Chapter 3 (draft): the Dependent Cast System

Mark Lemay

November 12, 2021

Part I
Introduction

We can now motivate one of the most fundamental problems with dependent type systems. Definitional equalities pervasive 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,

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

While "obviously" $1 + x = x + 1$, in the surface language definitional equality does not associate these two terms, $1_c +_c x \not\equiv x +_c 1_c$. Recall that **definitional equality** is the name for the conservative approximation of equality used internally by a dependent type system. Definitional equality 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 prevent further work. Especially frustrating from the programmer's 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? A more convenient system can assume the equalities hold until a concrete witness of inequality is discovered at runtime. A good dependently typed programming language should

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

If a meaningful problem occurs at runtime a clear error message that pinpoints the exact source of errors should be provided. For instance, if the $+_c$ function 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 the system give an error stating $9_c \neq 0_c = 8_c +_c 1_c$. 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 in general undecidable.
- Some amount of information must be maintained in the term so that good runtime error messages are possible.
- If equality checks are embedded into syntax, they may propagate into the type level.

I am unaware of research that directly addresses all of these concerns. Furthermore, we cannot work in the surface language directly, a new system is needed. This is the **cast system** described in this chapter. The cast system is comprised of

- The **cast language**, a dependently typed language with embedded runtime checks, that have evaluation behavior.
- The **elaboration procedure** that transforms appropriate untyped surface syntax into checked cast language terms.

source location information,			
ℓ			
variable contexts,			
H	$::=$	$\Diamond \mid H, x : A$	
expressions,			
a, b, A, B	$::=$	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	$::=$	.	current location
	$\mid$	$o.arg$	function type-arg
	$\mid$	$o.bod[a]$	function type-body

would $a_A :? :_{\ell,o} B$ be clearer syntax then $a ::_{A,\ell,o} B$?

would it be clearer to add the ℓ to the observation?

Figure 1.1: Cast Language Syntax

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

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 syntax for the cast language can be seen in figure 1.1. By design the cast language is almost identical to the surface language except that the cast construct has been added and cast annotations have been removed.

The cast language can assume arbitrary equalities over types, $A = B$, with a cast, $a ::_{A,\ell,o} B$ given

- a source location ℓ where it was asserted,
- a concrete observation o that would witness inequality,

- the type of the underlying term A,
- and the expected type of the term B.

Every time there is a definitional mismatch between the type inferred from a term and the type expected from the usage, the elaboration procedure will produce a cast.

Observations allow indexing into terms to pinpoint errors For instance if we want to highlight the C sub expression in $(x : A) \to (y : (x : B) \to \underline{C}) \to D$ we can use the observation $..Bod[x].arg.Bod[y]$. Since, in general, the C may specifically depend on x and $y$they are tracked as part of the observation. For instance, given the type $(X : \star) \to X$ we might want to point out A when $X = A \to B$ resulting in the type $(X : \star) \to (\underline{A} \to B)$, the observation would then read $..Bod[A \to B].arg$ recording the specific input that allows an argument to be inspected.

2 How Casts Should Reduce

How does the cast construct interact with the existing constructs? These are all the interactions that could cause a term to be stuck in evaluation or block type checking

$$\begin{array}{cc} \star :: B & \text{universe under cast} \\ ((x : A) \to B) :: C & \text{function type under cast} \\ (b :: C)\, a & \text{application blocked by cast} \end{array}$$

we can account for these by realizing obvious casts should evaporate, freeing up the underling term

$$\begin{array}{cccccc} \star :: \star & \rightsquigarrow & \star & & \\ \star :: B & \rightsquigarrow & \star :: B' & \text{when} & B \sim \\ \star ::_{\ell,o} B & \text{blame} & \ell, o & \text{when} & B \text{ cann} \\ ((x : A) \to B) :: \star & \rightsquigarrow & \star & & \\ ((x : A) \to B) :: C & \rightsquigarrow & ((x : A) \to B) :: C' & \text{when} & C \sim \\ ((x : A) \to B) ::_{\ell,o} C & \text{blame} & \ell, o & \text{when} & C \text{ cann} \\ \left(b ::_{(x:A') \to B'} (x : A) \to B\right) a & \rightsquigarrow & (b\ (a :: A')) ::_{B'[x:=a::A']} B\,[x := A] & & \\ (b :: C)\, a & \rightsquigarrow & (b :: C')\, a & \text{when} & C \sim \\ (b ::_{\ell,o} C)\, a & \text{blame} & \ell, o & \text{when} & C \text{ cannot be} \end{array}$$

The fully formal reduction rules are listed later, but their complete detail can be distracting. Type universes live in the type universe, so any cast that contradicts this should be blamed. Similarly for function types. Terms that take input must be functions, so any cast that contradicts this should blame the source location. The most interesting case is when a cast confirms that the applied term is a function, but with potentially different input and output types. Then we use the function cast to determine a reasonable cast over the argument, and maintain the appropriate cast over the resulting computation. This distribution is similar to the way higher order contracts invert the polarity of blame for the arguments of higher order funcitons [FF02] [1].

[1] and also found in gradual type systems[WF09]

Note that the rules outlined here are not deterministic since there are cases when we might blame or continue reducing the argument. One of the subtle innovations of the system described in this chapter is to completely separate blame from reduction. This sidesteps many of the complexities of having a reduction relevant `abort` term in a dependent type theory. As far as reduction is concerned, bad terms simply "get stuck" as it might on a variable assumed in the typing context. Otherwise the reduction behavior is well behaved. Terms my be blamable by the rules outlined in this chapter, but it is easy to imagine more sophisticated ways to extract blame without interfering with reduction.

cites

This outlines the minimum requirements for cast reductions, there are plausibly many additional reductions that could be considered. Some tempting reductions are

$$a ::_{C} C \quad \rightsquigarrow_{=} \quad a$$
$$a ::_{C'} C \quad \rightsquigarrow_{\equiv} \quad a \quad \text{when} \quad C' \equiv C$$

However these rules preclude extracting blame that may be embedded within the casts themselves. These rules also seem to complicate the type theory. Despite this we will use these reductions in examples to keep the book keeping to a minimum.

3 Examples

3.1 Higher Order Functions

Higher order functions are dealt with 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 $Bool$. 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,

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

3.2 Pretending $true = false$

Recall that we proved $\neg true_c \doteq_{\mathbb{B}_c} false_c$ in Chapter 2. What happens if it is assumed anyway? Every type equality assumption needs an underlying term, here we can choose $refl_{true_c:\mathbb{B}_c} : true_c \doteq_{\mathbb{B}_c} true_c$, and cast that term to $true_c \doteq_{\mathbb{B}_c} false_c$ resulting in $refl_{true_c:\mathbb{B}_c} ::_{true_c \doteq_{\mathbb{B}_c} true_c} true_c \doteq_{\mathbb{B}_c} false_c$. Recall that $\neg true_c \doteq_{\mathbb{B}_c} false_c$ is a short hand for $true_c \doteq_{\mathbb{B}_c} false_c \to \bot$. What if we try to use our term of type $true_c \doteq_{\mathbb{B}_c} false_c$ to get a term of type $\bot$?

$$
\begin{array}{rl}
 & (\lambda pr \Rightarrow pr\, toLogic\, t \\
\rightsquigarrow & \left(refl_{true_c:\mathbb{B}_c} :: \right. \\
true_c \doteq_{\mathbb{B}_c} false_c \coloneqq (C : (\mathbb{B}_c \to \star)) \to C\, true_c \to C\, false_c & \left(refl_{true_c:\mathbb{B}_c} ::_{true_c \doteq_{\mathbb{B}_c} true_c} \right. \\
true_c \doteq_{\mathbb{B}_c} true_c \coloneqq (C : (\mathbb{B}_c \to \star)) \to C\, true_c \to C\, true_c & \left(refl_{true_c:\mathbb{B}_c} ::_{(C:(\mathbb{B}_c \to \star)) \to C\, true_c} \right. \\
\rightsquigarrow_= & (refl_{true_c:\mathbb{B}_c}\, toLogic :: toLogic\, tr \\
refl_{true_c:\mathbb{B}_c} \coloneqq \lambda - \; cx \Rightarrow cx & ((\lambda - \; cx \Rightarrow cx)\; toLogic :: toLogic \\
\rightsquigarrow & ((\lambda cx \Rightarrow cx) :: toLogic\, true_c \\
\rightsquigarrow_= & ((\lambda cx \Rightarrow c \\
\rightsquigarrow & (tt_c : \\
toLogic \coloneqq \lambda b \Rightarrow b \star Unit_c \; \bot_c & (tt_c :: (\lambda b \Rightarrow \\
\rightsquigarrow & (tt_c :: (t \\
true_c \coloneqq \lambda - \; x - \Rightarrow x & (tt_c :: ((\lambda - \; x \\
\rightsquigarrow & (tt_c :: ((\lambda x \\
\rightsquigarrow & (tt_c :: ((\\
\rightsquigarrow & (\\
toLogic \coloneqq \lambda b \Rightarrow b \star Unit_c \; \bot_c & (tt_c :: U \\
\rightsquigarrow & (tt_c \\
false_c \coloneqq \lambda - \; - y \Rightarrow y & (tt_c :: U \\
\rightsquigarrow\rightsquigarrow\rightsquigarrow &
\end{array}
$$

As in the above the example, the term $(tt_c :: Unit_c) ::\bot_c$ has not yet "gotten stuck". Applying any type will uncover the error.

$$
\begin{array}{cc}
 & ((tt_c :: Unit_c) ::\bot_c) \star \\
\bot_c \coloneqq (X : \star) \to X & ((tt_c :: Unit_c) :: (X : \star) \to X) \star \\
\rightsquigarrow_= & ((tt_c :: Unit_c) \star) :: \star \\
Unit_c \coloneqq (X : \star) \to X \to X & ((tt_c :: ((X : \star) \to X \to X)) \star) :: \star \\
\rightsquigarrow_= & (tt_c \star) :: (\star \to \star) :: \star \\
tt_c \coloneqq \lambda - \; x \Rightarrow x & ((\lambda - \; x \Rightarrow x) \star) :: (\star \to \star) :: \star \\
\rightsquigarrow & (\lambda x \Rightarrow x) :: (\star \to \star) :: \star \\
\text{Blame!} & (\lambda x \Rightarrow x) :: (\star \underline{\to} \star) :: \underline{\star}
\end{array}
$$

The location and observation information that was left out of the computation could generate an error message like

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

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

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

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

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

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

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

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

TODO: remove regularity stuff

Figure 4.1: Cast Language Type Assignment Rules

when
$C \coloneqq \lambda b : \mathbb{B}_c.b \star \star \bot$
$C\, true_c = \bot \neq \star = C\, false_c$
Reminding the programmer that they should not confuse true with false.

4 Cast Language Type Assignment System

Recall that type soundness is the property for a typed programming language to exhibit. In a programming language, type soundness proves some undesirable behaviors are unreachable from a well typed term. How should this apply to the cast language, where bad behaviors are intended to be reachable? We allow bad behavior, but require that when a bad state is reached we blame the original faulty type annotations. Where the slogan for type soundness is "well typed terms don't get stuck", the slogan for cast soundness is "well typed terms don't get stuck without blame".

In Chapter 2 we proved type soundness for a minimal dependently typed language, with a progress and preservation style proof given a suitable definition of term equivalence. We can extend that proof to support cast soundness with only a few modifications.

The cast language supports its own type assignment system (figure 4.1). 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. TODO: highlight differnences form CH2

As before we need a suitable reduction relation to generate our proof of equality. TODO: revew par reduductions, highlight differnences form CH2

4.1 Definitional Equality

Address the irony of using a def eq to avoid the issues of a def eq

As in Chapter 2, $\Rightarrow_*$ can be shown to be confluent. The proofs follow the same structure, but since observations can contain terms, $\Rightarrow$ and max must be extended to observations. Proofs must be extended to mutually induct on observations, since they can contain expressions that could also reduce.

$$
\begin{array}{lcl}
max(\ (\mathsf{fun}\ f\ x \Rightarrow b)\ a\ ) = & max\,(b)\,[f := \mathsf{fun}\ f\ x \Rightarrow max \\
max(\ \left(b ::_{(x:A_1)\to B_1,\ell,o}\ (x : A_2) \to B_2\right) a\ ) = & \left(max\,(b)\ \left(max\,(a) ::_{max(A_2),\ell,m}\right.\right. \\
 & \left.\left. ::_{max(B_2)\left[x:=max(a)::_{max(A_2),\ell,max(o).arg} max(A_1)\right],\ell,max(o}\right. \\
max(\ b ::_{B_1,\ell,o} B_2\ ) = & max\,(b) ::_{max(B_1),\ell,max(o} \\
max(\ \ldots\ ) = & \ldots \\
max(\ .\ ) = & . \\
max(\ o.arg\ ) = & max\,(o)\,.arg \\
max(\ o.bod[a]\ ) = & max\,(o)\,.bod[ma
\end{array}
$$

If $a \Rrightarrow a'$ then $a' \Rrightarrow max\,(a)$. If $o \Rrightarrow o'$ then $o' \Rrightarrow max\,(o)$

by mutual induction on the derivations of $m \Rrightarrow m'$ and $o \Rrightarrow o'$.

The diamond property follows directly, if $a \Rrightarrow a'$, $a \Rrightarrow a''$, implies $a' \Rrightarrow max\,(a)$,$a'' \Rrightarrow max\,(a)$.

The diamond property implies the confluence of $\Rrightarrow_*$.

It follows that $\equiv$ is transitive.

4.1.1 Stability

$\forall A, B, C.\,(x : A) \to B \Rrightarrow_* C \Rightarrow \exists A', B'.C = (x : A') \to B' \wedge A \Rrightarrow_* A' \wedge B \Rrightarrow_* B'$

by induction on $\Rrightarrow_*$

Therefore the following rule is admissible

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

4.2 Preservation

4.2.1 Context Weakening

The following rule is admissible

$$\frac{H \vdash a : A}{H, H' \vdash a : A}$$

by induction on typing derivations

$$\frac{b \Rrightarrow b' \quad a \Rrightarrow a'}{(\mathsf{fun}\, f\, x \Rightarrow b)\, a \Rrightarrow b'\,[f := \mathsf{fun}\, f\, x \Rightarrow b', x := a']}\ \Rrightarrow\text{-fun-app-red}$$

$$\frac{b \Rrightarrow b' \quad a \Rrightarrow a' \quad A_1 \Rrightarrow A_1' \quad A_2 \Rrightarrow A_2' \quad B_1 \Rrightarrow B_1' \quad B_2 \Rrightarrow B_2' \quad o \Rrightarrow o'}{\begin{array}{c}\left(b :: _{(x:A_1)\to B_1,\ell,o}\ (x : A_2) \to B_2\right) a \Rrightarrow \\ \left(b'\, a' :: _{A_2',\ell,o.arg}\ A_1'\right) :: _{B_1'\left[x:=a' :: _{A_2',\ell,o.arg} A_1'\right],\ell,o'.bod[a']}\ B_2'\,[x := a']\end{array}}\ \Rrightarrow\text{-fun-::-red}$$

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

$$\frac{}{x \Rrightarrow x}\ \Rrightarrow\text{-var}$$

$$\frac{a \Rrightarrow a' \quad A_1 \Rrightarrow A_1' \quad A_2 \Rrightarrow A_2' \quad o \Rrightarrow o'}{a :: _{A_1,\ell,o} A_2 \Rrightarrow a' :: _{A_1',\ell,o'} A_2'}\ \Rrightarrow\text{-::}$$

$$\frac{}{\star \Rrightarrow \star}\ \Rrightarrow\text{-}\star$$

$$\frac{A \Rrightarrow A' \quad B \Rrightarrow B'}{(x : A) \to B \Rrightarrow (x : A') \to B'}\ \Rrightarrow\text{-fun-ty}$$

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

$$\frac{b \Rrightarrow b' \quad a \Rrightarrow a'}{b\, a \Rrightarrow b'\, a'}\ \Rrightarrow\text{-fun-app}$$

$$\frac{}{. \Rrightarrow .}\ \Rrightarrow\text{-obs-emp}$$

$$\frac{o \Rrightarrow o'}{o.arg \Rrightarrow o'.arg}\ \Rrightarrow\text{-obs-arg}$$

$$\frac{o \Rrightarrow o' \quad a \Rrightarrow a'}{o.bod[a] \Rrightarrow o'.bod[a']}\ \Rrightarrow\text{-obs-bod}$$

$$\frac{}{a \Rrightarrow_* a}\ \Rrightarrow_*\text{-refl}$$

$$\frac{a \Rrightarrow_* a' \quad a' \Rrightarrow a''}{a \Rrightarrow_* a''}\ \Rrightarrow_*\text{-trans}$$

$$\frac{a \Rrightarrow_* a'' \quad a' \Rrightarrow_* a''}{a \equiv a'}\ \equiv\text{-def}$$

$$\frac{}{\Diamond \equiv \Diamond}\ \equiv\text{-ctx-empty}$$

$$\frac{H \equiv H' \quad A \equiv A'}{H, x : A \equiv H', x : A'}\ \equiv\text{-ctx-ext}$$

Figure 4.2: Contextual Equivalence

4.2.2 Substitution Preservation

The following rule is admissible

$$\frac{H \vdash c : C \quad H, x : C, H' \vdash a : A}{H, H'\,[x \coloneqq c] \vdash a\,[x \coloneqq c] : A\,[x \coloneqq c]}$$

by induction over typing derivations

cast-: :	$H, x : C, H' \vdash a : A$, $H, x : C, H' \vdash A : \star$, $H, x : C, H' \vdash B : \star$, wellformed ℓ, o	
	$H, H'\,[x \coloneqq c] \vdash a\,[x \coloneqq c] : A\,[x \coloneqq c]$	by induction
	$H, H'\,[x \coloneqq c] \vdash A\,[x \coloneqq c] : \star$	by induction
	$H, H'\,[x \coloneqq c] \vdash B\,[x \coloneqq c] : \star$	by induction
	$H, H'\,[x \coloneqq c] \vdash a\,[x \coloneqq c] ::_{A[x:=c],\ell,o[x:=c]} B\,[x \coloneqq c]\ :\ B\,[x \coloneqq c]$	cast-: :
other rules	...	correspond to the

4.2.3 Context Preservation

As before the notion of definitional equality can be extended to cast contexts in 4.2.
the following rule is admissible

$$\frac{\Gamma \vdash n : N \quad \Gamma \equiv \Gamma'}{\Gamma' \vdash n : N}$$

by induction over typing derivations

cast-: :	$H \vdash a : A$, $H \vdash A : \star$, $H \vdash B : \star$, wellformed ℓ, o	
	$H' \vdash a : A$	by induction
	$H' \vdash A : \star$	by induction
	$H' \vdash B : \star$	by induction
	$H' \vdash a ::_{A,\ell,o} B\ :\ B$	cast-: :
other rules	...	correspond to the inductive cases in Chapter 2

4.2.4 Inversions

As before we show inversions on the term syntaxes, generalizing the induction hypothesis up to equality, when needed.

$$\frac{H \vdash \mathsf{fun}\, f\, x \Rightarrow a\ :\ C \quad C \equiv (x : A) \to B}{H, f : (x : A) \to B, x : A \vdash b : B}$$

by induction on the cast derivation

cast-fun	$H, f : (x : A') \to B', x : A' \vdash b \ : \ B'$, $(x : A') \to B' \equiv (x : A) \to B$	
	$A' \equiv A, \quad B' \equiv B$	by stability of fun-ty
	$H, f : (x : A') \to B', x : A' \equiv H, f : (x : A) \to B, x : A$	by reflexivity of $\equiv$, extended w
	$H, f : (x : A) \to B, x : A \vdash b \ : \ B'$	by context preservation
	$H, f : (x : A) \to B, x : A \vdash b \ : \ B$	ty-conv
cast-conv	$H \vdash \mathsf{fun}\, f\, x \Rightarrow b \ : \ C'$, $C' \equiv C$, $C \equiv (x : A) \to B$	
	$C' \equiv (x : A) \to B$	by transitivity
	...	by induction
other rules	impossible	the term position has the form

This allows us to conclude the corollary

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

The following rule is admissible

$$\frac{H \vdash (x : A) \to B \ : \ C \quad C \equiv \star}{H \vdash A : \star \quad H, x : A \vdash B : \star}$$

By induction on the typing derivations

cast-fun-ty	$H \vdash (x : A) \to B \ : \ C$	
		follows directly
cast-conv	$H \vdash (x : A) \to B \ : \ C'$, $C' \equiv C$	
	$C' \equiv \star$	by transitivity
	...	by induction
other rules	impossible	since the term position has the form $(x : A) \to B$

leading to the corollary

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

The following rule is admissible

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

By induction on the typing derivations

cast-::	$H \vdash a : A \quad H \vdash A : \star \quad H \vdash B : \star$	
		follows directly
cast-conv	$H \vdash a ::_{A,\ell,o} B \ : \ C'$, $C' \equiv C$	
	...	by induction
other rules	impossible	the term position has the form $a ::_{A,\ell,o} B$

Note that each of the output judgments are smaller then the input judgments, this allows other proofs to use induction on the output of this derivation.

4.2.5 $\Rrightarrow$-Preservation

The following rule is admissible

$$\frac{a \Rrightarrow a' \quad H \vdash a : A}{H \vdash a' : A}$$

by induction

cast-$\star$	$\Rrightarrow$-$\star$	$H \vdash \star : \star$, $\star \Rrightarrow \star$
cast-var	$\Rrightarrow$-var	$H \vdash x : A$, $x \Rrightarrow x$
ty-conv		$H \vdash a : A$, $A \equiv A'$
	all $\Rrightarrow$	$a \Rrightarrow a'$
		$H \vdash a' : A$
		$H \vdash a' : A'$
cast-: :		well formed ℓ, o
	$\Rrightarrow$-::-red	$H \vdash a : \star$, $a \Rrightarrow a'$
		$H \vdash a' : \star$
	$\Rrightarrow$-::	$H \vdash a : A_1$, $H \vdash A_2 : \star$, $a \Rrightarrow a'$, $A_1 \Rrightarrow A_1'$, $A_2 \Rrightarrow A_2'$, $o \Rrightarrow o'$
		$H \vdash a' : A_1$
		$H \vdash a' : A_1'$
		$H \vdash A_2' : \star$
		$H \vdash a' ::_{A_1', \ell, o'} A_2' : A_2'$
		$H \vdash a' ::_{A_1', \ell, o'} A_2' : A_2$
cast-fun-ty		$H \vdash A : \star$, $H, x : A \vdash B : \star$
	$\Rrightarrow$-fun-ty	$A \Rrightarrow A'$, $B \Rrightarrow B'$
		$H \vdash A' : \star$
		$H, x : A \vdash B' : \star$
		$H, x : A \equiv H, x : A'$
		$H, x : A' \vdash B' : \star$
		$H \vdash (x : A') \to B' : \star$
cast-fun		$H, f : (x : A) \to B, x : A \vdash a : A$
	$\Rrightarrow$-fun	$a \Rrightarrow a'$
		$H, f : (x : A) \to B, x : A \vdash a' : A$
		$H \vdash \mathsf{fun}\, f\, x \Rightarrow a' : (x : A) \to B$
cast-fun-app		
	$\Rrightarrow$-fun-app-red	$H \vdash \mathsf{fun}\, f\, x \Rightarrow b : (x : A) \to B$, $H \vdash a : A$, $a \Rrightarrow a'$, $b \Rrightarrow b'$
		$\mathsf{fun}\, f\, x \Rightarrow b \Rrightarrow \mathsf{fun}\, f\, x \Rightarrow b'$
		$H \vdash \mathsf{fun}\, f\, x \Rightarrow b' : (x : A) \to B$
		$H, f : (x : A) \to B, x : A \vdash a'$
		$H \vdash a' : A$
		$H \vdash b' [f \coloneqq \mathsf{fun}\, f\, x \Rightarrow b', x \coloneqq a'] : B [x \coloneqq a']$
		$B [x \coloneqq a'] \equiv M [x \coloneqq a]$
		$H \vdash b' [f \coloneqq \mathsf{fun}\, f\, x \Rightarrow b', x \coloneqq a'] : B [x \coloneqq A]$
	$\Rrightarrow$-fun-app	$H \vdash a : (x : A) \to B$, $H \vdash a : A$, $a \Rrightarrow a'$, $b \Rrightarrow b'$
		$H \vdash b' : (x : B) \to A$
		$H \vdash A' : A$
		$H \vdash b'\, a' : B [x \coloneqq a']$
		$B [x \coloneqq a'] \equiv B [x \coloneqq a]$
		$H \vdash b' [f \coloneqq \mathsf{fun}\, f\, x \Rightarrow b', x \coloneqq a'] : B [x \coloneqq a]$
	$\Rrightarrow$-fun-::-red	$H \vdash \left(b ::_{(x:A_1) \to B_1, \ell, o} (x : A_2) \to B_2\right) : (x : A_2) \to B_2$, $H \vdash a : A_2$,
		$b \Rrightarrow b'$, $a \Rrightarrow a'$, $A_1 \Rrightarrow A_1'$, $A_2 \Rrightarrow A_2'$, $B_1 \Rrightarrow B_1'$, $B_2 \Rrightarrow B_2'$, $o \Rrightarrow o'$,
		$H \vdash a' : A_2$
		$H \vdash a' : A_2'$
		$H \vdash b : (x : A_1) \to B_1$, $H \vdash (x : A_1) \to B_1 : \star$, $H \vdash (x : A_2) \to B_2 : \star$
		$(x : A_2) \to B_2 \Rrightarrow (x : A_2') \to B_2'$

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

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

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

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

Figure 4.3: Cast Language Values

4.3 Progress

Unlike the surface language, it is not longer practical to characterize values syntactically. Values are specified by judgments in figure 4.3. They are standard except for the Val-::, which states that a type ($\star$or function type) under a cast is not a value.

Small steps are listed in figure 4.3. They are standard for call-by-value except that casts can distribute over application, and casts can reduce when both types are $\star$.

In addition to small step and values we also specify blame judgments in figure 4.3. Blame tracks the information needed to create a good error message and is inspired by the many systems that use blame tracking [FF02, WF09, Wad15] and universes, only inequalities of the form $* \not\equiv A \to B$ 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.

$\rightsquigarrow$ preserves types since the following rule is admissible

$$\frac{m \rightsquigarrow m'}{m \Rrightarrow m'}$$

4.3.1 Canonical forms

As in Chapter 2 we will need technical lemmas that determine the form of a value of a type in the empty context. However, canonical function values look different because they must account for the possibility of blame arising from a stuck term.

If $\vdash a : A$, $a\,\mathbf{Val}$, and $A \equiv \star$ then either

- $a = \star$,
- or there exists C, B, such that $a = (x : C) \to B$

[3]well founded since cast-inversion produces smaller judgments

[4]well founded since cast-inversion produces smaller judgments

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

by induction on the cast derivation

cast-$\star$	$\vdash \star : \star$	follows since $a = \star$
cast-fun-ty	$\vdash (x : A) \to B \ : \star$	follows since $a = (x$
ty-conv	$\vdash a : A$, $a\,\mathbf{Val}$, $A \equiv A'$, $A' \equiv \star$	which concluded $\vdash$
	$A' \equiv \star$	by transitivity, sym
	$a = \star$ or there exists C, B, such that $a = (x : C) \to B$	by induction
cast-: :	$\vdash a : A_1$, $a ::_{A_1,\ell,o} A_2\,\mathbf{Val}$,$\vdash A_1 : \star$, $\vdash A_2 : \star$, $A_2 \equiv \star$	
	$a \not\equiv \star$, $a \not\equiv (x : C) \to B$, $a\,\mathbf{Val}$	since it must have b
	but $a = \star$ or there exists C, B, such that $a = (x : C) \to B$	by induction
	!	so cast-: : case was i
cast-fun	$f : (x : A) \to B, x : A \vdash b : B$, $(x : A) \to B \equiv \star$	
	$(x : A) \to B \not\equiv \star$	by the stability of $\equiv$
	!	so cast-fun case was
other rules	impossible	since they do not ty

Leading to the corollary (by reflexivity of $\equiv$),

If $\vdash\ A\ :\ \star$, and $A\,\mathbf{Val}$ then either $A\ =\ \star$, or there exists C, B, such that $A = (x : C) \to B$

Likewise

If $\vdash a\ :\ A$, $a\,\mathbf{Val}$, and $A \equiv (x : C) \to B$ then either

- $a = \mathsf{fun}\, f\, x \Rightarrow b$
- or $a = d ::_{D,\ell,o} (x : C') \to B'$, $d\,\mathbf{Val}$, $D\,\mathbf{Val}$, $C' \equiv C$, $B' \equiv B$

by induction on the cast derivation

cast-$\star$	$\star \equiv (x : C) \to B$	
	$\star \not\equiv (x : C) \to B$!	by the stability of $\equiv$
cast-fun-ty	$\star \equiv (x : C) \to B$	
	$\star \not\equiv (x : C) \to B$!	by the stability of $\equiv$
cast-fun	$f : (x : C') \to B', x : C' \vdash b : B'$, $(x : C') \to B' \equiv (x : C) \to B$	
		follows directly
cast-: :	$\vdash a : A_1$, $\vdash A_1 : \star$, $\vdash A_2 : \star$, $A_2 \equiv (x : C) \to B$	
	$a \not\equiv \star$, $a \not\equiv (x : C) \to B$, $a\,\mathbf{Val}$, $A_1\,\mathbf{Val}$, $A_2\,\mathbf{Val}$	since it must have been a value
	$A_2 = (x : C') \to B'$	by the stability of $\equiv$, $A_2\,\mathbf{Val}$[5]
	$C' \equiv C$, $B' \equiv B$	by the stability of $\equiv$
ty-conv	$\vdash a : A$, $A \equiv A'$, $A' \equiv (x : C) \to B$	
	$A \equiv (x : C) \to B$	by transitivity
	...	by induction
other rules	impossible	since they do not type values of

As a corollary (by reflexivity of $\equiv$)

If $\vdash a : (x : C) \to B$, and $a\,\mathbf{Val}$

- $a = \mathsf{fun}\, f\, x \Rightarrow b$

[5] TODO: make an explicit lemma?

- or $a = d ::_{D,\ell,o} (x : C') \to B'$, $d\,\mathbf{Val}$, $D\,\mathbf{Val}$, $C' \equiv C$, $B' \equiv B$

which further implies $a \not\equiv \star$, $a \not\equiv (x : C) \to C'$

4.3.2 Progress

If $\vdash a : A$ then either

- $a\,\mathbf{Val}$

- there exists a' such that $a \rightsquigarrow a'$

- or there exists ℓ, o such that $\mathbf{Blame}\,\ell\,o\,a$

As usual this follows form induction on the typing derivation

cast-var $\vdash \star : \star$

$\star\,\mathbf{Val}$

cast-var $\vdash x : A$

cast-conv $\vdash a : A'$, $A' \equiv A$

$a\,\mathbf{Val}$, there exists a' such that $a \rightsquigarrow a'$, or there exists ℓ, o such that $\mathbf{Blame}\,\ell\,o$

cast-:: $\vdash a : A$, $\vdash A : \star$, $\vdash B : \star$

$a\,\mathbf{Val}$, there exists a' such that $a \rightsquigarrow a'$, or there exists ℓ_a, o_a such that $\mathbf{Blame}\,\ell_a\,o_a$

$A\,\mathbf{Val}$, there exists A' such that $A \rightsquigarrow A'$, or there exists ℓ_A, o_A such that $\mathbf{Blame}\,\ell_A\,o_A$

$B\,\mathbf{Val}$, there exists B' such that $B \rightsquigarrow B'$, or there exists ℓ_B, o_B such that $\mathbf{Blame}\,\ell_B\,o_B$

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

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

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

if $a\,\mathbf{Val}$, $A\,\mathbf{Val}$, $B\,\mathbf{Val}$,

$A = \star$ or $A = (x : C_A) \to D_A$

$B = \star$ or $B = (x : C_B) \to D_B$

if $A = \star$, $B = \star$ $a ::$

if $A = (x : C_A) \to D_A$, $B = (x : C_B) \to D_B$

$a \not=$

$a ::$

if $A = \star$, $B = (x : C_B) \to D_B$ **Bl**

if $A = (x : C_A) \to D_A$, $B = \star$ **Bl**

if there exists ℓ_a, o_a such that $\mathbf{Blame}\,\ell_a\,o_a\,a$ **Bl**

if there exists ℓ_A, o_A such that $\mathbf{Blame}\,\ell_A\,o_A\,A$ **Bl**

if there exists ℓ_B, o_B such that $\mathbf{Blame}\,\ell_B\,o_B\,B$ **Bl**

cast-fun-ty $\vdash (x : A) \to B : \star$

$(x : A) \to B\,\mathbf{Val}$

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

$\mathsf{fun}\,f\,x \Rightarrow b\,\mathbf{Val}$

cast-fun-app $\vdash b : (x : A) \to B$, $\Gamma \vdash a : A$

$a\,\mathbf{Val}$, there exists a' such that $a \rightsquigarrow a'$, or there exists ℓ_a, o_a such that $\mathbf{Blame}$

$b\,\mathbf{Val}$, there exists b' such that $b \rightsquigarrow b'$, or there exists ℓ_b, o_b such that $\mathbf{Blame}\,\ell_b$

if $b \rightsquigarrow b'$, $b\,a \rightsquigarrow b'\,a$

if $b\,\mathbf{Val}$, $a \rightsquigarrow a'$, $b\,a \rightsquigarrow b\,a'$

if $b\,\mathbf{Val}$, $a\,\mathbf{Val}$, $b = \mathsf{fun}\,f\,x \Rightarrow c$ or $b = d_b ::_{D_b,\ell_b\,o_b} (x : A') \to B'$, c

if $b = \mathsf{fun}\,f\,x \Rightarrow c$ (fu

$b = d_b ::_{D_b,\ell_b,o_b} (x : A') \to B'$, $d_b\,\mathbf{Val}$, $D_b\,\mathbf{Val}$, A'

$\vdash$ *I*

D_b

if *I*

Bl

if *I*

let

$\left(d_b\right.$

$(d_b$

if there exists ℓ_b, o_b such that $\mathbf{Blame}\,\ell_b\,o_b\,b$ **Bl**

if there exists ℓ_a, o_a such that $\mathbf{Blame}\,\ell_a\,o_a\,a$ **Bl**

$$\frac{}{\Diamond \vdash} \text{ wf-empty}$$

$$\frac{H \vdash \quad H \vdash A : \star}{H, x : A \vdash} \text{ wf-ext}$$

Figure 4.4: context well formedness

4.4 Cast Soundness

We can now prove cast 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. This follows by iteration the progress and preservation lemmas.

4.5 Further Properties

Because of the conversion rule and non-termination, type-checking is undecidable. All previous arguments from chapter 2 apply to why this may be acceptable, but the most relevant is that the system should only be used through the Elaboration procedure described in the next section.

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

Just as there are many different flavors of definitional equality that could have been used in chapter 2, there are also many possible degrees that runtime equality can be enforced. The **Blame** relation in 4.3 outlines the minimal possible checking to support cast soundness. For instance[7], $\texttt{head}\, \mathbb{B}\, 1\ (\texttt{rep}\, \mathbb{B}\, \mathit{true}\, 0)$ will result in blame since 1and 0have different head constructors. But $\texttt{head}\, \mathbb{B}\, 1\ (\texttt{rep}\, \mathbb{B}\, \mathit{true}\, 9)$ will not result in blame since 1and 9 have the same head constructor and the computation can reduce to *true*.

It is likely that more aggressive checking is preferable in practice, especially in the presence of data types. That is why in our implementation we check equalities up to call-by-value.

This behavior is consistent with the conjectured partial correctness of logically unsound Call-by-Value execution for dependent types in [JZSW10].

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.

4.5.1 Regularity

We will often want to limit ourselves to well formed contexts, which are defined in 4.4 . This definition excludes nonsensical contexts like $x : 3_c \vdash$. Sensible contexts will be required for some further proofs.

Because the derivations were chosen to simplify the type soundness proof, the $\equiv$ relation is untyped and thus untyped terms can be associated with typed terms. For instance, $\star \equiv (\lambda - \Rightarrow \star)\,(\star\star)$, (you can replace $(\star\star)$ with any untypeable expression). Which leads to junk typing judgments like $\vdash \star : (\lambda - \Rightarrow \star)\,(\star\star)$, while not $\vdash (\lambda - \Rightarrow \star)\,(\star\star) : \star$. Because of this we will sometimes need to assume $H \vdash A : \star$, in addition $H \vdash a : A$.

Part III
Elaboration

5 Elaboration

Even though the cast language allows us to optimistically assert equalities, manually noting every cast would be unrealistically cumbersome. This bureaucracy is solved with an elaboration procedure that translates (untyped) terms from the surface language into the cast language. If the term is well typed in the surface language elaboration will produce a term without blamable 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. Elaboration serves a similar role as the bidirectional type system did in chapter 2.

or will after rewritten

First we enrich the surface language with location information, ℓ, at every position that could result in a type mismatch 5.1. Not that the location tags match exactly with the check annotations of the bidirectional system. We also insist that the set of locations is nonempty, and designate a specific null location . that can be used when we need to generate fresh terms, but have no sensible location information available . All the meta theory from chapter 2 goes through assuming that all locations are equivalent and by generating null locations when needed[8]. We will avoid writing these annotations when they are unneeded (explicitly in 5.2).

[7] assuming the data types of chapter 3

[8] For instance, the parallel reduction relation will associate all locations, $\frac{M \Rrightarrow M' \quad N \Rrightarrow N'}{(x:M_l)\to N_{l'} \Rrightarrow (x:M'_{l''})\to N'_{l'''}} \Rrightarrow$-fun-ty, so that the relation does not discriminate over syntaxes that come from different locations. While the max function will map terms into the null location, $max\,((x : M_\ell) \to N_{\ell'}) = max\,((x : max\,(M)_{.}) \to max\,(N)_{.})$ so that the output is unique.

source labels,			
ℓ	$::=$	...	
	$\mid$	.	no source label
expressions,			
m, n, M, N	$::=$	x	variable
	$\mid$	$m ::_\ell M^{\ell'}$	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

Figure 5.1: Surface Language Syntax, With Locations

$m ::_\ell M^{\ell'}$	written	$m ::_\ell M$	when	ℓ' is irrelevant
$m ::_\ell M$	written	$m :: M$	when	ℓ is irrelevant
$(x : M_\ell) \to N_{\ell'}$	written	$(x : M) \to N$	when	ℓ, ℓ' are irrelevant
$m_\ell\, n$	written	$m\, n$	when	ℓ is irrelevant

Figure 5.2: Surface Language Abbreviations

5.1 Examples

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

5.2 Elaboration

The rules for elaboration are presented in figure 5.3. Elaboration rules are written in a style of bidirectional type checking. However, unlike bidirectional type checking, when checking an inference elaboration adds a cast assertion that the two types are equal. Thus any conversion checking can be suspended until runtime.

There are several desirable properties of elaboration,

- elaborated terms preserve the erasure (defined in 5.4)

  1. if $H \vdash m\, \mathbf{Elab}\, a \overrightarrow{:} A$ then $|m| = |a|$
  2. if $H \vdash m\, a \overleftarrow{:}_{\ell,o} A\, \mathbf{Elab}\, a$ then $|m| = |a|$

by mutual induction on the **Elab** derivations

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

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

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

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

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

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

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

$$\frac{H \vdash m \,\mathbf{Elab}\, a \overrightarrow{:} A}{H \vdash m \overleftarrow{:_{\ell,o}} B \;\mathbf{Elab}\, (a ::_{A,\ell,o} B)}\ \overleftarrow{\mathbf{Elab}}\text{-cast}$$

$$\frac{H \vdash m \,\mathbf{Elab}\, a \overrightarrow{:} \star}{H \vdash m \overleftarrow{:_{\ell,o}} \star \;\mathbf{Elab}\, a}\ \overleftarrow{\mathbf{Elab}}\text{-conv-}\star$$

which syntax looks the best? on the left when input, or alway on the right like the typing judgment

Figure 5.3: Elaboration

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

Figure 5.4: Erasure

$\overrightarrow{\textbf{Elab}}$-var $|x| = x = |x|$

$\overrightarrow{\textbf{Elab}}$-$\star$ $|\star| = \star = |\star|$

$\overrightarrow{\textbf{Elab}}$-fun-ty $H \vdash M \overleftarrow{:_{\ell,\cdot}} \star\, \textbf{Elab}\ A \quad H, x : A \vdash N \overleftarrow{:_{\ell',\cdot}} \star\, \textbf{Elab}\ B$

$|M| = |A|$ by induction

$|N| = |B|$ by induction

$|(x : M_\ell) \to N_{\ell'}| = (x : |M|) \to |N| = (x : |A|) \to |B| = |(x : A) \to B|$

$\overrightarrow{\textbf{Elab}}$-fun-app $H \vdash m\, \textbf{Elab}\ b \overrightarrow{:} C \quad C \Rrightarrow_* (x : A) \to B \quad H \vdash n \overleftarrow{:_{\ell,.arg}} A\, \textbf{Elab}\, a$

$|m| = |b|$ by induction

$|n| = |a|$ by induction

$|m_\ell\, n| = |m|\,|n| = |b|\,|a| = |b\, a|$

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

$|m| = |b|$ by induction

$|\mathsf{fun}\, f\, x \Rightarrow m| = \mathsf{fun}\, f\, x \Rightarrow |m| = \mathsf{fun}\, f\, x \Rightarrow |b| = |\mathsf{fun}\, f\, x \Rightarrow b|$

$\overrightarrow{\textbf{Elab}}$-: : $H \vdash M \overleftarrow{:_\ell} \star\ \textbf{Elab}\ A \quad H \vdash m \overleftarrow{:_{\ell,\cdot}} A\, \textbf{Elab}\ a$

$|m| = |a|$ by induction

$|m ::_\ell M| = |m| = |a|$

$\overleftarrow{\textbf{Elab}}$-conv $H \vdash m\, \textbf{Elab}\ a \overrightarrow{:} A, \ldots$

$|m| = |a|$ by induction

$|m| = |a| = |a ::_{A,\ell,o} A'|$

- It follows that whenever an elaborated cast term evaluates, the corresponding surface term evaluates consistently
  1. if $H \vdash m\, \textbf{Elab}\ a \overrightarrow{:} A$, and $a \rightsquigarrow_* \star$ then $m \rightsquigarrow_* \star$
  2. if $H \vdash \textbf{Elab}\ m\, a \overrightarrow{:} A$, and $a \rightsquigarrow_* (x : A) \to B$ then there exists N and M such that $m \rightsquigarrow_* (x : N) \to M$

Since $a \rightsquigarrow_* a'$ implies $|a| \rightsquigarrow_* |a'|$ and $m \rightsquigarrow_* m'$ implies $|m| \rightsquigarrow_* |m'|$.

- elaborated terms are well-cast in a well formed ctx.

  1. for any $H \vdash a \,\mathbf{Elab}\, m \overrightarrow{:} A$ then $H \vdash a : A$, $H \vdash A : \star$
  2. for any $H \vdash$, $H \vdash A : \star$, $H \vdash a \overleftarrow{:}_{\ell,o} A \,\mathbf{Elab}\, m$ then $H \vdash a : A$

revise below, causing latex some issues

by mutual induction on **Elab** derivations

$\overrightarrow{\mathbf{Elab}}$-var	$x : A \in H$	
	$H \vdash A : \star$	by $H \vdash$
	$H \vdash x : A$	cast-var
$\overrightarrow{\mathbf{Elab}}$-fun-ty	$H \vdash M \overleftarrow{:}_{\ell,\cdot} \star \,\mathbf{Elab}\, A \quad H, x : A \vdash N \overleftarrow{:}_{\ell',\cdot} \star \,\mathbf{Elab}\, B$	
	$H \vdash A : \star$	by induction
	$H, x : A \vdash B : \star$	by induction
$\overrightarrow{\mathbf{Elab}}$-fun-app	$H \vdash m \,\mathbf{Elab}\, b \overrightarrow{:} C \quad C \Rrightarrow_* (x : A) \to B \quad H \vdash n \overleftarrow{:}_{\ell,.arg} A \,\mathbf{Elab}\, a$	
	$H \vdash b : C$, $H \vdash C : \star$	by induction
	$H \vdash (x : A) \to B : \star$	by $\Rrightarrow_*$ preservation
	$H \vdash A : \star$, $H, x : A \vdash B : \star$	by fun-ty inversion
	$H \vdash b : (x : A) \to B$	cast-conv
	$H \vdash a : A$	by induction
	$H \vdash B\,[x \coloneqq a] : \star$	by subst preservation
	$H \vdash b\,a \;:\; B\,[x \coloneqq a]$	cast-fun-app
$\overleftarrow{\mathbf{Elab}}$-fun	$H, f : (x : A) \to B, x : A \vdash$, $H, f : (x : A) \to B, x : A \vdash B : \star$	
	$H, f : (x : A) \to B, x : A \vdash m \overleftarrow{:}_{\ell,o.bod[x]} B \,\mathbf{Elab}\, b$	
		TODO revise!
	$H, f : (x : A) \to B, x : A \vdash A : \star$	by reg
	$H \vdash A : \star$	by removing free vars
	$H, x : A \vdash B : \star$	similarly with fun-ty in
	$H, f : (x : A) \to B, x : A \vdash b : B$	by induction
	$H \vdash \mathsf{fun}\, f\, x \Rightarrow b \;:\; (x : A) \to B$	cast-fun
$\overrightarrow{\mathbf{Elab}}$-::	$H \vdash M \overleftarrow{:}_{\ell} \star \,\mathbf{Elab}\, A \quad H \vdash m \overleftarrow{:}_{\ell,\cdot} A \,\mathbf{Elab}\, a$	
	$H \vdash A : \star$	by induction
	$H \vdash a : A$	by induction
$\overleftarrow{\mathbf{Elab}}$-conv	$H \vdash$, $H \vdash A' : \star$, $H \vdash m \,\mathbf{Elab}\, a \overrightarrow{:} A$, with some ℓ, o	
	$H \vdash a : A$, $H \vdash A : \star$	by induction
	$H \vdash a ::_{A,\ell,o} B \;:\; B$	cast-::

- I conjecture that every term well typed in the bidirectional surface language elaborates

  1. if $\Gamma \vdash$, then there exists H such that $\Gamma \,\mathbf{Elab}\, H$
  2. $\Gamma \vdash m \overrightarrow{:} M$ then there exists H, a and A such that $\Gamma \,\mathbf{Elab}\, H$, $H \vdash m \,\mathbf{Elab}\, a \overrightarrow{:} A$

3. $\Gamma \vdash m \overleftarrow{:} M$ and given ℓ, $o$then there exists H, $a$and A such that Γ **Elab** H, $H \vdash$ **Elab** $a\, m \overleftarrow{:}_{\ell o} A$

- I conjecture blame never points to something that checked in the bidirectional system

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

The last three guarantees are similar to the gradual guarantee [SVCB15] for gradual typing.

As formulated here the elaboration procedure is total. However in a realistic implementation the function application would reduce the function type to weak head normal form (to match a realistic bidirectional implementations) and will be undecidable for some pathological terms. Because the application rule needs to determine if a type level computation results in a function type. If that computation "runs forever", elaboration will "run forever".

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
Suitable Warnings

As presented here, not every cast corresponds to a reasonable warning. For instance, $(\lambda x \Rightarrow x) ::_{\star \to \star} \star \to \star$ is a possible output from elaboration. By the rules given the cast will not reduce without input, it is however inert, and vill not cause blame. In fact since the user only interacts with the surface language, any cast $a ::_A B$ where $|A| \equiv |B|$ will not produce an understandable warning, and further should not be a direct source of runtime errors. A reasonable first attempt would be to simply remove these trivial casts, but this ignores the possibility that casts themselves may contain problematic equalities. Currently the implementation leaves most casts intact and filters our equivalent casts from the warnings shown to the user

$$
\begin{array}{llll}
Warns(& a ::_{A,\ell,o} B &) = & \{(a,\ell,o,B)\} \cup Warns(a) \cup Warns(A) \cup Warns(B) \quad \text{if} \quad |A| \not\equiv \\
Warns(& a ::_{A,\ell,o} B &) = & Warns(a) \cup Warns(A) \cup Warns(B) \quad \text{if} \quad |A| \equiv \\
Warns(& \star &) = & \emptyset \\
Warns(& x &) = & \emptyset \\
Warns(& (x : A) \to B &) = & Warns(A) \cup Warns(B) \\
Warns(& \mathsf{fun}\, f\, x \Rightarrow b &) = & Warns(b) \\
Warns(& b\, a &) = & Warns(a) \cup Warns(b)
\end{array}
$$

Since the $\equiv$ relation is undecidable an approximation can be used in practice.

Part V
Related Work

5.3 Dependent types and equality

Difficulties in dependently typed equality have motivated many research projects [Pro13, SW15, CTW21]. However, these impressive efforts currently require a high level of expertise from programmers. 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. In the meantime systems should trust the programmer when they use an unverified equality, and use that advanced research to suppress warnings.

5.4 Contract Systems

Several of the tricks and notations in this chapter find their basis in the large amount of work on higher order contracts and related fields. Higher order contracts were introduced in [FF02] as a way to dynamically enforce invariants of software interfaces, specifically higher order functionsThe notion of blame dates at least that far back. Swapping the type cast of the input argument can be traced directly to that paper's use of blame contravarience, though it is presented in a much different way.

Contract semantics were revisited in [DFFF11, DTHF12] where a more specific correctness criteria based in blame is presented.

Contract systems still generally rely on users making annotating their invariants explicitly. Similar to how programmers might include asserts in conventional languages.

While there are similarities between contract systems and the cast system outlined here, the cast system is designed to address only issues with definitional equality. Definitional equality simply isn't relevant in the vast majority of contract systems.

5.4.1 Gradual Types

Types can be viewed as a very specific form of contracts, gradual type systems allow for a mixing of the static view of data and dynamic checking. Often type information can be inferred using standard techniques, allowing programmers to write fewer contract annotations.

Gradual type systems usually achieve this by adding in a ? meta character to denote imprecise typing information. The first plausible account of gradual typed semantic's appeared in [SVCB15] with the alliterative name of "the gradual guarantee" which has inspired some of the properties in this chapter.

Additionally some of the formalism from this chapter was inspired by the "Abstracting gradual typing" methodology [GCT16], where static evidence annotations become runtime checks.

This thesis borrows some notational conventions from gradual typing such as the $a :: A$ construct for type assertions.

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 intentional style of definitional equality, so that it is possible, in principle, to get $vec\,(1+x) \neq vec\,(x+1)$ as a runtime error. Additionally it is unclear if adding the ? meta-symbol into an already very complicated type theory, easier or harder from the programmers perspective.

This chapter has a much tighter scope then the other work cited here, dealing only with equational assumptions. Anyone using a dependent type system has already bought into the usefulness of types in general and will not want fragments of completely untyped code. The common motivation for gradual type systems to gradually convert a code base from untyped to (simply) typed is less plausible than gradually converting an untyped code base to use dependent types[9].

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. However it might be reasonable to characterize this work as gradualizing the definitional equality relation.

5.4.2 Blame

Blame is one of the key ideas explored in the contract type and gradual types literature[WF09, Wad15, AJSW17]. Often the reasonableness of a system can be

[9]Especially considering that most real-life codebases will use effects, while dependent types and effects are a complicated area of ongoing research

judged by the way blame is handled[Wad15]. Blame is treated in [Wad15] very similarly to the presentation in this chapter. Though in addition to merely blaming a source locations we ensure that a witnessing observation can also be made.

5.5 Refinement Style Approaches

In this thesis I describe 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`. Under this restriction, it is straightforward to check these decidable equalities and some additional properties hold at runtime. One specific approach is called **Hybrid Type Checking** methodology [Fla06]. Another notable example is [OTMW04] which describes a refinement system that limits predicates to base types. Another example is [LT17], a refinement system treated in a specifically gradual way. 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.

[DFFF11] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: No more scapegoating. *SIGPLAN Not.*, 46(1):215–226, January 2011.

[DTHF12] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In Helmut Seidl, editor, *Programming Languages and Systems*, pages 214–233, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[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.

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

[LT17] Nico Lehmann and Éric Tanter. Gradual refinement types. *SIGPLAN Not.*, 52(1):775–788, January 2017.

[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.

[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 VI
TODO

6 Alternative Elab formalization

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

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

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

$$\frac{H \vdash m \,\mathbf{Elab}\, b \overrightarrow{:} (x : A) \to B \quad H \vdash n \overleftarrow{:_{\ell,arg}} A \,\mathbf{Elab}\, a}{H \vdash (m_\ell\, n) \,\mathbf{Elab}\, (b\, a) \overrightarrow{:} B\,[x \coloneqq a]}\,\overrightarrow{\mathbf{Elab}}\text{-fun-app}$$

$$\frac{H \vdash n \,\mathbf{Elab}\, a \overrightarrow{:} C \quad C \Rrightarrow (x : A) \to B \quad H \vdash n \overleftarrow{:_{\ell,arg}} A \,\mathbf{Elab}\, a}{H \vdash (m_\ell\, n) \,\mathbf{Elab}\, (b\, a) \overrightarrow{:} B\,[x \coloneqq a]}\,\overrightarrow{\mathbf{Elab}}\text{-fun-app}$$

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

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

$$\frac{H \vdash m \,\mathbf{Elab}\, a \overrightarrow{:} A}{H \vdash m \overleftarrow{:_{\ell,o}} B \;\mathbf{Elab}\, (a ::_{A,\ell,o} B)}\,\overleftarrow{\mathbf{Elab}}\text{-cast}$$

The $\overleftarrow{\mathbf{Elab}}$-conv-$\star$ rule is a bit of a hack, and does not reflect the implementation. This formalization allows us to remove the special case of the conversion rule.The

above is more accurate, but harder to prove. and proofs are very sensitive to the exact formalization of bidirectional type checking. seems more complicated to prove, though such a proof might be more enlightening.

It may be possible to prove one of the elaboration properties by stages, for instance with a lemma like $\Gamma \vdash m : M$ (some constraint on gamma) $H \vdash m\,\mathbf{Elab}\; a \overrightarrow{:} A$ where $a$cannot cuase blame. Then use that as a lemma in the main thing

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.

If we did not allow general recursion (and the non-termination allowable by type-in-type), it seems elaboration would always terminate.

7 Other

- Salvage dictation notes?
- properly complete the "correctness guarantees"
  - the warning scheme hints at the induction invariant that is needed, not only are the casts types generated by well typed code similar, every cast inside of those labels are similar
  - this should hold for terms the TAS and check, which is prob easier to work with.
- TODO relate to https://hal.archives-ouvertes.fr/hal-01849166v3/document ?
- locations are "correct by construction"
- In every popular type system users are allowed to assume unsafe equalities. Thus ...

Todo list

Part VII
scratch

Part VIII
unused

TODO can $\Gamma \mathbf{Elab} H$ be made simpler?

by mutual induction on the bi-directional typing derivations and the well formedness of $\Gamma \vdash$

$\overrightarrow{ty}$-var	$x : M \in \Gamma$	
	...	mutual induction
	$x : A \in H$	
	$H \vdash x \,\mathbf{Elab}\, x \overrightarrow{:} A$	$\overrightarrow{\mathbf{Elab}}$-var
$\overrightarrow{ty}$-$\star$		
	...	mutual induction
	$H \vdash \star \,\mathbf{Elab}\, \star \overrightarrow{:} \star$	$\overrightarrow{\mathbf{Elab}}$-$\star$
$\overrightarrow{ty}$-: :	$\Gamma \vdash m \overleftarrow{:} M$, $\Gamma \vdash M \overleftarrow{:} \star$, for some ℓ	(ℓ comes from the syn
	$H \vdash m \overleftarrow{:_{\ell,.}} A \,\mathbf{Elab}\, a$	by induction, $o = .$
	$\Gamma \vdash M \overleftarrow{:} \star$	by regularity
	$H \vdash M \overleftarrow{:_{\ell.}} \star \;\mathbf{Elab}\; A'$	by induction, $o = .$
	m~a, M~A, m:M then a:A	TODO
	$H \vdash (m ::_\ell M) \,\mathbf{Elab}\, a \overrightarrow{:} A$	$\overrightarrow{\mathbf{Elab}}$-: :
$\overrightarrow{ty}$-fun-ty	$\Gamma \vdash M \overleftarrow{:} \star \quad \Gamma, x : M \vdash N \overleftarrow{:} \star$, with ℓ, ℓ'	
	$H \vdash M \overleftarrow{:_{\ell,.}} \star \,\mathbf{Elab}\, A$	by induction, $o = .$
	$H, x : A \vdash N \overleftarrow{:_{\ell',.}} \star \,\mathbf{Elab}\, B$	...M~A
	$H \vdash ((x : M_\ell) \to N_{\ell'}) \,\mathbf{Elab}\, ((x : A) \to B) \overrightarrow{:} \star$	$\overrightarrow{\mathbf{Elab}}$-fun-ty
$\overrightarrow{ty}$-fun-app	$\Gamma \vdash m \overrightarrow{:} (x : N) \to M \quad \Gamma \vdash n \overleftarrow{:} N$, with ℓ	
	$H \vdash m \,\mathbf{Elab}\, b \overrightarrow{:} C$	by induction
	$C \Rrightarrow_* (x : A)$	???
	$H \vdash n \overleftarrow{:_{\ell,.arg}} A \,\mathbf{Elab}\, a$	by induction
	$H \vdash (m_\ell \, n) \,\mathbf{Elab}\, (b \, a) \overrightarrow{:} B\,[x \coloneqq a]$	$\overrightarrow{\mathbf{Elab}}$-fun-app

TODO: Mutual cases

...

The cast language records all the potential type conflicts, and uses runtime an-

notations to monitor these discrepancies. If the types do not conflict at runtime, evaluation will continue, otherwise the system will correctly blame the source location of the conflict with a concrete inequality. Terms that type according to the cast language type system will be referred to as **well casts**.

$$
\begin{array}{rl}
 & (\lambda pr \Rightarrow pr\, toLogic\, tt_c)\left(r\right. \\
\rightsquigarrow & \left(refl_{true_c:\mathbb{B}_c} ::_{true_c \doteq_{\mathbb{B}} \right. \\
true_c \doteq_{\mathbb{B}_c} false_c \coloneqq (C : (\mathbb{B}_c \to \star)) \to C\, true_c \to C\, false_c & \left(refl_{true_c:\mathbb{B}_c} ::_{true_c \doteq_{\mathbb{B}_c} true_c} (C : \right. \\
true_c \doteq_{\mathbb{B}_c} true_c \coloneqq (C : (\mathbb{B}_c \to \star)) \to C\, true_c \to C\, true_c & \left(refl_{true_c:\mathbb{B}_c} ::_{(C:(\mathbb{B}_c \to \star)) \to C\, true_c \to C\, true_c} \right. \\
\text{let } toLogic' \coloneqq toLogic :: (\mathbb{B}_c \to \star) & \\
\rightsquigarrow & ((refl_{true_c:\mathbb{B}_c}\, toLogic' :: toLogic'\, true_c - \\
refl_{true_c:\mathbb{B}_c} \coloneqq \lambda -\ cx \Rightarrow cx & (((\lambda -\ cx \Rightarrow cx)\ toLogic' :: toLogic'\, true_c \\
\rightsquigarrow & (((\lambda cx \Rightarrow cx) :: toLogic'\, true_c \to to \\
\text{let } tt'_c \coloneqq tt_c :: toLogic\, true_c & \\
\rightsquigarrow & (((\lambda cx \Rightarrow cx)\, tt'_c :: \\
\rightsquigarrow & ((tt'_c :: toLog \\
 & (((tt_c :: toLogic'\, true_c \\
 & (((tt_c :: (toLogic :: (\mathbb{B}_c \to \star) \\
\rightsquigarrow & (((tt_c :: (toLogic\, (true_c :: \mathbb{B} \\
toLogic \coloneqq \lambda b \Rightarrow b \star Unit_c\ \bot_c & (((tt_c :: ((\lambda b \Rightarrow b \star Unit_c\ \bot_c)\, (tr \\
\rightsquigarrow & (((tt_c :: ((true_c :: \mathbb{B}_c) \star Unit_ \\
\rightsquigarrow & (((tt_c :: ((true_c :: \mathbb{B}_c) \star Unit_ \\
\mathbb{B}_c \coloneqq (X : \star) \to X \to X \to X & (((tt_c :: ((true_c :: ((X : \star) \to X \to X \to \\
\rightsquigarrow & (((tt_c :: ((true_c\, (\star :: \star) :: (\star \to \star \to \star \\
\vdash true_c \coloneqq \lambda -\ x - \Rightarrow x & (((tt_c :: (((\lambda -\ x - \Rightarrow x)\, (\star :: \star) :: (\star \to \star \\
\rightsquigarrow & (((tt_c :: (((\lambda x - \Rightarrow x) :: (\star \to \star \to \star \\
\rightsquigarrow & (((tt_c :: (((\lambda x - \Rightarrow x)\ (Unit_c :: \star) :: (\star \\
\rightsquigarrow & (((tt_c :: (((\lambda - \Rightarrow (Unit_c :: \star)) :: (\star \\
...\rightsquigarrow & (((tt_c :: ((Unit_c :: \star) :: \\
...\rightsquigarrow & (((tt_c :: Unit_c) :: \\
\multicolumn{2}{l}{(\lambda pr \Rightarrow pr\, toLogic\, tt_c)\left(refl_{true_c:\mathbb{B}_c} ::_{true_c \doteq_{\mathbb{B}_c} true_c} true_c \doteq_{\mathbb{B}_c} false_c\right)}
\end{array}
$$

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

reduces to

...

$\lambda b \Rightarrow b \star Unit_c\ \bot_c$

...

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

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

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

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

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

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

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

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

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

is elaborated to

7.1 Pretending $\star =_\star \bot$

Recall that we proved $\neg \star =_\star \bot$ what happens if we none the less assume it? Every type equality assumption needs an underlying term, here we can choose $refl_{\star:\star} : \star =_\star \star$, and cast that term to $\star =_\star \bot$ resulting in $refl_{\star:\star} ::_{\star =_\star \star} \star =_\star \bot$. Recall that $\neg \star =_\star \bot$ is a short hand for $\star =_\star \bot \to \bot$

spoofing an equality

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

elaborates to

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

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

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

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

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

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

$C \coloneqq \lambda x . x$

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

8 old proof

- Every term well typed in the surface language elaborates

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

TODO can $\Gamma\mathbf{Elab}H$ be made simpler?

by mutual induction on the bi-directional typing derivations and the well formedness of $\Gamma \vdash$

$$\frac{\Gamma \vdash m \overleftarrow{:} M}{\Gamma \vdash m :: M \overrightarrow{:} M}\ \overrightarrow{ty}\text{-}::$$

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

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

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

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

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

$$\frac{H \vdash m \,\mathbf{Elab}\, b \overrightarrow{:} C \quad C \Rrightarrow_* (x : A) \to B \quad H \vdash n \overleftarrow{:_{\ell,.arg}} A \,\mathbf{Elab}\, a}{H \vdash (m_\ell\, n) \,\mathbf{Elab}\, (b\, a) \overrightarrow{:} B\,[x \coloneqq a]}\ \overrightarrow{\mathbf{Elab}}\text{-fun-app}$$

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

$$\frac{H \vdash n \,\mathbf{Elab}\, a \overrightarrow{:} (x : A) \to B \quad H \vdash n \overleftarrow{:_{\ell,.arg}} A \,\mathbf{Elab}\, a}{H \vdash (m_\ell\, n) \,\mathbf{Elab}\, (b\, a) \overrightarrow{:} B\,[x \coloneqq a]}\ \overrightarrow{\mathbf{Elab}}\text{-fun-app}$$

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

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

$\overrightarrow{ty}$-var	$x : M \in \Gamma$	
	...	mutual induction
	$x : A \in H$	
	$H \vdash x \, \mathbf{Elab} \, x \overrightarrow{:} A$	$\overrightarrow{\mathbf{Elab}}$-var
$\overrightarrow{ty}$-$\star$		
	...	mutual induction
	$H \vdash \star \, \mathbf{Elab} \, \star \overrightarrow{:} \star$	$\overrightarrow{\mathbf{Elab}}$-$\star$
$\overrightarrow{ty}$-: :	$\Gamma \vdash m \overleftarrow{:} M$, for some ℓ	(ℓ comes from the syntactic exte
	$H \vdash m \overleftarrow{:_{\ell,.}} A \, \mathbf{Elab} \, a$	by induction, $o = .$
	$\Gamma \vdash M \overleftarrow{:} \star$	by regularity
	$H \vdash M \overleftarrow{:_{\ell.}} \star \; \mathbf{Elab} \; A'$	by induction[10]
	m~a, M~A, m:M then a:A	TODO
	$H \vdash (m ::_\ell M) \, \mathbf{Elab} \, a \overrightarrow{:} A$	$\overrightarrow{\mathbf{Elab}}$-: :
$\overrightarrow{ty}$-fun-ty	$\Gamma \vdash M \overleftarrow{:} \star \quad \Gamma, x : M \vdash N \overleftarrow{:} \star$, with ℓ, ℓ'	
	$H \vdash M \overleftarrow{:_{\ell,.}} \star \, \mathbf{Elab} \; A$	by induction, $o = .$
	$H, x : A \vdash N \overleftarrow{:_{\ell',.}} \star \, \mathbf{Elab} \; B$	...M~A
	$H \vdash ((x : M_\ell) \to N_{\ell'}) \, \mathbf{Elab} \, ((x : A) \to B) \overrightarrow{:} \star$	$\overrightarrow{\mathbf{Elab}}$-fun-ty
$\overrightarrow{ty}$-fun-app	$\Gamma \vdash m \overrightarrow{:} (x : N) \to M \quad \Gamma \vdash n \overleftarrow{:} N$, with ℓ	
	$H \vdash m \, \mathbf{Elab} \; b \overrightarrow{:} C$	by induction
	$C \Rrightarrow_* (x : A)$	???
	$H \vdash n \overleftarrow{:_{\ell,.arg}} A \, \mathbf{Elab} \, a$	by induction
	$H \vdash (m_\ell \, n) \, \mathbf{Elab} \, (b \, a) \overrightarrow{:} B \, [x \coloneqq a]$	$\overrightarrow{\mathbf{Elab}}$-fun-app

1. 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

[10] not well founded?