Dissertation Prospectus

A Full-Spectrum Dependently Typed Language for Programming with Dynamic Equality, Testing, and Runtime Proof Search

Mark Lemay

Department of Computer Science, Boston University

June 1, 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 syntax and methodology 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 type systems are used 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 can be difficult in these systems.

My solution to these problems is a dependent type system that is as permissive as possible and produces clear errors at runtime. The first ingredient is a dependently typed "surface" language with programmatic features that supports type sound, but logically unsound execution. This "surface" language can then be generalized into a dynamic "cast" language provisionally accepting many more equalities. Programmers trade poor static errors for precise counterexamples that are made available at runtime. Additionally, automated testing can provide feedback about the correctness of typing wether the programmer intends to create a formal proof, or merely assert a property holds. Finally, a runtime proof search feature will allow programmers to take advantage of this machinery for rapid prototyping.

Currently, the surface language is implemented and the cast language is being cleaned up. Several properties have been proven for these languages. Earlier prototypes of for automated testing and runtime proof search will need to be rewritten to accommodate changes changes to the cast 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 implement a provably correct sorting function

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

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 formalizations 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 use effects, it can be seen in some of the most popular dependently typed programming languages such as Agda and Idris.

Despite the potential, programmers often find these systems difficult to use. Consider the confusing error messages these languages are known to produce. For instance, in Agda this reasonable looking program

$$\begin{aligned}
&\texttt{Vec} : * \to \mathbb{N} \to * \\
&\texttt{rep} : (x : \mathbb{N}) \to \texttt{Vec}\,\mathbb{B}\,x \\
&\texttt{head} : (x : \mathbb{N}) \to \texttt{Vec}\,\mathbb{B}\ (1 + x) \to \mathbb{B} \\
&\not\vdash \lambda x.\texttt{head}\,x\ (\texttt{rep}\ (x + 1)) : \mathbb{N} \to \mathbb{B}
\end{aligned}$$

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, without telling why that property might not hold. If addition had a bug such that $x + 1 \neq 1 + x$, 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 been the goal for many research projects (to name a few [8, 30]However, it is unlikely those impressive efforts are suitable for non-exerts. Programmers unfamiliar with dependent type theory will 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 witness of inequality as a

runtime error. There is some evidence that specific examples like this can help clarify the error messages in OCaml[26] and there has been an effort to make refinement type error messages more concrete and other systems like Liquid Haskell[14]. 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 a reality. Section 2 will describe a conventional dependently typed "surface" language with some programmatic features. Section 3 describes an extension to surface language that supports dynamic equality via casts. Section 4 covers some of the other features a language in this style will support.

2 A Dependently Typed Surface Language

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

The surface 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, [25] [2, 1][25] but there is still a lot of work to be done. General effects, though undoubtedly useful, will not be considered in this proposal.

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

- Unrestricted recursion (no required termination checking)
- Type-in-Type (no hierarchy of universes)
- Unrestricted user defined dependent data types (no requirement of strict positivity)

Any one of these features can result in logical unsoundness[1], but they seem useful for mainstream functional programming. Additionally non-termination causes type-checking to be undecidable, though this has not caused any issues in practice, and even decidable type-checking in dependent type theory is computationally intractable. In spite of logical unsoundness, the resulting language still has type soundness[2].

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. Therefore, logical unsoundness seems suitable for a dependently typed programming language since 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 in order to preserve logical soundness.

This programming language uses bidirectional type checking to minimize the number of annotations required. Bidirectional type-checking is a popular technique for implementing dependently typed programming languages because it not as complicated as more sophisticated unification strategies, while still minimizing the annotations needed. This style of type checking usually only needs top level functions to be annotated[3]. Bidirectional type-checking splits the typing judgment into 2 separate judgments: the "infer" judgment if the type can be inferred from a term, and a "check" judgment for when term will be checked against a type. Inferences can be turned into checked judgments with an explicit equality

[1] Every type is inhabited by an infinite loop.
[2] No term typed in an empty context will "get stuck".
[3] Even in Haskell, with full Hindley-Milner type inference, this style of annotations is encouraged.

check.

The surface 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

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

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 [21]). In the 1980s, Cardelli explored the domain semantics of a system with general recursive dependent functions and Type-in-Type[5].

The first direct 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 [29]. At the time their language had several additional features not included in my surface language. Additionally my surface 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 high in retrospect. In general, the base language has been deeply informed by the Trellys Project[16][29][7, 6] [30] [28] and the Zombie Language[4] it produced.

Several implementations support this combination of features without proofs of type soundness. Coquand presented an early bidirectional algorithm to type-check a similar language [9]. Cayenne [4] is a Haskell like language that combined dependent types with type-in-type, 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.

[15] introduces a similar "partial correctness" criterion for dependent languages with non-termination run with Call-by-Value.

3 A Language with Dynamic Type Equality

A fundamental 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 computations they intend to equate for type checking. For instance, the surface language follows the common choice of $\alpha\beta$ equivalence of terms. However, this causes many acceptable programs to not type-check. For instance $\lambda x.\texttt{head}\, x\ (\texttt{rep}\ (x+1)) : \mathbb{N} \to \mathbb{B}$ will not type-check 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

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

equality relation intend to preserve decidable type checking and/or logical soundness, so equality will never be complete[5]. Since dependently typed languages with the practical features outlined in surface language are already incompatible with logical soundness and decidable type checking, these concerns no longer apply.

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

1. $\vdash a : A$ then
   (a) a is a value
   (b) or $a \rightsquigarrow a'$ where $\vdash a' : A$
   (c) or $a$can blame a source location with a specific witness of inequality

In short, type soundness holds, or a good error message can be given. In the example above $\lambda x.\texttt{head}\, x\ (\texttt{rep}\ (x+1)) : \mathbb{N} \to \mathbb{B}$ will not cause 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 source 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.

[5] I am unaware of any suitable notion of complete extensional equality for dependent type theory though it is considered briefly in [28] .

Cast soundness improves on the naive solution of 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").

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.

(Possibly untyped) terms in the surface language need to be converted into well-cast terms of the cast language. To do this I have defined an elaboration procedure, that maintains the information required to preserve cast checking. The elaboration procedure is inspired by bidirectional typing to

There are several basic properties that should hold when terms of the surface language elaborate into terms of the cast language.

1. The elaboration procedure only produces well cast terms
2. Every term typed in the surface language elaborates
3. Errors are not spurious, errors never point to anything that type-checked in the surface language
4. Whenever an elaborated cast term evaluates, the corresponding surface term evaluates consistently

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 [32, 31]. Blame tracking can help establish the reasonableness of monitoring systems by linking a dynamic failure directly to the broken invariant in source. 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 [27]. The implementation also takes inspiration from "Abstracting gradual typing"[13], 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 [12]. 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 [34] though it does not appear powerful enough to be characterized as a full-spectrum dependent type theory. [34] builds on earlier refinement type system work, which described itself as "dynamic". A notable example of dynamic checks in a refinement type system, is [24] 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 the function will verify that the `sort` correctly handled that input, gives an error, or non-terminate.

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

$$
\begin{aligned}
&data\,\texttt{Mult} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to * \,where \\
&\quad \texttt{base} : (x : \mathbb{N}) \to \texttt{Mult0}\,x\,0 \\
&\quad \texttt{suc} : (x\,y\,z : \mathbb{N}) \to \texttt{Mult}\,x\,y\,z \to \texttt{Mult}\ (1 + x)\ y\,(y + z)
\end{aligned}
$$

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.

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 [23].

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 test 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 to be managable, since otherwise, definitional properties of functions would need to be accounted for. My previous attempts to test dependent specifications were overwhelmed by definitional considerations. Using dynamic equality it is possible to 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:

- [14] presents a symbolic execution engine supporting Haskell's lazy execution and type system.

Higher order functions are not handled

- The draft work[33], handles higher order functions as inputs and provides a proof of completeness
- Symbolic execution for higher order functions for an untyped variant of PCF is described in [22]

4.2.3 Testing dependent types

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

- In [11] a QuickCheck style framework was added to an earlier version of Agda
- QuickChick[6] [10][19, 18, 17] 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

The surface language has been fully implemented. I have proven type soundness for the full surface language (follows a similar structure to [29]), Qiancheng Fu has mechanized my proof of type soundness for the function fragment in Coq. Details are in a paper currently under submission at TyDe workshop.

A prototype implementation for the cast language and elaboration procedure is almost completed. All properties mentioned in section 2 have been proven for the functional fragment. Qiancheng Fu has mechanized my proof of cast soundness in Coq. We conjecture that they can be extended to the full language.

A primitive proof search and a testing framework were implemented in an earlier prototype, but will need to be reimplemented to account for dynamic equality. Better search methods are planned.

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

5.2 Plan

- June
  - refine implementation
  - write chapters: on the surface language, and on the cast language
- July
  - finalize the implementation of symbolic execution
  - rework symbolic execution[20] to use dynamic equality
  - write chapter: testing and symbolic execution
- August
  - finalize the implementation of runtime proof search
  - rework symbolic execution[20] to use dynamic equality
  - write chapter: runtime proof-search
- September
  - Write a medium sized example in the language
  - Write the chapter walking thought that example
- October
  - Finalize: Introduction, Future work, Conclusion
- November: 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] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1):167–177, 1996.

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] Mark Lemay, Cheng Zhang, and William Blair. Extended abstract: Developing a dependently typed language with runtime proof search. *Workshop on Type-Driven Development*, 2020.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

6 Appendix

6.1 Surface Language (the function fragment)

6.1.1 Pre-syntax

source labels,			
ℓ			
variable contexts,			
Γ	$::=$	$\Diamond \mid \Gamma, x : M$	
expressions,			
m, n, h, M, N, H	$::=$	x	variable
	$\mid$	$m ::_\ell M$	annotation
	$\mid$	$\star$	type universe
	$\mid$	$(x : M_\ell) \to N_{\ell'}$	function type
	$\mid$	$\mathsf{fun}\, f\, x \Rightarrow m$	function
	$\mid$	$m_\ell\, n$	application
values,			
v	$::=$	$x \mid \star$	
	$\mid$	$(x : M_\ell) \to N_{\ell'}$	
	$\mid$	$\mathsf{fun}\, f\, x \Rightarrow m$	

6.1.2 Surface Language Judgments

$\Gamma \vdash m \overrightarrow{:} M$	The type M can be inferred form m in Γ
$\Gamma \vdash m \overleftarrow{:} M$	The m is checked against the type M in Γ
$x : M \in \Gamma$	x is in $\Gamma$with type M
$\Gamma \vdash$	Context Γ is well formed in the bidirectional system
$\Gamma \vdash m \ : \ M$	m has type M in Γ via type assignment
$\Gamma \vdash m \equiv m' \ : \ M$	m and m' share a reduct (ignoring location information)
$\Gamma \vdash m \Rrightarrow m' \ : \ M$	m goes to m' in 1 parallel step
$\Gamma \vdash m \Rrightarrow^* m' \ : \ M$	m goes to m' in 0 or more parallel steps
$m \rightsquigarrow m'$	m goes to m' in 1 small step
$m \rightsquigarrow^* m'$	m goes to m' in 0 or more small steps
$m \, Stuck$	m is not a value and does not small step

Only select definitions are given here, full definitions are available in a tech report. More polished variants are available in our TyDe submission, and Coq development.

6.1.3 Bidirectional Type System

$$\frac{x : M \in \Gamma}{\Gamma \vdash x \overrightarrow{:} M} var - ty$$

$$\frac{\Gamma \vdash m \overleftarrow{:} M \quad \Gamma \vdash M \overleftarrow{:} \star}{\Gamma \vdash m ::_\ell M \overrightarrow{:} M} :: -ty$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \star \overrightarrow{:} \star} \star - ty$$

$$\frac{\Gamma \vdash M \overleftarrow{:} \star \quad \Gamma, x : M \vdash N \overleftarrow{:} \star}{\Gamma \vdash (x : M) \to N \overrightarrow{:} \star} \Pi - ty$$

$$\frac{\Gamma \vdash m \overrightarrow{:} (x : N) \to M \quad \Gamma \vdash n \overleftarrow{:} N}{\Gamma \vdash m \, n \overrightarrow{:} M \, [x \coloneqq n]} \Pi - app - ty$$

$$\frac{\Gamma \vdash m \overleftarrow{:} M \quad \Gamma \vdash M \equiv M' : \star}{\Gamma \vdash m \overrightarrow{:} M'} conv$$

$$\frac{\Gamma, f : (x : N) \to M, x : N \vdash m \overleftarrow{:} M}{\Gamma \vdash \mathsf{fun} \, f \, x \Rightarrow m \overleftarrow{:} (x : N) \to M} \Pi - \mathsf{fun} - ty$$

6.1.4 Properties

Type Soundness, $\vdash m \overleftarrow{:} M$ and for all m', $m \rightsquigarrow^* m'$, then m' is not $Stuck$.

6.2 Cast Language (the function fragment)

6.2.1 Pre-syntax

variable contexts,			
H	$::=$	$\Diamond \mid H, x : A$	
expressions,			
a, b, c, A, B, C	$::=$	x	
	$\mid$	$a ::_{A,\ell o} B$	cast
	$\mid$	$\star$	
	$\mid$	$(x : A) \to B$	
	$\mid$	$\mathsf{fun}\, f\, x \Rightarrow b \mid b\, a$	
observations,			
o	$::=$	$. \mid o.arg$	function type-argument
	$\mid$	$o.bod[a]$	function type-body

6.2.2 Judgments

$H \vdash a : A$	a has type A in H
$x : A \in H$	x is in H with type A
$H \vdash$	Context H is well formed
$H \vdash a \equiv a' : A$	a and a' share a reduct and have type A in H
$a \Rrightarrow a'$	a goes to a' in 1 parallel step
$a \Rrightarrow^* a'$	a goes to a' in 0 or more parallel steps
$a \rightsquigarrow a'$	a goes to a' in 1 small step
$a \rightsquigarrow^* a'$	a goes to a' in 0 or more small steps
$a\, Val$	a is a value
$a\, Stuck$	a is not a value and does not small step
$Blame\, \ell\, o\, a$	a blames the location ℓ with observation o

6.2.3 Parallel reductions

$$\frac{b \Rrightarrow b' \quad a \Rrightarrow a'}{(\mathsf{fun}\, f\, x \Rightarrow b)\, a \Rrightarrow b'\, [f \coloneqq \mathsf{fun}\, f\, x \Rightarrow b', x \coloneqq a']}$$

$$\frac{b \Rrightarrow b' \quad a \Rrightarrow a' \quad o \Rrightarrow o' \quad A_1 \Rrightarrow A_1' \quad B_1 \Rrightarrow B_1' \quad A_2 \Rrightarrow A_2' \quad B_2 \Rrightarrow B_2'}{\big(b ::_{(x:A_1)\to B_1,\ell o} (x : A_2) \to B_2\big)\, a \;\Rrightarrow\; \big(b'\, a' ::_{A_2',\ell o'.arg} A_1'\big) ::_{B_1\left[x:=a'::_{A_2',\ell o'.arg} A_1'\right],\ell o'.bod[a']} B_2'\, [x \coloneqq a']}$$

$$\frac{a \Rrightarrow a'}{a ::_{\star,\ell o} \star \Rrightarrow a'}$$

$$\frac{}{x \Rrightarrow x}$$

$$\frac{a \Rrightarrow a' \quad A \Rrightarrow A' \quad B \Rrightarrow B' \quad o \Rrightarrow o'}{a ::_{A,\ell,o} B \Rrightarrow a' ::_{A',\ell,o'} B'}$$

$$\frac{}{\star \Rrightarrow \star}$$

$$\frac{A \Rrightarrow A' \quad B \Rrightarrow B'}{(x : A) \to B \Rrightarrow (x : A') \to B'}$$

$$\frac{b \Rrightarrow b'}{\mathsf{fun}\, f\, x \Rightarrow b \;\Rrightarrow\; \mathsf{fun}\, f\, x \Rightarrow b'}$$

$$\frac{b \Rrightarrow b' \quad a \Rrightarrow a'}{b\, a \Rrightarrow b'\, a'}$$

$$\frac{o \Rrightarrow o'}{o \Rrightarrow o'.arg}$$

$$\frac{o \Rrightarrow o' \quad a \Rrightarrow a'}{o.bod\,[a] \Rrightarrow o'.bod\,[a']}$$

$$\frac{}{. \Rrightarrow .}$$

6.2.4 Definitional Equality

$$\frac{B \Rrightarrow B'' \quad B' \Rrightarrow B''}{B \equiv B'}$$

from the confluence and reflexivity of $\Rrightarrow$, $\equiv$ is an equivalence (reflexive, commutative and transitive). Type constructors are definitionally unique, $*\not\equiv\Pi$

6.2.5 Typing rules

$$\frac{x : A \in H}{H \vdash x \;:\; A} var - ty$$

$$\frac{H \vdash a : A \quad H \vdash A : \star \quad H \vdash B : \star}{H \vdash a ::_{A,\ell,o} B \;:\; B} :: -ty$$

$$\frac{H \vdash}{H \vdash \star : \star} \star - ty$$

$$\frac{H \vdash A : \star \quad H, x : A \vdash B : \star}{H \vdash (x : A) \to B \ : \ \star} \Pi - ty$$

$$\frac{H, f : (x : A) \to B, x : A \vdash b : B}{H \vdash \mathsf{fun}\, f\, x \Rightarrow b \ : \ (x : A) \to B} \Pi - \mathsf{fun} - ty$$

$$\frac{H \vdash b : (x : A) \to B \quad H \vdash a : A}{H \vdash b\, a \ : \ B\,[x \coloneqq a]} \Pi - app - ty$$

$$\frac{H \vdash a : A \quad H \vdash A \equiv A' : \star}{H \vdash a : A'} conv$$

6.2.6 Values

$$\frac{}{\star Val} \star - Val$$

$$\frac{}{(x : A) \to B\, Val} \Pi - Val$$

$$\frac{}{\mathsf{fun}\, f\, x \Rightarrow b\, Val} \Pi - \mathsf{fun} - Val$$

$$\frac{\begin{array}{c} a\, Val \quad A\, Val \quad B\, Val \\ a \not\equiv \star \\ a \not\equiv (x : C) \to C' \end{array}}{a ::_{A,\ell o} B\, Val} :: - Val$$

6.2.7 Small Step Semantics

$$\frac{a\, Val}{(\mathsf{fun}\, f\, x \Rightarrow b)\, a \rightsquigarrow b\,[f \coloneqq \mathsf{fun}\, f\, x \Rightarrow b, x \coloneqq a]}$$

$$\frac{b\, Val \quad a\, Val}{\begin{array}{l} \big(b ::_{(x:A_1)\to B_1,\ell o} (x : A_2) \to B_2\big)\, a \rightsquigarrow \\ (b\, a ::_{A_2,\ell o.arg} A_1) ::_{B_1[x:=a::_{A_2,\ell o.arg} A_1],\ell o.bod[a]} B_2\,[x \coloneqq a] \end{array}}$$

$$\frac{a\, Val}{a ::_{\star,\ell o} \star \rightsquigarrow a}$$

$$\frac{a \rightsquigarrow a'}{a ::_{A,\ell o} B \rightsquigarrow a' ::_{A,\ell o} B}$$

$$\frac{a\,Val \quad A \rightsquigarrow A'}{a ::_{A,\ell o} B \rightsquigarrow a ::_{A',\ell o} B}$$

$$\frac{a\,Val \quad A\,Val \quad B \rightsquigarrow B'}{a ::_{A,\ell o} B \rightsquigarrow a ::_{A,\ell o} B'}$$

$$\frac{b \rightsquigarrow b'}{b\,a \rightsquigarrow b'\,a}$$

$$\frac{b\,Val \quad a \rightsquigarrow a'}{b\,a \rightsquigarrow b\,a'}$$

as usual $\frac{a \rightsquigarrow a'}{a \Rrightarrow a'}$ is derivable. However it is not the case that $\frac{a \Rrightarrow * a'}{a \rightsquigarrow * a'}$ since the operational semantics does not evaluate under binders or in observations.

6.2.8 Blame

$$\frac{}{Blame\ \ell\ o\ a ::_{(x:A)\to B,\ell o} \star}$$

$$\frac{}{Blame\ \ell\ o\ a ::_{\star,\ell o} (x : A) \to B}$$

structural rules

$$\frac{Blame\ \ell\ o\ a}{Blame\ \ell\ o\ a ::_{A,\ell' o'} B}$$

$$\frac{Blame\ \ell\ o\ A}{Blame\ \ell\ o\ a ::_{A,\ell' o'} B}$$

$$\frac{Blame\ \ell\ o\ B}{Blame\ \ell\ o\ a ::_{A,\ell' o'} B}$$

$$\frac{Blame\ \ell\ o\ b}{Blame\ \ell\ o\ (b\,a)}$$

$$\frac{Blame\ \ell\ o\ a}{Blame\ \ell\ o\ (b\,a)}$$

6.2.9 Properties

Cast Soundness, $\vdash a : A$ and for all a', $a \rightsquigarrow^* a'$, if $a'\ Stuck$ then there exists ℓ, o such that $Blame\ \ell\ o\ a'$

6.3 Elaboration (the function fragment)

6.3.1 Judgments

$H \vdash Elab\, m\, a \overrightarrow{:} A$	m elaborates into a with inferred type A in H
$H \vdash Elab\, n\, a \overleftarrow{:_{\ell o}} A$	m elaborates into a with type A because of ℓo in H
$Elab\, \Gamma\ H$	the context Γ elaborates into $\Gamma\ H$

6.3.2 Elaboration

$$\frac{x : A \in H}{H \vdash Elab\, x\, x \overrightarrow{:} A}$$

$$\frac{}{H \vdash Elab\ \star\ \star \overrightarrow{:} \star}$$

$$\frac{H \vdash Elab\ M\ A \overleftarrow{:_{\ell.}} \star \quad H, x : A \vdash Elab\ N\ B \overleftarrow{:_{\ell'.}} \star}{H \vdash Elab\ (x : M_\ell) \to N_{\ell'}\ \ (x : A) \to B \overrightarrow{:} \star}$$

$$\frac{H \vdash Elab\, m\, b \overrightarrow{:} C \quad C \equiv (x : A) \to B \quad H \vdash Elab\, n\, a \overleftarrow{:_{\ell.arg}} A}{H \vdash Elab\, m_\ell\, n\, b\, a \overrightarrow{:} B\,[x := a]}$$

$$\frac{H \vdash Elab\ M\ A \overleftarrow{:_{\ell.}} \star \quad H \vdash Elab\, m\, a \overleftarrow{:_{\ell.}} A}{H \vdash Elab\, m :_\ell M\ a \overrightarrow{:} A}$$

$$\frac{H \vdash Elab\, m\, a \overrightarrow{:} A}{H \vdash Elab\, m\, a :_{A,\ell o} A' \overleftarrow{:_{\ell o}} A'}$$

$$\frac{H, f : (x : A) \to B, x : A \vdash Elab\, m\, b \overleftarrow{:_{\ell o.bod[x]}} B}{H \vdash Elab\, \mathsf{fun}\, f\, x \Rightarrow m\ \mathsf{fun}\, f\, x \Rightarrow b \overleftarrow{:_{\ell o}} (x : A) \to B}$$

6.3.3 Erasure

This function that uniformly erases cast, location and type information from all variants of syntax

$$
\begin{array}{rcl}
|x| & = & x \\
|\star| & = & \star \\
|m ::_\ell M| & = & |m| \\
|(x : M_\ell) \to N_{\ell'}| & = & (x : |M|) \to |N| \\
|m_\ell\, n| & = & |m|\, |n| \\
|\mathsf{fun}\, f\, x \Rightarrow m| & = & \mathsf{fun}\, f\, x \Rightarrow |m| \\
|\Diamond| & = & \Diamond \\
|\Gamma, x : A| & = & |\Gamma|, x : |A| \\
|a ::_{A,M,\ell,o,N} B| & = & |a| \\
|(x : A) \to B| & = & (x : |A|) \to |B| \\
|\mathsf{fun}\, f\, x \Rightarrow b| & = & \mathsf{fun}\, f\, x \Rightarrow |b| \\
b\, a & = & |b|\, |a| \\
|H, x : M| & = & |H|, x : |M|
\end{array}
$$

6.3.4 Properties

Elaboration preserves erasure

- for any $Elab\, \Gamma\, H$, then $|H| = |\Gamma|$
- for any $H \vdash Elab\, a\, m \overrightarrow{:} A$, then $|m| = |a|$
- for any $H \vdash Elab\, a\, m \overleftarrow{:}_{\ell o} A$, then $|m| = |a|$

Elaboration is always well-cast

- for any $H \vdash Elab\, a\, m \overrightarrow{:} A$, then $H \vdash a : A$
- for any $H \vdash Elab\, a\, m \overleftarrow{:}_{\ell o} A$, then $H \vdash a : A$

Terms that are bidirectionally typed elaborate

- if $\Gamma \vdash m \overrightarrow{:} M$ then there exists a, A such that $\vdash Elab\, m\, a \overrightarrow{:} A$
- if $\Gamma \vdash m \overleftarrow{:} M$ and given ℓ then there exists a, A, o such that $H \vdash Elab\, a\, m \overleftarrow{:}_{\ell o} A$

Blame never points to something that checked in the bidirectional system

- if $\vdash m \overrightarrow{:} M$, and $\vdash Elab\, m\, a \overrightarrow{:} A$, then for no $a \rightsquigarrow^* a'$ will $Blame\, \ell\, o\, a'$

Evaluation of elaboration is consistent with unelaborated evaluation

- if $H \vdash Elab\, m\, a \overrightarrow{:} A$, and $a \rightsquigarrow^* \star$ then $m \rightsquigarrow^* \star$
- if $H \vdash Elab\, m\, a \overrightarrow{:} A$, and $a \rightsquigarrow^* (x : A) \to B$ then there exists N, M, ℓ, ℓ' such that $m \rightsquigarrow^* (x : N_\ell) \to M_{\ell'}$