ICFP 2019 Paper #137 Reviews and Comments
===
Paper #137 Program Synthesis by Type-Guided Abstraction Refinement

Review #137A
===

Overall merit

C. Weak paper, though I will not fight strongly against it.

Reviewer expertise

Y. I am knowledgeable in the area, though not an expert.

Paper summary

This paper proposes a method for generating a term of a given type that can use components in a given library. This paper aims to synthesise Haskell code, of which libraries contain polymorphic functions. Existing methods are not applicable to this setting; an approach based on proof search has a scalability issue and another method based on Petri-net reachability can deal with only monomorphic components. A naive approach is to regard a polymorphic function as a family of monomorphic functions, but this approach does not work as the expansion significantly increases the size of libraries (as the number of types is usually countably infinite, the result of the naive expansion consists of countably infinite components).

The method of this paper is based on the framework known as CEGER. So the method first tries to solve an abstracted problem, whose solution set is an overapproximation of the original problem, and if the found solution is spurious, then it refines the abstraction and again tries to solve it.

This paper introduces the notion of abstract types to describe an abstraction. The syntax of abstract types is basically the same as concrete types but it has a special type *, which matches any type. (In fact, this paper considers a bit more expressive syntax.) Then an abstract type naturally describes a set of concrete types. An abstraction is determined by a finite set {A1, ..., An} of abstract types such that every concrete type matches exactly one of Ai. In the abstracted problem, a component of type T -> U is regarded as that of type Ai -> Aj, where Ai and Aj match T and U, respectively. A polymorphic function is handled by expansion, but expansion in the abstracted problem is tractable.

The refinement process takes a spurious solution to the abstract problem and returns a refined abstraction. A spurious solution is a term e that is not typable but its untypability cannot be observed in the current abstraction. Then there should be a subterm e' that has type T but is required to have type T', where T and T' match the same abstract type Ai. Such a subterm can be found by the standard type-inference algorithm and the refinement is given by a sufficiently deep pattern that separate T and T'. This process should be inductively applied to subterms of e'.

This paper solves (abstracted) problems by reducing it to the Petri-net reachability problem. This idea have already be seen in the previous work for the monomorphic case. The Petri-net reachability problem is solved by describing "reachability in k steps" by a formula, which can be decided by an SMT solver, and iteratively increasing the bound k when no path is found. The logical formula naively encodes the existence of a path by using integer variables p[i,j] representing the number of tokens on the place i in the j-th step. This paper also discusses "incremental encoding ', where refinement only requires adding constraints, enabling us to use the incremental mode of an SMT solver.

The proposed method is implemented and experimental results are reported.

Comments for author

The topic of this paper is in the scope of ICFP. The problem addressed in this paper is important, and the ideas given in the paper are nontrivial and interesting. The paper is extremely well-written, and accessible to audiences of ICFP.

My concern is about correctness of theoretical results. This paper claims implicitly that, when a term e is used as a spurious solution for refinement, then the refined abstraction does not have e as a solution. This result is a key to the completeness result (Theorem 3.7, page 12). It seems to me that the following example is a counterexample to this claim. Assume three type constructors B, U and Z whose arities are 2, 1 and 0, respectively.

- Component library = { f : \forall a. B a a, g : \forall b. B (U b) b -> Z }
- Query type: Z
- Spurious solution: g f

I could not find any abstraction that refutes this spurious solution. Indeed, because any abstraction is a finite set of finite types, there exists a bound d (determined by an abstraction) such that two concrete types have the same abstraction whenever they are equivalent up to depth d. Since f : B (U^n Z) (U^n Z) and
g : B (U^(n+1) Z) (U^n Z) -> Z, one should have |- g f : Z in any abstraction.

I would like to know about the following points as well:
- I think the principal type of an expression may contain a type variable, but the algorithm in Fig.9 seems to assume that it is closed. For example, type(e') is passed to Split as the third argument (p.13, l.592) and the third argument of Split appears in the right-hand-side of the abstract unification relation (p.13, l.597), but the abstract unification relation assumes that only the left-hand-side contains a type variable (p.10, l.482). What does occur when e' contains a type variable?
- In the second case of Split (p.13, l.599), when (C A1 ... An) and (C B1 ... Bn) are the second and the third argument, the first step seems to collect all pairs (Ai, Bi) such that Ai does not match Bi. In general, there is no such pair even if (C A1 ... An) does not match (C B1 ... Bn). Consider, for example, (C a a) and (C int bool), where a is a type variable and int and bool are type constants. What does occur in this case?

Section 3 focuses on the first-order setting, but I am interested in a precise formulation of the higher-order setting.

In summary, I think the technical details of this paper are premature and I do not support this paper. I do not oppose to accepting this paper when the above points are addressed by author's response and/or the finial revision, because ideas in this paper is of practical interest.

* Minor comments

p.6, l.286
"of a given element a list" --> "in a list"?

p.9, Fig.6,
In the conclusion of the rule (COMP), the subscript on |- is missing.

p.10, l.455
The definition of the one-step transition of Petri-net looks wrong, when there is a place which is both input and output of a transition. This applies to the encoding in Section 4 as well.

p.14, l.679
"x \in {P \cup T}" --> "x \in P \cup T"

p.14, definition of \phi
(2) and (3) look wrong for the case of a place being both input and output of a transition.

p.19, l.906
The inhabitation problem corresponds to provability, not to satisfiability, of a logical formula.

Review #137B
==========

Overall merit

C. Weak paper, though I will not fight strongly against it.

Reviewer expertise

X. I am an expert in the subject area of this paper.

Paper summary

Author(s) present techniques for type-directed component-oriented synthesis.
The main technical contribution consists in a techique for performing type-guided
abstraction refinement ("tygar") to help scale up type-directed synthesis
in the presence of polymorphic datatypes and components. Polymorphic types are a fundamental challenge
for component-oriented type-based synthesis, because it is difficult to bound the set of instances
of (possibly large collections of) component types relevant to a given synthesis goal.
The specific approach of author(s) is motivated by stating that classical approaches founded on proof search in
intuitionistic logics (i.e. solving the inhabitation problem in a type theory) do not scale. Similarly, recent methods based on
graph-reachability do not elledgedly scale in the presence of polymorphic libraries. Type-guided abstraction refinement
as proposed by author(s) is based on the idea of introducing so-called abstract type constructors to represent potentially
unbounded sets of instances thereby increasing scalability. Abstract type constructors can be endowed with a refinement system
such that
iterative refinement of abstractions become possible. The key idea of the paper is to use such a system to perform
iterative inhabitant search starting from coarse (and efficient) abstractions, refining on-the-fly upon failed
inhabitation goals directing further refinement by diagnosing causes of inhabitation failure by means of proofs of untybeability
for solution candidates. A central technical idea in abstraction refinement as proposed here
is to use "negative information" in abstract type constructors such as an expression !{C1 ... Cn} denoting the set of types
whose outermost constructor is not among the C1, ..., Cn. That way, different levels of detail and different patterns of types
can be represented, underlying refinement. The described technique is developed in the setting of a Petri-net based
representation of component structure (so-called type transition nets, TTNs) which is adapted from the literature
in the form of a system ("SyPet") for syntesis of Java programs. A major challenge
discussed by author(s) consists in adapting such strategies to a setting with polymorphism and higher-order types.
Doing so is important in the setting of the present paper, because author(s) are interested in functional synthesis
for Haskell, implemented in a tool,H+, which takes as input a set of Haskell libraries and a
query type and (in the successful case) returns a Haskell term that uses functions from the provided libraries to implement the
query
type. The tool H+ is evaluated on a benchmark of 35 queries collected from Hoogle (a widely used online API search tool for
Haskell
libraries) together with a set of popular Haskell
libraries with a total of 244 components. Author(s) conclude that the experimental results demonstrate that tygar allows H+ to
rapidly return
well-typed terms, with a median time to first solution of 2.2 seconds.

Comments for author

On the positive side,
the general motivation from scalability in component-based synthesis in the presence of polymorphic and higher-order components
is
strong. In particular, it is indeed recognized as a major problem (which has been discussed in the literature several times) to
scale
synthesis from polymorphic libraries. The experimental results of the paper should be useful.
The paper is overall reasonably well written.

On the negative side, I have a number of reservations about the paper which I will now summarize:

1. The idea of using abstraction refinement in synthesis is not new per se, see Wang, Dillig, Singh. Program synthesis using
abstraction refinement, PACMPL 2, POPL (2018).

2. The specific proposal described by author(s) strikes me as being foremostly an engineering contribution which does not appear
to contain any one particularly striking new idea in way of a transferrable innovation (i.e. one which is independent
of the specific engineering framework chosen). Thus, for example, even though author(s) place their work in the context of

type-theoretic inhabitation, it does not seem to be possible to identify any transferrable type-theoretic contribution.
The formal framework contains only the first-order fragment of simple type theory, even though (as discussed by author(s)) the higher-order generalization appears to be non-obvious in the context of Petri-net models. I find it difficult to discover from the engineering descriptions what exactly is going on here.
In this sense, the work strikes me as being certainly not theoretically striking.
Part of the reason may be found in the somewhat peculiar mix of Petri-net technology and type structure which does not strike me as theoretically productive (but it is, of course, a framework in which one can engineer a number of techniques). Another aspect of this character of the work is found in the fact that the central algorithm is a semi-decision procedure (the underlying decision problem is undecidable, as shown in Appendix A). However, I could find no other treatment of this aspect than the remark (on p. 12, l. 578-579) that "In practice, it makes to impose an upper bound on the length of the solution, which then guarantees termination".

3. The bibliographic work in this paper strikes me as very incomplete. I will give a few representative examples but the list is not exhaustive. For example, I find the omission of any reference to the following work surprising: Frankle, Osera, Walker, Zdancewic: Example-directed synthesis: a type-theoretic interpretation. POPL 2016: 802-815.
True, this work is about example-directed synthesis, but it has a lot to say about type-directed synthesis in general (including, in fact,
a discussion of challenges in dealing with polymorphic libraries). As another example, I find the discussion of the combinatory approach referenced in Heineman et al. 2016 to be very truncated - the focus there on object-oriented synthesis is only one out of many further contributions in this line of work.

Review #137C
===

Overall merit

B. OK paper, but I will not champion it.

Reviewer expertise

Z. I am not an expert. My evaluation is that of an informed outsider.

Paper summary

The paper considers the problem of synthesizing programs that satisfy
a certain type signature given a library of primitive functions.
It extends on earlier work in being able to handle parametric polymorphism,
using a new technique of type-guided abstraction refinement (TYGAR) to
prevent search space explosion and make the synthesis feasible in practice.
Using their tool H+ the authors evaluate their approach on queries found
in practice.

Comments for author

The paper is well written with many concrete examples. The TYGAR technique
is novel and the authors demonstrate the usefulness using H+ to evaluate the
performance in practice. The SMT encoding, and the incremental refinement
is smart and it is great how well TYGAR fits with SMT solvers.

The evaluation is still a bit weak as there are only 35 queries evaluated
from the Hoogle set of queries. (I guess this is due to the need to manually
filter out invalid queries?). Also, there are just 6 queries to establish if
TYGAR finds correct/useful results where 2 of them fail to produce the correct
one within the first 5 solutions. The paper would be stronger if there would
be more queries in this set as this is what one actually wants in practice.
Moreover, the Hoogle set contains quite a few strange queries where the type
makes no sense to derive a reasonable program, like (13) `(a -> b) -> a -> a -> a`
(where the `a->b` cannot be used), or (19) `(a -> a) -> a -> Int -> a` where
the `Int` could be interpreted in many ways, or (32) `Maybe a -> b -> b`,
or (33) `(a -> b) -> a -> a`, or (37) `a -> [String] -> String`
(and 12, 32, 33, and 37) are so similar in this respect).

There are also still quite a few heuristics and limitations. Perhaps it would
be good to discuss these in the introduction or conclusion all together also
as pointers to future work. In particular, instantiating type variables with
function types, work with type classes, special handling of tuples,
and the splitting into two tiers.

Page 18; it is said that query (39) `(a->b, a) -> b` needs the Eq typeclass
but it seems just a curried application? Why would TYGAR fail in this case?
(I guess this is actually due to the function type inside the tuple but I
couldn't identify the query that needs Eq in that case, is it (37)?)