Gradual Correctness: a Dynamically Bidirectional Full-Spectrum Dependent Type Theory (Extended Abstract)

Mark Lemay
Qiancheng Fu
Hongwei Xi
Computer Science
Boston University
USA
lemay@bu.edu

Abstract

Dependent type systems are powerful tools to eliminate bugs from programs. Unlike other systems of formal methods, dependent type systems can re-use the methodology and syntax that functional programmers are already familiar with for the construction of formal proofs. However, implementations of these languages still have substantial usability issues arising from the conservative equality commonly used in intensional type theories, which can manifest as confusing error messages. In this paper we show how to take a full-spectrum dependently typed language and optimistically delay some equality checking until runtime. The advantage of our method is clear runtime error messages supported by blame that pinpoints the exact source of failure.

Keywords: Programing Languages, Dependent Types

1 Introduction

Programming is an error-filled process. While different formal methods approaches can make some error rare or impossible, they burden programers with complex additional syntax and semantics that can make them hard to work with. Dependent type systems offer a simpler approach. In a dependent type system, proofs and invariants can borrow from the syntax and semantics already familiar to functional programmers.

This promise of dependent types in a practical programming language has inspired research projects for decades. Several approaches have now been explored. The **full-spectrum** approach is a popular and parsimonious approach that allow computation to behave the same at the term and type level [Augustsson 1998; Brady 2013; Norell 2007; Sjöberg et al. 2012]. While this approach offers tradeoffs, it seems to be the most predictable from the programmer's perspective.

Authors' address: Mark Lemay; Qiancheng Fu; Hongwei Xi, Computer Science, Boston University, USA, lemay@bu.edu.

2021. 2475-1421/2021/1-ART1 $15.00
https://doi.org/

For instance, dependent types can prevent an out-of-bounds error when indexing into a length indexed list. The following type checks in virtually all full-spectrum dependent type systems

$$\begin{aligned}
\mathtt{Bool} &: *,\\
\mathtt{Nat} &: *,\\
\mathtt{Vec} &: * \to \mathtt{Nat} \to *,\\
\mathtt{add} &: \mathtt{Nat} \to \mathtt{Nat} \to \mathtt{Nat},\\
\mathtt{rep} &: (A : *) \to A \to (x : \mathtt{Nat}) \to \mathtt{Vec}\, A\, x,\\
\mathtt{head} &: (A : *) \to (x : \mathtt{Nat}) \to \mathtt{Vec}\, A\ (\mathtt{add}\, 1\, x) \to A
\end{aligned}$$

$$\vdash \lambda x.\mathtt{head}\,\mathtt{Bool}\, x\ (\mathtt{rep}\,\mathtt{Bool}\,\mathtt{true}\ (\mathtt{add}\, 1\, x))\ : \mathtt{Nat} \to \mathtt{Bool}$$

We are sure `head` never inspects an empty list because the `rep` function will always return a list of length $1 + x$. In a more polished implementation many arguments would be implicit and the above could be written as $\lambda x.\mathtt{head}\ (\mathtt{rep}\,\mathtt{true}\ (1 + x))\ : \mathtt{Nat} \to \mathtt{Bool}$.

Unfortunately, dependent types have yet to see widespread industrial use. Programmers often find dependent type systems difficult to learn and use. One of the reasons for this difficulty is that conservative assumptions about equality create subtle issues for users, and lead to some of the confusing error messages these languages are known to produce [Eremondi et al. 2019a].

The following will not type check in any conventional system with user defined addition,

$$\nvdash \lambda x.\mathtt{head}\,\mathtt{Bool}\, x\ (\mathtt{rep}\,\mathtt{Bool}\,\mathtt{true}\ (\mathtt{add}\, x\, 1))\ : \mathtt{Nat} \to \mathtt{Bool}$$

While "obviously" $1 + x \;=\; x + 1$, in the majority of dependently typed programming languages, $\mathtt{add}\, 1\, x \equiv \mathtt{add}\, x\, 1$ is not a definitional equality. This means a term of type $\mathtt{Vec}\,(\mathtt{add}\, 1\, x)$ cannot be used where a term of type $\mathtt{Vec}\,(\mathtt{add}\, x\, 1)$ is expected. Usually when dependent type systems encounter situations like this, they will give a type error and prevent evaluation. If the programmer made a mistake in the definition of addition such that $\mathsf{add}\ 1\ x \neq \mathsf{add}\ x\ 1$, no hints are given to correct the mistake. This increase of friction and lack of communication are key reasons that dependent types systems are not more widely used.

Instead why not sidestep static equality? We could assume the equalities hold and discover a concrete witness of inequality as a runtime error. Assuming there was a mistake in addition, we could instead provide a runtime error that gives an exact counter example. For instance, if the add function incorrectly computes $\mathsf{add}\ 8\ 1 = 0$ the above function will "get stuck" on the input 8. If that application is encountered at runtime we can give the error $\mathsf{add}\ 1\ 8 = 9 \neq 0 = \mathsf{add}\ 8\ 1$, a message that would have been the perfect type error. There is some evidence that specific examples like this can help clarify the type error messages in OCaml [Seidel et al. 2016] and there has been an effort to make refinement type error messages more concrete in other systems like Liquid Haskell [Hallahan et al. 2019].

Runtime type checking leads to a different workflow than traditional type systems. Instead of type checking first and only then executing the program, execution and type checking can both inform the programmer. Users can still be warned about uncertain equalities, but the warning need not block the flow of programming. Since the user can gradually correct their program as errors surface, we call this workflow **gradual correctness**.

Additionally, our approach avoids fundamental issues of definitional equality. No system will be able to statically verify every "obvious" equality for arbitrary user defined data types and functions, since general program equivalence is famously undecidable. By weakening the assumption that all equalities be decided statically, we can experiment with other advanced features without arbitrarily committing to which equalities are acceptable. Finally, we expect this approach to equality is a prerequisite for other desirable features such as a foreign function interface, runtime proof search, and a lightweight ability to test dependent type specifications.

Though gradual correctness is an apparently simple idea, there are several subtle issues that must be dealt with. While it is easy to check ground natural numbers for equality, even simply typed functions have undecidable equality. This means that we cannot just check types for equality at applications of higher order functions. Dependent functions mean that equality checks may propagate into the type level. Simply removing all type annotations will mean there is not enough information to construct good error messages. We are unaware of research that directly handles all of these concerns.

We solve these problems with a system of 2 dependently typed languages connected by an elaboration procedure.

- The surface language, a conventional full-spectrum dependently typed language (section 2)
  - the untyped syntax is used directly by the programer
  - the type theory is introduced to make formal comparisons
- The cast language, a dependently typed language with embedded runtime checks (section 3)
  - will actually be run
  - intended to be invisible to the programer
- An elaboration procedure that transforms untyped surface syntax into checked cast language terms (section 4)

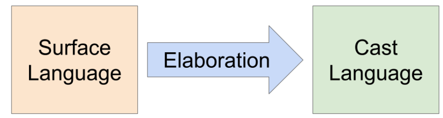

Figure 1. Architecture

The programmer uses the untyped syntax of the surface language to write programs that they intend to typecheck in the conventional dependently typed surface language. Programs that fail to typecheck under the conservative type theory of the surface language, are elaborated into the cast language. These cast language terms act exactly as typed surface language terms would, unless the programmer assumed an incorrect equality. If an incorrect equality is encountered, a clear runtime error message is presented against the static location of the error, with a counter example.

Our contributions are

- Metatheoretic properties of the cast language: culminating in a weakened form of type soundness[1], we call cast soundness (section 3)
- Metatheoretic properties of elaboration (section 4)
- A proposal of how to extend the system with data (section 5)
- Formalized Coq proofs[2] of the type soundness of the surface language, and cast soundness of the cast language
- A prototype implementation[3]

2 Surface Language

In an ideal world programmers would write perfect code with perfectly proven equalities. The surface language models this ideal, but difficult, system. Thus the surface language enforces definitional equality, and is a standard well behaved core calculus. We intend for programmers to "think" in the surface language and hope the machinery of later sections

[1]well typed programs will not "get stuck"
[2]https://github.com/qcfu-bu/dtest-coq
[3]https://github.com/marklemay/dDynamic

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	$::=$	$x \mid \star$	
	$\mid$	$(x : M_{\ell}) \to N_{\ell'}$	
	$\mid$	$\mathsf{fun}\, f\, x \Rightarrow m$	

Figure 2. Surface Language Pre-Syntax

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

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

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

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

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

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

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

Figure 3. Surface Language Bidirectional Typing Rules

reinforces an understanding of the surface type system, while being transparent to the programmer.

The surface language presented here is a minimal intensional dependent type theory. We allow some programmatic features, since we hope to target functional programmers who are not type theory experts. The language allows general recursion, since general recursion is useful for general purpose functional programming. It also supports type-in-type, since we believe it simplifies the system for programmers and makes the metatheory slightly easier.

The pre-syntax can be seen in figure 2. Location data ℓ is marked at every position in syntax where the type of a term may be contradicted by the type expected at that position. When unnecessary the location information ℓ will be left implicit.

The surface language supports bidirectional type-checking over the pre-syntax with the rules in figure 3. Bidirectional type-checking is a form of lightweight type inference, and strikes a good compromise between the needed type annotations and the simplicity of the theory. This is accomplished by breaking typing judgments into 2 forms:

- Inference judgments where type information propagates out of a term, $\overrightarrow{:}$ in our notation.
- And Checking judgments where a type is checked against a term, $\overleftarrow{:}$ in our notation.

Except for the features already noted, our bidirectional rules are standard.

The language has type soundness, well typed terms will never "get stuck" in the surface language. This can be shown by generalizing from the bidirectional system to a corresponding type assignment system. The type assignment system can be shown sound using a progress and preservation style proof. The key is to show that computation is confluent and use that computation to generate the definitional equality relation. This allows definitional equality to distinguish constructors while still being easy to prove an equivalence. As usual, computation can be shown confluent using parallel-reductions [Takahashi 1995]. These techniques are known (a similar proof is in [Sjöberg et al. 2012]), but we want to call attention to this style of proof as especially elegant and relatively easy to work with. We have mechanized the type soundness of the type assignment system (without location data) in Coq. Type checking is undecidable because of our addition of general recursion and type-in-type. However, since the user is not expected to type-check their program directly this should not cause any issues in practice.

Unfortunately, the system is logically unsound (every type is trivially inhabited with recursion), since our language attempts to be more oriented to programs than proofs. We expect this is acceptable.

It might seem restrictive that the surface language only supports dependent recursive functions. However, this is extremely expressive: church style data can be encoded, as can calculus of construction style predicates, recursion can simulate induction, and type-in-type allows large elimination (see [Cardelli 1986] for examples). This is still inconvenient, so we have implemented dependent data in our prototype. We suggest ways dependent data could be added to the theory in Section 4.

It should be noted that similar systems have been studied going back to [Martin-Löf 1972] before type-in-type was known to be unsound. The semantics was further explored in [Cardelli 1986] and an early bidirectional type-checking algorithm for a similar language is specified in [Coquand 1996]. Cayenne [Augustsson 1998], a Haskell-like language, combined dependent types with type-in-type and non-termination. It was more recently explored in the context of call by value evaluation in [Jia et al. 2010] and [Sjöberg variable contexts,

$$
\begin{array}{llll}
H & ::= & \Diamond \mid H, x : A & \\
\multicolumn{4}{l}{\text{expressions,}} \\
a, b, A, B & ::= & x & \\
 & \mid & a ::_{A,\ell,o} B & \text{cast} \\
 & \mid & \star & \\
 & \mid & (x : A) \to B & \\
 & \mid & \mathsf{fun}\ f\ x \Rightarrow b & \\
 & \mid & b\ a & \\
\multicolumn{4}{l}{\text{observations,}} \\
o & ::= & . \mid o.arg & \text{function type-arg} \\
 & \mid & o.bod[a] & \text{function type-body}
\end{array}
$$

Figure 4. Cast Language Pre-Syntax

et al. 2012]. Though not novel, we believe our Coq proof to be the clearest formal exposition to date.

3 Cast Language

The cast language is a version of the surface language that supports runtime checks. These runtime checks remember the sources of potential inequalities and what observation is needed to witness them. Observations can be refined over evaluation. The cast language has its own type system to ensure that error information is maintained consistently. To avoid confusion we will refer to terms that type according to the cast language type system as well-cast. If a term that is well-cast "gets stuck" there will always be enough information to blame a source location with a witness of inequality.

The pre-syntax for the cast can be seen in figure 4. Every questionable equality records a source location where it was asserted and a concrete observation that would witness inequality. Unlike the surface language, locations can only appear under casts. In addition to the intended type, casts also record the type of the underlying term.

The cast language supports its own type assignment system (figure 5). The system ensures that computation will not get stuck without enough information for errors.

Values are specified by judgments in (figure 6). They are standard except that casts of values that cannot step are also considered values. Small steps are listed in figure 7. They are standard for call-by-value except that casts can distribute over application, and casts can reduce when both types are $\star$. We deal with higher order functions by distributing function casts around applications. If an application happens to a cast of function type, the argument and body cast is separated and the argument cast is swapped. For instance in

$$
\begin{aligned}
&\left((\lambda x \Rightarrow x \&\& x) ::_{Bool \to Bool,\ell,.}\ Nat \to Nat\right) 7 \\
\rightsquigarrow\ &\left((\lambda x \Rightarrow x \&\& x)\ \left(7 ::_{Nat,\ell,.arg}\ Bool\right)\right) ::_{Bool,\ell,.bod[7]}\ Nat \\
\rightsquigarrow\ &\left(\left(7 ::_{Nat,\ell,.arg}\ Bool\right) \&\&\ \left(7 ::_{Nat,\ell,.arg}\ Bool\right)\right) \\
&::_{Bool,\ell,.bod[7]}\ Nat
\end{aligned}
$$

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

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

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

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

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

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

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

Figure 5. Cast Language Type Assignment Rules

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

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

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

$$\frac{\begin{array}{c} a\ \textbf{Val} \quad A\ \textbf{Val} \quad B\ \textbf{Val} \\ a \neq \star \\ a \neq (x : C) \to C' \end{array}}{a ::_{A,\ell o} B\ \textbf{Val}}\ ::\text{-Val}$$

Figure 6. Cast Language Values

if evaluation gets stuck on && and we can blame the argument of the cast for equating Nat and $Bool$. This is analogous to the swapping of blame parity in higher order contract systems [Findler and Felleisen 2002] and gradual type systems [Wadler and Findler 2009].

The body casts record the symbolic arguments so if the function is not simply typed there is enough information to give a good error. For instance in the $.bod[7]$ observation.

Because casts can be embedded inside of casts, types themselves need to normalize and casts need to simplify. Since our theoretical language has one universe of types, type casts only need to simplify themselves when a term of type $\star$ is

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

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

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

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

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

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

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

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

Figure 7. Cast Language Small Step

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

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

$$\frac{\mathbf{Blame}\ \ell\ o\ a}{\mathbf{Blame}\ \ell\ o\ \left(a ::_{A,\ell',o'} B\right)}$$

$$\frac{\mathbf{Blame}\ \ell\ o\ A}{\mathbf{Blame}\ \ell\ o\ \left(a ::_{A,\ell',o'} B\right)}$$

$$\frac{\mathbf{Blame}\ \ell\ o\ B}{\mathbf{Blame}\ \ell\ o\ \left(a ::_{A,\ell',o'} B\right)}$$

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

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

Figure 8. Cast Language Blame

cast to $\star$. For instance,

$$\begin{aligned}&\left((\lambda x \Rightarrow x) ::_{(Bool\to Bool)::_{\star,\ell,.arg}\star,\ell,.} Nat \to Nat\right) 7\\ \rightsquigarrow\ &((\lambda x \Rightarrow x) ::_{Bool\to Bool} Nat \to Nat)\ 7\\ \rightsquigarrow\ &\left((\lambda x \Rightarrow x)\left(7 ::_{Nat,\ell,.arg} Bool\right)\right) ::_{Bool,\ell,.bod[7]} Nat\\ \rightsquigarrow\ &\left(\left(7 ::_{Nat,\ell,.arg} Bool\right)\right) ::_{Bool,\ell,.bod[7]} Nat\end{aligned}$$

In addition to small step and values we also specify blame judgments in figure 8. Blame tracks the information needed to create a good error message and is inspired by the many systems of blame tracking. The first 2 rules of the blame judgment witness concrete type inequalities. With only dependent functions and universes these are the only inequalities that can be witnessed. The rest of the blame rules extract concrete witnesses from larger terms.

The cast language supports a weaker form of type soundness. $\lozenge \vdash c : C$ implies that for any c', $c \rightsquigarrow^* c'$, if **Stuck** c' then **Blame** $\ell\ o\ c'$ where **Stuck** c' means c' is not a value and c' does not step. A well cast term (in an empty context) will never get stuck without a location to blame and an observation that witnesses it. We will refer to this as cast soundness to distinguish it from the other types systems already presented. Cast soundness follows from a progress and preservation-like proof:

- The (unlisted) rule for definitional equality is generated by a system of parallel reductions.
  - Dince the parallel reductions are confluent, definitional equality is an equivalence.
  - Definitional equality can distinguish between head constructors.
- The cut lemma holds because of this
- Cast-preservation holds because of this
- A cast version of the canonical forms lemma holds
  - If $\lozenge \vdash c : C$, and c **Val** and $C \equiv \star$ then either: c is $\star$, c is $(x : A) \to B$ for some A, B, or **Blame** $\ell\ o\ c$ for some ℓ, o
  - If $\lozenge \vdash c : C$, and c **Val** and $C \equiv (x : A) \to B$ then either: c is $\mathsf{fun}\, f\, x \Rightarrow b$, or c is $b ::_{C,\ell,o} (x : A_2) \to B_2$, with C **Val**, b' **Val**, $(x : A_2) \to B_2 \equiv (x : A) \to B$

We have formalized these proofs (without location data) in our Coq development.

Because of the conversion rule and non-termination, type-checking is undecidable. Since the user will not type-check against this system directly we consider this acceptable.

As in the surface languages, the cast language is logically unsound.

Just as there are many different flavors of statically typed equality, there are also many possible choices to enforce runtime equality. We have outlined the minimal possible checking to support cast soundness. However, we suspect that more aggressive checking may be preferable in practice, especially in the presence of data types.

It should be noted that unlike static type-checking, these runtime checks have runtime costs. Though pathological, in the worst case a cast assumption alone may cause nontermination at runtime. It would be relatively easy to remove the casts of definitionally equal terms, reducing the runtime costs of terms that type-check in the surface language. Additionally if non-definitional equalities are proven elsewhere the corresponding casts can be removed, and this is a feature we hope to support in future work.

4 Elaboration

Using the cast language manually would be cumbersome and complicated. To avoid this we have an elaboration procedure that translates (possibly untyped) terms from the surface

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

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

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

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

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

$$\frac{H \vdash \mathbf{Elab}\, m\, a \overrightarrow{:} A}{H \vdash \mathbf{Elab}\, m\ \left(a ::_{A,\ell,o} A'\right) \overleftarrow{:_{\ell,o}} A'}$$

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

Figure 9. Elaboration

language into the cast language. The elaboration procedure maps well typed terms of the surface language into Cast language terms that will not cause a blameable error. Terms that do not type because of dubious type assumptions are mapped with enough information to point out the original source and a witnessing observation. For example,

$\vdash (\lambda x \Rightarrow 7) ::_{\ell} \mathbb{B} \to \mathbb{B}$ elaborates to $\vdash (\lambda x \Rightarrow 7 ::_{\mathbb{N}.\ell,.bod[x]} \mathbb{B})$

$f : \mathbb{B} \to \mathbb{B} \vdash f_{\ell} 7\ : \mathbb{B}$ elaborates to $f : \mathbb{B} \to \mathbb{B} \vdash f\left(7 ::_{\mathbb{N}.\ell,.arg} \mathbb{B}\right)\ : \mathbb{B}$

$f : \mathbb{N} \to \mathbb{B} \to \mathbb{B} \vdash f_{\ell} 7_{\ell'} 3\ : \mathbb{B}$ elaborates to $f : \mathbb{N} \to \mathbb{B} \to \mathbb{B} \vdash f\left(7 ::_{\mathbb{N}.\ell,.arg} \mathbb{N}\right)\left(3 ::_{\mathbb{N}.\ell',.arg} \mathbb{B}\right)\ : \mathbb{B}$

The elaboration procedure is written in a bidirectional style (figure 9) that minimizes the annotations required and propagates type inference in a sensible way. The rules roughly correspond to the bidirectional rules of the surface language. Because the application rule follows this bidirectional style, we may need to determine if a type level computation results in a function type. This computation causes the relation to be undecidable, but is the only source of undecidability.

When interpreted as a procedure, this undecidability could cause non-termination. Since the application requires that the elaborated type of the function position reaches at least weak head normal form. This only happens in pathological cases. If we did not allow general recursion (and the non-termination allowable by type-in-type), we suspect elaboration would always terminate.

Unlike in gradual typing, we cannot elaborate arbitrary untyped syntax. The underlying type of a cast needs to be known so that a function type can swap its argument type at application. For instance, $\lambda x \Rightarrow x$ will not elaborate since the intended type is not known. Though arbitrary syntax does not elaborate, experimental testing suggests that a majority of randomly generated terms can be elaborated, while only a small minority of terms would type-check in the surface language. The programmer can make any term elaborate if they annotate the intended type. For instance, $(\lambda x \Rightarrow x) :: * \to *$ will elaborate.

We can show several desirable properties of elaboration,

1. Every term elaborated into the cast language is well-cast.
   a. for any $\mathbf{Elab}\, \Gamma\, H$, then $H \vdash$
   b. for any $H \vdash \mathbf{Elab}\, a\, m \overrightarrow{:} A$, then $H \vdash a : A$
   c. for any $H \vdash \mathbf{Elab}\, a\, m \overleftarrow{:_{\ell o}} A$, then $H \vdash a : A$
2. Every term typed by the bidirectional system elaborates
   a. if $\Gamma \vdash$, then there exists H such that $\mathbf{Elab}\, H\, \Gamma$
   b. if $\Gamma \vdash m \overrightarrow{:} M$ then there exists a and A such that $\vdash \mathbf{Elab}\, m\, a \overrightarrow{:} A$
   c. if $\Gamma \vdash m \overleftarrow{:} M$ and given ℓ then there exists a, A, and o such that $H \vdash \mathbf{Elab}\, a\, m \overleftarrow{:_{\ell o}} A$
3. Blame never points to something that checked in the bidirectional system
   a. if $\vdash m \overrightarrow{:} M$, and $\vdash \mathbf{Elab}\, m\, a \overrightarrow{:} A$, then for no $a \rightsquigarrow^{*} a'$ will $\mathbf{Blame}\ \ell\ o\ a'$
4. Whenever an elaborated cast term evaluates, the corresponding surface term evaluates consistently
   a. if $H \vdash \mathbf{Elab}\, m\, a \overrightarrow{:} A$, and $a \rightsquigarrow^{*} \star$ then $m \rightsquigarrow^{*} \star$
   b. if $H \vdash \mathbf{Elab}\, m\, a \overrightarrow{:} A$, and $a \rightsquigarrow^{*} (x : A) \to B$ then there exists N and M such that $m \rightsquigarrow^{*} (x : N) \to M$

The last three guarantees are inspired by, but are less composable than, the gradual guarantee [Siek et al. 2015] for gradual typing.

The first property follows from mutual induction on elaboration judgments.

The 2nd property follows by mutual induction on the bidirectional typing judgments.

Property 3 follows from elaborations preserving erasure (figure 10), and the type soundness of the surface language.

Property 4 follows from elaborations preserving erasure.

5 Adding Data

In addition to the dependent functions already described, our implementation supports dependent data types. While adding data does not increase the theoretical power of the language, it is essential for realistic programming. For instance, the example in the introduction relies on the data types `Bool` , `Nat` and `Vec` (figure 11). Surprisingly, our handling of data requires significant modifications to the theory described in sections 2-4.

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

Figure 10. Erasure

```
data Bool : * {
 | True : Bool
 | False : Bool
};

data Nat : * {
 | Z : Nat
 | S : Nat -> Nat
};

data Vec : (A : *) -> Nat -> * {
 | Nil  : (A : *) -> Vec A Z
 | Cons : (A : *) -> A -> (x : Nat)
          -> Vec A x -> Vec A (S x)
};
```

Syntactic sugar expands decimal numbers in source code into their unary representation.

Figure 11. Definitions of Common Data Types

Data in dependent type theories is often characterized by how data can be used. For example, the functions used in the introductory example all need to inspect data (figure 13). Our implementation eliminates data with case syntax that branches off of every constructor. cases do not support nested pattern matching.

The case construct allows large eliminations with arbitrary computations, supporting dependence on both the parameters of the type constructor and on the scrutinee itself. Unless the elimination is simply typed the motive must be expressed explicitly. Since non-termination is already allowed, we do not require data definitions to be strictly positive. Recursive and mutually recursive data is available without additional syntax.

We have proven type soundness of the surface language with data (a similar system is described and proven in [Sjöberg et al. 2012]).

```
add : Nat -> Nat -> Nat;
add x y = case x {
 | Z   => y
 | S p => S (add x y)
};

rep : (A : *) -> A -> (x : Nat) -> Vec A x;
rep A a x = case x <x': Nat => Vec A x'> {
 | Z   => Nil A
 | S p => Cons A a p (rep p)
};

head : (A : *) -> (x : Nat) -> Vec A (S x)
      -> A;
head A x v = case v
 < _ : Vec A' x' =>
   case x'{
     Z   => Unit
     S _ => A'
   }
 > {
 | Nil _         => tt
 | Cons A' a _ _ => a
};
```

Figure 12. Definitions of Functions Using Data

$$
\frac{\begin{array}{c}
\Gamma \vdash n \overrightarrow{:} D\, \overline{P} \\
\mathsf{data}\, D\, \Delta\, \left\{ \overline{\mid d_i\, \Theta_i \to D \overline{m}_i} \right\} \in \Gamma \\
\Gamma, \overline{y} : \Delta, x : D\overline{y} \vdash M \overleftarrow{:} \star \\
\forall i.\, \Gamma, \overline{z}_i : \Theta_i \vdash h_i \overleftarrow{:} M\, [x := d\overline{z}_i, \overline{y} := \overline{m}_i]
\end{array}}{\Gamma \vdash \mathsf{case}\, n\, \langle x : D\, \overline{y}.M \rangle\ \mathsf{of}\, \left\{ \overline{\mid d_i \overline{z}_i \Rightarrow h_i} \right\} \overrightarrow{:} M \left[x := n, \overline{y} := \overline{P} \right]}
$$

Where Δ and Θ represent telescope syntax. Lists of variables and terms are $\overline{x}$, $\overline{m}$ respectively. Contexts must be extended to allow both data definitions and abstract data definitions (for recursive data types).

Figure 13. Surface Language Elimination Typing

$$
\begin{array}{rcll}
a, b, A, B, C & ::= & \ldots & \\
& \mid & D\, \overline{a} & \text{data constructor} \\
& \mid & T\, \overline{a} & \text{type constructor} \\
& \mid & a ::_{A,C} B & \text{cast} \\
& \mid & a \sim_{\ell o} b & \text{asserted equivalence} \\
o & ::= & \ldots & \text{observation} \\
& \mid & o.App[a] & \text{application} \\
& \mid & o.TCon[i] & \text{type constructor arg.} \\
& \mid & o.DCon[i] & \text{data constructor arg.}
\end{array}
$$

Figure 14. Cast Language extended syntax (Selected)

$$\frac{\dots}{(\mathsf{fun}\, f\, x \Rightarrow b) \sim_{\ell o} (\mathsf{fun}\, f\, x \Rightarrow b') \rightsquigarrow \left(\mathsf{fun}\, f\, x \Rightarrow b \sim_{\ell o.App[x]} b'\right)}$$

$$\frac{\dots}{(T\, \overline{a}) \sim_{\ell o} \left(T\, \overline{b}\right) \rightsquigarrow \left(T\, \overline{a \sim_{\ell o.TCon[i]} b}\right)}$$
for each i in the list

$$\frac{\dots}{(D\, \overline{a}) \sim_{\ell o} \left(D\, \overline{b}\right) \rightsquigarrow \left(D\, \overline{a \sim_{\ell o.DCon[i]} b}\right)}$$
for each i in the list

$$\frac{\dots}{\left(a ::_{A,C} A'\right) \sim_{\ell o} b \rightsquigarrow a \sim_{\ell o} b}$$

$$\frac{\dots}{a \sim_{\ell o} \left(b ::_{B,C} B'\right) \rightsquigarrow a \sim_{\ell o} b}$$

Figure 15. Cast Language extended steps (Selected)

Unfortunately, adding cast support was surprisingly difficult. Since the motives are unrestricted and can depend on multiple variables, it is possible for cast errors to occur in indirect ways, and more type observations must be allowed. Our implementation overcomes this by adding a new syntactic construct that asserts untyped expressions are equal (figure 14), this allows blame to directly point to the specific index of type or data constructors. This new assertion syntax ignores cast checks to avoid getting stuck (figure 15). This approach is harder to justify theoretically, but we conjecture that the system extended with data supports all the properties we have proven for the functional fragment.

Adding general dependent data to full-spectrum dependently typed systems is surprisingly subtle. Even simple accounts add significant bookkeeping to the meta theory, and introduce awkward issues like positivity checking. Convenient dependently typed languages should support parameterized data[4], the syntax should support pattern matching and the language should support implicit arguments. We have decided to keep as simple an implementation as possible, for now, even though they are inconvenient to use.

6 Prior Work

6.1 Contract Systems, Gradual Types, and Blame

This paper has been influenced by the large amount of work done on higher order contracts [Findler and Felleisen 2002], gradual types [Garcia et al. 2016; Siek et al. 2015] [Garcia et al. 2016; Siek et al. 2015] and especially blame [Ahmed et al. 2017; Wadler 2015; Wadler and Findler 2009]. Most work in those areas focuses on simply typed languages that are not necessarily pure.

The implementation also takes inspiration from "Abstracting gradual typing" [Garcia et al. 2016], where static evidence annotations become runtime checks. Unlike some impressive attempts to gradualize the polymorphic lambda calculus [Ahmed et al. 2017], our system does not attempt to enforce any parametric properties of the base language. It is unclear if such a restriction would be desirable for a dependently typed language in practice.

A system for gradual dependent types has been presented in [Eremondi et al. 2019b]. That paper is largely concerned with establishing decidable type checking via an approximate term normalization. However, that system retains the definitional style of equality, so that it is possible, in principle, to get $vec\,(1 + x) \neq vec\,(x + 1)$ as a runtime error.

While the gradual typing goals of mixing static certainty with runtime checks are similar to our work here, the approach and details are different. Instead of trying to strengthen untyped languages by adding types we take a dependent type system and weaken its equalities. This leads to different trade-offs in the design space. For instance, we cannot support completely unannotated code, but we do not need to complicate the type language with wildcards for uncertainty. We think keeping the surface type language as simple as possible is important for dependent type systems that are already quite complicated.

6.2 Refinement Style Approaches

In this paper we have described a full-spectrum dependently typed language. This means computation can appear uniformly in both term and type position. An alternative approach to full-spectrum dependent types is refinement type systems. Refinement type systems restrict type dependency, possibly to specific base types such as int or bool. It is then straightforward to check these decidable equalities at runtime in what has been coined the Hybrid Type Checking methodology [Flanagan 2006].

A notable example is [Ou et al. 2004] which describes a refinement system that limits predicates to base types.

A refinement type system with higher order features is gradualized in [Zalewski et al. 2020] and builds on earlier refinement type system work.

7 Future Work

There are several directions we would like to take this research

- The prototype implementation should be improved
  - While the runtime error contains much of the theoretical information to give a good error message

[4] https://agda.readthedocs.io/en/v2.6.1.3/language/data-types.html#parametrized-datatypes

our implementation could greatly improve the actual presentation of error messages by making them match the example in the introduction.
  - Ideally allow more direct use though a web interface or through the language server protocol.
  - The interface should give notifications when an uncertain cast assumption is encountered.
  - While we explicitly allow logically unsound programs, we think adding static analysis to verify the parts of the code that are logically sound would be a sensible improvement. This would allow proofs of equality to remove runtime checks.
- Our implementation of data is missing several features that functional programmers expect, such as pattern matching. It would be good to know in which situations the motives and nested patterns can be elaborated into a cast. We are hopeful that improvements in this direction could be made.
- Given that this style of typing does not exclude all errors, we would like to add mechanisms to better support conventional testing, and hopefully symbolic execution. We consider the work here as a prerequisite to automated testing so that definitional equalities can be ignored. An interesting further extension to symbolic execution is runtime proof search.
- While our proofs show that our theory isn’t too unreasonable, our theory is very syntactic. We would like to have a more semantic understanding of gradual correctness that matches our intuitions about program equality.
- While we have mechanized many parts of the metatheory in Coq, we would like to fully mechanize every proof listed in this paper.
- There has been recent work to combine dependent type systems with effects [Ahman 2017]. A large hurdle for a practical implantation is the difficulty of dealing with equality of effectful computations. We may be able to use our cast methodology to avoid dealing with effectfull equalities directly.
- Currently our theory does not enforce the parametric constraints that a functional programmer might expect. Parametricity in dependent type systems has been studied in theory [Bernardy et al. 2010]. But it is still unclear what the most convenient form of parametricity is in practice. It would be good to at least communicate this better to programmers. This topic deserves more research generally.
- Finally, we would like to verify that programmers do find this system helpful with a usability study. It would be straightforward to compare programer performance between the entire system and the surface language alone.

8 Conclusion

In order for dependent types to fulfill their potential, we must continue to make them easier to use. Better error messages and more flexible evaluation seem some of the necessary ingredients of a widespread dependent language.

9 Acknowledgments

Thanks to the anonymous reviewers for their helpful feedback and correcting many of our unintentional errors. Our implementation was greatly assisted by unbound-generics[5]. Our Coq mechanizations were assisted by Autosubst [Schäfer et al. 2015]. We thank Stephanie Savir and Cheng Zhang for proofreading a draft of this paper.

References

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

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, with and without Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 39 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110283

Lennart Augustsson. 1998. Cayenne a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP ’98)*. Association for Computing Machinery, New York, NY, USA, 239–250. https://doi.org/10.1145/289423.289451

Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2010. Parametricity and Dependent Types. *SIGPLAN Not.* 45, 9 (Sept. 2010), 345–356. https://doi.org/10.1145/1932681.1863592

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming* 23, 5 (2013), 552–593.

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

Thierry Coquand. 1996. An algorithm for type-checking dependent types. *Science of Computer Programming* 26, 1 (1996), 167–177. https://doi.org/10.1016/0167-6423(95)00021-6

Joseph Eremondi, Wouter Swierstra, and Jurriaan Hage. 2019a. A framework for improving error messages in dependently-typed languages. *Open Computer Science* 9, 1 (2019), 1–32.

Joseph Eremondi, Éric Tanter, and Ronald Garcia. 2019b. Approximate Normalization for Gradual Dependent Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 88 (July 2019), 30 pages. https://doi.org/10.1145/3341692

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming* (Pittsburgh, PA, USA) *(ICFP ’02)*. Association for Computing Machinery, New York, NY, USA, 48–59. https://doi.org/10.1145/581478.581484

Cormac Flanagan. 2006. Hybrid Type Checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL ’06)*. Association for Computing Machinery, New York, NY, USA, 245–256. https://doi.org/10.1145/1111037.1111059

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL ’16)*. Association for Computing Machinery, New York, NY, USA, 429–442. https://doi.org/10.1145/2837614.2837670

[5]https://github.com/lambdageek/unbound-generics

William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. 2019. Lazy Counterfactual Symbolic Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 411–424. https://doi.org/10.1145/3314221.3314618

Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg, and Stephanie Weirich. 2010. Dependent Types and Program Equivalence. *SIGPLAN Not.* 45, 1 (Jan. 2010), 275–286. https://doi.org/10.1145/1707801.1706333

Per Martin-Löf. 1972. *An intuitionistic theory of types.* Technical Report. University of Stockholm.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types. In *Exploring New Frontiers of Theoretical Informatics*, Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell (Eds.). Springer US, Boston, MA, 437–450.

Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *Interactive Theorem Proving*, Christian Urban and Xingyuan Zhang (Eds.). Springer International Publishing, Cham, 359–374.

Eric L. Seidel, Ranjit Jhala, and Westley Weimer. 2016. 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* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 228–242. https://doi.org/10.1145/2951913.2951915

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

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

M. Takahashi. 1995. Parallel Reductions in λ-Calculus. *Information and Computation* 118, 1 (1995), 120–127. https://doi.org/10.1006/inco.1995.1057

Philip Wadler. 2015. A Complement to Blame. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 309–320. https://doi.org/10.4230/LIPIcs.SNAPL.2015.309

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

Jakub Zalewski, James McKinna, J. Garrett Morris, and Philip Wadler. 2020. λdB: Blame tracking at higher fidelity. In *Workshop on Gradual Typing (WGT20)*. Association for Computing Machinery, New Orleans, 171–192.