Chapter 6 (draft): Notes, Future work, Conclusion

Mark Lemay

December 31, 2021

Introduce this section

Part I

Symbolic Execution

One of the advantages of type checking is the immediacy of feedback. We have outlined here a system that will give warning messages immediately, but requires evaluation to give the detailed error messages that are most helpful when correcting a program. This is especially important if the user wants to use the system as a proof language, and will not generally execute their proofs. A symbolic evaluation system recaptures some of that quicker feed back, by having a system that passively tries to find errors. This ideal workflow appears in table 1.

Since this procedure operates over the cast language, we must decide what constitutes a reasonable testing environment

A one hole context can be defined for the cast languages presented in this thesis $C[-]$ and we can say that an error is observable if $\vdash a : A$ and $\vdash C[a] : B$ and $C[a] \rightsquigarrow_* b$ and $\mathbf{Blame}\, \ell\, o\, b$ for some $\ell \in \mathit{lables}\,(a)$ but $\ell \notin \mathit{lables}\,(C[-])$. cite

This process is semi-decidable in general (by enumerating all well typed syntax). But testing every term is infeasibly inefficient for functions and types. An approximate approach can build partial contexts, that are correct up to some constraint. For instance,

- If we have the "empty" type, $\lambda x \Rightarrow \star ::_\ell x \quad : (x : \star) \to x$, then we can observe an error by applying x of type $x : \star$ to the term and $x = - \to -$, which would correspond to the context $[\lambda x \Rightarrow \star ::_\ell x]\, (\star \to \star) \rightsquigarrow_* \star ::_{\star,\ell} (\star \to \star)$ and $\mathbf{Blame}\, \ell\, .\, \star ::_{\star,\ell} (\star \to \star)$

- This reasoning can be extended to higher order functions $\lambda f \Rightarrow \star ::_\ell f \star :: \star \quad : (\star \to \star) \to \star$, then we can observe an error by applying f of type $f : \star \to \star$ to the term and $f \star = - \to -$, which would correspond to the context $[\lambda f \Rightarrow \star ::_\ell f \star :: \star]\, (\lambda x \Rightarrow (\star \to \star)) \rightsquigarrow_* \star ::_{\star,\ell} (\star \to \star)$ and $\mathbf{Blame}\, \ell\, .\, \star ::_{\star,\ell} (\star \to \star)$

- Observing a higher order input directly, $\lambda\, f \Rightarrow f\, (!!) \quad : (\star \to \star) \to \star$, and can observe an error by inspecting

```
               edit program
                |
                |
                v
 testing   <-- Elaboration                          ^
    |  ^        |                  |                | runtime error
    -           |  no warnings     |  type warnings |
                v                  v                |
                   run program     __________________
```

better graphics

Figure 1: Ideal Workflow

its input, which would correspond to the context $[\lambda\, f \Rightarrow f\,(!!)]\,(\lambda x \Rightarrow x) \rightsquigarrow_{*} !!$ which is blamable. Where !! will stand infor any blamabme term (here $!! = \star :: (\star \to \star) :: \star$)

- Similarly with dependent types, $!! \to \star \quad : \star$, can observe an error by inspecting its input, which would correspond to the context $((\lambda x \Rightarrow x :: \star) :: [!! \to \star] :: (\star \to \star))\,\star \rightsquigarrow_{*} \star :: [!!] :: \star$ which is blamable.
- Similarly we will allow extraction from the dependent body of a function type $(b : \mathbb{B}_c) \to b\,\star\,!!\,\star \quad : \star$, by symbolically applying b where $b : \mathbb{B}_c, b\,\star = y$ leaving $y\,!!\,\star$ wich can obseve blame via y's first argument.
- data can be observed incrementally, and paths along data will confirm that the data conductors are consistent

The procedure insists that the following constraints hold,

- variables and assignments are type correct
- observable different outputs must come from observably different inputs (in the case of dependent function types, the argument should be considered as an input)
-

um out-

This seems to be good because the procedure,

- can guide toward the labels of interest, for instance we can move to labels that have not yet observed a concrete error. Terms without lables can be skipped entirely.
- can choose assignments strategically avoiding or activating blame as desired
- Since examples are built up partially the partial contexts can avoid introducing their own blame by construction
- handle higher order functions, recursions, and self reference gracefully. For instance $f : Nat \to Nat$, $f\,(f\,0) = 1$ and $f\,(3) = 3$ if there is an assignment that implies$f\,0 \neq 3$

However the procedure is unsound, it will flag errors that are not possible to realize in a context,

- However since there is no way for a terms within the cast language to "observe" a distinction between the type formers plausible environments cannot always be realized back to a term that would witness the bad behavior. For instance, the environment $F : \star \to \star, F\,\star = \star \to \star, F\,\,(\star \to \star) = \star$, which might be needed to explore the term with casts like $\lambda F \Rightarrow ... :: \star :: F\,(\star \to \star) ...... :: (\star \to \star) :: F\,\star \quad : (\star \to \star) \to ...$, cannot be realized as a closed term. In this way the environment is stronger then the cast language. The environment reflects a term language that has a type case construct.

add non termination as a source of unrealizability

- Additionally, working symbolically can move past evaluations that would block blame in a context. For instance that procedure outlined above would allow !! to be reached in $(\lambda x f \Rightarrow f(!!))\,loop \quad : \star \to \star$ even though there is no context that would allow this to cause blame.

add por as a source of unrealizability

- Subtly a version of parallel-or can be generated via assignment even though such a term is unconstructable in the languge, $por : \mathbb{B} \to \mathbb{B} \to \mathbb{B}, por\ loop\ tt = tt, por\ tt\ loop = tt, por\ ff\ ff = ff$. Here all assignments are well typed, and each output can be differentiated by a different input.

pro-
age,
puter
d in
ghton.us/research/por.pdf

1 Related and Future Work

While formalizing a complete and efficient testing procedure along these lines is still future work. There are likely insights to be gained from the large body of research on symbolic execution, especially work that deals with typed higher order functions. A fully formal account would deal with a formal semantics of the cast language.

1.1 Testing

Many of the testing strategies for typed functional programming trace their heritage to **property based** testing in QuickCheck [CH01]. Property based testing involves writing functions that encode the properties of interest, and then randomly testing those functions.

- QuickChick [1] [DHL$^+$14][LPP17, LGWH$^+$17, Lam18] uses type-level predicates to construct generators with soundness and completeness properties, but without support for higher order functions. However, testing requires building types classes that establish the properties needed by the testing framework such as decidable equality. This is presumably out of reach of novice Coq users.
  - Current work in this area uses coverage guided techniques in [LHP19] like those in symbolic execution. More recently Benjamin Pierce has used Afl on compiled Coq code as a way to generate counter examples[2].

review thi
there's a
now?

- [DHT03] added QuickCheck style testing to Agda 1.

1.2 Symbolic Execution

Symbolic evaluation is a technique to efficiently extract errors from programs. Usually this happens in the context of an imperative language with the assistance of an SMT solver. Symbolic evaluation can be supplemented with other techniques and a rich literature exists on the topic.

The situation described in this chapter is unusual from the perspective of symbolic execution:

- the number of blamable source positions is limited by the location tags. Thus the search is error guided, rather then coverage guided.
- The language is dependently typed. Often the language under test is untyped.
- The language needs higher order execution. often the research in this area focuses on base types that are efficiently handleable with an SMT solver.

This limits the prior work to relatively few papers

- A Symbolic execution engine for Haskell is presented in [HXB$^+$19], but at the time of publication it did not support higher order functions.
- A system for handling higher order functions is presented in [NTHVH17], however the system is designed for Racket and is untyped. Additionally it seems that there might be a state space explosion in the presence of higher order functions.
- [**?**] extended and corrected some issues with [NTHVH17], but still works in a untyped environment. The authors note that there is still a lot of room to improve performance.
- Closest to the goal here, [LT20] uses game semantics to build a symbolic execution engine for a subset of ML with some nice theoretical properties.
- An version of the above procedure was experimented with in the extended abstract , however conjectures made in that preliminary work were false (the procedure was unsound). The underlying languages has improved substantially since that work.

cite

The subtle unsoundness hints that the approach presented here could be revised in terms of games semantics (perhaps along lines like [LT20]). Though game semantics for dependent types is a complicated subject in and of itself . Additionally is seems helpful to allow programmers to program their own solvers that could greatly increase the search speed.

cite

[1] https://github.com/QuickChick/QuickChick

[2] https://www.youtube.com/watch?v=dfZ94N0hS4I

Part II

Runtime Poof Search

Just as "obvious" equalities are missing from the definitional relation, "obvious" proofs and programs are not always conveniently available to the programmer. For instance, in Agda it is possible to write a sorting function quickly using simple types. With effort is it possible to prove that sorting procedure correct by rewriting it with the necessarily dependently typed invariants. However, very little is offered in between. The problem is magnified if module boundaries hide the implementation details of a function, since those details are exactly what is needed to make a proof! This is especially important for larger scale software where a library may require proof terms that while true are not provable from the exports of other libraries.

The solution proposed here is additional syntax that will search for a term of the type when resolved at runtime. Given the sorting function

$$\texttt{sort} : \texttt{List}\,\mathbb{N} \to \texttt{List}\,\mathbb{N}$$

and given the first order predicate that

$$\texttt{IsSorted} : \texttt{List}\,\mathbb{N} \to *$$

then it is possible to assert that sort behaves as expected with

$$\lambda x.? : (x : \texttt{List}\,\mathbb{N}) \to \texttt{IsSorted}\,(\texttt{sort}x)$$

This term will act like any other function at runtime, given a List input the function will verify that the sort correctly handled that input, or the term will give an error, or non-terminate.

Additionally, this would allow simple prototyping form first order specification. For instance,

$$\begin{aligned} &\textit{data}\ \texttt{Mult} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to *\ \textit{where} \\ &\qquad \texttt{base} : (x : \mathbb{N}) \to \texttt{Mult}\ 0\ x\ 0 \\ &\qquad\ \ \texttt{suc} : (x\,y\,z : \mathbb{N}) \to \texttt{Mult}\,x\,y\,z \to \texttt{Mult}\ (1+x)\ y\,(y+z) \end{aligned}$$

can be used to prototype

$$\texttt{div} = \lambda z.\lambda x.\texttt{fst}\left(? : \sum y : \mathbb{N}.\texttt{Mult}x\,y\,z\right)$$

The symbolic execution described above can precompute many of these solutions in advance. In some cases it is possible to find and report a contradiction.

Experiments along these lines have been limited to ground data types, and a an arbitrary solution is fixed for every type. Ground data types do not need to to worry about the path equalities since all the constructors will be concrete.

Non ground data can be very hard to work with when functions or function types are considered. For instance,

$$? : \sum f : \mathbb{N} \to \mathbb{N}.\texttt{Id}\,(f, \lambda x.x+1)\,\&\texttt{Id}\,(f, \lambda x.1+x)$$

It is tempting to make the ? operator sensitive to more then just the type. For instance,

```
n  :  Nat;
n  =  ?;

pr  :  Id  Nat  n  1;
pr  =  refl  Nat  1;
```

will likely give the waning error "n =?= 1 in Id Nat $\underline{\text{n}}$ 1". It will then likely give the runtime error "0=!=1". Since the only information to solve ? is the type Nat and an arbitrary term of type nat will be 0. Most users would expect n to be solved for 1.

However constraints assigned in this manner can be extremely non-local. For instance,

```
n : Nat;
n = ?;

...

pr : Id Nat n 1;
pr = refl Nat 1;

...

pr2 : Id Nat n 2;
pr2 = refl Nat 2;
```

And thing become even more complicated when solving is interleaved with computation. For instance,

```
n : Nat;
n = ?;

prf : Nat -> Nat ;
prf x = (\ _ => x) (refl Nat x : Id Nat n x);
```

2 Prior Work

Proof search is often used for static term generation in dependently typed languages (for instance Coq tactics). A first order theorem prover is attached to Agda in [Nor07]. However it is rare to make those features available at runtime.

Logic programing languages such as Prolog[3], Datalog[4], and miniKanren[5] use "proof search" as their primary method of computation. Dependent data types can be seen as a kind of logical programming predicate. The Twelf project[6] makes use of runtime proof search and has some support for dependent types, but the underling theory cannot be considered full-spectrum. The only full spectrum systems that preport to handle solving in this way are the gradual dependent type work, though it is unclear how that work handles the non locality of constraints given their local ? operator.

cite the C

Part III
Convenience

implicit function arguments

Part IV
Future work

3 Effects

The last and biggest hurdle to bring dependent types into a mainstream programming language is by providing a reasonable way to handle effects. Though dependent types and effects have been studied I am not aware of any full spectrum system that has implemented those systems. It is not even completely clear how to best to add an effect system into Haskell, the closest "mainstream" language to the one studied here.

While trying to carefully to avoid effects in this thesis, we still have encountered 2 important effects, non-termination and blame based error.

[3] https://www.swi-prolog.org/

[4] https://docs.racket-lang.org/datalog/

[5] http://minikanren.org/

[6] http://twelf.org/wiki/Main_Page

3.1 Errors

The current system handles blame based runtime errors and static warnings in a unique way. There is no control flow for errors built into the reduction relations, and there is no way to handle an error. Every potential error is linked to a static warning. There are a few features that would be good to experiment with

Allow users to provide proofs of equality to remove warnings by having them define and annotate an appropriate identity type. This would allow the language to act more like an extensional type theory. Just as with ETT many desirable properties such as function extensionality would not be provable. Programmers could justify these proofs as a way to make symbolic execution faster. We have pushed this to future work since their are already many explored strategies for dealing with equality in an Intentional type theory that are completely compatible with the current implementation.

Currently blame based errors aren't handled. Programmers may want to use the information from a bad cast to build the final output, it might even be possible to capture the witness of the bad cast. For instance,

```
f : Vec y -> string;

h : (x : Nat) -> string;
h x =
  handle{
    f (rep x)
  } pr : x != y => "whoops" ;
```

Though additional research would be need for exactly the form the contradictions should take.

However handling effects in a dependent type system is subtle[7] since the handling construct can observe differences that should not otherwise be apparent. This is most clearly seen in the generalization of Herbelin's paradox presented in[PT20]. The paradox can be instantiated to the system presented in this thesis with an additional handling construct and assuming a lazy evaluation order,

e

```
h : (u : Unit) -> Bool;
h u =
  handle{
    case u {
      | tt => true
    }
  } _ => false ;

hIsTrue : (u : Unit) -> Id Bool true (h u);
hIsTrue u =
  case u <u -> Id Bool true (h u)>{
    | tt => refl Bool true
  };

hIsTrue !! : Id Bool true false
```

Interestingly this term is not as bad as the paradox makes out, the term "gets stuck" but with a well blamed runtime error and a static warning is given.

3.2 Non-termination

Non-termination is allowed, but it would be better to have it work in the same framework as equational constraints, namely warn when non-termination is possible, and try to find slow running code via symbolic execution. Then we could say without caveat "programs without warnings are proofs". It might be possible for users to supply their own external proofs of termination[CSW14], or termination metrics.

on casts:
ach to
n general

3.3 Other effects

One of the difficulties of an effect system for dependent types is expressing the definitional equalities of the effect modality. Is print "hello";print "world" = print "helloworld" : IO Unit? by delaying equality checks to runtime these

[7] everything about dependent types is subtle

issues can be avoided until the research space is better explored. Effects risk making computation mathematically inelegant. In this thesis we avoided this inelegance for an error effect with a blame relation. This strategy could perhaps be applied to more interesting effect systems.

Both the symbolic execution and search above could be considered in terms of an effect in an effect system. Proof search could be localized though an effect modality, avoiding the non local examples above.

4 User studies

The main proposition of this work is that it will make dependent types easier to learn and use. This should be demonstrated empirically with user studies.

5 Semantics

The semantics explored in this thesis have been operational, this has lead to serviceable, but cumbersome, proofs. Ideally the denotational semantics of a typed language in this style should be explored. While there has been some exploration into untyped contextual equivalence for dependent types [Sjö15, JZSW10], untyped contextual equivalence is a weak relation . A notion of dependently typed contextual equivalence would be an interesting and helpful direction for further study.

Ian A. Ma
olyn L. Ta
alence in
guages wi
nal of Fun
gramming
1991

References

[CH01] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2001.

[CSW14] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. *ACM SIGPLAN Notices*, 49(1):33–45, 2014.

[DHL$^+$14] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Quickchick: Property-based testing for coq. In *The Coq Workshop*, 2014.

[DHT03] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining testing and proving in dependent type theory. In *International Conference on Theorem Proving in Higher Order Logics*, pages 188–203. Springer, 2003.

[HXB$^+$19] William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. Lazy counterfactual symbolic execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 411–424, New York, NY, USA, 2019. Association for Computing Machinery.

[JZSW10] Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg, and Stephanie Weirich. Dependent types and program equivalence. *SIGPLAN Not.*, 45(1):275–286, January 2010.

[Lam18] Leonidas Lampropoulos. Random testing for language design. 2018.

[LGWH$^+$17] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C Pierce, and Li-yao Xia. Beginner's luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 114–129, 2017.

[LHP19] Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[LPP17] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

[LT20] Yu-Yang Lin and Nikos Tzevelekos. Symbolic Execution Game Semantics. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[NTHVH17] Phuc C Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Higher order symbolic execution for contract verification and refutation. *Journal of Functional Programming*, 27, 2017.

[PT20] Pierre-Marie Pédrot and Nicolas Tabareau. The fire triangle how to mix substitution, dependent elimination, and effects. 2020.

[Sjö15] Vilhelm Sjöberg. A dependently typed language with nontermination. 2015.

Part V

TODO

Todo list

Part VI

notes

Part VII

unused

More subtle is that the procedure described here will allow f to observe parallel or, even though parallel or cannot be constructed within the language. This hints that the approach presented here could be revised in terms of games semantics (perhaps along lines like [LT20]). Though game semantics for dependent types is a complicated subject in and of itself .