Dissertation Prospectus

A Full-Spectrum Dependently Typed Language for Testing with Dynamic Equality

Mark Lemay

Department of Computer Science, Boston University

March 8, 2021

Abstract

Dependent type systems are a powerful tool to eliminate bugs from programs. Unlike other systems of formal methods, dependent types systems can re-use the methodology and syntax that functional programmers are already familiar with for the construction of formal proofs. This insight has lead to several full-spectrum languages that try to present programmers with a consistent view of proofs and programs. However, these languages still have substantial usability issues: missing features like general recursion, confusingly conservative equality, and no straightforward way to test and use properties that have not yet been proven.

These issues are not superficial, but are tied to some of the conventional assumptions of dependent type theory. Logical soundness is essential when dependent types are considered as a mathematical foundation, but may be too restrictive in a programming language. Conservative equality is easy to hand-wave away informally, but it makes many reasonable programs not type check. Users often experience these language properties as inexplicable static errors, and cannot debug their programs using conventional dynamic techniques. In practice exploratory programming is difficult in these systems.

A possible solution is to design a dependently typed language with programmatic features that supports type sound, but logically unsound execution. This language can then be generalized into a dynamic language with more permissive type checking. Programmers can trade poor static errors for precise counterexamples that are made available at runtime. Progress has been made by formalizing the intended relationship between the 2 languages, implementing prototypes of those languages, and proving some properties hold over the formal language.

Contents

1 Introduction

The Curry-Howard correspondence identifies functions with theorems, providing a mutually beneficial link between well explored areas of math and software engineering. This connection is most pronounced in dependent typed systems that provide a common language to write programs about proofs and proofs about programs. Specifically, dependent type systems allow types to depend on terms, supporting an extremely powerful system of specification, computation and proof evidence.

For instance, in a dependently typed language it is possible to prove the correctness of a sorting function

$$sort\ :\ (input : List\,\mathbb{N}) \to \Sigma ls : List\,\mathbb{N}.IsSorted\,input\,ls$$

by providing an appropriate term of that type. Unlike other systems of formal methods, the additional logical power does not require the programmer understand any additional syntax or semantics. From the programmer's perspective the function arrow and the implication arrow are the same. The proof $IsSorted$ is no different then any other term of a datatype like $List$ or $\mathbb{N}$.

The promise of dependent types in a practical programming language has inspired research projects for decades. There have been many formalization and prototypes that make different compromises in the design space. One of the most popular styles is "full-spectrum" dependent types, these languages tend to have a minimalist approach: computation can appear anywhere in a term or type. Such a design purposely exposes the Curry-Howard correspondence, as opposed to merely using it as a convenient logical foundation: a proof has the exact same syntax and behavior as a program. Even though this style makes writing efficient programs hard, and drastically complicates the ability to encode effects, it can be seen in some of the most popular dependently typed programming languages such as Agda and Idris.

Despite the potential, users often find these systems difficult to use. The common symptom of these issues can be seen in the confusing error messages these languages produce. For instance in Agda this reasonable looking program

$$
\begin{aligned}
\texttt{Vec} &: * \to \mathbb{N} \to * \\
\texttt{rep} &: (x : \mathbb{N}) \to \texttt{Vec}\,\mathbb{B}\,x \\
\texttt{head} &: (x : \mathbb{N}) \to \texttt{Vec}\,\mathbb{B}\ (1 + x) \to \mathbb{B} \\
&\nvdash \lambda x.\texttt{head}\,x\ (\texttt{rep}\ (x + 1))\ :\ \mathbb{N} \to \mathbb{B}
\end{aligned}
$$

will give the error "x + 1 != suc x of type $\mathbb{N}$ when checking that the expression rep (x + 1) has type Vec $\mathbb{B}$ (1 + x)". The error is confusing since it objects to an intended property of addition, and if addition were buggy no hints are given to fix the problem. While an expert in type theory can appreciate the subtleties of definitional equality, programmers would prefer an error message that gives a specific instance of x where $x + 1 \neq 1 + x$ or be allowed to run their program.

Strengthening the equality relation in dependently typed languages has motivated many research projects (to name a few [8, 28]). However, it is unlikely those impressive efforts are suitable for non-exerts, since programmers expect the data types and functions they define to have the properties they were intended to have. None of these projects makes the underling equality less complicated. No system will be able to verify every "obvious" equality for arbitrary user defined data types and functions statically, since program equivalence is famously undecidable.

Alternatively we could assume the equalities hold and discover a concrete inequality as a runtime error. There is some evidence that specific examples like this can help clarify the error messages in OCaml[24] and there has been an effort to make refinement type error messages more concrete and other systems like Liquid Haskell[13]. This leads to a different workflow then traditional type systems, instead of type checking first and only then executing the program, execution and type checking can both inform the programmer.

Several steps have been taken to make this new workflow possible. Section 2 will describe a conventional dependently typed base language with some programmatic features. Section 3 describes a dynamic extension to that language that supports dependent equality and the conditions that would make an implementation reasonable. Section 4 covers some of the other features a language in this style could support.

2 A Dependently Typed Base Language

The dynamic dependent equality of the next section is hard to study without a more conventional dependently typed language to serve as a reference. This base language contains the key features that should be supported: user defined dependent functions and dependent datatypes. The base language uses a "full-spectrum" style. The dynamic language from the next section can be elaborated from and compared to this concrete implementation.

The language is pure in the sense of Haskell, supporting only non-termination and unchecked errors as effects. Combining other effects with full-spectrum dependent types is substantially more difficult because effectful equality is hard to characterize for individual effects and especially hard for effects in combination. Several attempts have been made to combine dependent types with more effects, [23] [2, 1][23] but there is still a lot of work to be done. Effects, though undoubtedly useful, are not considered in the base language.

Since this work emphasizes programming over theorem proving, the language contains these logically dubious features:

- Unrestricted recursion (no required termination checking)

- Unrestricted user defined dependent data types (no requirement of strict positivity)
- Type-in-Type (no predictive hierarchy of universes)

Any one of these features can result in logical unsoundness[1], but they seem helpful for mainstream functional programming. In spite of logical unsoundness, the resulting language still has type soundness[2]. This seems suitable for a programming language since logically sound proofs can still be encoded and logical unsoundness can be discovered through traditional testing, or warned about in a non-blocking way. Importantly, no desirable computation is prevented.

Logical soundness seems not to matter in programming practice. For instance, in ML the type $\texttt{f} : \texttt{Int} - > \texttt{Int}$ does not imply the termination of `f 2`. While unproductive non-termination is always a bug, it seems an easy bug to detect and fix when it occurs. In mainstream languages, types help to communicate the intent of termination, even though termination is not guaranteed by the type system.

The base language still supports a partial correctness property for first order data types when run with Call-by-Value. For instance:

$$\vdash M : \sum x : \mathbb{N}. \texttt{IsEven}\, x$$

$\texttt{fst}\, M$ may not terminate, but if it does, $\texttt{fst}\, M$ will be an even $\mathbb{N}$. However, this property does not extend to functions

$$\vdash M : \sum x : \mathbb{N}. (y : \mathbb{N}) \to x \leq y$$

it is possible that $\texttt{fst}\, M \equiv 7$ if

$$M \equiv \langle 7, \lambda y. \texttt{loopForever} \rangle$$

[1]Every type is inhabited by an infinite loop.

[2]No term with a reduct that applies an argument to a non-function in the empty context will type.

2.1 Prior work

While many of these features have been explored in theory and implemented in practice, I am unaware of any development with exactly this formulation.

Unsound logical systems that are acceptable programming languages go back to at least Church's lambda calculus which was originally intended to be a logical foundation for mathematics. In the 1970s, Martin Lof proposed a system with Type-in-Type that was shown logically unsound by Girard (as described in the introduction in [19]). In the 1980s, Cardelli explored the domain semantics of a system with general recursive dependent functions and Type-in-Type[5].

The first proof of type soundness for a language with general recursive dependent functions, Type-in-Type, and dependent data that I am aware of came form the Trellys Project [27]. At the time their language had several additional features, not included in the base language. Additionally the base language uses a simpler notion of equality and dependent data resulting in an arguably simpler proof of type soundness. Later work in the Trellys Project[7, 6] used modalities to separate the terminating and non terminating fragments of the language, though the annotation burden seems too high in practice. In general, the base language has been deeply informed by the Trellys Project[15][27][7, 6] [28] [26] and the Zombie Language[3] it produced.

Several implementations support this combination of features without proofs of type soundness. Cayenne [4] is a Haskell like language that combined dependent types with and data and non-termination. Agda supports general recursion and Type-in-Type with compiler flags, and can simulate some non-positive data types using coinduction. Idris supports similar "unsafe" features.

[14] introduces a similar "partial correctness" criterion for dependent languages with non-termination.

3 A Language with Dynamic Type Equality

A key issue with dependent type theories is the characterization of definitional equality. Since computation can appear at the type level, and types must be checked for equality, dependent type theories must define what computational equalities they intend to equate for type checking. For instance, the base language follows the common choice of $\alpha\beta$ equivalence of terms. However, this causes many programs to not type-check:

[3]https://github.com/sweirich/trellys

$\lambda x.\texttt{head}\, x\ (\texttt{rep}\ (x+1))\ :\ \mathbb{N} \to \mathbb{B}$ Since $1+x$ does not have the same normalized syntax as $x+1$.

This is a widely recognized issue with dependent type theories. However, most attempts to improve the equality relation intend to preserve decidable type checking and/or logical soundness, so equality will never be complete[4]. Since dependently typed languages with the practical features outlined in base language are already incompatible with logical soundness and decidable type checking, these concerns no longer apply.

The base language can be extended to a cast language that supports the expectation of the original typing. Many programs that do not type in the base language can be elaborated into the cast language. The cast language has a weaker notion of type soundness such that

1. $\vdash_c e' : M'$ then
   (a) $e' \downarrow v'$ and $\vdash_c v' : M'$
   (b) or $e' \uparrow$
   (c) or $e' \downarrow blame$

Type soundness holds, or av inequality can be witnessed at a specific source location. In the example above $\lambda x.\texttt{head}\, x\ (\texttt{rep}\ (x+1))\ :\ \mathbb{N} \to \mathbb{B}$ will not emit any type checking errors or runtime errors (though a static warning may be given).

If the example is changed to

$$\lambda x.\texttt{head}\, x\ (\texttt{rep}\, x)\ :\ \mathbb{N} \to \mathbb{B}$$

and the function is called at runtime, the blame tracking system will blame the exact static location that uses unequal types with a direct proof of inequality, allowing an error like "failed at application $\left(\texttt{head}\, x : \texttt{Vec}\, \underline{(1+x)}\, \mathbb{B} \to ...\right) (rep\, x : \texttt{Vec}\, \underline{x}\, \mathbb{B})$ since when $x = 3$, $1 + x = 4 \neq 3 = x$", regardless of when the function was called in the program and where the discrepancy was discovered. This improves on the naive solution of completely ignoring type annotations for execution, since without type soundness, the returned error may appear unrelated to the problematic type assumption ("3 applied to 7 but 7 is not a function").

[4] I am also unaware of any suitable notion of complete extensional equality for dependent type theory though it is considered briefly in [26] .

Just as standard type theories allow many possible characterizations of equality that support logical soundness, there are many choices of runtime checking that support this weaker notion of type soundness. The minimal choice is likely too permissive in practice: in the example above, it would only flag an error when the function is applied to 0. Alternatively, I conjecture that runtime-checking that matches the partial correctness criteria above would be reasonably intuitive. Extending checks into non-dependent function types also seems reasonable, and would allow simple types to be completely checked.

Taking inspiration from the "gradual guarantee" of gradual typing, there are several basic properties in addition to weakened type soundness that this cast language hopes to fulfill:

1. $\vdash e : M$, $elab\,(M, *) = M'$, and $elab\,(e, M') = e'$ then $\vdash_c e' : M'$.
2. $\vdash_c e' : M'$ and $e' \downarrow blame$ then there is no $\vdash e : M$ such that $elab\,(M, *) = M'$, $elab\,(e, M') = e'$
3. $\vdash_c e' : *$ and $elab\,(e, *) = e'$ then
   (a) if $e' \downarrow *$ then $e \downarrow *$
   (b) if $e' \downarrow (x : M') \to N'$ then $e \downarrow (x : M) \to N$
   (c) if $e' \downarrow TCon\,\overline{M'}$ then $e \downarrow TCon\,\overline{M}$

The first condition states that every typed term in the base language can be embedded in the cast language. The second condition shows that errors are not spurious. The third condition shows that except for error, observations are consistent with the base language (with large eliminations, term constructor observations are also consistent).

3.1 Prior work

It is unsurprising that dynamic equality shares many of the same concerns as the large amount of work for contracts, hybrid types, gradual types, and blame. In fact, this work could be seen as gradualizing the Reflection Rule in Extensional Type Theory.

Blame has been strongly advocated for in [30, 29]. Blame tracking can help establish the reasonableness of monitoring systems by linking a dynamic failure directly to the broken static invariant. Blame is also a key ingredient of clear runtime error messages. However, as many authors have noticed, proving blame correctness is tedious and error prone, it is often only conjectured.

The basic correctness conditions are inspired by the Gradual Guarantee [25]. The implementation also takes inspiration from "Abstracting gradual typing"[12], where static evidence annotations become runtime checks. Unlike some impressive attempts to gradualize the polymorphic lambda calculus [3], dynamic equality does not attempt to preserve any parametric properties of the base language. It is unclear if such a restriction to parametric properties would be desirable for a dependently typed language.

A direct attempt has been made to gradualize a full spectrum dependently typed language to an untyped lambda calculus using the AGT philosophy in [11]. However that system retains the intentional style of equality and user defined data types are not supported. The paper is largely concerned with establishing decidable type checking via an approximate term normalization.

A refinement type system with higher order features is gradualized in [32] though it does not appear powerful enough to be characterized as a full-spectrum dependent type theory. [32] builds on earlier refinement type system work, which described itself as "dynamic". A notable example is [22] which describes a refinement system that limits predicates to base types.

4 Further Features

Dynamic equality appears to be a prerequisite for other useful features.

4.1 Prototyping proofs and programs

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 "correct" are not constructable 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 it will verify that the `sort` function correctly handles that input, gives an error, or non-terminate.

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

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

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 term search can be surprisingly subtle. For instance,

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

depends on the definitional properties of functions. To avoid this subtly I plan to only support term search over first order data.

A primitive proof search was implemented in an earlier prototype, better search methods could be incorporated in future work.

4.1.1 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 [21].

Twelf made use of runtime proof search but the underling theory cannot be considered full-spectrum.

4.2 Testing dependent programs

Both dynamic equalities and dynamic proof search vastly weaken the guarantees of normal dependent type systems. Programmers still would like evidence of correctness, even while they intend to provide full proofs in the future. However, there are few options available in full-spectrum dependently typed languages aside from costly and sometimes unconstructable proofs.

The mainstream software industry has a similar need for evidence of correctness, and has standardized on testing done in a separate execution phase. Given the rich specifications that dependent types provide it is possible to improve on the hand crafted tests used by most of the industry. Instead we can use a type directed symbolic execution, to run questionable equalities over concrete values and precompute the searches of proof terms. Precomputed proof terms can be cached, so that exploration is not too inefficient in the common case of repeating tests at regular intervals of code that is mostly the same. Precomputed terms can be made available at runtime, excusing the inefficient search procedure.

Interestingly dynamic equality is necessary for testing like this, since otherwise, definitional properties of functions would need to be accounted for. Previous attempts to test dependent specifications were overwhelmed by definitional considerations. Using dynamic equality it is possible only consider the extensional behavior of functions.

Finally, future work can add more advanced methods of testing and proof generation. This architecture should make it easier to change exploration and search procedures without changing the underlining definitional behavior.

4.2.1 Prior work

4.2.2 Symbolic Execution

Most research for Symbolic Execution targets popular imperative languages (like C) and uses SMT solvers to efficiently explore conditional branches that depend on base types. Most work does not support higher order functions or makes simplifying assumptions about the type system. There are however some relevant papers:

- [13] presents a symbolic execution engine supporting Haskell's lazy execution and type system. Higher order functions are not handled
- The draft work[31], handles higher order functions as inputs and provides a proof of completeness
- Symbolic execution for higher order functions for a limited untyped variant of PCF is described in [20]

4.2.3 Testing dependent types

There has been a long recognized need for testing in addition to proving in dependent type systems

- In [10] a QuickCheck style framework was added to an earlier version of Agda
- QuickChick[5] [9][18, 17, 16] is a research project to add testing to Coq. 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.

5 Status and Plan

5.1 Status

I am rewriting a prototype for dynamic equality to support for all the features of the base language. The fragment for dependent function types and type universes satisfies all the conditions above. I believe that all conditions can be made to hold with data types as well.

The proof search prototype written for the extended abstract presented last summer, will need to be corrected to account for dynamic equality.

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

5.2 Plan

- March: re-implement data types in prototype
- April: extend proofs to data types
- May: Submit to TyDe 2021
- June: Rewrite the proof search and testing prototype
- July: Draft thesis
- August: Defend

References

[1] Danel Ahman. Fibred computational effects. *arXiv preprint arXiv:1710.02594*, 2017.

[2] Danel Ahman. Handling fibred algebraic effects. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, 2017.

[3] Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: Parametricity, with and without types. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017.

[4] Lennart Augustsson. Cayenne a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 239–250, New York, NY, USA, 1998. Association for Computing Machinery.

[5] Luca Cardelli. *A Polymorphic [lambda]-calculus with Type: Type*. Technical Report, DEC SRC, 130 Lytton Avenue, Palo Alto, CA 94301. May. SRC Research Report, 1986.

[6] Chris Casinghino. Combining proofs and programs. 2014.

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

[8] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. The taming of the rew: A type theory with computational assumptions. *Proceedings of the ACM on Programming Languages*, 2021.

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

[10] 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.

[11] Joseph Eremondi, Éric Tanter, and Ronald Garcia. Approximate normalization for gradual dependent types. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[12] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 429–442, New York, NY, USA, 2016. Association for Computing Machinery.

[13] 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.

[14] Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg, and Stephanie Weirich. Dependent types and program equivalence. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 275–286, 2010.

[15] Garrin Kimmell, Aaron Stump, Harley D Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins, and Ki Yung Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. In *Proceedings of the sixth workshop on Programming languages meets program verification*, pages 15–26, 2012.

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

[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.

[18] 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.

[19] Per Martin-Löf. An intuitionistic theory of types. In Giovanni Sambin and Jan M. Smith, editors, *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 127–172. Oxford University Press, 1998.

[20] 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.

[21] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Citeseer, 2007.

[22] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics*, pages 437–450, Boston, MA, 2004. Springer US.

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

[24] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 228–242, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined Criteria for Gradual Typing. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 274–293, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

[27] Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. Irrelevance, heterogeneous equality, and call-by-value dependent type systems. *Mathematically Structured Functional Programming*, 76:112–162, 2012.

[28] Vilhelm Sjöberg and Stephanie Weirich. Programming up to congruence. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 369–382, 2015.

[29] Philip Wadler. A Complement to Blame. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 309–320, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[30] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 1–16, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[31] Shu-Hung You, Robert Bruce Findler, and Christos Dimoulas. Dynamic symbolic execution of higher-order functions, 2020.

[32] Jakub Zalewski, James McKinna, J. Garrett Morris, and Philip Wadler. Î»db: Blame tracking at higher fidelity. 2020. First ACM SIGPLAN Workshop on Gradual Typing 2020, WGT 2020 ; Conference date: 19-01-2020 Through 25-01-2020.