf{Gen}$$

$$\frac{\langle \Gamma \rangle \wedge \langle r_1 \rangle \Rightarrow \langle r_2 \rangle}{\Gamma \vdash \{\nu \ : \ B \mid r_1\} <: \{\nu \ : \ B \mid r_2\}} \ \mathsf{BaseSub} \qquad \frac{\Gamma \vdash P'_x <: P_x \quad \Gamma; x : P'_x \vdash P <: P'}{\Gamma \vdash \{x : P_x \to P\} <: \{x : P'_x \to P'\}} \ \mathsf{FunSub}$$

$$\frac{\forall\, i.\ \Gamma \vdash \{x : P_{ix} \to P_i\} <: \{x : P'_{i_x} \to P'_i\}}{\Gamma \vdash \oplus_i \ \{x : P_{ix} \to P_i\} <: \oplus_i \ \{x : P'_{i_x} \to P'\}} \ \mathsf{ConcFunSub}$$

Fig. 4. Dependent typing rules

Fig. 4 defines the dependent type inference rules; these rules are adapted from [23], generalized to deal with richer path and context-sensitive types. Syntactically, $\Gamma \vdash e : P$ states that expression e has type dependent type P under type environment Γ. Our typing rules are refinements of the ML typing rules. If $\Gamma \vdash e : P$ then $\mathit{Shape}(\Gamma) \vdash e : \mathit{Shape}(P)$. $\Gamma; \mathtt{x} : P$ defines the type environment that extends the sequence Γ with a binding for $\mathtt{x}$ to P. The rules for variables, constants, let-expressions are standard. Rule Fun associates a context-sensitive dependent function type with an abstraction. The structure of this type is determined by the different contexts in which the abstraction is applied (Γ_i) generated

from rule Conc described below. The first judgment in the antecedent considers the type of the abstraction body in all type environments Γ_i enriched by a type binding of bound variable x with dependent type P_{i_x}. The second judgment asserts that P_{i_e}, the type associated with the body of the abstraction, be a subtype of the return type of the abstraction. Rule Fix defines recursive functions in the obvious way. Rule App establishes a subtyping relation between the actual and formal parameters in the application. The abstract labels that subscript function identifiers in the rules are used to express context-sensitivity but are not part of the program syntax, and are constructed during the interprocedural type refinement phase.

In the If rule, we independently infer types P_t and P_f for branch expressions e_t and e_f, resp. Then, the dependent type of the entire expression is given using operator $\mathcal{C}$ that enforces the guard expression (or its negation) p (or $\neg p$) to be a precondition of the corresponding type:

$$\mathcal{C}(p, \{\tau \mid r_1\}, \{\tau \mid r_2\}) = \{\tau \mid p \Rightarrow r_1 \wedge \ \neg p \Rightarrow r_2\}$$
$$\mathcal{C}(p, \oplus_i\{x : P_1 \to P_{r_1}\}, \oplus_i\{x : P_2 \to P_{r_2}\}) = \oplus_i\{x : \mathcal{C}(p, P_1, P_2) \to \ \mathcal{C}(p, P_{r_1}, P_{r_2})\}$$

There are two rules for extracting and generating context-sensitive dependent type functions. A term f with type $\oplus_i \ \{x : P_{i_x} \to P_i\}$ reflects the type of all uses of f in different contexts; the type at a given context can be indexed by the label at the use (rule Pick). Conversely, we can construct the concatanation of the types at each context to yield the actual type of the function (rule Conc). The subtype judgment in rule Assert enforces that the assertion predicate p hold. Polymorphic instantiation and generalization are defined in the standard way.

There are three subtyping rules. In rule Base Subtyping, the premise check $\langle\Gamma\rangle \wedge \langle r_1\rangle \Rightarrow \langle r_2\rangle$ requires that the conjunction of environment formula $\langle\Gamma\rangle$ and r_1 imply r_2. As in [23], $\langle\Gamma\rangle$ is defined as a first order logic formula:

$$\bigwedge\{r \mid r \ \in \Gamma\} \wedge \bigwedge\{[[x/\nu]\ r] \mid x \ : \ \{\nu \ : \ B \mid r\} \in \Gamma\}$$

Rule FunSubtype defines the usual subtyping relation on functions and rule ConcFun generalizes this rule to deal with context-sensitivity. These three rules implicitly encode subtyping chains, allowing specifications to be propagated across function boundaries.

Our semantics enjoys the usual progress and preservation properties; evaluation preserves types, and well-typed programs do not get stuck. (An assertion violation causes the program to halt with the special value fail.)

Theorem 1 (Dependent Type Safety)

1. *(Preservation) If* $\Gamma \vdash e : P$ *and* $e \hookrightarrow e'$ *then* $\Gamma \vdash e' : P$
2. *(Progress) If* $\Gamma \vdash e : P$*, where* $e \ \neq$ fail *then* e *is either a constant or an abstraction, or there exists an* e' *such that* $e \hookrightarrow e'$.

4 Verification Procedure

Our verification system consists of (a) a type-checking algorithm that encodes intra-procedural path constraints and generates verification conditions whose validity can be checked by a first-order decision procedure, and (b) a counterexample guided dependent type refinement loop that uses the counterexample yielded by a verification failure to strengthen existing invariants, and propagate new ones inter-procedurally via dependent subtyping chains.

4.1 Dependent Type Checking

The first step of our verification procedure is to assign each function a dependent type *template* as described earlier. By applying our inference rules, with the type template, given a type environment Γ and expression e, we can construct dependent types for local expressions and derive a set of subtyping constraints, which will be subsequently used to generate verification conditions (VC).

There are three verification conditions generated from the type checking rules. First, a subtyping constraint introduced by an assert expression:

$$\Gamma \vdash \{\mathsf{bool} \mid \mathsf{true}\} <: \{\mathsf{bool} \mid p\}$$

entails a verification condition that checks the validity of p under the path constraints and type bindings defined by Γ. Second, the subtyping constraint associated with function abstraction:

$$\Gamma; x : P_{T_x} \vdash P_{T_e} <: P_T$$

establishes a verification condition on the post-condition of this abstraction that requires it be consistent with the invariants inferred for its body. Third, the subtyping constraint associated with function application:

$$\Gamma \vdash P_{T_y} <: P_{T_x}$$

entails a verification condition that checks that the specification of the function's pre-condition subsumes the invariants associated with the argument at the call.

A solution in our system is defined by a refinement environment Σ that maps refinement variables κ to refinements. We lift this notion to dependent types $\Sigma(P_T)$ and type environment $\Sigma(\Gamma)$ by substituting each place holder κ with $\Sigma(\kappa)$ appearing in P_T and Γ. A verification condition c is valid if $\Sigma(c)$ is valid. We say Σ satisfies a subtyping constraint $\Gamma \vdash P_{T1} <: P_{T2}$ if $\Sigma(\Gamma) \vdash \Sigma(P_{T1}) <: \Sigma(P_{T2})$. Σ is a valid solution if it satisfies all subtype constraints.

Like [23], we deconstruct arbitrary subtyping constraints to base subtyping constraints (Fig. 4). According to the Base Subtyping rule, the verification condition formula is generated as

$$\langle \Sigma(\Gamma) \rangle \wedge \langle \Sigma(r_1) \rangle \Rightarrow \langle \Sigma(r_2) \rangle$$

To allow our verification engine to deal with unknown higher-order functions, we encode higher-order functions into an uninterpreted form. Suppose the type

of function f is $x_0 : P_{x_0} \to \cdots \to x_n : P_{x_n} \to P_f$. We encode P_f to be $\{\nu = \mathsf{R}_f(\mathsf{arg}_0(f), \ldots, \mathsf{arg}_n(f))\}$; here, R_f and arg_i are uninterpreted terms representing the result of function f and the i^{th} argument supplied to f at a call. Applications of higher-order function f are encoded by substituting actuals for the appropriate (suitably encoded) formal as $\mathsf{Encode}(f)$. This gives us the ability to verify a function modularly without having to know the set of definitions referenced by a functional argument or result. For example, for the program shown in Fig. 1, the variable `r` in the let-binding, `r = g x`, is encoded as $[\mathtt{x}/\mathsf{arg}_0(\mathtt{g})]\mathsf{R}_\mathtt{g}(\mathsf{arg}_0(\mathtt{g}))$, which is simply $\mathsf{R}_g(\mathtt{x})$. The subtyping constraint built for checking the post-condition during the verification of `f`, leads to the construction of the verification condition:

$$((\mathtt{x} \geq 0 \wedge \mathtt{r} = \mathsf{R}_g(\mathtt{x})) \Rightarrow \nu = \mathtt{r}) \wedge ((\neg(\mathtt{x} \geq 0) \wedge \mathtt{s} \geq 0) \Rightarrow \nu = \mathtt{s}) \Rightarrow (\nu \geq 0)$$

4.2 Dependent Type Refinement

The heart of our counterexample-guided type refinement loop is given in Fig. 5. Our refinement algorithm exploits the dependent type template and subtyping constraints generated from type inference rules and finally returns solution Σ. In `Solve`, our method iteratively type checks each procedure of the given program using the subtyping rules listed in Fig. 4 until a fix-point is reached. When a procedure cannot be typed with the set of current refinements, our method supplies the unverified procedure's type environment Γ, the current refinement map Σ, its type template $x : P_{T_x} \to P_T$, the unverified function $\lambda x.e$, and the verification conditions C constructed for the function to `Refine` which can then proceed to strengthen the function's dependent type.

Counterexample Generation. Our refinement algorithm first constructs a counterexample *ce* for an unverified verification condition. The counterexample is derived by solving the negation of the desired verification condition:

$$\langle \Sigma(\Gamma) \rangle \wedge \langle \Sigma(r_1) \rangle \wedge \neg \langle \Sigma(r_2) \rangle$$

The encoding of Γ and r_1 reflects path information; by the structure of the rules in Fig. 4, the encoding of refinement r_2, on the other hand, reflects a safety property that is implied by $\langle \Gamma \rangle \wedge \langle r_1 \rangle$. Thus, an assignment to this formula leads to a counterexample of a possible safety violation; this counterexample path is represented as a straight-line program.

A path expression of the form: "`if` p `then` e_t `else` e_f" is translated to: "`assume` p; e_t" if an assignment from the VC evaluates p to `true` and "`assume` $\neg p$; e_f" otherwise. Consider our example from Fig. 1. A first-order decision procedure would find an assignment to the the negation of the VC as an error witness, e.g., `r = -1` and `x = 1`. The representation of the counterexample path of procedure `f` given in Fig. 1 is thus:

```
fun f g x = assume (x >= 0); let r = g x in r
```

```
Refine (Γ, Σ, {x : P_Tx → P_T}, λx.e, C) =
  if exists c ∈ C such that Σ(c) is not valid with a witness of ce
  then
    let Σ' = case c of
      | Γ ⊢ {ν : B|p1} <: {ν : B|r2} ⇒
        let pred = case r2 of | p2 ⇒ r2 | _ ⇒ Σ[r2]
        in Strengthen ({x : P_Tx → P_T}, Σ, wp (ce, pred), r2)
      | Γ ⊢ {ν : B| κ1} <: {ν : B| κ2} ⇒
        Σ[κ1 ↦ (Σ[κ1]) ∧ (Σ[κ2])]
    in Refine (Γ, Σ', {x : P_Tx → P_T}, λx.e, C)
  else Σ

Solve (procedures as List[Γ, {x : P_Tx → P_T}, λx.e, C], Σ) =
  if exists (Γ, {x : P_Tx → P_T}, λx.e, C) for a procedure needs to be checked
  then
    Solve (procedures, Refine (Γ, Σ, {x : P_Tx → P_T}, λx.e, C))
  else Σ
```

Fig. 5. Type refinement algorithm

According to the two different forms of subtyping constraints generated, dependent types can be refined from the counterexample path in one of two ways: weakest precondition generation or procedure specification propagation.

Weakest Precondition Generation. In this setting, the constraint is of the form: $\Gamma \vdash \{\nu : B \mid p_1\} <: \{\nu : B \mid r_2\}$, corresponding to the first case in `Refine` in Fig. 5, where p_1 is a concrete predicate and r_2 is either a concrete predicate or a refinement variable or a selection of refinement variable. This constraint is generated when based typed expression is supplied as function argument or return or establishing assertions. Our type refinement in this case can be implemented by a backward symbolic analysis analogous to weakest precondition generation, operating over a counterexample path. Recall that the weakest pre-condition of an expression S is a function $wp(S, Q)$ mapping the post-condition Q to a pre-condition P, ensuring the execution of S terminates in a final state satisfying Q. Similarly, our weakest precondition generation simply pushes up post-conditions backwards, substituting terms for values in the presumed post-condition based on the structure of the term used to generate the pre-condition.

Our weakest precondition semantics is extended to deal with counterexample paths that include unknown function calls but for which context information constraining their arguments or results is available. Here, we can only strengthen relevant signatures, deferring the re-verification of the procedure being invoked until it becomes known. The called function's post-condition will be eventually propagated via dependent subtyping chains back to the procedures that flow into this call-site; in doing so, pre-conditions of these functions could be strengthened, requiring re-verification of the calling contexts in which they occur to ensure that these contexts imply the pre-condition. Such flows are handled directly by the subtyping chains analyzed by the refinement phase. For a called higher-order

$$
\begin{aligned}
\mathsf{wp}(e, \phi) = \ & \mathsf{case}\ e\ \mathsf{of} \\
& \mid \lambda x.e \Rightarrow \mathsf{wp}(e, \phi) \\
& \mid \texttt{assume}\ \psi;\ e \Rightarrow (\psi \Rightarrow \mathsf{wp}(e, \phi)) \\
& \mid \texttt{let}\ x\ =\ e'\ \mathsf{in}\ e \Rightarrow \mathsf{wp}(x\ =\ e',\ \mathsf{wp}(e, \phi)) \\
& \mid v = x \Rightarrow [x/v]\phi \\
& \mid v = c \Rightarrow [c/v]\phi \\
& \mid v = f\overrightarrow{d} \Rightarrow [\mathsf{Encode}(f)/v]\phi \\
& \mid \_ \Rightarrow \mathsf{wp}(\nu\ =\ e, \phi)
\end{aligned}
$$

Fig. 6. Weakest precondition generation definition

function f, we use $\mathsf{Encode}\ (f)$ to represent its value. The definition of our wp is given in Fig. 6. Consider the example in Fig. 1. The post-condition inferred is $\nu \geq 0$. We can infer the precondition shown in Section 2 by applying our wp rules as follows:

$$
\begin{aligned}
& \mathsf{wp}(\ \texttt{assume}\ (\texttt{x} \geq 0); \texttt{let}\ \texttt{r}\ =\ \texttt{g}\ \ \texttt{x}\ \texttt{in}\ \texttt{r}\ ), \nu \geq 0) = \\
& \mathsf{wp}(\ \texttt{assume}\ (\texttt{x} \geq 0), \mathsf{wp}(\texttt{let}\ \texttt{r}\ =\ \texttt{g}\ \ \texttt{x}\ \texttt{in}\ \texttt{r}\ , \nu \geq 0)) = \\
& \mathsf{wp}(\ \texttt{assume}\ (\texttt{x} \geq 0), \mathsf{wp}(\ \texttt{r}\ =\ \texttt{g}\ \ \texttt{x}\ , (\mathsf{wp}(\nu = \texttt{r}\ , \nu \geq 0)))) = \\
& \mathsf{wp}(\ \texttt{assume}\ (\texttt{x} \geq 0), \mathsf{wp}(\ \texttt{r}\ =\ \texttt{g}\ \ \texttt{x}\ , \texttt{r} \geq 0)) = \\
& \mathsf{wp}(\ \texttt{assume}\ (\texttt{x} \geq 0), \mathsf{R}_g(\texttt{x}) \geq 0) = \\
& \texttt{x} \geq 0 \Rightarrow \mathsf{R}_g(\texttt{x}) \geq 0
\end{aligned}
$$

Thus g's specification is strengthened to $\texttt{g} : \{\{\nu \geq 0\} \rightarrow \{\nu \geq 0\}\}$.

When a function call `f(x)` is encountered and the abstraction to which `f` is bound is known precisely (e.g., based on a syntactic or control-flow analysis preprocessing phase), our method strengthens the post-condition of the function's body of `f` to that available at the call. `wp` recursively applies our verification technique to refine the function's precondition based on the post-condition defined by the context in which it is called. `wp` can then be executed from this call site operating on the rest of statements of the counterexample beyond the call site and the newly strengthened precondition.

Procedure Specification Propagation. In this setting, the subtyping constraint is of the form: $\Gamma \vdash \{\nu\ :\ B \mid \kappa_1\} <: \{\nu\ :\ B \mid \kappa_2\}$, corresponding to the second case in `Refine` in Fig. 5, Refinement variables are introduced when defining dependent type templates; this occurs during inference of function abstraction and fix expressions. Ensuring the subtyping constraint holds requires that any instantiation of κ_2 be propagated to κ_1. This enables refinements associated with the post-condition of a higher order function to be propagated into the real function body, and conversely to propagate refinements associated with a function's pre-condition back to the parameters of higher order function.

Consider how we might verify the program shown in Fig. 2. Our method initially infers a dependent type template for `f` as $\{\{\kappa_{1_1} \to \kappa_{1_2}\} \to \kappa_2 \to \kappa_{\mathtt{f}}\}$. The assertion in `main` drives a new post-condition $\{\nu \geq 0\}$ for `twice`, and hence $\mathtt{f}_2$ which is the second the call to `f`, instantiating $\kappa_{\mathtt{f}}$ to $\{\mathsf{true} \oplus \nu \geq 0\}$. This constraint is then propagated to the post-condition of `neg` since `neg` subtypes to `f` at the call site of `twice` in `main`. The weakest pre-condition backward analysis of our system then strengthens the pre-condition for `neg` and propagates it back to `f`, instantiating $\{\kappa_{1_2}\}$ to $\{\mathsf{true} \oplus \nu \leq 0\}$. In `twice`, our technique needs to ensure, at the second call site of $\mathtt{f}_2$, the actual higher-order function `p` subtypes to the first argument of `f` where `p` is derived from the first call to `f` notated as $\mathtt{f}_1$. The subtyping relation can then be expressed as $\Gamma \vdash \{\kappa_2.1 \to \kappa_{\mathtt{f}}.1\} <: \{\kappa_{1_1}.2 \to \kappa_{1_2}.2\}$. The post-condition in $\kappa_{1_2}.2$ ($\{\nu \leq 0\}$) is then propagated to $\kappa_{\mathtt{f}}.1$, which becomes $\{\nu \leq 0 \oplus \nu \geq 0\}$. Finally, the context-sensitive type for `f` is derived as

$$\mathtt{f}_{\mathtt{arg}_1} : \{\{\mathsf{true} \oplus \mathsf{true}\} \to \{\nu \geq 0 \ \oplus \ \nu \leq 0\}\} \to \mathtt{f}_{\mathtt{arg}_2} : \{\mathsf{true} \oplus \mathsf{true}\} \to \{\nu \leq 0 \ \oplus \ \nu \geq 0\}$$

4.3 Correctness

We provide two correctness results for our verification algorithm $\mathcal{V}(\Gamma, \mathit{Prog})$ where Γ is top-level typing environment and *Prog* is a program[1]. The first (*Soundness*) states that the dependent types inferred by our verification procedure are consistent with our type rules. The second (*Weak*) states that our procedure generates the least type necessary to discharge the subtyping constraints collected by the inference algorithm. In the following, $R(\Gamma)$ recursively extracts dependent base types $\{\nu : B|\kappa\}$ from the domain of Γ.

Theorem 2 (Verification Algorithm)

1. *(Soundness) Let* $((\ldots, \{\Gamma, x : P_{T_x} \to P_T, \lambda x.e, C\}, \ldots), \Sigma)$ *be the result of* $\mathcal{V}(\Gamma, \mathit{Prog})$*. Then, provided* $\mathcal{V}(\Gamma, \mathit{Prog})$ *terminates,* $\Sigma(\Gamma) \vdash \ \lambda x.e \ : \{x : \Sigma(P_{T_x}) \to \Sigma(P_T)\}$.
2. *(Weak) And, for all other valid solution* Σ'*, the algorithm generates the weakest solution:* $\forall c$ *as* $\{\Gamma \vdash \{\nu : B|r_1\} <: \{\nu : B|r_2\}\} \in C$*, and* $\forall \ \{\nu : B|\kappa\} \in \{R(\Gamma) \cup \{\nu : B|r_1\}\}$*,* $\Sigma'(\Gamma) \vdash \{\nu : B|\Sigma'(\kappa)\} <: \{\nu : B|\Sigma(\kappa)\}$.

4.4 Invariant Generation

Because our technique does not guarantee termination given the undecidability of automatically synthesizing loop invariants, the size of a dependent function type may grow into an infinite representation, and a fixed-point may never be reached. Consider the ML program fragment shown in Fig. 7 adapted from [13]. The procedure `iteri` visits the elements of a list `xs`, applying function `f` to each element and its index in the list. Procedure `mask` calls `iteri` when the length of its array and list arguments are the same. It supplies function `g` as the

[1] The proof can be found in `www.cs.purdue.edu/homes/zhu103/pubs/vmcai13full.pdf`

higher-order argument to `iteri` which performs some computation involving a list and array element at the same index. We desire to verify the array bound safety property $\texttt{j} < \mathsf{len}(\texttt{a})$ for the array access in procedure g (Note $\texttt{j} \geq 0$ can be directly proved by our method introduced in Section 4.2).

```
fun iteri i xs f =
  case xs of
    [ ] => ()
  | x :: xs' =>  (f i x; iteri (i+1) xs' f)

fun mask a xs =
  let g j y = ··· y ··· Array.sub (a, j) ··· in
    if Array.length a = List.length xs then
      iteri 0 xs g
    else () end
```

Fig. 7. A program that has a non-trivial loop invariant

During the course of verifying this program, we would need to discharge a specification that forms a pre-condition for `iteri` asserting that $\mathsf{len}(\texttt{xs}) \neq 0 \Rightarrow \texttt{i} < \mathsf{len}(\texttt{a})$. However, verifying this specification requires a theorem prover to conclude that $\mathsf{len}(\texttt{xs})\text{-}1 \neq 0 \Rightarrow \texttt{i}+1 < \mathsf{len}(\texttt{a})$ as precondition for the recursive call to `iteri (i+1) xs'`. In trying to discover a counter-example to this claim, a theorem prover would likely generate an infinite number of pre-conditions, $\mathsf{len}(\texttt{xs}) - k \neq 0 \Rightarrow \texttt{i} + k < \mathsf{len}(\texttt{a})$ where $k = 0, 1, 2 \cdots$ What is required is a sufficiently strong invariant that can be used to validate the required safety properties. While programmers could certainly write such specifications if necessary, we follow the idea of interpolation-based model-checking [21] to automatically infer them when possible.

When our mainline verification algorithm diverges or reaches a pre-determined timebound during the analysis of a recursive procedure, it is unrolled incrementally together with its calling context. Our method then infers dependent type templates and generates subtyping constraints for the k-unrolled procedures. Pre-conditions of the higher order functions used in recursive procedure are propagated via subtyping chain from that of the real function they represent for. Post-conditions of the higher order functions are also propagated from that of the real function which can be obtained from our type inference algorithm. We then exploit a technique described in [27] to infer dependent types from the collected base subyping constraints. The basic idea is to use the interpolation of the first-order formulas derived from the subtyping constraints to deduce an instantiation for a given type refinement variable κ. We desire that the prover returns a more suitable refinement beyond that yielded by a weakest precondition generator. Refinements synthesized from k-unrolled non-recursive procedures are folded back to the original procedure as candidates.

For example, suppose our method discovers that it must unroll the recursive procedure `iteri` two times, obtaining the program shown below:

```
fun iteri0 i0 xs0 f0 =
  case xs0 of
    [ ] => ()
  | x0 :: xs0' =>  (f0 i0 x0; iteri1 (i0+1) xs0' f0)

fun iteri1 i1 xs1 f1 =
  case xs1 of
    [ ] => ()
  | x1 :: xs1' =>  (f1 i1 x1; iteri2 (i1+1) xs1' f1)

fun iteri2 i2 xs2 f2 = halt
```

Fig. 8. Unrolling a recursive procedure to enable loop invariant discovery using interpolation

Here, `halt` is a special term, representing a termination point. Because we maintain the original calling context of `iteri`, we have $\mathsf{len}\ \mathtt{a} = \mathsf{len}\ \mathtt{xs}$ in the typing environment and leverage subtyping constraints to establish that the actual `g` subtypes to the formal `f0`. We infer refinements for this unrolled excerpt using the obtained base subtyping constraints. We thus have the following subtyping constraint:

$$\mathtt{i1} : \kappa_{\mathtt{i1}},\ \mathtt{i0} : \kappa_{\mathtt{i0}},\ \mathtt{xs0} : \kappa_{\mathtt{xs0}}, \mathsf{len}\,(\mathtt{xs0}) = \mathsf{len}\,(\mathtt{xs0}') + 1, \mathsf{len}\,(\mathtt{xs}) = \mathsf{len}\,(\mathtt{a})$$
$$\vdash \{\nu = \mathtt{xs0}'\} <: \kappa_{\mathtt{xs1}}$$

that establishes that the actual `xs0'` given to `iteri1` subtypes to the formal `xs1`. In the body of `iteri1`, there is another constraint for the call to `f1 i1`:

$$\mathtt{i1} : \kappa_{\mathtt{i1}},\ \mathtt{xs1} : \kappa_{\mathtt{xs1}}, \mathsf{len}\,(\mathtt{xs1}) = \mathsf{len}\,(\mathtt{xs1}') + 1 \vdash \{\nu' = i1\} <: \{\nu' < \mathsf{len}\,(\mathtt{a})\}$$

Because we have already inferred the dependent type for procedure `g` before typing `iteri` and obtained precondition $\nu' < \mathtt{len(a)}$ for its first argument, we can use it to also serve as the precondition of the first argument of `f1` propagated through the subtyping chains.

We extend the above constraints into first order logic formulas:

$$\{\mathtt{i1} = \mathtt{i0} + 1 \ \wedge\ \mathtt{i0} = 0 \ \wedge\ \mathtt{xs0} = \mathtt{xs} \ \wedge\ \mathsf{len}\,(\mathtt{xs0}) = \mathsf{len}\,(\mathtt{xs0}') + 1 \wedge$$
$$\mathsf{len}\,(\mathtt{xs}) = \mathsf{len}\,(\mathtt{a}) \ \wedge\ \nu = \mathtt{xs0}'\}^{(a)} \Rightarrow \kappa_{xs1}$$
$$\kappa_{xs1} \Rightarrow \{\mathtt{i1} = \mathtt{i0} + 1 \ \wedge\ \nu = \mathtt{xs1} \ \wedge\ \mathsf{len}\,(\mathtt{xs1}) = \mathsf{len}\,(\mathtt{xs1}') + 1 \wedge$$
$$\nu' = \mathtt{i1} \Rightarrow \nu' < \mathsf{len}\,(\mathtt{a})\}^{(b)}$$

The unknown refinement represented by $\kappa_{\mathtt{xs1}}$ is indeed an interpolation of formula (a) and formula (b) and can be inferred by feeding them into an appropriate

interpolation theorem prover [21] which may return $\mathsf{len}(\nu) + \mathtt{i1} = \mathsf{len}\,(\mathtt{a})$ as result. Our method then yields $\mathsf{len}(\nu) + \mathtt{i} = \mathsf{len}\,(\mathtt{a})$ (discarding subscript) as a refinement candidate of the second argument `xs` of procedure `iteri`.

After candidate refinement synthesis, our method then applies an elimination procedure [23] to filter out incorrect candidates. If the original procedure is still not typable, the process is repeated, unrolling it $k+1$ times. For this example, with the above refinement candidate, we can correctly verify the pre-condition of `f` in `iteri`. Since the theorem prover can use the case condition to know $\mathsf{length}(\mathtt{xs}) > 0$ and based on the invariant $\mathtt{i} + \mathsf{len}(\mathtt{xs}) = \mathsf{len}(\mathtt{a})$, it can determine that $\mathtt{i} < \mathsf{len}(\mathtt{a})$ must hold. Our method finally generates the appropriate dependent type for `iteri` as:

$$\begin{aligned}\mathtt{iteri} : \mathtt{i} : \mathsf{int} \to \{\mathtt{xs} \ : \ '\mathtt{a}\ \mathsf{list}|\ \mathtt{i} + \mathsf{len}(\nu) = \mathsf{len}(\mathtt{a})\} \to \\ \{\mathtt{f} \ : \ \{\mathtt{f}_{\mathtt{arg}_1} \ : \ \mathsf{int} \mid 0 \leq \nu < \mathsf{len}(\mathtt{a})\} \to '\mathtt{a} \to \mathsf{unit}\} \to \mathsf{unit}\end{aligned}$$

Note the invariant generation module is only invoked when our system diverges during the verification of a recursive procedure. We differ from [27] in two respects: first, [27] does not use an elimination procedure since it tries to infer dependent types for the original program using a whole program analysis; second, we only infer refinement candidates for a non-recursive unrolled code fragment instantiated upon divergence, instead of the original whole program, greatly reducing the number of instances where interpolation computation is required.

5 Implementation

We have implemented our verification system in Popeye. Popeye takes as input an SML program (not necessarily closed) and outputs specifications inferred for the procedures defined by the program. We have provided specifications for built-in primitive datatypes as well as arrays, lists, tuples, and records that are used to bootstrap the inference procedure. The Yices theorem prover is used as the verification engine. CSIsat [5] is employed to generate interpolations when inferring candidate refinements for recursive procedures and loops. The implementation is incorporated within the MLton whole-program optimizing compiler toolchain and consists of roughly 14KLOC written in SML[2].

5.1 Case Study: Bit Vectors

To gauge Popeye's utility, we applied it to an open-source bit vector library (bitv) [6] (version 0.6). A bit vector is represented as a record of two fields, `bits`, an array containing vector's elements, and `length`, an integer that represents the number of bits that the vector holds. Operations on bit vectors should enforce the invariant that $\mathtt{(bits.length - 1)} \cdot \mathtt{b} < \mathtt{bits.length} \cdot \mathtt{b}$, where `b` is a constant that defines the number of bits intended to be stored per array element.

[2] The Popeye implementation is available at `http://code.google.com/p/popeye-type-checker/`

This invariant is assumed for all procedures. POPEYE successfully type checks the program combined with 5 manually generated preconditions (for recursive procedures as prover [5] cannot deal with mod operation heavily used in the library) by relatively longer verification time than that of DSOLVE [23] in this benchmark; however DSOLVE requires manual addition of extra 14 user-supplied qualifiers.

Bug Detection. Without any programmer annotations, POPEYE discovered an array out-of-bounds error that occurs in the `blit` function:

```
fun blit {bits=b1, length=l1} {bits=b2, length=l2}
         ofs1 ofs2 n =
  if n < 0 || ofs1 < 0 || ofs1 + n > l1
           || ofs2 < 0 || ofs2 + n > l2
  then assert false
  else unsafe_blit b1 ofs1 b2 ofs2 n
```

This function calls `unsafe_blit` only if a guard condition that checks that all offset value and the number of bits (`n`) to be copied are positive, and that the range of the copy fit within the bounds of the source and target vectors. The counterexample reported for blit procedure corresponds to an input as {length (b1)=2, length (b2)=0, l1=60, ofs1=32, l2=0, ofs2=0, len=0}. The guard holds under this assignment, but because `unsafe_blit` attempts to access the offset in the target bit-vector that is the starting point for the copy, before initiating the copy loop, an array out-of-bounds exception gets thrown. In this example, POPEYE reports a test case that serves as a witness to the bug, and can help direct the programmer to identify the source of the error. The primary novelty of this technique in this regard is its ability to generate a precise counterexample path with concrete inputs that serve as a witness to the violation without requiring explicit user confirmation as DSOLVE.

Complex Refinement Generation. Procedure `unsafe_blit` found in this library tries to copy `n` bits starting at offset `ofs1` from bit-vector `v1` to bit-vector `v2` with target offset `ofs2`. POPEYE discovers the following precondition:

$$((\mathsf{ofs2} + \mathsf{n}) - 1)/\mathsf{b}) < \mathsf{v2.length}$$

This is a non-trivial specification comprised of refinements that we believe would be difficult, in general, for programmers to construct. Systems such as DSOLVE require users to provide these qualifiers explicitly. The ability to generate non-trivial refinements automatically only using counterexamples is an important distinguishing feature of our approach compared to e.g., Liquid Types.

5.2 Experimental Results

To test its accuracy, we have applied POPEYE to a number of synthetic SML programs from the benchmark suite used to evaluate MOCHI [18]. While these

benchmarks are small (typically less than 100 LOC), they exercise complex control- and dataflow, and exploit higher-order procedures heavily, in ways intended to make dependent type inference challenging. Details of these benchmarks are provided in [18]. In the table, column num_ref denotes the number of refinements discovered by POPEYE. num_cegar shows how many iterations of the refinement loop were necessary for POPEYE to converge. prover_call gives the number of theorem prover calls; there are typically more prover calls than CEGAR loop iterations because the results of a counterexample usually entails propagation of newly discovered invariants to other contexts, thus requiring reverification (and hence additional theorem prover calls). cegar_time shows the time spent on refinement loops. run_time gives the total running time taken.

The first seven benchmarks shown in Table 1 cannot be verified by DSOLVE using its default set of simple qualifiers since either context-sensitive dependent types or non-trivial invariants are required. The last two of these seven (suffixed with `-e`) are buggy, and thus cannot cannot also be automatically proved by DSOLVE. The last two benchmarks requires recursive procedure invariants which can be synthesized by our invariant generation module. Here, a single unrolling of the recursive procedure in repeat-e was sufficient to witness the error; in contrast, POPEYE required three unrollings of the recursive procedure in array-init to find a suitable set of candidate refinements. We note that MOCHI fails to verify the `array-init` program. While MOCHI can also verify the first eight benchmarks in this table, its formulation is a bit more complex than ours, and does not easily generalize to deal with data structures and user-level datatypes.

6 Related Work

There has been much work on the use of dependent types for checking complex safety properties of ML programs. Freeman and Pfenning [10] describe a refinement type inference scheme defined in terms of an abstract interpretation over a programmer-specified lattice of refinements for each ML type, and a restricted use of intersection types to combine these refinements that still preserves decidability of type inference. DML [28] is a conservative extension of ML's type system that supports type checking of programmer-specified dependent types; the system supports a form of partial type inference whose solution depends upon the set of refinements found in a linear constraint domain.

To reduce the annotation burden imposed by systems like DML, Liquid Types [23,14] requires programmers to only specify simple candidate qualifiers from which more complex dependent types defined as conjunctions of these refinements are inferred by a whole program abstract interpretation. Our approach differs from liquid types in four important respects: (1) we attempt to infer refinements, (2) a counterexample path together with a test case can be reported as a program bug witness; (3) the type refinement fixpoint loop enables compositional verification, propagating specifications via dependent subtyping chains on demand; (4) the dependent types we inferred are context-sensitive.

Broadly related to our goals, HMC [13] also borrows techniques from imperative program verification to verify functional programs. It does so by reducing

Table 1. Benchmark Results

Program	num_ref	num_cegar	prover_call	cegar_time	run_time
fhnhn	3	4	35	0s	0.014s
neg	15	20	230	0.004s	0.18s
max	10	11	175	0.005s	0.95s
r-file	11	21	205	0.012s	1.56s
r-lock	10	18	108	0.006s	0.60s
r-lock-e	13	18	113	0.01s	0.68s
repeat-e	39	18	237	0.11s	4.87s
list-zip	2	4	149	0.01s	1.55s
array-init	35	106	3617	0.03	102.3s

the problem of checking the satisfiability of the constraints generated in a liquid type system to a safety checking problem of a simple imperative program. However, the translated imperative program loses the structure of the original source semantics. Thus, it is not obvious how we might convert a counterexample reported in the translated program into the original source for debugging.

Terauchi [26] also proposes a counterexample-guided dependent type inference scheme, albeit based on a whole-program analysis. A counterexample in his approach is an "unwound" slice of the program that is untypable using the current set of candidate types, rather than a counterexample path. Since the unfolded program may be involved in multiple program paths, many of which may not be relevant to the verification obligation, it would appear that the size of the constraint sets that needs to be solved may become quite large.

There has been much recent interest in using higher-order recursion schemes [17,19] to define expressive model-checkers for functional programs. In [18,22], predicate abstraction is proposed to abstract higher-order program with infinite domains like integers to a finite data domain; the development in these papers is limited to pure functional programs without support for data structures. Model checking arbitrary μ-calculus properties of finite data programs with higher order functions and recursions can be reduced to model checking for higher-order recursion schemes [17]. Finding suitable refinements relies on a similar constraint solving to [26,27] for a straight-line higher-order counterexample program. Such techniques involve substantial re-engineering of first-order imperative verification tools to adapt them for a higher-order setting.

One important motivation for our work is to reuse well-studied imperative program verification techniques. For example, predicate abstraction [11] has been effectively harnessed by tools such as SLAM [2] and BLAST [4] to verify complex properties of imperative programs with intricate shape and aliasing properties. Software verification tools, such as Boogie [3], ESC/Java [9] and CALYSTO [1] construct first order logic formula to encode a program's control flow. If a verification condition, expressed via programmer-specified assertions or specifications, cannot be discharged, the counterexample path can be used to refine and strengthen it.

7 Conclusion

In this paper, we present a compositional inter-procedural verification technique for functional programs. We use dependent type checking rules to generate dependent type templates for local expressions inside a procedure. Dependent subtyping rules are then used to generate verification conditions. From an unprovable verification condition, we can construct a counterexample path to infer dependent types for procedure arguments and results, and to propagate inferred specifications between procedures and call-sites where they are applied. Thus, our technique effectively leverages a variety of strategies used in the verification of first-order imperative programs within a higher-order setting.

Acknowledgements. We thank Ranjit Jhala, Aditya Nori, and Francesco Zappa-Nardelli for many useful comments and discussions. This work was supported in part by the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370.

References

1. Babić, D., Hu, A.J.: Structural Abstraction of Software Verification Conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 366–378. Springer, Heidelberg (2007)
2. Ball, T., Bounimova, E., Kumar, R., Levin, V.: SLAM2: Static Driver Verification with Under 4% False Alarms. In: FMCAD, pp. 35–42 (2010)
3. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
4. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The Software Model Checker Blast: Applications to Software Engineering. Int. J. Softw. Tools Technol. Transf. 9, 505–525 (2007)
5. Beyer, D., Zufferey, D., Majumdar, R.: CSIsat: Interpolation for LA+EUF. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 304–308. Springer, Heidelberg (2008)
6. `http://www.lri.fr/~filliatr/software.en.html`
7. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
8. Damas, L., Milner, R.: Principal Type-Schemes for Functional Programs. In: POPL, pp. 207–212 (1982)
9. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI, pp. 234–245 (2002)
10. Freeman, T., Pfenning, F.: Refinement Types for ML. In: PLDI, pp. 268–277 (1991)
11. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
12. Jagannathan, S., Weeks, S.: A Unified Treatment of Flow Analysis in Higher-Order Languages. In: POPL, pp. 393–407 (1995)

13. Jhala, R., Majumdar, R., Rybalchenko, A.: HMC: Verifying Functional Programs Using Abstract Interpreters. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 470–485. Springer, Heidelberg (2011)
14. Kawaguci, M., Rondon, P., Jhala, R.: Type-based Data Structure Verification. In: PLDI, pp. 304–315 (2009)
15. Knowles, K., Flanagan, C.: Type Reconstruction for General Refinement Types. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 505–519. Springer, Heidelberg (2007)
16. Kobayashi, N.: Model-Checking Higher-Order Functions. In: PPDP, pp. 25–36 (2009)
17. Kobayashi, N.: Types and Higher-Order Recursion Schemes for Verification of Higher-Order Programs. In: POPL, pp. 416–428 (2009)
18. Kobayashi, N., Sato, R., Unno, H.: Predicate Abstraction and CEGAR for Higher-Order Model Checking. In: PLDI, pp. 222–233 (2011)
19. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order Multi-Parameter Tree Transducers and Recursion Schemes for Program Verification. In: POPL, pp. 495–508 (2010)
20. Martin-Löf, P.: Constructive Mathematics and Computer Programming (312), 501–518 (1984)
21. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
22. Ong, C.H.L., Ramsay, S.J.: Verifying Higher-Order Functional Programs with Pattern-Matching Algebraic Data Types. In: POPL, pp. 587–598 (2011)
23. Rondon, P., Kawaguci, M., Jhala, R.: Liquid Types. In: PLDI, pp. 159–169 (2008)
24. Sharir, M., Pnueli, A.: Two Approaches to Interprocedural Data Flow Analysis. In: Program Flow Analysis (1981)
25. Shivers, O.: Control-Flow analysis in Scheme. In: PLDI, pp. 164–174 (1988)
26. Terauchi, T.: Dependent types from Counterexamples. In: POPL, pp. 119–130 (2010)
27. Unno, H., Kobayashi, N.: Dependent Type Inference with Interpolants. In: PPDP, pp. 277–288 (2009)
28. Xi, H., Pfenning, F.: Dependent Types in Practical Programming. In: POPL, pp. 214–227 (1999)