# Chapter 3

October 23, 2021

# Part I
# Introduction

We can now motivate the most fundemental problems with dependent type systems. Definitional equalityis prervasive and unintuitive.

For instance, dependent types can prevent an out-of-bounds error when indexing into a length indexed list.

$$\begin{aligned}
\texttt{Vec} &: 1_c \to \mathbb{N}_c \to *, \\
\texttt{rep} &: (X : *) \to X \to (y : \mathbb{N}_c) \to \texttt{Vec}\, X\, y, \\
\texttt{head} &: (X : *) \to (y : \mathbb{N}_c) \to \texttt{Vec}\, X\, (1_c +_c y) \to X
\end{aligned}$$

$$\vdash \lambda x \Rightarrow \texttt{head}\, \mathbb{B}_c\, x\ (\texttt{rep}\, \mathbb{B}_c\, true_c\ (1_c +_c x)) : \mathbb{N}_c \to \mathbb{B}_c$$

`head` is a function that expects a list of length $1_c +_c y$, making it impossible for `head` to inspect an empty list. Luckily the `rep` function will return a list of length $1_c +_c y$, exactly the type that is required.

Unfortunately, the following will not type check in the surface language,

$$\not\vdash \lambda x \Rightarrow \texttt{head}\, \mathbb{B}_c\, x\ (\texttt{rep}\, \mathbb{B}_c\, true_c\ (x +_c 1_c)) : \mathbb{N}_c \to \mathbb{B}_c$$

While "obviously" $1 + x = x + 1$, in the surface language, $1_c +_c x \not\equiv x +_c 1_c$. Recall that **definitional equality** is the name for the conservative approximation of equality used internally by dependent type systems. This prevents the use of a term of type $\texttt{Vec}\, (1_c +_c x)$ where a term of type $\texttt{Vec}\, (x +_c 1_c)$ is expected. Usually when dependent type systems encounter situations like this, they will give a type error and block further evaluation. Especially fustraitng from the programmers perspective, if the programmer made a mistake in the definition of addition, such that for some $x$, $1_c +_c x \not\equiv x +_c 1_c$, the system will not provide hints on which $x$ break this equality. The lack of clear error messages and the requirement to prove obvious equalities is a problem in all dependent type systems not just the surface language.

Why not sidestep definitional equality? Why not assume the equalities hold until a concrete witness of inequality is discovered? A good dependently typed programming language should

- Provisionally allow obvious like $1_c +_c x = x +_c 1_c$ so programming is not blocked.
- Give static warnings about potential type inequalities; users can decide if they are worth verifying.
- Give runtime errors, with a concrete witness of inequality, if one of the provisional equalities is shown not to hold in practice.

For instance, if the $+_c$ incorrectly computes $8_c +_c 1_c = 0_c$ the above function will "get stuck" on the input $8_c$. If that application is encountered at runtime we can give an error stating $9_c \neq 0_c = 8_c +_c 1_c$.

have motivated many research projects [Pro13, SW15, CTW21]. However, these impressive efforts are still only usable by experts. Further, since program equivalence is undecidable in general, no system will be able to statically verify every "obvious" equality for arbitrary user defined data types and functions. The conventional work is important, but it is not yet usable, in te meatime systems should trust the programmer when they use an unverified equality.

be turned into warnings. If a meaningful problem occurs at runtime a clear error message that pinpoints the exact source of errors should be provided. This way runtime errors and static warnings both inform the programmer. Since the user can gradually correct their program as errors surface, we call this workflow **gradual correctness**.

Though gradual correctness is an apparently simple idea, there are several subtle issues that must be dealt with.

- While it is easy to test when natural numbers are equal at runtime, testing that 2 functions are equal is impossible in general.
- Erasing too much type information will make good runtime error messages impossible.
- If equality checks are embedded into syntax, they may propagate into the type level.

We are unaware of research that directly handles all of these concerns.

If we want the above, we cannot work in the surface language directly. Our system needs

- The cast language, a dependently typed language with embedded runtime checks, that will be evaluated.
- An elaboration procedure that transforms untyped surface syntax into checked cast language terms.

| | | | |
|---|---|---|---|
| location information, | | | |
| $\ell$ | ::= | ... | |
| | \| | . | no source label |
| variable contexts, | | | |
| $H$ | ::= | $\Diamond \mid H, x : A$ | |
| expressions, | | | |
| $a, b, A, B$ | ::= | $x$ | |
| | \| | $a ::_{A,\ell,o} B$ | cast |
| | \| | $\star$ | |
| | \| | $(x : A) \to B$ | |
| | \| | $\mathsf{fun}\, f\, x \Rightarrow b$ | |
| | \| | $b\, a$ | |
| observations, | | | |
| o | ::= | . | |
| | \| | $o.arg$ | function type-arg |
| | \| | $o.bod[a]$ | function type-body |

Figure 1: Cast Language Syntax

We show that a novel form of type soundness holds. Instead of "well typed terms don't get stuck", we prove "well cast terms don't get stuck without blame". We call this **cast soundness**.

Additionally, by construction, blame (in the sense of gradual typing) is reasonably handled. Several graduality properties hold for the system overall.

# Part II
# Cast Language

## 1 Syntax

The cast language records all the potential type conflicts, and uses runtime annotations to monitor these disdrepencies. If the types do not conflict at runtime, evaluation will continue, otherwise the system will correctly blame the source location of the conflict. Terms that type according to the cast language type system as well-casts.

The syntax for the cast language can be seen in figure 1. Every time there is a type mismatch between the type of the term and the type expected from the usage, a cast is elaborated with

- a source location $\ell$ where it was asserted,
- a concrete observation $o$ that would witness inequality,
- the type of the underlieing term,

- and the expected type of the term.

# 2 Examples

## 2.1 Pretending $\star =_\star \bot$

spoofing an equality

$(\lambda pr : (\star =_\star \bot)\, .pr\ (\lambda x.x)\ \bot \quad : \neg\star =_\star \bot)\, refl_{\star:\star}$

elaborates to

$(\lambda pr : (\star =_\star \bot)\, .pr\ (\lambda x.x)\ \bot \quad : \neg\star =_\star \bot)\, (refl_{\star:\star} :: (\star =_\star \star) =_l (\star =_\star \bot))$

$refl_{\star:\star} :: (\star =_\star \star) =_l (\star =_\star \bot)\ (\lambda x.x)\ \bot \quad :\bot$

$(\lambda C : (\star \to \star)\, .\lambda x : C\ \star\, .x :: (\Pi C : (\star \to \star)\, .C \star \to C \star) =_l (\Pi C : (\star \to \star)\, .C \star \to C\ \bot))\ (\lambda x.x)\ \bot$ $:\bot$

$(\lambda x : \star .x :: (\star \to \star) =_{l,bod} (\star \to \bot))\ \bot \quad :\bot$

$(\bot :: \star =_{l,bod.bod} \bot) \quad :\bot$

note that the program has not yet "gotten stuck". to exercise this error, $\bot$ must be eliminated, this can be done by tying to summon another type by applying it to $\bot$

$((\bot :: \star =_{l,bod.bod} \bot) \quad :\bot)\ \star$

$((\Pi x : \star .x) :: \star =_{l,bod.bod} (\Pi x : \star .x))\ \star$

the computation is stuck, and the original application can be blamed on account that the "proof" has a discoverable type error at the point of application $l$

$\Pi C : (\star \to \star)\, .C \star \to \underline{C \star} \neq \Pi C : (\star \to \star)\, .C \star \to \underline{C\ \bot}$

when

$C \coloneqq \lambda x.x$

$C\ \bot = \bot \neq \star = C \star$

## 2.2 Pretending $true_c =_{\mathbb{B}_c} false_c$

spoofing an equality, evaluating $\neg true_c =_{\mathbb{B}_c} false_c$ with an incorrect proof.

$(\lambda pr : (\Pi C : (\mathbb{B}_c \to \star)\, .C\, true_c \to C\, false_c)\, .pr\ (\lambda b : \mathbb{B}_c .b \star \star \bot)\ \bot \quad : \neg true_c =_{\mathbb{B}_c} false_c)\, refl_{true_c:\mathbb{B}_c}$

is elaborated to

$(\lambda pr : (\Pi C : (\mathbb{B}_c \to \star)\, .C\, true_c \to C\, false_c)\, .pr\ (\lambda b : \mathbb{B}_c .b \star \star \bot)\ \bot \quad : \neg true_c =_{\mathbb{B}_c} false_c)\, (refl_{true_c:\mathbb{B}_c} ::$

$(refl_{true_c:\mathbb{B}_c} :: true_c =_{\mathbb{B}_c} true_c =_l true_c =_{\mathbb{B}_c} false_c)\ (\lambda b : \mathbb{B}_c .b \star \star \bot)\ \bot$

$((\lambda C : (\mathbb{B}_c \to \star)\, .\lambda x : C\, true_c .x) :: \Pi C : (\mathbb{B}_c \to \star)\, .C\, true_c \to C\, true_c =_l \Pi C : (\mathbb{B}_c \to \star)\, .C\, true_c \to C\, false$

$((.\lambda x : \star .x) :: \star \to \star =_{l,bod} \star \to \bot)\ \bot$

$(\bot :: \star =_{l,bod.bod} \bot)$

As in the above the example has not yet "gotten stuck". As above, applying $\star$ will discover the error, which would result in an error like

$\Pi C : (\mathbb{B}_c \to \star)\, .C\, true_c \to \underline{C\, true_c} \neq \Pi C : (\mathbb{B}_c \to \star)\, .C\, true_c \to \underline{C\, false_c}$

when

$C \coloneqq \lambda b : \mathbb{B}_c .b \star \star \bot$

$C\, true_c = \bot \neq \star = C\, false_c$

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

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

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

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

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

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

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

TODO: remove regularity stuff

Figure 2: Cast Language Type Assignment Rules

# 3 Cast Soundness

The cast language supports its own type assignment system (figure 2). This system ensures that computations will not get stuck without enough information for good runtime error messages. Specifically computations will not get stuck without a source location and a witness of inequality.

Unlike that surface language it is not longer practical to characterize values syntactically. Values are specified by judgments in (figure 3). They are standard except for the ::-Val, which states that a type under a cast is not a value.

Small steps are listed in figure 4. 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

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

if evaluation gets stuck on && and we can blame the argument of the cast for equating $Nat$ and $Bool$. This is similar to how blame parity is swapped in

$$\frac{}{\star\,\mathbf{Val}}\ \star\text{-Val}$$

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

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

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

Figure 3: Cast Language Values

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

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

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

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

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

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

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

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

Figure 4: Cast Language Small Step

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

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

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

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

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

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

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

higher order contract systems [FF02] and gradual type systems [WF09]. The body observation records the argument the function is called with. For instance in the *.bod*[7] observation. In a dependently typed function the exact argument of use may be important to give a good error. Because casts can be embedded inside of casts, types themselves need to normalize and casts need to simplify. Since our system has one universe of types, type casts only need to simplify themselves when a term of type $\star$ is cast to $\star$. For instance,

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

In addition to small step and values we also specify blame judgments in figure 3. Blame tracks the information needed to create a good error message and is inspired by the many systems of blame tracking [FF02, WF09, Wad15]. With only dependent functions and universes, only $*\not\simeq\to$ inequalities that can be witnessed. The first 2 rules of the blame judgment witness these concrete type inequalities, The rest of the blame rules will recursively extract concrete witnesses from larger terms.

The cast language supports a weaker form of type soundness.

For any $\Diamond \vdash c : C$, $c'$, $c \rightsquigarrow^* c'$, if $\mathbf{Stuck}\,c'$ then $\mathbf{Blame}\,\ell\,o\,c'$, where $\mathbf{Stuck}\,c'$ 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 property as "cast soundness", and it can be shown with a "progress and preservation"-like proof.

Because of the conversion rule and non-termination, type-checking is undecidable.

Prevous arguents apply.

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

Just as there are many different flavors of defineitional 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. That is why in our implementation we check equalities up to call-by-value.

Unlike static type-checking, these runtime checks have runtime costs. Since the language allows nontermination, checks can take forever to resolve. We don't expect this to be an issue in practice, since we could limit the number of steps allowed. Additionally, our implementation avoids casts when it knows that the types are equal.

# Part III
# Elaboration

## 4 Elaboration

Even though cast language saves us from reasoning about obvious equality, manually noting down every cast would be cumbersome. The elaboration procedure translates (untyped) terms from the surface language into the cast language. If the term is well typed in the serface langugage elaboration will produce a term without blameable errors. Terms with unproven equality in types are mapped to a cast with enough information to point out the original source when an inequality is witnessed.

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

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

The rules for elaboration are presented in figure 5. Elaboration rules are written in a style of bidirectional type checking. However, unlike bidirectional type checking, when inferring a check we add a cast assertion that the 2 types are equal.

There are 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 $H \vdash a : A$

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

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

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

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

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

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

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

Figure 5: Elaboration

   (c) for any $H \vdash \mathbf{Elab}\, a\, m \overleftarrow{:_{\ell o}} A$, then $H \vdash a : A$

2. Every term well typed in the surface language 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 $\ell$ 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 $\mathbf{Blame} \ell\, o a'$ occur
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 similar to the gradual guarantee [SVCB15] for gradual typing.

- The first property follows from mutual induction on elaboration judgments.

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

Figure 6: Erasure

- The 2nd property follows by mutual induction on the bidirectional typing judgments.
- Property 3 follows from elaborations preserving erasure (figure 6), and the type soundness of the surface language.
- Property 4 follows from elaborations preserving erasure.

The elaboration relation is mostly well behaved, but as presented here, it is undecidable for some pathological terms. Because the application rule follows the bidirectional style, we may need to determine if a type level computation results in a function type. If that computation "runs forever", elaboration will "run forever". 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. Fortunately, our 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.

# Part IV
# Related Work

## 4.1 Contract Systems, Gradual Types, and Blame

This paper has been influenced by the large amount of work done on higher order contracts [FF02], gradual types [SVCB15, GCT16], and especially blame [WF09, Wad15, AJSW17]. Most work in those areas focuses on simply typed languages that are not necessarily pure.

The implementation also takes inspiration from "Abstracting gradual typing" [GCT16], where static evidence annotations become runtime checks. A system for gradual dependent types has been presented in [ETG19]. 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 allow more permissive 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. Further we assume someone using a dependent type system feels positively about types in general and will not want fragments of completely typed code.

## 4.2 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 dependent types is found in refinement type systems. Refinement type systems restrict type dependency, possibly to specific base types such as int or bool. It is straightforward to check these decidable equalities at runtime in the "Hybrid Type Checking" methodology [Fla06]. A notable example is [OTMW04] which describes a refinement system that limits predicates to base types. A refinement type system with higher order features is gradualized in [ZMMW20].

# References

[AJSW17] 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.

[CTW21] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. The taming of the rew: A type theory with computational assumptions. In *ACM Symposium on Principles of Programming Languages*, 2021.

[ESH19] Joseph Eremondi, Wouter Swierstra, and Jurriaan Hage. A framework for improving error messages in dependently-typed languages. *Open Computer Science*, 9(1):1–32, 2019.

[ETG19] Joseph Eremondi, Éric Tanter, and Ronald Garcia. Approximate normalization for gradual dependent types. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. Association for Computing Machinery.

[Fla06] Cormac Flanagan. Hybrid type checking. In *ACM Symposium on Principles of Programming Languages*, POPL '06, pages 245–256, New York, NY, USA, 2006. Association for Computing Machinery.

[GCT16] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *ACM Symposium on Principles of Programming Languages*, POPL '16, pages 429–442, New York, NY, USA, 2016. Association for Computing Machinery.

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

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

[OTMW04] 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.

[Pro13] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.

[SJW16] 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.

[SVCB15] 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.

[SW15] 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.

[Wad15] 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.

[WF09] 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.

[ZMMW20] Jakub Zalewski, James McKinna, J. Garrett Morris, and Philip Wadler. λdb: Blame tracking at higher fidelity. In *Workshop on Gradual Typing*, WGT20, pages 171–192, New Orleans, January 2020. Association for Computing Machinery.

# Part V
# TODO

- Salvage dictation notes?
- where shoud I talk about error msgs?

In every popular type system users are allowed to asume unsafe equalities. Thus

| | | | |
|---|---|---|---|
| source labels, | | | |
| $\ell$ | ::= | ... | |
| | \| | . | no source label |
| type contexts, | | | |
| $\Gamma$ | ::= | $\Diamond \mid \Gamma, x : M$ | |
| expressions, | | | |
| $m, n, M, N$ | ::= | $x$ | variable |
| | \| | $m ::_\ell M$ | annotation |
| | \| | $\star$ | type universe |
| | \| | $(x : M_\ell) \to N_{\ell'}$ | function type |
| | \| | $\mathsf{fun}\, f\, x \Rightarrow m$ | function |
| | \| | $m_\ell\, n$ | application |

Figure 7: Surface Language Syntax

# Part VI
# scratch

## 5 error

If programmers found dependent type systems easier to learn and use, software could become more reliable. Unfortunately, dependent type systems have yet to see widespread use in industry. One source of difficulty is the conservative equality checking required by most dependent type systems. This conservative equality is a source of some of the confusing error messages dependent type systems are known for [ESH19].

This error will help the programmer fix the bug in `add`. There is evidence that specific examples like this can help clarify the type error messages in OCaml [SJW16] and there has been an effort to make refinement type error messages more concrete in other systems like Liquid Haskell [HXB+19].

## 6 ...

CAST LANGUAGE
- CAST LANGUGAE EXMAPLES
- CAST SOUNDNESS
- ELABORATION
- GRADUAL CORECTNESS
- EXTEND THE SURFACE SYNTAX WITH LOCATIONS
- ELABORATION SYSTEM
- S
- ...

| | | | | |
|---|---|---|---|---|
| $(x : M) \to N$ | written | $M \to N$ | when | $x \notin fv\,(N)$ |
| $\mathsf{fun}\, f\, x \Rightarrow m$ | written | $\lambda x \Rightarrow m$ | when | $f \notin fv\,(m)$ |
| $\ldots\, x \Rightarrow \lambda y \Rightarrow m$ | written | $\ldots\, x\, y \Rightarrow m$ | | |
| $x$ | written | $-$ | when | $x \notin fv\,(m)$ when $x$ binds $m$ |
| $m ::_\ell M$ | written | $m :: M$ | when | $\ell$ is irrelevant |
| $(x : M_\ell) \to N_{\ell'}$ | written | $(x : M) \to N$ | when | $\ell$, $\ell'$ are irrelevant |
| $m_\ell\, n$ | written | $m\, n$ | when | $\ell$ is irrelevant |

where $fv$ is a function that returns the set of free variables in an expression

Figure 8: Surface Language Abbreviations

A similar "partial correctness" criterion for dependent languages with non-termination run with Call-by-Value is presented in [JZSW10].