Irrelevance, Heterogenous Equality, and Call-by-value Dependent Type Systems

Vilhelm Sjöberg
University of Pennsylvania
vilhelm@cis.upenn.edu

Chris Casinghino
University of Pennsylvania
ccasin@cis.upenn.edu

Ki Yung Ahn
Portland State University
kya@cs.pdx.edu

Nathan Collins
Portland State University
nathan.collins@gmail.com

Harley D. Eades III
University of Iowa
harley-eades@uiowa.edu

Peng Fu
University of Iowa
peng-fu@uiowa.edu

Garrin Kimmell
University of Iowa
garrin-kimmell@uiowa.edu

Tim Sheard
Portland State University
sheard@cis.pdx.edu

Aaron Stump
University of Iowa
astump@acm.org

Stephanie Weirich
University of Pennsylvania
sweirich@cis.upenn.edu

We present a full-spectrum dependently typed core language which includes both *nontermination* and *computational irrelevance* (a.k.a. erasure), a combination which has not been studied before. The two features interact: to protect type safety we must be careful to only erase terminating expressions. Our language design is strongly influenced by the choice of CBV evaluation, and by our novel treatment of propositional equality which has a heterogenous, completely erased elimination form.

Note to reviewers This document consists of the paper itself (pages 1-22) followed by a technical appendix with detailed proofs. The appendix supplies additional details but is not necessary for understanding the paper.

1 Introduction

The Trellys project is a collaborative effort to design a new dependently typed programming language. Our goal is to bridge the gap between ordinary functional programming and program verification with dependent types. Programmers should be able to port their existing functional programs to Trellys with minor modifications, and then gradually add more expressive types as appropriate.

This goal has implications for our design. First, and most importantly, we must consider **nontermination** and other effects. Unlike Coq and Agda, functional programming languages like OCaml and Haskell allow general recursive functions, so to accept functional programs 'as-is' we must be able to turn off the termination checker. We also want to use dependent types even with programs that may diverge.

Second, the fact that our language includes effects means that order of evaluation matters. We choose **call-by-value** order, both because it has a simple cost model (enabling programmers to understand the running time and space usage of their programs), and also because CBV seems to to work particularly well for nonterminating dependent languages (as we explain in section 1.1).

Submitted to:
MSFP 2012

Finally, to be able to add precise types to a program without slowing it down, we believe it is essential to support **computational irrelevance**—expressions in the program which are only needed for type-checking should be erased during compilation and require no run-time representation. We also want to reflect irrelevance in the type system, where it can also help reason about a program.

These three features interact in nontrivial ways. Nontermination makes irrelevance more complicated, because we must be careful to only erase terminating expressions. On the other hand CBV helps, since it lets us treat variables in the typing context as terminating.

To study this interaction, we have designed a full-spectrum, dependently-typed core language with a small-step call-by-value operational semantics. This language is inconsistent as a logic, but very expressive as a programming language: it includes general recursion, datatypes, abort, large eliminations and "Type-in-Type".

The subtleties of adding irrelevance to a dependent type system all have to do with equality of expressions. Therefore many language design decisions are influenced by our **novel treatment of propositional equality**. This primitive equality has two unusual features: it is computationally irrelevant (equality proofs do not need to be examined during computation), and it is "very heterogenous" (we can state *and* use equations between terms of different types).

This paper discusses some of the key insights that we have gained in the process of this design. In particular, the contributions of this paper include:

1. The presence of nontermination means that the application rule must be restricted. This paper presents the most generous application rule to date (section 2.1).
2. Our language includes a primitive equality type, which may be eliminated in an irrelevant manner (section 2.2).
3. The equality type is also "very heterogenous", and we design a new variation of the elimination rule, "n-ary conv", to better exploit this feature (section 2.4). We also discuss how to add type annotations to this rule (section 2.5).
4. We support irrelevant arguments and data structure components. We show by example that in the presence of nontermination/abort the usual rule for irrelevant function application must be restricted, and propose a new rule with a value restriction (section 2.3).
5. We prove that our language is type safe (section 3).

The design choices for each language feature affects the others. By combining concrete proposals for evaluation-order, erasure, and equality in a single language, we have found interactions that are not apparent in isolation.

1.1 CBV, nontermination, and "partial correctness"

There is a particularly nice fit between nonterminating dependent languages and CBV evaluation, because the strictness of evaluation partially compensates for the fact that all types are inhabited.

For example, consider integer division. Suppose the standard library provides a function

```
div : Nat → Nat → Nat
```

which performs truncating division and throws an error if the divisor is zero. If we are concerned about runtime errors, we might want to be more careful. One way to proceed is to define a wrapper around `div`, which requires a proof of div's **precondition** that the denominator be non-zero:

```
safediv : Nat → (y:Nat) → (p: isZero y = false) → Nat
safediv = λx:Nat.λy:Nat.λp:(isZero y = false).div x y
```

Programs written using `safediv` are guaranteed to not divide by zero, even though our language is inconsistent as a logic. This works because λ-abstractions are strict in their arguments, so if we provide an infinite loop as the proof in `safediv 1 0 loop` the entire expression diverges and never reaches the division. In the `safediv` example, strictness was a matter of expressivity, since it allowed us to maintain a strong invariant. But when type conversion is involved, strictness is required for type safety. For example, if a purported proof of $\mathsf{Bool} = \mathsf{Nat}$ were not evaluated strictly, we could use an infinite loop as a proof and try to add two booleans. This is recognized by, e.g. GHC Core, which does most evaluation lazily but is strict when computing type-equality proofs [27].

While strict λ-abstractions give preconditions, strict data constructors can be used to express **postconditions**. For example, we might define a datatype characterizing what it means for a string (represented as a list of characters) to match a regular expression

```
data Match : String → Regexp → * where
  MChar : (x:Char) → Match (x::nil) (RCh x)
  MStar0 : (r:Regexp) → Match (nil) (RStar r)
  MStar1 : (r:Regexp) → (s s':String) →
   Match s r → Match s' (RStar r) → Match (s ++ s') (RStar r)
  ...
```

and then define a regexp matching function to return a proof of the match

```
match : (s:String) → (r:Regexp) → Maybe (Matches s r)
```

Such a type can be read as a partial correctness assertion: we have no guarantee that the function will terminate, but if it does and says that there was a match, then there really was. Even though we are working in an inconsistent logic, if the function returns at all we know that the constructors of `Match` were not given bogus looping terms.

Compared to normalizing languages, the properties our types can express are limited in two ways. First, of course, there is no way to state total correctness. Second, we are limited to predicates that can be witnessed by a first-order type like `Match`. In Coq or Agda we could give `match` the more informative type

```
match : (s:String) → (r:Regexp) → Either (Matches s r) (Matches s r → False)
```

which says that it is a decision-procedure. But in our language a value of type `Matches s r →False` is not necessarily a valid proof, since it could be a function that always diverges when called.

2 Language Design

We now go on the describe the syntax and type system of our language, focusing on its novel contributions.

The syntax of the language is shown in figure 1. Terms, types, and sorts are collapsed into one syntactic category as in the presentation of the lambda cube [5], but by convention we use uppercase metavariables A, B for expressions that are types. Some of the expressions are standard: the type of types $\star$ [9], variables, recursive definitions, error, the usual dependent function type, function definition, and function application. The language also includes expressions dealing with irrelevance, datatypes, and propositional equality; these will be explained in detail in the following subsections.

$$
\begin{array}{rcl}
x,y,f,p & \in & \text{variables} \\
D & \in & \text{data types, including } \mathsf{Nat} \\
d & \in & \text{constructors, including } \mathsf{0} \text{ and } \mathsf{S} \\
i,j & \in & \text{natural numbers}
\end{array}
$$

$$
\begin{array}{llcl}
\text{expressions} & a,b,A,B & ::= & \star \mid x \mid \mathsf{rec}\, f : A.v \mid \mathsf{abort}_A \\
 & & \mid & (x{:}A) \to B \mid \lambda x : A.a \mid a\, b \\
 & & \mid & [x{:}A] \to B \mid \lambda [x : A].a \mid a\, [b] \\
 & & \mid & D\, \overline{A_i} \mid d\, [\overline{A_i}]\, \overline{a_i} \mid \mathsf{case}\, a\, \mathsf{as}\, [y]\, \mathsf{of}\, \{\, \overline{d_j \Delta_j \Rightarrow b_j}^{j \in 1..k}\, \} \\
 & & \mid & a = b \mid \mathsf{join}_{a=b}\, i\, j \mid \mathsf{conv}\, a\, \mathsf{at}\, [\sim P_1/x_1] \ldots [\sim P_i/x_i] A \\
\text{telescopes} & \Delta & ::= & \cdot \mid (x : A)\Delta \mid [x : A]\Delta \\
\text{expression lists} & \overline{a_i} & ::= & \cdot \mid a\, \overline{a_i} \mid [a]\, \overline{a_i} \\
\text{conv proofs} & P & ::= & v \mid [a = b] \\
 & & & \\
\text{values} & v & ::= & \star \mid x \mid \mathsf{rec}\, f : A.v \\
 & & \mid & (x{:}A) \to B \mid \lambda x : A.a \\
 & & \mid & [x{:}A] \to B \mid \lambda [x : A].a \\
 & & \mid & D\, \overline{A_i} \mid d\, [\overline{A_i}]\, \overline{v_i} \\
 & & \mid & a = b \mid \mathsf{join}_{a=b}\, i\, j \mid \mathsf{conv}\, v\, \mathsf{at}\, [\sim P_1/x_1] \ldots [\sim P_i/x_i] A
\end{array}
$$

Figure 1: Syntax of the annotated language

The typing judgment is written $\Gamma \vdash a : A$. The full definition can be found in appendix A.5. In the rest of the paper we will highlight the interesting rules when we describe the corresponding language features. The typing contexts Γ are lists containing variable declarations and datatype declarations (discussed in section 2.3):

$$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, \mathsf{data}\, D\, \Delta\, \mathsf{where}\, \{\, \overline{d_i : \Delta_i \to\ D\, \Delta}^{i \in 1..j}\, \}$$

In order to study computational irrelevance and erasure, we define a separate language of *unannotated expressions* ranged over by metavariables m, M. The unannotated language captures runtime behavior; its definition is similar to the annotated language but with computationally irrelevant subexpressions (e.g. type annotations) removed. This is the language for which we define the operational semantics (the step relation $\rightsquigarrow_{\mathsf{cbv}}$ in figure 2). The annotated and unannotated languages are related by an *erasure operation* $|\cdot|$, which takes an expression a and produces an unannotated expression $|a|$ by deleting all the computationally irrelevant parts (figure 4). To show type safety we define an unannotated typing relation $H \vdash m : M$ and prove preservation and progress theorems for unannotated terms.

The relation $\rightsquigarrow_{\mathsf{cbv}}$ models runtime evaluation. However, in the specification of the type system we use a more liberal notion of *parallel reduction*, denoted $\rightsquigarrow_{\mathsf{p}}$. The difference is that $\rightsquigarrow_{\mathsf{p}}$ allows reducing under binders, e.g. $(\lambda x.1+1) \rightsquigarrow_{\mathsf{p}} (\lambda x.2)$ even though $(\lambda x.1+1)$ is already a CBV value. The main reason for introducing $\rightsquigarrow_{\mathsf{p}}$ in addition to $\rightsquigarrow_{\mathsf{cbv}}$ is for the metatheory: in order to characterize when two expressions are provably equal (lemma 10) we need a notion of reduction that satisfies the substitution properties in section 3.2, and we defined $\rightsquigarrow_{\mathsf{p}}$ accordingly. But because $\rightsquigarrow_{\mathsf{p}}$ allows strictly more reductions than $\rightsquigarrow_{\mathsf{cbv}}$, defining the type system is terms of $\rightsquigarrow_{\mathsf{p}}$ lets the programmer write more programs. Since the type safety proof does not become harder, we pick the more expressive type system.

In summary, we use the following judgments:

$$\begin{array}{llcl}
\text{expressions} & m,n,M,N & ::= & \star \mid x \mid \mathsf{rec}\, f.u \mid \mathsf{abort} \\
 & & \mid & (x{:}M) \to N \mid \lambda x.m \mid m\, n \\
 & & \mid & [x{:}M] \to N \mid \lambda [].m \mid m[] \\
 & & \mid & D\overline{M_i} \mid d\,\overline{m_i} \mid \mathsf{case}\, n\, \mathsf{of}\, \{\, \overline{d_j \overline{x_{ij}} \Rightarrow m_j}^{j\in 1..k} \,\} \\
 & & \mid & m = n \mid \mathsf{join} \\
\text{telescopes} & \Xi & ::= & \cdot \mid (x : M)\Xi \mid [x : M]\Xi \\
\text{expression lists} & \overline{m_i} & ::= & \cdot \mid m\, \overline{m_i} \mid []\, \overline{m_i} \\
 & & & \\
\text{values} & u & ::= & \star \mid x \mid \mathsf{rec}\, f.u \\
 & & \mid & (x{:}M) \to N \mid \lambda x.m \\
 & & \mid & [x{:}M] \to N \mid \lambda [].m \\
 & & \mid & D\overline{M_i} \mid d\,\overline{u_i} \\
 & & \mid & m = n \mid \mathsf{join} \\
\text{evaluation contexts} & \mathscr{E} & ::= & \bullet \mid \bullet\, m \mid u\, \bullet \mid \bullet\, [] \mid d\, \overline{u_i}\, \bullet\, \overline{m_i} \mid \mathsf{case}\, \bullet\, \mathsf{of}\, \{ d_j \overline{x_{ij}} \Rightarrow m_j \}
\end{array}$$

$\boxed{m \rightsquigarrow_{\mathsf{cbv}} m'}$

$$\frac{}{(\lambda x.m)\; u \rightsquigarrow_{\mathsf{cbv}} [u/x]m}\,\text{SC\_APPBETA} \qquad \frac{}{(\mathsf{rec}\, f.u_1)\; u_2 \rightsquigarrow_{\mathsf{cbv}} ([\mathsf{rec}\, f.u_1/f]u_1)\; u_2}\,\text{SC\_APPREC}$$

$$\frac{}{(\lambda [].m)[] \rightsquigarrow_{\mathsf{cbv}} m}\,\text{SC\_IAPPBETA} \qquad \frac{}{(\mathsf{rec}\, f.u_1)[] \rightsquigarrow_{\mathsf{cbv}} ([\mathsf{rec}\, f.u_1/f]u_1)[]}\,\text{SC\_IAPPREC}$$

$$\frac{}{\mathsf{case}\, (d_l\, \overline{u_i})\, \mathsf{of}\, \{\, \overline{d_j \overline{x_{ij}} \Rightarrow m_j}^{j\in 1..k} \,\} \rightsquigarrow_{\mathsf{cbv}} [\overline{u_i}/\overline{x_{il}}]m_l}\,\text{SC\_CASEBETA}$$

$$\frac{m \rightsquigarrow_{\mathsf{cbv}} n}{\mathscr{E}[m] \rightsquigarrow_{\mathsf{cbv}} \mathscr{E}[n]}\,\text{SC\_CTX} \qquad \frac{}{\mathscr{E}[\mathsf{abort}] \rightsquigarrow_{\mathsf{cbv}} \mathsf{abort}}\,\text{SC\_ABORT}$$

Figure 2: Syntax and operational semantics of the unannotated language

$\Gamma \vdash a : A$	Typing of annotated expressions
$H \vdash m : M$	Typing of unannotated expressions
$m \rightsquigarrow_{\mathsf{cbv}} m'$	(Runtime, deterministic CBV) evaluation
$m \rightsquigarrow_{\mathsf{p}} m'$	(Typechecking-time, nondeterministic) parallel reduction

Nontermination and Error Before moving on to the more novel parts of the language we mention how recursive definitions and error terms are formalized. Recursive definitions are made using the $\mathsf{rec}\, f : A.v$ form, with the typing rule

$$\frac{\begin{array}{c}\Gamma, f : A \vdash v : A \quad \Gamma \vdash A : \star \\ A \text{ is } (x{:}A_1) \to A_2 \text{ or } [x{:}A_1] \to A_2\end{array}}{\Gamma \vdash \mathsf{rec}\, f : A.v : A}\text{T\_REC}$$

For simplicity the rule restricts A so that a `rec` can only have a function type, disallowing (for example) recursive types or pairs of mutually recursive functions. A typical use of the form will look like $\mathsf{rec}\, f : (x{:}A) \to B.\lambda x : A.b$. Rec-expressions are values, and a rec-expression in an evaluation context steps by the rule $(\mathsf{rec}\, f.u_1)\ u_2 \rightsquigarrow_{\mathsf{cbv}} ([\mathsf{rec}\, f.u_1/f]u_1)\ u_2$. This maintains the invariant that CBV evaluation only substitutes values for variables.

In addition to nonterminating expressions, we include explicit error terms abort_A, which can be given any well-formed type.

$$\frac{\Gamma \vdash A : \star}{\Gamma \vdash \mathsf{abort}_A : A}\text{T\_ABORT}$$

An abort expression propagates past any evaluation context by the rule $\mathscr{E}[\mathsf{abort}] \rightsquigarrow_{\mathsf{cbv}} \mathsf{abort}$. This is a standard treatment of errors. General recursion already lets us give a looping expression any type in any context, so it is not surprising that this is type safe. However, note that the stepping rule for abort could be considered an extremely simple control effect. We will see that this is already enough to influence the language design.

2.1 CBV Program Equivalence meets the Application rule

Adding more effects to a dependently typed language requires being more restrictive about what expressions the type system equates. Pure, strongly normalizing languages can allow arbitrary β-reductions when comparing types, for example reducing $(\lambda x.m)\ n$ either to $[n/x]m$ or by reducing n. This works because any order of evaluation gives the same answer. In our language that is not the case, e.g. $(\lambda x.3)$ abort evaluates to abort under CBV but to 3 under CBN. We can not have both equations $((\lambda x.3)\ \mathsf{abort}) = \mathsf{abort}$ and $((\lambda x.3)\ \mathsf{abort}) = 3$ at the same time, since by transitivity all terms would be equal. Our type system must commit to a particular order of evaluation.

Therefore, as in previous work [13], our type system uses a notion of equality that respects CBV contextual equivalence. Two terms can by proven equal if they have a common reduct under **CBV parallel reduction** $\rightsquigarrow_{\mathsf{p}}$. This relation is similar to $\rightsquigarrow_{\mathsf{cbv}}$, except that it permits evaluation under binders and subexpressions can be evaluated in parallel. The rules for λ-abstractions and applications are shown in figure 3 (the remaining rules are in the appendix, section A.4). In particular, the typechecker can only carry out a β-reduction of an application or case expression if the argument or scrutinee is a value. Note, however, that values include variables. Treating variables as values is safe due to the CBV semantics, and it is crucial when reasoning about open terms. For example, to typecheck the usual append function we want $\mathsf{Vec\ Nat}\ (0+x)$ and $\mathsf{Vec\ Nat}\ x$ to be equal types.

$$\frac{}{m \rightsquigarrow_{\mathsf{p}} m}\text{SP\_REFL} \qquad \frac{m \rightsquigarrow_{\mathsf{p}} m'}{\lambda x.m \rightsquigarrow_{\mathsf{p}} \lambda x.m'}\text{SP\_ABS} \qquad \frac{\begin{array}{l} m \rightsquigarrow_{\mathsf{p}} m' \\ n \rightsquigarrow_{\mathsf{p}} n' \end{array}}{m\, n \rightsquigarrow_{\mathsf{p}} m'\, n'}\text{SP\_APP}$$

$$\frac{\begin{array}{l} m \rightsquigarrow_{\mathsf{p}} m' \\ u \rightsquigarrow_{\mathsf{p}} u' \end{array}}{(\lambda x.m)\; u \rightsquigarrow_{\mathsf{p}} [u'/x]m'}\text{SP\_APPBETA} \qquad \frac{\begin{array}{l} u_1 \rightsquigarrow_{\mathsf{p}} u_1' \\ u_2 \rightsquigarrow_{\mathsf{p}} u_2' \end{array}}{(\mathsf{rec}\, f.u_1)\; u_2 \rightsquigarrow_{\mathsf{p}} ([\mathsf{rec}\, f.u_1'/f]u_1')\; u_2'}\text{SP\_APPREC}$$

Figure 3: Parallel reduction $\rightsquigarrow_{\mathsf{p}}$ (Excerpt).

The possibility that expressions may have effects restricts the application rule of a dependent type system. The typical rule for typing applications in pure languages is

$$\frac{\begin{array}{l} \Gamma \vdash a : (x{:}A) \to B \\ \Gamma \vdash b : A \end{array}}{\Gamma \vdash a\, b : [b/x]B}$$

However, this rule does not work if b may have effects, because then the type $[b/x]B$ may not be well-formed. Although we know by regularity that $(x{:}A) \to B$ is well-formed, the derivation of $\Gamma \vdash (x{:}A) \to B : \star$ may involve reductions, and substituting a non-value b for x may block a β-reduction that used to have x as an argument. Intuitively this makes sense: under CBV-semantics, a is really called on the *value* of b, so the type B should be able to assume that x is an (effect-free) value. Our fix is to add a premise that the result type is well-formed. This additional premise is exactly what is required to prove type safety.

$$\frac{\begin{array}{l} \Gamma \vdash a : (x{:}A) \to B \\ \Gamma \vdash a : A \\ \Gamma \vdash [a/x]B : \star \end{array}}{\Gamma \vdash a\, b : [a/x]B}\text{T\_APP}$$

This rule is simple, yet expressive. Previous work [28, 12, 26] uses a more restrictive typing for applications, splitting it into two rules: one which permits only *value dependency*, and requires the argument to be a value, and one which allows an application to an arbitrary argument when there is no dependency.

Lemma 1. *The following rules are admissible.*

$$\frac{\begin{array}{l} \Gamma \vdash a : (x{:}A) \to B \\ \Gamma \vdash v : A \end{array}}{\Gamma \vdash a\, v : [v/x]B}\text{APP\_VAL} \qquad \frac{\begin{array}{l} \Gamma \vdash a : A \to B \\ \Gamma \vdash b : A \end{array}}{\Gamma \vdash a\, b : B}\text{APP\_NONDEP}$$

Both of these rules are special cases of our rule above.

2.2 Equality and irrelevant type conversions

One crucial point in the design of a dependently typed language is the elimination form for propositional equality, *conversion*. Given an expression $\Gamma \vdash a : A$ and a proof $\Gamma \vdash b : (A = A')$, we should be able to convert the type of a to A'. We write this operation as $\mathsf{conv}\, a\, \mathsf{by}\, b$.

$$\begin{array}{lll}
|\star| = \star & |x| = x & |\mathsf{rec}\, f : A.v| = \mathsf{rec}\, f.|v| \\
|\mathsf{abort}_A| = \mathsf{abort} & & \\
|(x{:}A) \to B| = (x{:}|A|) \to |B| & |\lambda x : A.a| = \lambda x.|a| & |a\, b| = |a|\ |b| \\
|[x{:}A] \to B| = [x{:}|A|] \to |B| & |\lambda [x : A].a| = \lambda [].|a| & |a\ [b]| = |a|[] \\
|D\, \overline{A_i}| = D\, |\overline{A_i}| & |d\ [\overline{A_i}]\ \overline{a_i}| = d\, |\overline{a_i}| & \\
|\mathsf{case}\, a\, \mathsf{as}\, [y]\, \mathsf{of}\, \{\, \overline{d_j\, \Delta_j \Rightarrow b_j}^{j \in 1..k}\, \}| & \multicolumn{2}{l}{= \mathsf{case}\, |a|\, \mathsf{of}\, \{\, \overline{d_j\, \overline{x_{ij}} \Rightarrow |b_j|}^{j \in 1..k}\, \}} \\
 & \multicolumn{2}{l}{\quad \text{where } \overline{x_{ij}} \text{ are the relevant variables of } \Delta_j} \\
|a = b| = |a| = |b| & |\mathsf{join}_{a=b}\, i\, j| = \mathsf{join} & |\mathsf{conv}\, a\, \mathsf{at}\, [\sim P_1/x_1] \ldots [\sim P_i/x_i] A| = |a| \\
 & & \\
|\cdot| = \cdot & |a\, \overline{a_i}| = |a|\ |\overline{a_i}| & |[a]\ \overline{a_i}| = []\ |\overline{a_i}|
\end{array}$$

Figure 4: The erasure function $|\cdot|$

In most languages, the proof b in such a conversion affects the operational semantics of the expression; we say that it is *computationally relevant*. For example, in Coq the operational behavior of `conv` is to first evaluate b until it reaches $\mathsf{refl\_eq}$, the only constructor of the equality type, and then step by $\mathsf{conv}\, a\, \mathsf{by}\, \mathsf{refl\_eq} \rightsquigarrow a$.

However, relevance can get in the way of reasoning about programs. Equations involving `conv` such as $(\mathsf{conv}\, a\, \mathsf{by}\, b) = a$ are not easily provable in Coq unless b is $\mathsf{refl\_eq}$. Indeed, because Coq's built-in equality is homogeneous, such equalities are often difficult even to state. This issue can be a practical problem when reasoning about programs operating on indexed data. One workaround is to assert additional axioms about equality and conversion, such as Streicher's Axiom K [23]. The situation is frustrating because the computationally relevant behavior of conversion does not actually correspond to the compiled code. Coq's extraction mechanism will erase b and turn $\mathsf{conv}\, a\, \mathsf{by}\, b$ into just a. But the Coq typechecker does not know about extraction.

Our language integrates extraction into the type-system, similarly to ICC* [7]. Specifically, we define an **erasure function** $|\cdot|$ which takes an annotated expression a and produces an unannotated expression $m \equiv |a|$. The definition of $|\cdot|$ is given in Figure 4. In most cases it just traverses a, but it erases type annotations from abstractions, it deletes irrelevant arguments (see section 2.3), and it completely deletes conversions leaving just the subject of the cast.

The unannotated system is used to determine when expressions are equal.

$$\frac{|a| \rightsquigarrow_{\mathsf{p}}^{*} n \quad |b| \rightsquigarrow_{\mathsf{p}}^{*} n \qquad \Gamma \vdash a = b : \star}{\Gamma \vdash \mathsf{join} : a = b}\ \text{JOIN\_NO\_ANNOT}$$

The rule says that the term join is a proof of an equality $a = b$ if the *erasures* of the expressions a and b parallel-reduce to a common reduct. Therefore, when reasoning about a program we can completely ignore the parts of it that will not remain at runtime. (The rule presented above is somewhat simplified from our actual system—it is type safe, but as we discuss in section 2.5 it needs additional annotations to make type checking algorithmic.)

Erasing conversions requires a corresponding restriction on the `conv` typing rule. As we noted before, conversion must evaluate equality proofs strictly in order to not be fooled by infinite loops, but if the proofs are erased there is nothing left at runtime to evaluate. The fix is to restrict the proof term to be a syntactic value:

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash v : A = B \quad \Gamma \vdash B : \star}{\Gamma \vdash \mathsf{conv}\, a\, \mathsf{by}\, v : B}\ \text{VCONV}$$

(We will discuss the third premise $\Gamma \vdash B : \star$ in section 2.4). In the case where the proof is a variable (for instance, the equalities that come out of a dependent pattern match), the value restriction is already satisfied. Otherwise (for example, when the proof is the application of a lemma) we can satisfy the requirement by rewriting $\mathsf{conv}\, a\, \mathsf{by}\, b$ to $\mathsf{let}\, x = b\, \mathsf{in}\, \mathsf{conv}\, a\, \mathsf{by}\, x$, making explicit the sequencing that Coq integrates into the evaluation rule.

Most languages make conversion computationally relevant in order to ensure strong normalization for open terms. If conversion is irrelevant, then in a context containing the assumption $\mathsf{Nat} = (\mathsf{Nat} \to \mathsf{Nat})$ it is possible to type the looping term $(\lambda x.x\, x)\ (\lambda x.x\, x)$ since evaluation does not get stuck on the assumption. Of course, in our language expressions are not normalizing in the first place.

Making conversions completely erased blurs the usual distinction between *definitional* and *propositional* equality. Typically, definitional equality is a decidable comparison which is automatically applied everywhere, while propositional equality uses arbitrary proofs but has to be marked with a explicit elimination form.

There are two main reasons languages use a distinguished definitional equality in addition to the propositional one, but neither of them applies to our language. First, if there exists a straightforward algorithm for testing definitional equality (e.g., just reduce both sides to normal form, as in PTSs [5]), then it is convenient for the programmer to have it applied automatically. However, our language has non-terminating expressions, and we don't want the type checker to loop trying to normalize them.

Second, languages where the use of propositional equalities is computationally relevant and marked need automatic conversion for a technical reason in the preservation proof. As an application steps, $m\, n \rightsquigarrow_{\mathsf{cbv}} m\, n'$, its type changes from $[n/x]N$ to $[n'/x]N$ and has to be converted back to $[n/x]N$. Because the operational semantics does not introduce any explicit conversion into the term, this conversion needs to be automatic. However, in our language uses of propositional equations are never marked, so we can use the propositional equality at this point in the proof.

2.3 Irrelevant arguments, and reasoning about indexed data

Above, we discussed how conversions get erased. Our language also includes a more general feature where arguments to functions and data constructors can be marked as irrelevant so that they are erased as well.

To motivate this feature, we consider *vectors* (i.e. length-indexed lists). Suppose we have defined the usual vector data type and append function, with types

```
Vec (a:*) :  Nat → * where
  nil  : Vec a 0
  cons : (n:Nat) → Vec a n → Vec a (S n)

app : (n1 n2 : Nat) → (a : *) → Vec a n1 → Vec a n2 → Vec a (n1+n2)
app n1 n2 a xs ys =
  case xs of
    nil → ys
    (cons n x xs) → cons a (n+n2) x (app n n2 a xs ys)
```

Having defined this operation, we might wish to prove that the append operation is associative. This amounts to defining a recursive function of type

```
app-assoc : (n1 n2 n3:Nat) →
            (v1 : Vec a n1) → (v2 : Vec a n2) → (v3 : Vec a n3) →
            (app a n1 (n2+n3) v1 (app a n2 n3 v2 v3))
              =  (app a (n1+n2) n3 (app a n1 n2 v1 v2) v3)
```

If we proceed by pattern-matching on `v1`, then when `v1 =cons n v` we have to show, after reducing the RHS, that

```
   (cons a (n + (n2 + n3)) x (app a    n     (n2 + n3)        v         (app a n2 n3 v2 v3)))
 = (cons a ((n + n2) + n3) x (app a (n + n2)    n3     (app a n n2 v v2)          v3))
```

By a recursive call/induction hypothesis, we have that the tails of the vectors are equal, so we are *almost* done. . . except we also need to show

```
n + (n2 + n3) = (n + n2) + n3
```

which requires a separate lemma about associativity of addition. In other words, when reasoning about indexed data, we are also forced to reason about their indices. In this case it is particularly frustrating because these indices are completely determined by the shape of the data—a Sufficiently Smart Compiler would not even need to keep them around at runtime [8]. Unfortunately, nobody told our typechecker that.

The solution is to make the length argument to cons an **irrelevant argument**. We change the definition of `Vec` to syntactically indicate that `n` is irrelevant by surrounding it with square brackets.

```
data Vec' (a:*) : Nat → * where
  nil' : Vec' a 0
  cons' : [n:Nat] → a → Vec' a n → Vec' a (S n)
```

Irrelevant constructor arguments are not represented in memory at run-time, and equations between irrelevant arguments are trivially true since our T_JOIN rule is stated using erasure.

The basic building block of irrelevance is irrelevant function types $[x{:}M] \to N$, which are inhabited by irrelevant λ-abstractions $\lambda[x:A].b$ and eliminated by irrelevant applications $a\ [b]$. The introduction rule for irrelevant λs is similar to the rule for normal λs, with one restriction:

$$\frac{\Gamma, x : A \vdash b : B \qquad x \notin \mathsf{FV}(|b|)}{\Gamma \vdash \lambda[x:A].b : [x{:}A] \to B}\ \text{T\_IABS}$$

The free variable condition ensures that the argument x is not used at runtime, since it does not remain in the erasure of the body b. So x can only appear in irrelevant positions in b, such as type annotations and proofs for conversions. On the other hand, x is available at type-checking time, so it can occur freely in the type B.

Since the bound variable is not used at runtime, we can erase it, leaving only a placeholder for the abstraction or application: $|\lambda[x:A].a|$ goes to $\lambda[].|a|$ and $|a\ [b]|$ goes to $|a|[]$. As a result, the term b is not present in memory and does not get in the way of equational reasoning.

The reason we leave placeholders is to ensure that syntactic values get erased to syntactic values. Since we make conversion irrelevant this invariant is needed for type-safety [22]. For example, using a hypothetical equality we can type the term

$$\lambda[p : \mathsf{Bool} = \mathsf{Nat}].1 + \mathsf{conv}\,\mathsf{true}\,\mathsf{by}\,p : [p{:}\mathsf{Bool} = \mathsf{Nat}] \to \mathsf{Nat}.$$

In our language this term erases to the value $\lambda[].1+\mathsf{true}$. On the other hand, if it erased to the stuck expression $1+\mathsf{true}$ then progress would fail.

Value restriction The treatment of erasure as discussed so far is closely inspired by ICC* [7] and EPTS [17], while a related system is described by Abel [1] and implemented in recent versions of Agda.

However, the presence of nontermination adds a twist because normal and irrelevant arguments have different evaluation behavior. In a CBV language, normal arguments are evaluated to values, but irrelevant arguments just get erased. So similarly to erased conversions we need to be careful—while we argued earlier that $(\lambda x : (\mathsf{Bool} = \mathsf{Nat}).a)$ (loop) will not lead to type error thanks to our CBV semantics, the same reasoning clearly does not work for $(\lambda[x : \mathsf{Bool} = \mathsf{Nat}].a)$ $[\mathsf{loop}]$. To maintain the invariant that variables always stand for values, we restrict the irrelevant application rule to only allow values in the argument position:

$$\frac{\begin{array}{l}\Gamma \vdash a : [x{:}A] \to B \\ \Gamma \vdash v : A\end{array}}{\Gamma \vdash a\ [v] : [v/x]B}\ \text{T\_IAPP}$$

Restricting arguments to values here limits the practical use of irrelevant arguments. For example, even if we make the length argument to `cons'` irrelevant, we cannot make the length arguments to `app` irrelevant. The problem is that in the recursive case we would want to return

```
cons' a [n+n2] x' (app a [n] xs' [n2] ys)
```

but `n+n2` is not a value. To make the function typecheck we must work around the restriction by computing the value of the sum at runtime.

However, to ensure type safety we believe it is enough to ensure that erased expressions have normal forms. In this paper we use a syntactic value check as the very simplest example of a termination analysis. For a full language, we would mark certain expressions as belonging to a terminating sublanguage (with, perhaps, the full power of Type Theory available for termination proofs). It is not hard to conclude that addition terminates.

This restriction is necessary because allowing nonterminating expressions to be erased would break type safety for our language. The problem is not only infinite loops directly inhabiting bogus equalities like $\mathsf{Bool} = \mathsf{Nat}$ (as above). The following counter-example shows that we can get in trouble even by erasing an abort of type Nat. First, note that since the reduction relation treats variables as values, $(\lambda x.\mathsf{Bool})\ x \rightsquigarrow_{\mathsf{p}} \mathsf{Bool}$. So we have $\mathsf{join} : ((\lambda x : \mathsf{Nat}.\mathsf{Bool})\ x = (\lambda x : \mathsf{Nat}.\mathsf{Nat})\ x) = (\mathsf{Bool} = \mathsf{Nat})$. Then the following term typechecks:

$$\begin{array}{l}\lambda[x : \mathsf{Nat}].\lambda p : ((\lambda x : \mathsf{Nat}.\mathsf{Bool})\ x) = ((\lambda x : \mathsf{Nat}.\mathsf{Nat})\ x).\mathsf{conv}\, p\, \mathsf{by}\, \mathsf{join} \\ \qquad : [x{:}\mathsf{Nat}] \to (p{:}(\lambda x : \mathsf{Nat}.\mathsf{Bool})\ x = (\lambda x : \mathsf{Nat}.\mathsf{Nat})\ x) \to (\mathsf{Bool} = \mathsf{Nat})\end{array}$$

On the other hand, by our reduction rule for error terms, $(\lambda x.\mathsf{Bool})\ \mathsf{abort} \rightsquigarrow_{\mathsf{p}} \mathsf{abort}$, so

$$\mathsf{join} : ((\lambda x : \mathsf{Nat}.\mathsf{Bool})\ \mathsf{abort}_{\mathsf{Nat}}) = ((\lambda x : \mathsf{Nat}.\mathsf{Nat})\ \mathsf{abort}_{\mathsf{Nat}}).$$

So if we allowed $\mathsf{abort}_{\mathsf{Nat}}$ to be given as an irrelevant argument, then we could write a terminating proof of $\mathsf{Bool} = \mathsf{Nat}$. Note that all the equality proofs involved are just join, so this example does not depend on conversions being computationally irrelevant. This illustrates a general issue when combining effects and irrelevance.

Datatypes In addition to irrelevant λ-abstractions we also allow irrelevant arguments in data types, like `Vec'`. Datatype declarations have the form

$$\mathsf{data}\, D\,\Delta\, \mathsf{where}\,\{\,\overline{d_i : \Delta_i \to\ D\,\Delta}^{i\in 1..j}\,\}$$

The rules for datatypes in dependently-typed languages often look intimidating. We tried to make ours as simple as we could. First, to reduce clutter we write the rules using **telescope notation**. Metavariables Δ range over lists of relevance-annotated variable declarations like $(x : A)[y : B](z : C)$, also known as telescopes, while overlined metavariables $\overline{a_i}$ range over lists of terms. Depending on where in an expression they occur, telescope metavariables stand in for either declarations or lists of variables, according to the following scheme: if Δ is $(x : A)[y : B](z : C)$ and $\overline{a_i}$ is $a,[b],c$, then...

...this:	is shorthand for this:
$a_1\,\Delta$	$a_1\;x\;[y]\;z$
$\Delta \to A_1$	$(x{:}A) \to [y{:}B] \to (z{:}C) \to A_1$
$[\overline{a_i}/\Delta]a_1$	$[a/x][b/y][c/z]a_1$
Γ,Δ	$\Gamma, x : A, y : B, z : C$
$\Gamma \vdash \overline{a_i} : \Delta$	$\Gamma \vdash a : A \wedge \Gamma \vdash b : [a/x]B \wedge \Gamma \vdash c : [b/y][a/x]C$

We also simplify the rules by having **parameters but not indices**. Each datatype has a list of parameters Δ, and these are instantiated uniformly (i.e. the type of each data constructor d_i ends with $D\,\Delta$, the type constructor D applied to a list of variables). This restriction does not limit expressivity, because we can elaborate non-uniform indexes into a combination of parameters and equality proofs (this is how Haskell GADTs are elaborated into GHC Core) [25, 14]. For example, the declaration of `Vec'` above can be reformulated without indices as

```
data Vec' (a:*) (n:Nat) where
  nil'  : [p:n=0] → Vec' a n
  cons' : [m:Nat] → [p:n=S m] → a → Vec' a m → Vec' a n
```

To make the statement of the canonical forms lemma simpler (see lemma 11 below) we require constructors to be fully applied, so they do not pollute the function space. In other words, d by itself is not a well-formed expression, it must be applied to a list of parameters and a list of arguments $d\;[\overline{A_i}]\;\overline{a_i}$.

In the corresponding elimination form (the case expression $\mathsf{case}\, b\, \mathsf{as}\, [y]\, \mathsf{of}\, \{\,\overline{d_i\,\Delta_i \Rightarrow a_i}^{i\in 1..l}\,\}$) the programmer must write one branch $d_i\,\Delta_i \Rightarrow a_i$ for each constructor of the datatype D. The branch only introduces pattern variables for the constructor arguments, as the parameters are fixed throughout the case. However, the parameters are used to refine the context that the match is checked in: if $\Gamma \vdash b : D\,\overline{B_i}$, then for each case we check

$$\Gamma, [\overline{B_i}/\Delta]\Delta_i, y : b = d_i\,\Delta_i \vdash a_i : A$$

The context also introduces an equality proof y which can by used (in irrelevant positions) to exploit the new information about which constructor matched.

So far, this is a fairly standard treatment of datatypes. However, we want to point out how **irrelevant parameters and constructor arguments** work.

First, parameters to data constructors are always irrelevant, since they are completely fixed by the types. The erasure operation simply deletes them: $|d\;[\overline{A_i}]\;\overline{a_i}|$ is $d\,|\overline{a_i}|$. On the other hand, it never makes sense for parameters to data*type* constructors to be irrelevant. For example, if the parameters to `Vec` were made irrelevant, the join rule would validate $\Gamma \vdash \mathsf{join} : (\mathsf{Vec}\;[\mathsf{Nat}]\;[1]) = (\mathsf{Vec}\;[\mathsf{Bool}]\;[2])$, which would defy the point of having the parameters in the first place. This is reflected in our typing rule for datatype constructors,

$$\frac{\begin{array}{l}\mathsf{data}\, D\, \Delta\, \mathsf{where}\, \{\, \overline{d_i : \Delta_i \to\ D\, \Delta}^{i \in 1..j}\, \} \in \Gamma \\ \Gamma \vdash \overline{A_i} : \Delta^+\end{array}}{\Gamma \vdash D\, \overline{A_i} : \star}\ \text{T\_TCON}$$

where we look up the telescope Δ from the datatype declaration but change all irrelevant variables to relevant ones (an operation we write as Δ^+).

Finally, arguments to data constructors d_i can be marked as relevant or not in the telescope Δ_i, and this is automatically reflected in the typing rule for constructor application and erasure. For example, given the above declaration of `Vec'`, the annotated expression

```
cons' [Bool] [1] [0] [join] true (nil' [Bool] [0] [join])
```

is well-typed and erases to `cons' [] [] true (nil' [])`. However, making a constructor argument erased carries a corresponding restriction in the case statement: since the argument has no run-time representation it may only be used in irrelevant positions. For example, in a case branch

```
  cons' [m:Nat] [p:n=S m] (x:a) (xs:Vec' a m) ⇒ ...body...
```

the code in the body can use `x` without restrictions but can only use `m` in irrelevant positions such as type annotations and conversions. With the original `Vec` type we could write a constant-time length function by projecting out `m`, but that is not possible with `Vec'`.

2.4 Very heterogenous equality and *n*-ary conversion

The `app-assoc` example also illustrates a different problem with indexed data: the two sides of the equation have different types (namely `Vec (n1+(n2+n3))` versus `Vec ((n1+n2)+n3)`)—so, e.g., the usual equality in Coq does not even allow writing down the equation! We need some form of **heterogenous equality**. The most popular choice for this is JMeq [15], which allows you to *state* any equality, but only *use* them if both sides have the same type. Massaging goals into a form where the equalities are usable often requires certain tricks and idioms (see e.g. [10], chapter 10).

For this language, we wanted something simpler. Like JMeq, we allow stating any equation as long as the two sides are well-typed. Our formation rule for the equality type is

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash a = b : \star}\ \text{T\_EQ}$$

Unlike JMeq, however, conversion can use an equality even if the two sides have different types. This is similar to the way equality is handled in Guru [24], although the details differ.

We showed a simplified version of our conversion rule on page 9. The full rule has a few more parts. First, in order to be able to change only parts of a type, we phrase the rule in terms of substituting into a template A.

$$\frac{\begin{array}{l}\Gamma \vdash a : [B_1/x]A \quad \Gamma \vdash v : B_1 = B_2 \\ \Gamma \vdash [B_2/x]A : \star\end{array}}{\Gamma \vdash \mathsf{conv}\, a\, \mathsf{by}\, v : [B_2/x]A}\ \text{CONV\_SUBST}$$

For example, given a proof of $y = 0$ we can convert the type $\mathsf{Vec\ Nat}\ (y+y)$ to $\mathsf{Vec\ Nat}\ (y+0)$ using the template $\mathsf{Vec\ Nat}\ (y+x)$.

We need the premise $\Gamma \vdash [B_2/x]A : \star$ for two reasons. First, since our equality is heterogenous, we do not know that B_2 is a type even if B_1 is. It is perfectly possible to write a function that takes a proof of $\mathsf{Nat} = 3$ as an argument (although it will never be possible to actually call it). But even if equality were homogenous we would still need the wellformedness premise for the same reason we need it in the application rule. If B_1 is a value and B_2 is not, then $[B_2/x]A$ is not guaranteed to be well-formed.

Next, to achieve the full potential of our flexible elimination rule we find it is not sufficient to eliminate one equality at a time. For a simple example, consider trying to prove $f\ x = f'\ x'$ in the context

$$f : A \to B, f' : A' \to B, x : A, x' : A', p : f = f', q : x = x'$$

Note that there is no equation relating A and A'. Using one equality at a time, the only way to make progress is by transitivity, that is by trying to prove $f\ x = f\ x'$ and $f\ x' = f'\ x'$. However, the intermediate expression $f\ x'$ is not well-typed so the attempt fails. To make propositional equality a congruence with respect to application, we are led to a conversion rule that allows eliminating several equations at once.

$$\frac{\begin{array}{l}\Gamma \vdash v_1 : A_1 = B_1 \quad \ldots \quad \Gamma \vdash v_i : A_i = B_i \\ \Gamma \vdash a : [A_1/x_1] \ldots [A_i/x_i]A \\ \Gamma \vdash [B_1/x_1] \ldots [B_i/x_i]A : \star\end{array}}{\Gamma \vdash \mathsf{conv}\, a\, \mathsf{by}\, v_1 \ldots v_i : \star}\ \textsc{conv\_multisubst}$$

Of course, the above example is artificial: we don't really expect that a programmer would often want to prove equations between terms of unrelated types. A more practical motivation comes from proofs about indexed data like vectors, where A might be `Vec (n+(n2+n3))` and A' be `Vec ((n+n2)+n3)`. In such an example, A and A' are indeed provably equal, but our n-ary conversion rule obviates the need to prove that proof.

The fact that our conversion can use heterogenous equations also has a downside: we are unable to support certain type-directed equality rules. In particular, adding *functional extensionality* would be unsound. Extensionality implies $(\lambda x : (1 = 0).1) = (\lambda x : (1 = 0).0)$ since the two functions agree on all arguments (vacuously). But our annotation-ignoring equality shows $(\lambda x : (1 = 0).1) = (\lambda x : \mathsf{Nat}.1)$, so by transitivity we would get $(\lambda x : \mathsf{Nat}.1) = (\lambda x : \mathsf{Nat}.0)$ (and from there, to $1 = 0$).

2.5 Annotating equality and conversion

Ultimately, the unannotated language is the most interesting artifact, since that is what actually gets executed. The point of defining an annotated language is to make it convenient to write down typings of unannotated terms. (We could consider the annotated terms as reified typing derivations). We designed the annotated language by starting with the unannotated language and adding just enough annotations that a typechecker traversing an annotated term will always know what to do. For most language constructs this was straightforward, e.g. adding a type annotation to λ-abstractions. The annotated programs get quite verbose, so for a full language more sophisticated methods like bidirectional type checking, local type inference, or unification-based inference would be helpful, but these techniques are beyond the scope of this paper.

However, it is worth discussing how nontermination and irrelevance affect the T_CONV and T_JOIN rules. The conv rule in the erased language, including n-ary substitution, looks like

$$\frac{\begin{array}{l}H \vdash u_1 : M_1 = N_1 \quad \ldots \quad H \vdash u_i : M_i = N_i \\ H \vdash m : [M_1/x_1] \ldots [M_i/x_i]M \\ H \vdash [N_1/x_1] \ldots [N_i/x_i]M : \star\end{array}}{H \vdash m : [N_1/x_1] \ldots [N_i/x_i]M}\ \textsc{et\_conv}$$

To guide the typechecker, in addition to the annotated version of m we need to supply the (annotated versions of) the proof values u_i and the (annotated version of) the "template" type M that we are substituting into. A first attempt at a corresponding annotated rule would look like the CONV_MULTISUBST rule we showed above.

However, that rule needs one modification. In order to correspond exactly to the unannotated conv rule it should ignore expressions in irrelevant positions. For example, consider proving the equation $f\ [A]\ a = f\ [B]\ b$, which erases to $f[]\ |a| = f[]\ |b|$. The unannotated conv rule only requires a proof of $|a| = |b|$, so in the annotated language we should not have to provide a proof involving A and B. Therefore, in the annotated rule we allow two kinds of evidence P: either a value v which is a proof of an equation, or just an annotation $[a = b]$ stating how an irrelevant subexpression should be changed. We also need to specify the template that the substitution is applied to. As a matter of concrete syntax, we prefer writing the evidence P_i interleaved with the template, marking it with a tilde. So our final annotated rule looks like this:

$$P \quad ::= \quad v \mid [a = b]$$

$$\frac{\begin{array}{l}\forall i.\ \ ((P_i \text{ is } v_i \ \text{ and } \Gamma \vdash v_i : A_i = B_i) \text{ or } (P_i \text{ is } [A_i = B_i] \ \text{ and } x_i \notin \mathsf{FV}(|A|)\ ))\\ \Gamma \vdash a : [A_1/x_1]\ldots[A_i/x_i]A\\ \Gamma \vdash [B_1/x_1]\ldots[B_i/x_i]A : \star\end{array}}{\Gamma \vdash \mathsf{conv}\, a\, \mathsf{at}\, [\sim P_1/x_1]\ldots[\sim P_i/x_i]A : [B_1/x_1]\ldots[B_i/x_i]A}\text{T\_CONV}$$

For example, if `a : Vec A x` and `y : x =3`, then `conv a at Vec A ~y` has type `Vec A 3`.

Next, consider the equality introduction rule. In the unannotated language it is simply

$$\frac{\begin{array}{l}m \curlyvee n\\ H \vdash m = n : \star\end{array}}{H \vdash \mathsf{join} : m = n}\text{ET\_JOIN}$$

This is very similar to what other dependent languages, such as PTSs, offer. In those languages, this rule may be implemented by evaluating both sides to normal forms and comparing. Unfortunately, in the presence of nontermination there is no similarly simple algorithm—the parallel reduction relation is nondeterministic, and since we are not guaranteed to hit a normal form we would have to search through all possible evaluation orders.

One possibility would be to write down the expression to be reduced, and tag sub-expressions of it with how many steps to take, perhaps marked with tildes. In our experiments with a prototype typechecker for our language, we have adopted a simpler scheme. The join rule only does deterministic CBV evaluation for at most a specified number of steps. So our annotated type rule looks like

$$\frac{\begin{array}{l}|a| \rightsquigarrow^{i}_{\mathsf{cbv}} n \quad |b| \rightsquigarrow^{j}_{\mathsf{cbv}} n\\ \Gamma \vdash a = b : \star\end{array}}{\Gamma \vdash \mathsf{join}_{a=b}\, i\, j : a = b}\text{T\_JOIN}$$

where i and j are integer literals. In the common case when both a and b quickly reach normal forms, the programmer can simply pick high numbers for the step counts, and in the concrete syntax we treat `join` as an abbreviation for `join 100 100`. When we want to prove equations between terms that are already values, we can use `conv` to change subterms of them. For example, to prove the equality $\mathsf{Vec}\, A\ (1+1) = \mathsf{Vec}\, A\ 2$ we write

```
conv (join : Vec A 2 = Vec A 2) at (Vec A 2 = Vec A ~(join : 1+1 = 2))
```

Not every parallel reduction step can be reached this way, since substitution is capture-avoiding. For instance, we cannot show an equation like $(\lambda x.(\lambda y.y)\ x) = (\lambda x.x)$. In practice, we have not found this restriction limiting.

3 Metatheory

The main technical contribution of this paper is a proof of type safety for our language via standard preservation and progress theorems. The full proof can be found in the appendix. In this section, we highlight the most interesting parts of it.

3.1 Annotated and unannotated type systems

While the description of the language so far has been in terms of a type system for annotated terms, we have also developed a type system for the unannotated language, and it is the unannotated system that is important for the metatheoretical development.

The unannotated typing judgment is of the form $H \vdash m : M$, where the metavariable H ranges over unannotated typing environments (i.e., environments of assumptions $x : M$). Below we give an outline of the rules. The complete definition can be found in the appendix (section A.6). The unannotated type system is designed to satisfy the following property, stating that erasure preserves well-typedness:

Lemma 2 (Annotation soundness)**.** *If* $\Gamma \vdash a : A$ *then* $|\Gamma| \vdash |a| : |A|$.

In practice, the unannotated rules simply mirror the annotated rules, except that all the terms in them have been erased. As an example, compare the annotated and unannotated versions of the IABS rule:

$$\frac{\Gamma, x : A \vdash b : B \qquad x \notin \mathsf{FV}(|b|)}{\Gamma \vdash \lambda[x : A].b : [x{:}A] \to B}\ \textsc{T\_IABS} \qquad \frac{H, x : M \vdash n : N \qquad x \notin \mathsf{FV}(n)}{H \vdash \lambda[].n : [x{:}M] \to N}\ \textsc{ET\_IABS}$$

Since our operational semantics is defined for unannotated terms, the preservation and progress theorems will be also stated in terms of unannotated terms.

3.2 Properties of parallel reduction

The key intuition in our treatment of equality is that, in an empty context, propositional equality coincides with joinability under parallel reduction. As a result, we will need some basic lemmas about parallel reduction throughout the proof. These are familiar from, e.g., the metatheory of PTSs, with the slight difference that the usual substitution lemma is replaced with two special cases because we work with CBV reduction.

Lemma 3 (Substitution of $\rightsquigarrow_{\mathsf{p}}$)**.** *If* $N \rightsquigarrow_{\mathsf{p}} N'$*, then* $[N/x]M \rightsquigarrow_{\mathsf{p}} [N'/x]M$.

Lemma 4 (Substitution into $\rightsquigarrow_{\mathsf{p}}$)**.** *If* $u \rightsquigarrow_{\mathsf{p}} u'$ *and* $m \rightsquigarrow_{\mathsf{p}} m'$*, then* $[u/x]m \rightsquigarrow_{\mathsf{p}} [u'/x]m'$.

Lemma 5 (Confluence of $\rightsquigarrow_{\mathsf{p}}$)**.** *If* $m \rightsquigarrow_{\mathsf{p}}^* m_1$ *and* $m \rightsquigarrow_{\mathsf{p}}^* m_2$*, then there exists some* m' *such that* $m_1 \rightsquigarrow_{\mathsf{p}}^* m'$ *and* $m_2 \rightsquigarrow_{\mathsf{p}}^* m'$.

Definition 6 (Joinability)**.** *We write* $m_1 \curlyvee m_2$ *if there exists some* n *such that* $m_1 \rightsquigarrow_{\mathsf{p}}^* n$ *and* $m_1 \rightsquigarrow_{\mathsf{p}}^* n$.

Using the above lemmas it is easy to see that $\curlyvee$ is an equivalence relation, and that $m_1 \curlyvee m_2$ implies $[m_1/x]M \curlyvee [m_2/x]M$.

3.3 Preservation

For the preservation proof we need the usual structural properties: weakening and substitution. Weakening is standard, but somewhat unusually substitution is restricted to substituting values u into the judgment, not arbitrary terms. This is because our equality is CBV, so substituting a non-value could block reductions and cause types to no longer be equal.

Lemma 7 (Substitution). *If $H_1, x_1 : M_1, H_2 \vdash m : M$ and $H_1 \vdash u_1 : M_1$, then $H_1, [u_1/x_1]H_2 \vdash [u_1/x_1]m : [u_1/x_1]M$.*

Preservation also needs an inversion lemmas for λs, irrelevant λs, rec, and data constructors. They are similar, and we show the one for λ-abstractions as an example.

Lemma 8 (Inversion for λs). *If $H \vdash \lambda x.n : M$, then there exists m_1, M_1, N_1, such that $H \vdash \mathsf{join} : M = (x{:}M_1) \to N_1$ and $H, x : M_1 \vdash n : N_1$.*

Notice that this is weaker statement than in a language with computationally relevant conversion. For example, in a PTS we would have that M is β-convertible to the type $(x{:}M_1) \to N_1$, not just provably equal to it. But in our language, if the context contained the equality $(\mathsf{Nat} \to \mathsf{Nat}) = \mathsf{Nat}$, then we could show $H \vdash \lambda x.x : \mathsf{Nat}$ using a (completely erased) conversion. As we will see, we need to add extra injectivity rules to the type system to compensate.

Now we are ready to show the preservation theorem. For type safety we are only interested in preservation for $\rightsquigarrow_{\mathsf{cbv}}$, but it is convenient to generalize the theorem to $\rightsquigarrow_{\mathsf{p}}$.

Theorem 9 (Preservation).
If $H \vdash m : M$ and $m \rightsquigarrow_{\mathsf{p}} m'$, then $H \vdash m' : M$.

The proof is mostly straightforward by an induction on the typing derivation. There are some wrinkles, all of which can be seen by considering some cases for applications. The typing rule looks like

$$\frac{\begin{array}{l} H \vdash m : (x{:}M) \to N \\ H \vdash n : M \\ H \vdash [n/x]N : \star \end{array}}{H \vdash m\, n : [n/x]N}\ \text{ET\_APP}$$

First consider the case when $m\ n$ steps by congruence, $m\ n \rightsquigarrow_{\mathsf{p}} m\ n'$. Directly by IH we get that $H \vdash n' : M$, but because of our CBV-style application rule we also need to establish $H \vdash [n'/x]N : \star$. But by substitution of $\rightsquigarrow_{\mathsf{p}}$ we know that $[n/x]N \rightsquigarrow_{\mathsf{p}} [n'/x]N$, so this also follows by IH (this is why we generalize the theorem to $\rightsquigarrow_{\mathsf{p}}$).

This showed $H \vdash m\, n' : [n'/x]N$, but we needed $H \vdash m\, n' : [n/x]N$. Since $[n/x]N \rightsquigarrow_{\mathsf{p}} [n'/x]N$ we have $H \vdash \mathsf{join} : [n'/x]N = [n/x]N$, and we can conclude using the conv rule. This illustrates how fully erased conversions generalize the β-equivalence rule familiar from PTSs.

Second, consider the case when an application steps by β-reduction, $(\lambda x.m_0)\ u \rightsquigarrow_{\mathsf{p}} [u/x]m_0$, and we need to show $H \vdash [u/x]m_0 : [u/x]N$. The inversion lemma for $\lambda x.m_0$ gives $H, x : M_1 \vdash m_0 : N_1$ for some $H \vdash \mathsf{join} : (x{:}M) \to N = (x{:}M_1) \to N_1$. Now we need to convert the type of u to $H \vdash u : M_1$, so that we can apply substitution and get $H \vdash [u/x]m_0 : [u/x]N_1$, and finally convert back to $[u/x]N$. To do this we need to decompose the equality proof from the inversion lemma into proofs of $M = M_1$ and $[u/x]N_1 = [u/x]N$. We run into the same issue in the cases for irrelevant application and pattern matching on datatypes. So we add a set of injectivity rules to our type system to make these cases go through (figure 5).

$$\frac{H \vdash u_1 : D\overline{n_i} = D\overline{n_i}'}{H \vdash \mathsf{join} : n_k = n_k'}\text{ET\_INJTCON} \qquad \frac{H \vdash u_1 : (x{:}M_1) \to N_1 = (x{:}M_2) \to N_2}{H \vdash \mathsf{join} : M_1 = M_2}\text{ET\_INJDOM}$$

$$\frac{\begin{array}{l} H \vdash u_1 : (x{:}M) \to N_1 = (x{:}M) \to N_2 \\ H \vdash u : M \end{array}}{H \vdash \mathsf{join} : [u/x]N_1 = [u/x]N_2}\text{ET\_INJRNG}$$

Figure 5: Injectivity rules (the two rules for $[x{:}M_1] \to N_1$ are similar and not shown)

3.4 Progress

As is common in languages with dependent pattern matching, when proving progress we have to worry about "bad" equations. Specifically, this shows up in the canonical forms lemma. We want to say that if a closed value has a function type, then it is actually a function. However, what if we had a proof of $\mathsf{Nat} = (\mathsf{Nat} \to \mathsf{Nat})$? To rule that out, we start by proving a lemma characterizing when two expressions can be propositionally equal. From now on, H_D denotes a context which is empty except that it may contain datatype declarations.

Lemma 10 (Soundness of equality)**.** *If $H_D \vdash u : M$ and $M \curlyvee (m_1 = n_1)$, then $m_1 \curlyvee n_1$.*

The proof is by induction on $H_D \vdash u : M$. It is not hard, but it is worth describing briefly. To rule out rules like λ-abstraction, we need to know that it is never the case that $(x{:}M) \to N \curlyvee (m_1 = n_1)$, which follows because $\leadsto_{\mathsf{p}}$ preserves the top-level constructor of a term. To handle the injectivity rules, we need to know that $\curlyvee$ is injective in the sense that $(x{:}M_1) \to N_1 \curlyvee (x{:}M_2) \to N_2$ implies $M_1 \curlyvee M_2$; again this follows by reasoning about $\leadsto_{\mathsf{p}}$. Finally, consider the conversion rule. The case looks like

$$\frac{\begin{array}{l} H_D \vdash u_1 : M_1 = N_1 \quad \ldots \quad H_D \vdash u_i : M_i = N_i \\ H_D \vdash u : [M_1/x_1] \ldots [M_i/x_i]M \\ H_D \vdash [N_1/x_1] \ldots [N_i/x_i]M : \star \end{array}}{H_D \vdash u : [N_1/x_1] \ldots [N_i/x_i]M}\text{ET\_CONV}$$

We have as an assumption that $[N_1/x_1] .. [N_i/x_i]M \curlyvee (m_1 = n_1)$, and the result would follow from the IH for u if we knew that $[M_1/x_1] \ldots [M_i/x_i]M \curlyvee (m_1 = n_1)$. But by the IHs for u_i we know that $N_i \curlyvee M_i$, so this follows by substitution and transitivity of $\curlyvee$.

With the soundness lemma in hand, canonical forms and progress follow straightforwardly.

Lemma 11 (Canonical forms)**.** *Suppose $H_D \vdash u : M$.*

1. *If $M \curlyvee (x{:}M_1) \to M_2$, then u is either $\lambda x.u_1$ or $\mathsf{rec}\, f.u_1$.*
2. *If $M \curlyvee [x{:}M_1] \to M_2$, then u is either $\lambda[].u_1$ or $\mathsf{rec}\, f.u_1$.*
3. *If $M \curlyvee D\overline{M_i}$ then u is $d\overline{u_i}$, where $\mathsf{data}\, D\, \Xi\, \mathsf{where}\, \{\, \overline{d_i : \Xi_i \to\ D\ \Xi}^{i \in 1..j}\, \} \in H_D$ and d is one of the d_j.*

Theorem 12 (Progress)**.** *If $H_D \vdash m : M$, then either m is a value, m is abort, or $m \leadsto_{\mathsf{cbv}} m'$ for some m'.*

4 Related Work

Dependent types with nontermination While there are many examples of languages that combine nontermination with dependent or indexed types, most take care to ensure that nonterminating expressions can not occur inside types. They do this either by making the type language completely separate from the expression language (e.g. DML [30], ATS [29], Ωmega [21], Haskell with GADTs [19]), or by restricting dependent application to values or "pure" expressions (e.g. DML [14], F* [26], Aura [12], and [18]).

In our language, types and expressions are unified and types can even be computed by general recursive functions. In this area of the design space, the most comparable languages are Cayenne [4], Cardelli's Type:Type language [9], and ΠΣ [2]. However, none of them have the particular combination of features that we discuss in this paper, i.e. irrelevance, CBV, and a built-in propositional equality.

$\lambda^{\cong}$[13] is a CBV dependently typed language with nontermination, which used CBV-respecting parallel reduction as one possible definitional equivalence. It proposed an application rule which is more expressive than just value-dependency, but not as simple as the one in this paper. $\lambda^{\cong}$ is not as expressive as our language (no polymorphism, propositional equality, or Type-in-Type), and has no notion of irrelevance.

Irrelevance We already mentioned ICC* [7], PTS [17], and Abel's system [1]. One of the key differences between the systems is whether the variable x in an irrelevant arrow type $[x:A] \to B$ is allowed to occur freely in B ("Miquel[16]-style irrelevance", our choice) or only in irrelevant positions in B ("Pfenning[20]-style"). Agda implements the latter because it interacts better with type-directed equality [1], whereas our equality is not type-directed.

Equality The usual equality type in Coq and Agda's standard libraries is homogenous and has a computationally relevant conversion rule. These languages also provide the heterogenous JMeq [15], which we discussed above.

Extensional Type Theory, e.g. Nuprl [11], is similar to our language in that conversion is computationally irrelevant and completely erased. ETT terms are similar to our unannotated terms, while our annotated terms correspond to ETT typing derivations. On the other hand, the equational theory of ETT is different from our language, e.g. it can prove extensionality while our equality cannot.

Observational Type Theory [3] also proves $(\mathsf{conv}\, a\, \mathsf{by}\, b) = a$, but in a more sophisticated way than by erasing the conversion. Instead it provides a set of axioms and ensures that those axioms can never block reduction. It is inherently type-directed, which means that it validates extensionality but cannot make use of equations between expressions of genuinely different types.

Guru [24], like our language, can eliminate equalities where the two sides have different types, and equalities are proved by joinability without any type-directed rules. However, unlike our language the equality formation rule does not require that the equated expressions are even well-typed. This can be annoying in practice, because simple programmer errors are not caught by the type system. Guru does not have our n-ary conv rule.

GHC Core [25, 27] is similar to our language in not having a separate notion of definitional and propositional equality. Instead, all type equivalences—which are implicit in Haskell source—must be justified by the typechecker by explicit proof terms. As in our language, the presence of nontermination means that proof terms must by computed at runtime, but there is no notion of irrelevance.

5 Conclusions and Future Work

In this paper, we combined **computational irrelevance** and **nontermination** in a dependently typed programming language.

In defining the language, we made concrete choices about evaluation order and treatment of conversion. Our evaluation order is CBV, and this is reflected in the equations that language can prove (including an inherently CBV rule for error expressions). An effectful language needs a restriction on the application rule, and we propose a particularly simple yet expressive one.

Our conversion rule has a novel combination of features: the equality proof is computationally irrelevant, conversion can use equalities where the two sides have different types, and conversion can use multiple equalities at once. These features are all aimed at making reasoning about programs easier.

We then proposed typing rules for irrelevant function and constructor arguments. We gave examples showing that in contrast to previous work in pure languages, irrelevant application must be restricted, and described a value-restricted version.

In future work, we plan to integrate this design with the larger Trellys project. The Trellys language will be divided into two fragments: a "programmatic" fragment that will resemble the language presented here, and a "logical" fragment that will be restricted to ensure consistency. While designing an expressive and consistent logical fragment will involve substantial additional challenges, the present work has provided a solid foundation by identifying and solving many problems that arise from Trellys's previously unseen combination of features.

Acknowledgments

This work was supported by the National Science Foundation (NSF grant 0910500).

References

[1] Andreas Abel (2011): *Irrelevance in Type Theory with a Heterogeneous Equality Judgement*. In: *Foundations of Software Science and Computational Structures (FOSSACS), part of ETAPS 2011*, pp. 57–71.

[2] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh & Nicolas Oury (2010): ΠΣ*: Dependent Types Without the Sugar*. *Functional and Logic Programming* , pp. 40–55.

[3] Thorsten Altenkirch, Conor McBride & Wouter Swierstra (2007): *Observational equality, now!* In: *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, ACM, New York, NY, USA, pp. 57–68.

[4] Lennart Augustsson (1998): *Cayenne – a Language With Dependent Types*. In: *ICFP*, ACM, pp. 239–250.

[5] Henk Barendregt, S. Abramsky, D. M. Gabbay & T. S. E. Maibaum (1992): *Lambda Calculi with Types*. In: *Handbook of Logic in Computer Science*, Oxford University Press, pp. 117–309.

[6] H.P. Barendregt (1984): *The Lambda Calculus: Its Syntax and Semantics*. In J. Barwise, D. Kaplan, H. J. Keisler, P. Suppes & A.S. Troelstra, editors: *Studies in Logic and the Foundation of Mathematics*, 103, North-Holland.

[7] B. Barras & B. Bernardo (2008): *The Implicit Calculus of Constructions as a Programming Language with Dependent Types*. In Roberto M. Amadio, editor: *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008*, *Lecture Notes in Computer Science* 4962, Springer, pp. 365–379.

[8] Edwin Brady, Conor Mcbride & James Mckinna (2004): *Inductive families need not store their indices*. In: *Types for Proofs and Programs, Torino, 2003, volume 3085 of LNCS*, Springer-Verlag, pp. 115–129.

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

[10] Adam Chlipala (2011): *Certified programming with dependent types*. Available at `http://adam.chlipala.net/cpdt`.

[11] R. Constable & the PRL group (1986): *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall.

[12] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr & Steve Zdancewic (2008): *AURA: A Programming Language for Authorization and Audit*. In: *Proc. of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Victoria, British Columbia, Canada.

[13] Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg & Stephanie Weirich (2010): *Dependent types and program equivalence*. In: *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 275–286.

[14] Daniel R. Licata & Robert Harper (2005): *A Formulation of Dependent ML with Explicit Equality Proofs*. Technical Report CMU-CS-05-178, Carnegie Mellon University Department of Computer Science.

[15] Conor McBride (2002): *Elimination with a Motive*. In Paul Callaghan, Zhaohui Luo, James McKinna & Robert Pollack, editors: *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00)*, *LNCS* 2277, Springer-Verlag.

[16] Alexandre Miquel (2001): *The Implicit Calculus of Constructions - Extending Pure Type Systems with an Intersection Type Binder and Subtyping*. In: *Proc. of 5th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'01, Krakow*, Springer-Verlag, pp. 2–5.

[17] N. Mishra-Linger & T. Sheard (2008): *Erasure and Polymorphism in Pure Type Systems*. In Roberto M. Amadio, editor: *Foundations of Software Science and Computational Structures, 11th International Conference (FOSSACS)*, *Lecture Notes in Computer Science* 4962, Springer, pp. 350–364.

[18] Xinming Ou, Gang Tan, Yitzhak Mandelbaum & David Walker (2004): *Dynamic typing with dependent types*. In: *IFIP International Conference on Theoretical Computer Science*, pp. 437–450.

[19] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich & Geoffrey Washburn (2006): *Simple unification-based type inference for GADTs*. pp. 50–61.

[20] Frank Pfenning (2001): *Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory*. In: *Symposium on Logic in Computer Science (LICS)*.

[21] T. Sheard (2006): *Type-level Computation Using Narrowing in Ωomega*. In: *The 1st workshop on Programming Languages meets Program Verification (PLPV)*, pp. 105–128.

[22] Vilhelm Sjöberg & Aaron Stump (2010): *Equality, Quasi-Implicit Products, and Large Eliminations*. In: *ITRS 2010*.

[23] Thomas Streicher (1993): *Investigations into Intensional Type Theory*. Habilitation Thesis, Ludwig Maximilian Universität.

[24] A. Stump, M. Deters, A. Petcher, T. Schiller & T. Simpson (2009): *Verified Programming in Guru*. In T. Altenkirch & T. Millstein, editors: *Programming Languges meets Program Verification (PLPV)*, pp. 49–58.

[25] Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones & Kevin Donnelly (2007): *System F with type equality coercions*. In Franois Pottier & George C. Necula, editors: *Proceedings of TLDI 07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, ACM, pp. 53–66, doi:http://doi.acm.org/10.1145/1190315.1190324.

[26] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan & Jean Yang (2011): *Secure Distributed Programming with Value-dependent Types*. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, ACM, pp. 285–296.

[27] Dimitrios Vytiniotis & Simon Peyton Jones (2011): *Practical aspects of evidence-based compilation in System FC*. Unpublished.

[28] Dimitrios Vytiniotis & Stephanie Weirich (2007): *Dependent types: Easy as PIE*. In: *8th Symp. on Trends in Functional Programming*.

[29] Hongwei Xi (2004): *Applied Type System*. In: *Proceedings of TYPES 2003*, pp. 394–408.

[30] Hongwei Xi & Frank Pfenning (1999): *Dependent Types in Practical Programming*. pp. 214–227.

A Full Language Specification

A.1 Syntax

varlist, $\overline{x_i}$, $\overline{y_i}$	$::=$		
	$\mid$	$\overline{x_i}^{\,i\in 1\ldots j}$	
tele, Δ	$::=$		telescope
	$\mid$		empty telescope
	$\mid$	$(x:A)\Delta$	relevant binding
	$\mid$	$[x:A]\Delta$	irrelevant binding
decl	$::=$		
	$\mid$	$x:A$	
	$\mid$	$\mathsf{data}\,D\,\Delta\,\mathsf{where}\,\{\,\overline{d_i:\Delta_i\to\ D\,\Delta}^{\,i\in 1..j}\,\}$	
	$\mid$	$\mathsf{data}\,D\,\Delta$	
env, Γ	$::=$		typing environment
	$\mid$		
	$\mid$	$\Gamma, decl$	
exp, a, b, c, A, B, C	$::=$		
	$\mid$	$\star$	type
	$\mid$	x	variable
	$\mid$	$D\,\overline{A_i}$	datatype
	$\mid$	$d\ [\overline{A_i}]\ \overline{a_i}$	data
	$\mid$	$\mathsf{rec}\,f:A.v$	recursive definition
	$\mid$	$\lambda x:A.a$	lambda-abstraction
	$\mid$	$\lambda[x:A].a$	irrelevant lambda-abstraction
	$\mid$	$a\,b$	application
	$\mid$	$a\,[b]$	implicit application
	$\mid$	$(x{:}A)\to B$	function type
	$\mid$	$[x{:}A]\to B$	irrelevant function type
	$\mid$	$\mathsf{case}\,a\,\mathsf{as}\,[y]\,\mathsf{of}\,\{\,\overline{d_j\,\Delta_j\Rightarrow b_j}^{\,j\in 1..k}\,\}$	pattern matching
	$\mid$	$a=b$	equality proposition
	$\mid$	$\mathsf{join}_{a=b}\ i\ j$	equality proof
	$\mid$	$\mathsf{conv}\,a\,\mathsf{at}\,[\sim P_1/x_1]\ldots[\sim P_i/x_i]A$	
	$\mid$	abort_A	failure
P	$::=$		Proofs used in conv rule
	$\mid$	v	
	$\mid$	$[a=b]$	

val, v	::=		Values
	|	x	
	|	$\star$	
	|	$(x{:}A) \to B$	
	|	$[x{:}A] \to B$	
	|	$a = b$	
	|	$\mathsf{conv}\, v\, \mathsf{at}\, [\sim P_1/x_1] \ldots [\sim P_i/x_i] A$	
	|	$\mathsf{join}_{a=b}\, i\, j$	
	|	$D\, \overline{A_i}$	
	|	$d\, [\overline{A_i}]\, \overline{v_i}$	
	|	$\lambda x : A.a$	
	|	$\lambda [x : A].a$	
	|	$\mathsf{rec}\, f : A.v$	
explist, $\overline{a_i}$, $\overline{b_i}$, $\overline{A_i}$, $\overline{B_i}$	::=		
	|		
	|	$a\, \overline{a_i}$	
	|	$[a]\, \overline{a_i}$	
vallist, $\overline{v_i}$	::=		
	|		
	|	$v\, \overline{v_i}$	
	|	$[v]\, \overline{v_i}$	
etele, Ξ	::=		unannotated telescope
	|		empty telescope
	|	$(x : M)\Xi$	relevant binding
	|	$[x : M]\Xi$	irrelevant binding
edecl	::=		
	|	$x : M$	
	|	$\mathsf{data}\, D\, \Xi\, \mathsf{where}\, \{ \overline{d_i : \Xi_i \to\ D\ \Xi}^{i \in 1..j} \}$	
	|	$\mathsf{data}\, D\, \Xi$	
edeclD	::=		
	|	$\mathsf{data}\, D\, \Xi\, \mathsf{where}\, \{ \overline{d_i : \Xi_i \to\ D\ \Xi}^{i \in 1..j} \}$	
eenv, H	::=		typing environment
	|		
	|	H, edecl	

$eenvD$, H_D	$::=$		typing environment containing only datatype declarat
	$\vert$		
	$\vert$	$H, edeclD$	
$eexp$, m, n, M, N	$::=$		
	$\vert$	$\star$	type
	$\vert$	x	variable
	$\vert$	$D\,\overline{M_i}$	datatype
	$\vert$	$d\,\overline{m_i}$	data
	$\vert$	$\mathsf{rec}\, f.u$	recursive definition
	$\vert$	$\lambda x.m$	lambda-abstraction
	$\vert$	$\lambda [].m$	irrelevant lambda-abstraction
	$\vert$	$m\, n$	application
	$\vert$	$m[]$	irrelevant application
	$\vert$	$(x{:}M) \rightarrow N$	function type
	$\vert$	$[x{:}M] \rightarrow N$	irrelevant function type
	$\vert$	$\mathsf{case}\, n\, \mathsf{of}\, \{\, \overline{d_j\, \overline{x_{ij}} \Rightarrow m_j}^{j \in 1..k} \,\}$	pattern matching
	$\vert$	$m = n$	equality proposition
	$\vert$	join	equality proof
	$\vert$	abort	failure
$eval$, u	$::=$		Values
	$\vert$	x	
	$\vert$	$\star$	
	$\vert$	$(x{:}M) \rightarrow N$	
	$\vert$	$[x{:}M] \rightarrow N$	
	$\vert$	$m = n$	
	$\vert$	join	
	$\vert$	$D\,\overline{M_i}$	
	$\vert$	$d\,\overline{u_i}$	
	$\vert$	$\mathsf{rec}\, f.u$	
	$\vert$	$\lambda x.m$	
	$\vert$	$\lambda [].m$	
$eexplist$, $\overline{m_i}$, $\overline{n_i}$, $\overline{M_i}$, $\overline{N_i}$	$::=$		
	$\vert$		
	$\vert$	$m\, \overline{m_i}$	
	$\vert$	$[]\, \overline{m_i}$	
$evallist$, $\overline{u_i}$	$::=$		
	$\vert$		

	$\vert$	$u\,\overline{u_i}$	
	$\vert$	$[]\,\overline{u_i}$	
evalctx, $\mathscr{E}$	::=		Evaluation contexts
	$\vert$	$\bullet$	
	$\vert$	$\bullet m$	
	$\vert$	$u\bullet$	
	$\vert$	$\bullet[]$	
	$\vert$	$\mathsf{case}\ \bullet\ \mathsf{of}\ \{d_j\,\overline{x_{ij}} \Rightarrow m_j\}$	
	$\vert$	$d\,\overline{u_i}\bullet\overline{m_i}$	
varset	::=		

A.2 Erasure function

The erasure function $|a|$ is defined by:

$$
\begin{array}{ll}
|\star| & = \star \\
|x| & = x \\
|D\,\overline{A_i}| & = D\,|\overline{A_i}| \\
|d\ [\overline{A_i}]\ \overline{a_i}| & = d\,|\overline{a_i}| \\
|\mathsf{rec}\,f : A.v| & = \mathsf{rec}\,f.|v| \\
|\lambda x : A.a| & = \lambda x.|a| \\
|\lambda [x : A].a| & = \lambda [].|a| \\
|a\ b| & = |a|\ |b| \\
|a\ [b]| & = |a| [] \\
|(x{:}A) \to B| & = (x{:}|A|) \to |B| \\
|[x{:}A] \to B| & = [x{:}|A|] \to |B| \\
|a = b| & = |a| = |b| \\
|\mathsf{join}_{a=b}\ i\ j| & = \mathsf{join} \\
|\mathsf{case}\,a\,\mathsf{as}\,[y]\,\mathsf{of}\,\{\,\overline{d_j\,\Delta_j \Rightarrow b_j}^{j\in 1..k}\,\}| & = \mathsf{case}\,|a|\,\mathsf{of}\,\{\,\overline{d_j\,\overline{x_{ij}} \Rightarrow |b_j|}^{j\in 1..k}\,\} \\
 & \quad \text{where } \overline{x_{ij}} \text{ are the relevant variables of } \Delta_j \\
|\mathsf{conv}\,a\,\mathsf{at}\,[\sim P_1/x_1]\ldots[\sim P_i/x_i]A| & = |a| \\
|\mathsf{abort}_A| & = \mathsf{abort} \\
 & \\
|\cdot| & = \cdot \\
|a\ \overline{a_i}| & = |a|\ |\overline{a_i}| \\
|[a]\ \overline{a_i}| & = []\ |\overline{a_i}|
\end{array}
$$

A.3 CBV evalution

$\boxed{m \rightsquigarrow_{\mathsf{cbv}} n}$

$$\frac{}{(\lambda x.m)\ u \rightsquigarrow_{\mathsf{cbv}} [u/x]m}\ \text{SC\_APPBETA}$$

$$\frac{}{(\mathsf{rec}\,f.u_1)\ u_2 \rightsquigarrow_{\mathsf{cbv}} ([\mathsf{rec}\,f.u_1/f]u_1)\ u_2}\ \text{SC\_APPREC}$$

$$\frac{}{(\lambda[].m)[] \rightsquigarrow_{\mathsf{cbv}} m}\text{SC\_IAPPBETA}$$

$$\frac{}{(\mathsf{rec}\, f.u_1)[] \rightsquigarrow_{\mathsf{cbv}} ([\mathsf{rec}\, f.u_1/f]u_1)[]}\text{SC\_IAPPREC}$$

$$\frac{}{\mathsf{case}\,(d_l\,\overline{u_i})\,\mathsf{of}\,\{\,\overline{d_j\,\overline{x_{ij}} \Rightarrow m_j}^{j\in 1..k}\,\} \rightsquigarrow_{\mathsf{cbv}} [\overline{u_i}/\overline{x_{il}}]m_l}\text{SC\_CASEBETA}$$

$$\frac{}{\mathscr{E}[\mathsf{abort}] \rightsquigarrow_{\mathsf{cbv}} \mathsf{abort}}\text{SC\_ABORT}$$

$$\frac{m \rightsquigarrow_{\mathsf{cbv}} n}{\mathscr{E}[m] \rightsquigarrow_{\mathsf{cbv}} \mathscr{E}[n]}\text{SC\_CTX}$$

A.4 Parallel reduction

$\boxed{m \rightsquigarrow_{\mathsf{p}} n}$

$$\frac{}{m \rightsquigarrow_{\mathsf{p}} m}\text{SP\_REFL}$$

$$\frac{u \rightsquigarrow_{\mathsf{p}} u'}{\mathsf{rec}\, f.u \rightsquigarrow_{\mathsf{p}} \mathsf{rec}\, f.u'}\text{SP\_REC}$$

$$\frac{m \rightsquigarrow_{\mathsf{p}} m'}{\lambda x.m \rightsquigarrow_{\mathsf{p}} \lambda x.m'}\text{SP\_ABS}$$

$$\frac{\begin{array}{c}M \rightsquigarrow_{\mathsf{p}} M' \\ N \rightsquigarrow_{\mathsf{p}} N'\end{array}}{(x{:}M) \to N \rightsquigarrow_{\mathsf{p}} (x{:}M') \to N'}\text{SP\_PI}$$

$$\frac{\begin{array}{c}M \rightsquigarrow_{\mathsf{p}} M' \\ N \rightsquigarrow_{\mathsf{p}} N'\end{array}}{[x{:}M] \to N \rightsquigarrow_{\mathsf{p}} [x{:}M'] \to N'}\text{SP\_IPI}$$

$$\frac{\begin{array}{c}m \rightsquigarrow_{\mathsf{p}} m' \\ n \rightsquigarrow_{\mathsf{p}} n'\end{array}}{m = n \rightsquigarrow_{\mathsf{p}} m' = n'}\text{SP\_EQ}$$

$$\frac{\begin{array}{c}m \rightsquigarrow_{\mathsf{p}} m' \\ n \rightsquigarrow_{\mathsf{p}} n'\end{array}}{m\ n \rightsquigarrow_{\mathsf{p}} m'\ n'}\text{SP\_APP}$$

$$\frac{\begin{array}{c}m \rightsquigarrow_{\mathsf{p}} m' \\ u \rightsquigarrow_{\mathsf{p}} u'\end{array}}{(\lambda x.m)\ u \rightsquigarrow_{\mathsf{p}} [u'/x]m'}\text{SP\_APPBETA}$$

$$\frac{\begin{array}{c}u_1 \rightsquigarrow_{\mathsf{p}} u_1' \\ u_2 \rightsquigarrow_{\mathsf{p}} u_2'\end{array}}{(\mathsf{rec}\, f.u_1)\ u_2 \rightsquigarrow_{\mathsf{p}} ([\mathsf{rec}\, f.u_1'/f]u_1')\ u_2'}\text{SP\_APPREC}$$

$$\frac{m \rightsquigarrow_{\mathsf{p}} m'}{m[] \rightsquigarrow_{\mathsf{p}} m'[]}\text{SP\_IAPP}$$

$$\frac{m \rightsquigarrow_{\mathsf{p}} m'}{(\lambda [].m)[] \rightsquigarrow_{\mathsf{p}} m'}\text{SP\_IAPPBETA}$$

$$\frac{u_1 \rightsquigarrow_{\mathsf{p}} u_1'}{(\mathsf{rec}\, f.u_1)[] \rightsquigarrow_{\mathsf{p}} ([\mathsf{rec}\, f.u_1'/f]u_1')[]}\text{SP\_IAPPREC}$$

$$\frac{\forall i.\;\; M_i \rightsquigarrow_{\mathsf{p}} M_i'}{D\,\overline{M_i} \rightsquigarrow_{\mathsf{p}} D\,\overline{M_i}'}\text{SP\_TCON}$$

$$\frac{\forall i.\;\; m_i \rightsquigarrow_{\mathsf{p}} m_i'}{d\,\overline{m_i} \rightsquigarrow_{\mathsf{p}} d\,\overline{m_i}}\text{SP\_DCON}$$

$$\frac{\begin{array}{l} m \rightsquigarrow_{\mathsf{p}} m' \\ \forall j.\;\; m_j \rightsquigarrow_{\mathsf{p}} m_j' \end{array}}{\mathsf{case}\, m\, \mathsf{of}\, \{\, \overline{d_j\,\overline{x_{ij}} \Rightarrow m_j}^{\,j\in 1..k}\, \} \rightsquigarrow_{\mathsf{p}} \mathsf{case}\, m'\, \mathsf{of}\, \{\, \overline{d_j\,\overline{x_{ij}} \Rightarrow m_j'}^{\,j\in 1..k}\, \}}\text{SP\_CASE}$$

$$\frac{\begin{array}{l} \forall i.\;\; u_i \rightsquigarrow_{\mathsf{p}} u_i' \\ m_l \rightsquigarrow_{\mathsf{p}} m_l' \end{array}}{\mathsf{case}\, (d_l\,\overline{u_i})\, \mathsf{of}\, \{\, \overline{d_j\,\overline{x_{ij}} \Rightarrow m_j}^{\,j\in 1..k}\, \} \rightsquigarrow_{\mathsf{p}} [\overline{u_i}'/\overline{x_{il}}]m_l'}\text{SP\_CASEBETA}$$

$$\frac{}{\mathscr{E}[\mathsf{abort}] \rightsquigarrow_{\mathsf{p}} \mathsf{abort}}\text{SP\_ABORT}$$

$\boxed{n \curlyvee m}$

$$\frac{\begin{array}{l} m_1 \rightsquigarrow_{\mathsf{p}}^{*} n \\ m_2 \rightsquigarrow_{\mathsf{p}}^{*} n \end{array}}{m_2 \curlyvee m_2}\text{J\_JOIN}$$

A.5 Annotated type system

$\boxed{\Gamma \vdash a : A}$

$$\frac{\vdash \Gamma}{\Gamma \vdash \star : \star}\text{T\_TYPE}$$

$$\frac{\begin{array}{l} \Gamma \vdash b : D\,\overline{B_i} \\ \Gamma \vdash A : \star \\ \mathsf{data}\, D\,\Delta\, \mathsf{where}\, \{\, \overline{d_i : \Delta_i \to\; D\,\Delta}^{\,i\in 1..l}\, \} \in \Gamma \\ \forall i.\;\; \Gamma, [\overline{B_i}/\Delta]\Delta_i, y : b = d_i\,\Delta_i \vdash a_i : A \\ \forall i.\;\; \{y\} \cup dom-(\Delta_i)\ \#\ \mathsf{FV}\,(|a_i|) \end{array}}{\Gamma \vdash \mathsf{case}\, b\, \mathsf{as}\, [y]\, \mathsf{of}\, \{\, \overline{d_i\,\Delta_i \Rightarrow a_i}^{\,i\in 1..l}\, \} : A}\text{T\_CASE}$$

$$\frac{\begin{array}{l} x : A \in \Gamma \\ \vdash \Gamma \end{array}}{\Gamma \vdash x : A}\text{T\_VAR}$$

$$\frac{\Gamma \vdash A : \star \quad \Gamma, x : A \vdash B : \star}{\Gamma \vdash (x{:}A) \to B : \star}\text{T\_PI}$$

$$\frac{\Gamma \vdash A : \star \quad \Gamma, x : A \vdash B : \star}{\Gamma \vdash [x{:}A] \to B : \star}\text{T\_IPI}$$

$$\frac{\begin{array}{l}\mathsf{data}\, D\, \Delta\, \mathsf{where}\, \{\, \overline{d_i : \Delta_i \to\ D\, \Delta}^{i \in 1..j}\, \} \in \Gamma \\ \Gamma \vdash \overline{A_i} : \Delta^{+}\end{array}}{\Gamma \vdash D\, \overline{A_i} : \star}\text{T\_TCON}$$

$$\frac{\begin{array}{l}\mathsf{data}\, D\, \Delta \in \Gamma \\ \Gamma \vdash \overline{A_i} : \Delta^{+}\end{array}}{\Gamma \vdash D\, \overline{A_i} : \star}\text{T\_ABSTCON}$$

$$\frac{\begin{array}{l}\mathsf{data}\, D\, \Delta\, \mathsf{where}\, \{\, \overline{d_i : \Delta_i \to\ D\, \Delta}^{i \in 1..j}\, \} \in \Gamma \\ \Gamma \vdash \overline{A_i} : \Delta^{+} \\ \Gamma \vdash \overline{a_i} : [\overline{A_i}/\Delta]\Delta_i\end{array}}{\Gamma \vdash d_k\, [\overline{A_i}]\, \overline{a_i} : D\, \overline{A_i}}\text{T\_DCON}$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x : A.b : (x{:}A) \to B}\text{T\_ABS}$$

$$\frac{\begin{array}{l}\Gamma, x : A \vdash b : B \\ x\ \notin \mathsf{FV}\,(|b|)\end{array}}{\Gamma \vdash \lambda[x : A].b : [x{:}A] \to B}\text{T\_IABS}$$

$$\frac{\begin{array}{l}\Gamma, f : A \vdash v : A \quad \Gamma \vdash A : \star \\ A \text{ is } (x{:}A_1) \to A_2 \text{ or } [x{:}A_1] \to A_2\end{array}}{\Gamma \vdash \mathsf{rec}\, f : A.v : A}\text{T\_REC}$$

$$\frac{\begin{array}{l}\Gamma \vdash a : (x{:}A) \to B \\ \Gamma \vdash a : A \\ \Gamma \vdash [a/x]B : \star\end{array}}{\Gamma \vdash a\, b : [a/x]B}\text{T\_APP}$$

$$\frac{\begin{array}{l}\Gamma \vdash a : [x{:}A] \to B \\ \Gamma \vdash v : A\end{array}}{\Gamma \vdash a\, [v] : [v/x]B}\text{T\_IAPP}$$

$$\frac{\Gamma \vdash A : \star}{\Gamma \vdash \mathsf{abort}_A : A}\text{T\_ABORT}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash a = b : \star}\text{T\_EQ}$$

$$\frac{\begin{array}{l}|a| \leadsto^{i}_{\mathsf{cbv}} n \quad |b| \leadsto^{j}_{\mathsf{cbv}} n \\ \Gamma \vdash a = b : \star\end{array}}{\Gamma \vdash \mathsf{join}_{a=b}\, i\, j : a = b}\text{T\_JOIN}$$

$$\frac{\begin{array}{l}\forall i.\ \ ((P_i \text{ is } v_i \text{ and } \Gamma \vdash v_i : A_i = B_i) \text{ or } (P_i \text{ is } [A_i = B_i] \text{ and } x_i\ \notin \mathsf{FV}\,(|A|)\,)) \\ \Gamma \vdash a : [A_1/x_1] \ldots [A_i/x_i]A \\ \Gamma \vdash [B_1/x_1] \ldots [B_i/x_i]A : \star\end{array}}{\Gamma \vdash \mathsf{conv}\, a\, \mathsf{at}\, [\sim P_1/x_1] \ldots [\sim P_i/x_i]A : [B_1/x_1] \ldots [B_i/x_i]A}\text{T\_CONV}$$

$$\frac{\Gamma \vdash v_1 : ((x{:}A_1) \to B_1) = ((x{:}A_2) \to B_2)}{\Gamma \vdash \mathsf{join} : A_1 = A_2}\text{T\_INJDOM}$$

$$\frac{\Gamma \vdash v_1 : ((x{:}A) \to B_1) = ((x{:}A) \to B_2) \quad \Gamma \vdash v : A}{\Gamma \vdash \mathsf{join} : [v/x]B_1 = [v/x]B_2}\text{T\_INJRNG}$$

$$\frac{\Gamma \vdash v_1 : ([x{:}A_1] \to B_1) = ([x{:}A_2] \to B_2)}{\Gamma \vdash \mathsf{join} : A_1 = A_2}\text{T\_IINJDOM}$$

$$\frac{\Gamma \vdash v_1 : ([x{:}A] \to B_1) = ([x{:}A] \to B_2) \quad \Gamma \vdash v : A}{\Gamma \vdash \mathsf{join} : [v/x]B_1 = [v/x]B_2}\text{T\_IINJRNG}$$

$$\frac{\Gamma \vdash v_1 : D\,\overline{A_i} = D\,\overline{A_i}'}{\Gamma \vdash \mathsf{join} : A_k = A'_k}\text{T\_INJTCON}$$

$\boxed{\vdash \Gamma}$ Γ is a well-formed environment

$$\frac{}{\vdash}\text{ENV\_WF\_EMPTY}$$

$$\frac{\begin{array}{l}\vdash \Gamma \quad x \notin \mathsf{dom}(\Gamma) \\ \Gamma \vdash A : \star\end{array}}{\vdash \Gamma, x : A}\text{ENV\_WF\_VAR}$$

$$\frac{\begin{array}{l}\vdash \Gamma, \Delta \quad D \notin \mathsf{dom}(H) \\ \overline{\Gamma, \mathsf{data}\, D\Delta, \Delta \vdash \Delta_i \to D\Delta : \star}^{i \in 1..j}\end{array}}{\vdash \Gamma, \mathsf{data}\, D\Delta\, \mathsf{where}\, \{\, \overline{d_i : \Delta_i \to\ D\,\Delta}^{i \in 1..j} \}}\text{ENV\_WF\_DTYPE}$$

$\boxed{\Gamma \vdash \overline{a_i} : \Delta}$

$$\frac{}{\Gamma \vdash :}\text{TL\_EMPTY}$$

$$\frac{\begin{array}{l}\Gamma \vdash a : A \\ \Gamma \vdash A : \star \\ \Gamma \vdash \overline{a_i} : [a/x]\Delta\end{array}}{\Gamma \vdash a\,\overline{a_i} : (x : A)\Delta}\text{TL\_CONS}$$

$$\frac{\begin{array}{l}\Gamma \vdash a : A \\ \Gamma \vdash A : \star \\ \Gamma \vdash \overline{a_i} : [a/x]\Delta\end{array}}{\Gamma \vdash [a]\,\overline{a_i} : [x : A]\Delta}\text{TL\_ICONS}$$

A.6 Unannotated type system

$\boxed{H \vdash m : M}$

$$\frac{\vdash H}{H \vdash \star : \star}\text{ET\_TYPE}$$

$$\frac{\begin{array}{l} H \vdash n : D\,\overline{n_i} \\ H \vdash M : \star \\ \mathsf{data}\,D\,\Xi\,\mathsf{where}\,\{\,\overline{d_i : \Xi_i \to\ D\,\Xi}^{i\in 1..l}\,\} \in H \\ \forall i.\ \ H, [\overline{n_i}/\Xi]\Xi_i, y : n = d_i\,\Xi_i \vdash m_i : M \\ \forall i.\ \ \{y\} \cup \mathsf{dom}^-(\Xi_i)\ \#\ \mathsf{FV}\,(m_i) \\ \overline{x_i}_i \text{ is } \mathsf{dom}^+(\Xi_i) \end{array}}{H \vdash \mathsf{case}\,n\,\mathsf{of}\,\{\,\overline{d_i\,\overline{x_i}_i \Rightarrow m_i}^{i\in 1..l}\,\} : M}\ \textrm{ET\_CASE}$$

$$\frac{\begin{array}{l} x : M \in H \\ \vdash H \end{array}}{H \vdash x : M}\ \textrm{ET\_VAR}$$

$$\frac{H \vdash M : \star \quad H, x : M \vdash N : \star}{H \vdash (x{:}M) \to N : \star}\ \textrm{ET\_PI}$$

$$\frac{H \vdash M : \star \quad H, x : M \vdash N : \star}{H \vdash [x{:}M] \to N : \star}\ \textrm{ET\_IPI}$$

$$\frac{\begin{array}{l} \mathsf{data}\,D\,\Xi\,\mathsf{where}\,\{\,\overline{d_i : \Xi_i \to\ D\,\Xi}^{i\in 1..j}\,\} \in H \\ H \vdash \overline{M_i} : \Xi^+ \end{array}}{H \vdash D\,\overline{M_i} : \star}\ \textrm{ET\_TCON}$$

$$\frac{\begin{array}{l} \mathsf{data}\,D\,\Xi \in H \\ H \vdash \overline{M_i} : \Xi^+ \end{array}}{H \vdash D\,\overline{M_i} : \star}\ \textrm{ET\_ABSTCON}$$

$$\frac{\begin{array}{l} \mathsf{data}\,D\,\Xi\,\mathsf{where}\,\{\,\overline{d_i : \Xi_i \to\ D\,\Xi}^{i\in 1..j}\,\} \in H \\ H \vdash \overline{M_i} : \Xi^+ \\ H \vdash \overline{m_i} : [\overline{M_i}/\Xi]\Xi_i \end{array}}{H \vdash d_k\,\overline{m_i} : D\,\overline{M_i}}\ \textrm{ET\_DCON}$$

$$\frac{H, x : M \vdash n : N}{H \vdash \lambda x.n : (x{:}M) \to N}\ \textrm{ET\_ABS}$$

$$\frac{\begin{array}{l} H, x : M \vdash n : N \\ x \notin \mathsf{FV}\,(n) \end{array}}{H \vdash \lambda[].n : [x{:}M] \to N}\ \textrm{ET\_IABS}$$

$$\frac{\begin{array}{l} H, f : M \vdash u : M \\ H \vdash M : \star \\ M \text{ is } (x{:}M_1) \to M_2 \text{ or } [x{:}M_1] \to M_2 \end{array}}{H \vdash \mathsf{rec}\,f.u : M}\ \textrm{ET\_REC}$$

$$\frac{\begin{array}{l} H \vdash m : (x{:}M) \to N \\ H \vdash n : M \\ H \vdash [n/x]N : \star \end{array}}{H \vdash m\,n : [n/x]N}\ \textrm{ET\_APP}$$

$$\frac{\begin{array}{l} H \vdash m : [x{:}M] \to N \\ H \vdash u : M \end{array}}{H \vdash m[] : [u/x]N}\ \textrm{ET\_IAPP}$$

$$\frac{H \vdash M : \star}{H \vdash \mathsf{abort} : M}\text{ET\_ABORT}$$

$$\frac{H \vdash m : M \quad H \vdash n : N}{H \vdash m = n : \star}\text{ET\_EQ}$$

$$\frac{\begin{array}{l} m \curlyvee n \\ H \vdash m = n : \star \end{array}}{H \vdash \mathsf{join} : m = n}\text{ET\_JOIN}$$

$$\frac{\begin{array}{l} H \vdash u_1 : M_1 = N_1 \quad \ldots \quad H \vdash u_i : M_i = N_i \\ H \vdash m : [M_1/x_1]\ldots[M_i/x_i]M \\ H \vdash [N_1/x_1]\ldots[N_i/x_i]M : \star \end{array}}{H \vdash m : [N_1/x_1]\ldots[N_i/x_i]M}\text{ET\_CONV}$$

$$\frac{H \vdash u_1 : (x{:}M_1) \to N_1 = (x{:}M_2) \to N_2}{H \vdash \mathsf{join} : M_1 = M_2}\text{ET\_INJDOM}$$

$$\frac{\begin{array}{l} H \vdash u_1 : (x{:}M) \to N_1 = (x{:}M) \to N_2 \\ H \vdash u : M \end{array}}{H \vdash \mathsf{join} : [u/x]N_1 = [u/x]N_2}\text{ET\_INJRNG}$$

$$\frac{H \vdash u_1 : [x{:}M_1] \to N_1 = [x{:}M_2] \to N_2}{H \vdash \mathsf{join} : M_1 = M_2}\text{ET\_IINJDOM}$$

$$\frac{\begin{array}{l} H \vdash u_1 : [x{:}M] \to N_1 = [x{:}M] \to N_2 \\ H \vdash u : M \end{array}}{H \vdash \mathsf{join} : [u/x]N_1 = [u/x]N_2}\text{ET\_IINJRNG}$$

$$\frac{H \vdash u_1 : D\,\overline{n_i} = D\,\overline{n_i}'}{H \vdash \mathsf{join} : n_k = n'_k}\text{ET\_INJTCON}$$

$\boxed{\vdash H}$ H is a well-formed environment

$$\frac{}{\vdash}\text{EENV\_WF\_EMPTY}$$

$$\frac{\begin{array}{l} \vdash H \quad x \notin \mathsf{dom}(H) \\ H \vdash M : \star \end{array}}{\vdash H, x : M}\text{EENV\_WF\_VAR}$$

$$\frac{\begin{array}{l} \vdash H, \Xi \\ D \notin \mathsf{dom}(H) \\ \forall i.\ d_i \notin \mathsf{dom}(H) \\ \forall i.\ H, \mathsf{data}\, D\, \Xi, \Xi \vdash \Xi_i \to D\, \Xi : \star \end{array}}{\vdash H, \mathsf{data}\, D\, \Xi\, \mathsf{where}\, \{ \overline{d_i : \Xi_i \to D\, \Xi}^{i \in 1..j} \}}\text{EENV\_WF\_DTYPE}$$

$\boxed{H \vdash \overline{m_i} : \Xi}$

$$\frac{}{H \vdash :}\text{ETL\_EMPTY}$$

$$\frac{\begin{array}{l} H \vdash m : M \\ H \vdash M : \star \\ H \vdash \overline{m_i} : [m/x]\Xi \end{array}}{H \vdash m\,\overline{m_i} : (x:M)\Xi}\text{ETL\_CONS}$$

$$\frac{\begin{array}{l} H \vdash u : M \\ H \vdash M : \star \\ H \vdash \overline{m_i} : [u/x]\Xi \end{array}}{H \vdash []\,\overline{m_i} : [x:M]\Xi}\text{ETL\_ICONS}$$

B Proofs

B.1 Correctness of erasure

Lemma 13. *If $\Gamma \vdash a : A$, then $|\Gamma| \vdash |a| : |A|$.*

Proof. Easy induction. □

B.2 Facts about parallel reduction

Definition 14. *The head constructor of an expression is defined as follows:*

- *The head constructor of $\star$ is $\star$.*
- *The head constructor of Nat is Nat.*
- *The head constructor of $(x\!:\!M) \to N$ is $\to$.*
- *The head constructor of $[x\!:\!M] \to N$ is $[] \to$.*
- *The head constructor of $D\overline{M_i}$ is D.*
- *The head constructor of $d\overline{m_i}$ is d.*
- *The head constructor of $a = b$ is $=$.*
- *Other expressions do not have a head constructor.*

We write $\mathsf{hd}\,(M)$ for the partial function mapping M to its head constructor.

Lemma 15. *If $m \rightsquigarrow_{\mathsf{p}} m'$ and $\mathsf{hd}\,(m)$ is defined, then $\mathsf{hd}\,(m) = \mathsf{hd}\,(m')$.*

Proof. By inspecting the definition of $\rightsquigarrow_{\mathsf{p}}$ we see that it always preserves the head constructor of a term. □

Lemma 16. *If $m \curlyvee m'$, then m and m' do not have different head constructors.*

Proof. Expanding the definition of $\curlyvee$ we know that $m \rightsquigarrow_{\mathsf{p}}^{*} n$ and $m' \rightsquigarrow_{\mathsf{p}}^{*} n$ for some n. If m and m' had (defined and) different head constructors, then by repeatedly applying Lemma 15 we would get that n had two different head constructors, which is impossible. □

Lemma 17 (Injectivity of $\curlyvee$)**.**

- *If $m_1 = n_1 \curlyvee m_2 = n_2$, then $m_1 \curlyvee m_2$ and $n_1 \curlyvee n_2$.*
- *If $D\overline{M}_{i1} \curlyvee D\overline{M}_{i2}$, then $M_{i1} \curlyvee M_{i2}$.*

- *If* $(x{:}M_1) \to N_1 \curlyvee (x{:}M_2) \to N_2$ *then* $M_1 \curlyvee M_2$ *and* $N_1 \curlyvee N_2$.
- *If* $[x{:}M_1] \to N_1 \curlyvee [x{:}M_2] \to N_2$ *then* $M_1 \curlyvee M_2$ *and* $N_1 \curlyvee N_2$.

Proof. The lemma is proven in the same way for all the different types of expressions, so we only show the proof for (1). Expanding the definition of $\curlyvee$, we have that $m_1 = n_1 \rightsquigarrow_{\mathsf{p}}^{*} N$ and $m_2 = n_2 \rightsquigarrow_{\mathsf{p}} N$ for some N.

By lemma 16 we know that N has the shape $n = m$. So it suffices to prove that, for any n_1, m_1, if $n_1 = m_1 \rightsquigarrow_{\mathsf{p}}^{*} n = m$, then $n_1 \rightsquigarrow_{\mathsf{p}}^{*} n$. This follows by an easy induction on the chain of reduction, since at each step the only reduction rule that can apply is congruence. □

Lemma 18. *If* $u \rightsquigarrow_{\mathsf{p}} u'$ *and* $m \rightsquigarrow_{\mathsf{p}} m'$*, then* $[u/x]m \rightsquigarrow_{\mathsf{p}} [u'/x]m'$.

Proof. By induction on $m \rightsquigarrow_{\mathsf{p}} m'$ □

Lemma 19. *If* $M \curlyvee M'$*, then* $[u/x]M \curlyvee [u/x]M'$.

Proof. Expanding the definition of $\curlyvee$ we get $M \rightsquigarrow_{\mathsf{p}}^{*} M_1$ and $M' \rightsquigarrow_{\mathsf{p}} M_1$ for some M_1. Repeatedly applying Lemma 18 we then get $[u/x]M \rightsquigarrow_{\mathsf{p}}^{*} [u/x]M_1$ and $[u/x]M' \rightsquigarrow_{\mathsf{p}}^{*} [u/x]M_1$ as required. □

Lemma 20 (One-step diamond property for $\rightsquigarrow_{\mathsf{p}}$)**.** *If* $m \rightsquigarrow_{\mathsf{p}} m_1$ *and* $m \rightsquigarrow_{\mathsf{p}} m_2$*, then there exists some* m' *such that* $m_1 \rightsquigarrow_{\mathsf{p}} m'$ *and* $m_2 \rightsquigarrow_{\mathsf{p}} m'$.

Proof. By induction on the structure of m. We only show the case when m is an application $m_1\ m_2$, as this case contains all the ideas of the proof.

Case m **is** $m_1\ m_2$ We consider all possible pairs of ways that $m_1\ m_2$ can reduce.

- One reduction is SC_REFL. This case is trivial.
- Both reductions are SC_APP. That is to say, $m_1\ m_2 \rightsquigarrow_{\mathsf{p}} m_{11}\ m_{21}$ and $m_1\ m_2 \rightsquigarrow_{\mathsf{p}} m_{12}\ m_{22}$, where $m_1 \rightsquigarrow_{\mathsf{p}} m_{11}$, $m_1 \rightsquigarrow_{\mathsf{p}} m_{21}$, $m_2 \rightsquigarrow_{\mathsf{p}} m_{21}$ and $m_2 \rightsquigarrow_{\mathsf{p}} m_{22}$.
  By the induction hypothesis for m_1, there exists m_1', such that $m_{11} \rightsquigarrow_{\mathsf{p}} m_1'$ and $m_{21} \rightsquigarrow_{\mathsf{p}} m_2'$. Similarly for m_2. So by SC_APP we have $m_{11}\ m_{21} \rightsquigarrow_{\mathsf{p}} m_1'\ m_2'$ and $m_{12}\ m_{22} \rightsquigarrow_{\mathsf{p}} m_1'\ m_2'$ as required.
- One reduction is SC_APPBETA. So it must be the case that $m_1\ m_2$ is $(\lambda x.m_0)\ u$. By considering cases, we see that only only possibilities for the other reduction is SC_APPBETA and SC_APP. In the case when the other reduction is SC_APP, we see that the only way that $\lambda x.m_0$ can step is by congruence when $m_0 \rightsquigarrow_{\mathsf{p}} m_{02}$. So we have:

$$\begin{aligned} (\lambda x.m_0)\ u &\rightsquigarrow_{\mathsf{p}} [u_1/x]m_{01} && \text{where } m_0 \rightsquigarrow_{\mathsf{p}} m_{01} \text{ and } u \rightsquigarrow_{\mathsf{p}} u_1. \\ (\lambda x.m_0)\ u &\rightsquigarrow_{\mathsf{p}} (\lambda x.m_{02})\ u_2 && \text{where } m_0 \rightsquigarrow_{\mathsf{p}} m_{02} \text{ and } u \rightsquigarrow_{\mathsf{p}} u_2. \end{aligned}$$

  Now by IH we get m_0' and u'. By substitution (lemma 18) we get $[u_1/x]m_{01} \rightsquigarrow_{\mathsf{p}} [u'/x]m_0'$, while by SC_APPBETA we get $(\lambda x.m_{02})\ u_2 \rightsquigarrow_{\mathsf{p}} [u'/x]m_0'$. So the terms are joinable as required. On the other hand, if both the reductions are by SC_APPREC, then we have

$$\begin{aligned} (\lambda x.m_0)\ u &\rightsquigarrow_{\mathsf{p}} [u_1/x]m_{01} && \text{where } m_0 \rightsquigarrow_{\mathsf{p}} m_{01} \text{ and } u \rightsquigarrow_{\mathsf{p}} u_1. \\ (\lambda x.m_0)\ u &\rightsquigarrow_{\mathsf{p}} [u_2/x]m_{02} && \text{where } m_0 \rightsquigarrow_{\mathsf{p}} m_{02} \text{ and } u \rightsquigarrow_{\mathsf{p}} u_2. \end{aligned}$$

Then by IH we again get m'_0 and u', and by substitution (twice), the two terms are again joinable at $[u'/x]m'_0$.

- One reduction is SC_APPREC. So $m_1\ m_2$ must be $(\mathsf{rec} f.u_1)\ u_2$. By considering cases, we see that the other reduction must be either SC_APPREC or SC_APP.
  If the other rule is SC_APP we note that the only way $\mathsf{rec} f.u_1$ can step is by congruence to $\mathsf{rec} f.u_1 \rightsquigarrow_{\mathsf{p}} \mathsf{rec} f.u_{12}$, so we have

$$\begin{aligned}(\mathsf{rec} f.u_1)\ u_2 \rightsquigarrow_{\mathsf{p}} ([\mathsf{rec} f.u_{11}/f]u_{11})\ u_{21} &\qquad \text{where } u_1 \rightsquigarrow_{\mathsf{p}} u_{11} \text{ and } u_2 \rightsquigarrow_{\mathsf{p}} u_{21}\\ (\mathsf{rec} f.u_1)\ u_2 \rightsquigarrow_{\mathsf{p}} (\mathsf{rec} f.u_{21})\ u_{22} &\qquad \text{where } u_1 \rightsquigarrow_{\mathsf{p}} u_{12} \text{ and } u_2 \rightsquigarrow_{\mathsf{p}} u_{22}\end{aligned}$$

  Now, by IH we have u'_1 and u'_2. By congruence, $\mathsf{rec} f.u_{11} \rightsquigarrow_{\mathsf{p}} \mathsf{rec} f.u'_1$, so by substitution (lemma 18) we get $[\mathsf{rec} f.u_{11}/f]u_{11} \rightsquigarrow_{\mathsf{p}} [\mathsf{rec} f.u'_1/f]u'_1$, and then by congruence $([\mathsf{rec} f.u_{11}/f]u_{11})\ u_{21} \rightsquigarrow_{\mathsf{p}} ([\mathsf{rec} f.u'_1/f]u'_1)\ u'_2$. Meanwhile, by SC_APPREC we have $(\mathsf{rec} f.u_{21})\ u_{22} \rightsquigarrow_{\mathsf{p}} ([\mathsf{rec} f.u'_1/f]u'_1)\ u'_2$ as required.
  On the other hand, if both reductions where by SC_APPREC, then we proceed in the same way, but conclude by using the substitution lemma for both expressions.
- One reduction is SC_ABORT. So $m_1\ m_2$ must be $\mathsf{abort}\ m_2$ or $u_1\ \mathsf{abort}$. Then by considering possible cases, we see that the other reduction must be SC_ABORT or SC_APP (the β-rules cannot match because abort is not a value). If the other rule is SC_AVORT we are trivially done, if it is SC_APP then the term steps to $u'_1\ \mathsf{abort}$, which can step to abort as required.

□

Lemma 21 (Confluence of $\rightsquigarrow_{\mathsf{p}}$)**.** *If $m \rightsquigarrow_{\mathsf{p}}^* m_1$ and $m \rightsquigarrow_{\mathsf{p}}^* m_2$, then there exists some m' such that $m_1 \rightsquigarrow_{\mathsf{p}}^* m'$ and $m_2 \rightsquigarrow_{\mathsf{p}}^* m'$.*

Proof. This is a simple corollary of the 1-step version (lemma 20), by "diagram-chasing to fill in the rectangle" (see e.g. [6], lemma 3.2.2). □

Lemma 22 ($\curlyvee$ is an equivalence relation)**.**

1. *For any m, $m \curlyvee m$.*
2. *If $m \curlyvee n$ then $n \curlyvee m$.*
3. *If $m_1 \curlyvee m_2$ and $m_2 \curlyvee m_3$, then $m_1 \curlyvee m_3$.*

Proof. (1) and (2) are immediate just by expanding the definition of $m \curlyvee n$ and $m \rightsquigarrow_{\mathsf{p}}^* n$.

For (3), by expanding the definition we have some n_1 and n_2 such that $m_1 \rightsquigarrow_{\mathsf{p}}^* n_1$, $m_2 \rightsquigarrow_{\mathsf{p}}^* n_1$, $m_2 \rightsquigarrow_{\mathsf{p}}^* n_2$ and $m_3 \rightsquigarrow_{\mathsf{p}}^* n_2$. So by confluence (lemma 21) applied to the two middle ones, there exists some n such that $n_1 \rightsquigarrow_{\mathsf{p}}^* n$ and $n_2 \rightsquigarrow_{\mathsf{p}}^* n$. Then we have $m_1 \rightsquigarrow_{\mathsf{p}}^* n$ and $m_3 \rightsquigarrow_{\mathsf{p}}^* n$ as required. □

Lemma 23. *If $N \rightsquigarrow_{\mathsf{p}} N'$, then $[N/x]M \rightsquigarrow_{\mathsf{p}} [N'/x]M$.*

Lemma 24. *If $N \curlyvee N'$, then $[N/x]M \curlyvee [N'/x]M$.*

Proof. Expanding the definition of $\curlyvee$ we have $N \rightsquigarrow_{\mathsf{p}}^* N_1$ and $N' \rightsquigarrow_{\mathsf{p}} N_1$ for some N_1. Now repeatedly apply Lemma 23, to get $[N/x]M \rightsquigarrow_{\mathsf{p}}^* [N_1/x]M$ and $[N'/x]M \rightsquigarrow_{\mathsf{p}}^* [N_1/x]M$. □

Lemma 25. *If $m \rightsquigarrow_{\mathsf{p}} m'$, then $\mathsf{FV}(m') \subseteq \mathsf{FV}(m)$.*

B.3 Structural properties

Lemma 26 (Free variables in typing judgments)**.** *If* $H \vdash m : M$*, then* $\mathsf{FV}(m) \subseteq \mathsf{dom}(H)$ *and* $\mathsf{FV}(M) \subseteq \mathsf{dom}(H)$.

Lemma 27 (Regularity for contexts)**.** *If* $H \vdash m : M$ *then* $\vdash H$.

Lemma 28 (Regularity for variable lookup)**.** *If* $H_1, x : M, H_2 \vdash n : N$*, then* $H_1 \vdash M : \star$.

Lemma 29 (Context conversion)**.** *If* $H, x : M, H' \vdash n : N$ *and* $H \vdash \mathsf{join} : M = M'$ *and* $H \vdash M' : \star$*, then* $H, x : M', H' \vdash n : N$.

Lemma 30 (Substitution)**.** *Suppose* $H_1 \vdash u_1 : M_1$*. Then,*

- *If* $H_1, x_1 : M_1, H_2 \vdash m : M$*, then* $H_1, [u_1/x_1]H_2 \vdash [u_1/x_1]m : [u_1/x_1]M$.
- *If* $\vdash H_1, x_1 : M_1, H_2$*, then* $\vdash H_1, [u_1/x_1]H_2$.

Lemma 31 (Regularity)**.** *If* $H \vdash m : M$*, then* $H \vdash M : \star$.

Lemma 32 (Data constructors are unique in the environment)**.** *If* $\vdash H$*, and* $\mathsf{data}\, D\, \Xi\, \mathsf{where}\, \{\, \overline{d_i : \Xi_i \to D\, \Xi}^{i \in 1..j} \,\} \in H$ *and* $\mathsf{data}\, D'\, \Xi'\, \mathsf{where}\, \{\, \overline{d'_i : \Xi'_i \to D'\, \Xi'}^{i \in 1..j} \,\} \in H$*, and* $d_k = d'_l$*, then* $D = D'$ *and* $\Xi = \Xi'$ *and* $\Xi_k = \Xi'_l$.

B.4 Inversion Lemmas

We need one inversion lemma for each introduction form that has a computationally irrelevant eliminator. These proofs are all similar, so we only show the representative case for λ.

We first need some basic facts about equality.

Lemma 33 (Inversion for equality)**.** *If* $H \vdash m = n : M$ *then,* $H \vdash m : \star$ *and* $H \vdash n : \star$.

Proof. Induction of $H \vdash m = n : M$. The only cases where the subject of the conclusion of the rule is an equality are ET_EQ (where we get the result as a premise to the rule) and ET_CONV (direct by induction). □

Lemma 34 (Proof irrelevance for equality proofs)**.** *If* $H \vdash u : M = N$*, then* $H \vdash \mathsf{join} : M = N$

Proof. By regularity (lemma 31) we know $H \vdash M = N : \star$, so by inversion (lemma 33) we have $H \vdash M : \star$. So by ET_TJOIN, $H \vdash \mathsf{join} : M = M$.

Now by ET_TCONV we get $H \vdash \mathsf{join} : M = N$ by using the assumed proof u to change M to N. □

Lemma 35 (Propositional equality is an equivalence relation)**.**

- *If* $H \vdash m : M$*, then* $H \vdash \mathsf{join} : m = m$.
- *If* $H \vdash u : m_1 = m_2$*, then* $H \vdash \mathsf{join} : m_2 = m_1$.
- *If* $H \vdash u : m_1 = m_2$ *and* $H \vdash u' : m_2 = m_3$*, then* $H \vdash \mathsf{join} : m_1 = m_3$.

Proof. (1) is just a special case of ET_JOIN.

(2) We have $H \vdash \mathsf{join} : m_1 = m_1$, so we can use the assumed proof to change the left m_1 to an m_2.

(3) Use the assumed proof u to change the type of u'. □

Lemma 36 (Inversion for λ)**.** *If* $H \vdash \lambda x.n : M$*, then* $H \vdash \mathsf{join} : (x{:}M_1) \to N_1 = M$ *for some* M_1 *and* N_1*, and* $H, x : M_1 \vdash n : N_1$.

Proof. By induction on $H \vdash \lambda x.n : M$. Only two typing rules can have a λ as the subject of the conclusion.

Case ET_ABS The rule looks like

$$\frac{H, x : M \vdash n : N}{H \vdash \lambda x.n : (x\!:\!M) \to N}\,\text{ET\_ABS}$$

By ET_JOIN we have $H \vdash \mathsf{join} : (x\!:\!M) \to N = (x\!:\!M) \to N$, and we have $H, x : M \vdash n : N$ as a premise to the rule.

Case ET_CONV The rule looks like

$$\frac{\begin{array}{l} H \vdash u_1 : M_1 = N_1 \quad \ldots \quad H \vdash u_i : M_i = N_i \\ H \vdash m : [M_1/x_1] \ldots [M_i/x_i]M \\ H \vdash [N_1/x_1] \ldots [N_i/x_i]M : \star \end{array}}{H \vdash m : [N_1/x_1] \ldots [N_i/x_i]M}\,\text{ET\_CONV}$$

By IH we get that $[M_1/x_1] \ldots [M_i/x_i]M$ is propositionally equal to an arrow type, with n being typeable at the "unwrapping" of that type. So if we can show that $[N_1/x_1] \ldots [N_i/x_i]M$ is propositionally equal to that same arrow type, then we are done.

But note that by regularity, inversion and reflexivity (lemmas 31, 33, 35) we have $H \vdash \mathsf{join} : [M_1/x_1] \ldots [M_i/x_i]M = [M_1/x_1] \ldots [M_i/x_i]M$. By applying ET_CONV using the proof $u_1 \ldots u_i$ we get $H \vdash \mathsf{join} : [M_1/x_1] \ldots [M_i/x_i]M = [N_1/x_1] \ldots [N_i/x_i]M$. Then by transitivity (lemma 35) we have that $[N_1/x_1] \ldots [N_i/x_i]M$ is propositionally equal to the arrow type as required.

□

The remaining inversion lemmas follow a similar pattern, so we omit the proofs.

Lemma 37 (Inversion for irrelevant λ)**.** *If $H \vdash \lambda[].n : M$, then $H \vdash \mathsf{join} : [x\!:\!M_1] \to N_1 = M$ for some M_1 and N_1, and $H, x : M_1 \vdash n : N_1$ where $x \notin \mathsf{FV}(n)$.*

Lemma 38 (Inversion for rec)**.** *If $H \vdash \mathsf{rec} f.u : M$, then $H \vdash \mathsf{join} : M = M_1$ and $H, f : M_1 \vdash u : M_1$ for some M_1 such that $H \vdash M_1 : \star$ and M_1 is an relevant or irrelevant arrow type.*

Lemma 39 (Inversion for dcon)**.** *If $H \vdash d\,\overline{m_i} : M$, then $H \vdash \mathsf{join} : D\,\overline{N_i} = M$ for some $\overline{N_i}$ such that:*

- $\mathsf{data}\, D\, \Xi\, \mathsf{where}\, \{\, \overline{d_i : \Xi_i \to D\, \Xi}^{\,i \in 1..j}\, \} \in H$ *and d is d_l for one of the constructors in the declaration.*
- $H \vdash \overline{N_i} : \Xi$
- $H \vdash \overline{m_i} : [\overline{N_i}/\Xi]\Xi_l$

B.5 Preservation

Lemma 40 (A conversion rule for value lists)**.** *If $H \vdash \overline{u_i} : [\overline{M_i}/\overline{y_i}]\Xi$ and $\forall i.\ H \vdash \mathsf{join} : M_i = N_i$ and $\vdash H, [\overline{N_i}/\overline{y_i}]\Xi$, then $H \vdash \overline{u_i} : [\overline{N_i}/\overline{y_i}]\Xi$.*

Proof. We proceed by induction on the structure of Ξ.

Case empty. Trivial.

Case $(x : M)\Xi$**.** By inversion on the assumed judgments, we know

$$\frac{\begin{array}{l} H \vdash u : [\overline{M_i / y_i}]M \\ H \vdash [\overline{M_i / y_i}]M : \star \\ H \vdash \overline{u_i} : [u/x][\overline{M_i / y_i}]\Xi \end{array}}{H \vdash u\,\overline{u_i} : (x : [\overline{M_i / y_i}]M)[\overline{M_i / y_i}]\Xi}\ \textsc{etl\_cons}$$

and

$$\vdash H, x : [\overline{N_i / y_i}]M, [\overline{N_i / y_i}]\Xi.$$

By inversion on this, we have $H \vdash [\overline{N_i / y_i}]M : \star$.

Now since we know $\forall i.\ H \vdash \mathsf{join} : M_i = N_i$ and $H \vdash [\overline{N_i / y_i}]M : \star$, then by ET_CONV we have $H \vdash u : [\overline{N_i / y_i}]M$.

By substitution (lemma 30) we get $\vdash H, [u/x][\overline{N_i / y_i}]\Xi$. We know u is well-typed, so by ET_JOIN we have $H \vdash \mathsf{join} : u = u$. Then by IH, taking the multi-substitution to be $[u/x][\overline{M_i / y_i}]$, we get $H \vdash \overline{u_i} : [u/x][\overline{N_i / y_i}]\Xi$. So re-applying ETL_CONS we get

$$H \vdash u\,\overline{u_i} : (x : [\overline{N_i / y_i}]M)[\overline{N_i / y_i}]\Xi$$

as required.

Case $[x : M]\Xi$**.** This case is similar. Inversion on the first assumed judgement now gives

$$\frac{\begin{array}{l} H \vdash u : [\overline{M_i / y_i}]M \\ H \vdash [\overline{M_i / y_i}]M : \star \\ H \vdash \overline{u_i} : [u/x][\overline{M_i / y_i}]\Xi \end{array}}{H \vdash [\,]\,\overline{u_i} : (x : [\overline{M_i / y_i}]M)[\overline{M_i / y_i}]\Xi}\ \textsc{etl\_cons}$$

By reasoning as in the previous case we get $H \vdash u : [\overline{N_i / y_i}]M$ and $H \vdash [\overline{N_i / y_i}]M : \star$ and $H \vdash \overline{u_i} : [u/x][\overline{N_i / y_i}]\Xi$. Then re-apply ETL_ICONS.

□

Theorem 41 (Preservation)**.**

1. *If $H \vdash m : M$ and $m \rightsquigarrow_{\mathsf{p}} m'$, then $H \vdash m' : M$.*
2. *If $H \vdash \overline{m_i} : [n_1/y_1] \ldots [n_l/y_l]\Xi$ and $\forall i.\ m_i \rightsquigarrow_{\mathsf{p}} m_i'$ and $\forall j.\ n_j \rightsquigarrow_{\mathsf{p}} n_j'$, then $H \vdash \overline{m_i}' : [n_1'/y_1] \ldots [n_l'/y_l]\Xi$.*

Proof. By mutual induction on the two judgments. The cases for $H \vdash m : M$ are:

Cases ET_TYPE, ET_VAR, ET_ABORT, ET_JOIN, ET_INJDOM, ET_INJRNG, ET_IINJDOM, ET_IINJRNG, ET_INJTCON.

These expressions can not step except by SP_REFL, so the result is trivial.

Case ET_CASE. The rule looks like

$$\frac{\begin{array}{l} \Gamma \vdash b : D\,\overline{B_i} \\ \Gamma \vdash A : \star \\ \mathsf{data}\,D\,\Delta\,\mathsf{where}\,\{\overline{d_i : \Delta_i \to\ D\,\Delta}^{i \in 1..l}\} \in \Gamma \\ \forall i.\ \Gamma, [\overline{B_i}/\Delta]\Delta_i, y : b = d_i\,\Delta_i \vdash a_i : A \\ \forall i.\ \{y\} \cup dom-(\Delta_i)\ \#\ \mathsf{FV}(|a_i|) \end{array}}{\Gamma \vdash \mathsf{case}\,b\,\mathsf{as}\,[y]\,\mathsf{of}\,\{\overline{d_i\,\Delta_i \Rightarrow a_i}^{i \in 1..l}\} : A}\ \textsc{t\_case}$$

We consider the ways the expression $\mathsf{case}\,n\,\mathsf{of}\,\{\overline{d_j\,\overline{x_{ij}} \Rightarrow m_j}^{j \in 1..k}\}$ may step:

- To $\mathsf{case}\, n' \,\mathsf{of}\, \{\overline{d_j \overline{x_{ij}} \Rightarrow m'_j}^{j\in 1..k}\}$ by SP_CASE when $n \rightsquigarrow_{\mathsf{p}} n'$ and $\forall j.\ m_j \rightsquigarrow_{\mathsf{p}} m'_j$.
  By IH we get $H \vdash n : D\overline{n_i}$. Also by IH, for each j we have

  $$H, [\overline{n_i}/\Xi]\Xi_j, y : n = d_j\,\Xi_j \vdash m'_j : M.$$

  Now by regularity (lemma 28) and inversion (lemma 33) we know that $d_j\,\Xi_j$ is welltyped in the context $H, [\overline{n_i}/\Xi]\Xi_j$. And we already observed that n and n' are welltyped. So $n = d_j\,\Xi_j$ and $n' = d_j\,\Xi_j$ are wellformed equations. Since $n = d_j\,\Xi_j \rightsquigarrow_{\mathsf{p}} n' = d_j\,\Xi_j$, by ET_JOIN we have $H, [\overline{n_i}/\Xi]\Xi_j \vdash \mathsf{join} : (n = d_j\,\Xi_j) = (n' = d_j\,\Xi_j)$. So by context conversion (lemma 29) we have

  $$H, [\overline{n_i}/\Xi]\Xi_j, y : n' = d_j\,\Xi_j \vdash m'_j : M.$$

  Then we can re-apply ET_CASE to get the required

  $$H \vdash \mathsf{case}\, n' \,\mathsf{of}\, \{\overline{d_j \overline{x_{ij}} \Rightarrow m'_j}^{j\in 1..k}\} : M.$$

- To $[\overline{u_i}'/\overline{x_{il}}]m'_l$ by SP_CASEBETA when n is $d_l\,\overline{u_i}$, and $\forall i.\ u_i \rightsquigarrow_{\mathsf{p}} u'_i$ and $m_l \rightsquigarrow_{\mathsf{p}} m'_l$. Notice that the step rule in particular requires that d_l is one of the branches of the case expression.
  By inversion (lemma 39) on the premise $H \vdash d\,\overline{u_i} : D\overline{n_i}$, we know that $H \vdash n : D\overline{N_i}$ with $H \vdash \mathsf{join} : D\overline{N_i} = D\overline{n_i}$, and $H \vdash \overline{N_i} : \Xi$ and $H \vdash \overline{u_i} : [\overline{N_i}/\Xi]\Xi_i$. (We know that the D, Ξ and Ξ_i that come out of the lemma are the same as the ones in the typing rule because data constructors have a unique definition in the context (lemma 32)).
  By the rule INJTCON we get $H \vdash \mathsf{join} : N_i = n_i$ for each i. So by value-list conversion (lemma 40) we have $H \vdash \overline{u_i} : [\overline{n_i}/\Xi]\Xi_l$.
  We next claim that $H, y : d_l\,\overline{u_i} = d_l\,\overline{u_i}' \vdash [\overline{u_i}'/\Xi_l]m'_l : M$. To show this we prove a more general claim: for any prefix $u'_1 \ldots u'_k$ of $\overline{u_i}$, and supposing Ξ_l has the form $(x_1 : M_1)\ldots(x_k : M_k)\Xi_0$, we have

  $$H, [u'_1/x_1]\ldots[u'_k/x_k][\overline{n_i}/\Xi]\Xi_0, y : [u'_1/x_1]\ldots[u'_k/x_k](d_l\,\overline{u_i} = d_l\,\Xi_l) \vdash [u'_1/x_1]\ldots[u'_k/x_k]m_l : [u'_1/x_1]\ldots[u'_k/x_k]M$$

  This follows by induction on k (by applying substitution, lemma 30, k times). So in particular, we have

  $$H, y : [\overline{u_i}'/\Xi_l](d_l\,\overline{u_i} = d_l\,\Xi_l) \vdash [\overline{u_i}'/\Xi_l]m_l : [\overline{u_i}'/\Xi_l]M$$

  But by the premises $H \vdash M : \star$ and $H \vdash d_l\,\overline{u_i} : D\overline{n_i}$ together with lemma 26 we know that x_l are not free in $d_l\,\overline{u_i}$ or M, so this simplifies to

  $$H, y : d_l\,\overline{u_i} = d_l\,\overline{u_i}' \vdash [\overline{u_i}'/\Xi_l]m'_l : M$$

  as we claimed.
  Next, we know $d_l\,\overline{u_i}$ is well-typed because that is a premise to the rule. By the mutual IH we have that $H \vdash d_l\,\overline{u_i}' : D\overline{n_i}$, so $d_l\,\overline{u_i}'$ is well-typed too. So by ET_JOIN we have $H \vdash \mathsf{join} : d_l\,\overline{u_i} = d_l\,\overline{u_i}'$. Then by substitution (lemma 30) again we have

  $$H \vdash [\mathsf{join}/y][\overline{u_i}'/\Xi_l]m'_l : [\mathsf{join}/y]M.$$

  But as a side-condition to the rule (plus lemma 25) we know that $y \notin \mathsf{FV}(m'_l)$, and y is a bound variable which we can pic so that $y \notin \mathsf{FV}(M)$. So we have in fact show the required

  $$H \vdash [\overline{u_i}'/\Xi_l]m'_l : M.$$

- To abort by SP_ABORT. By regularity (lemma 31) on the original typing derivation we know that $H \vdash M : \star$, so by ET_ABORT we have $H \vdash \mathsf{abort} : M$ as required.

Case ET_PI. The rule looks like

$$\frac{H \vdash M : \star \quad H, x : M \vdash N : \star}{H \vdash (x{:}M) \to N : \star}\text{ET\_PI}$$

The only way $(x{:}M) \to N$ can step (except trivially by SP_REFL) is by SP_PI:

$$(x{:}M) \to N \rightsquigarrow_{\mathsf{p}} (x{:}M') \to N' \qquad \text{where } M \rightsquigarrow_{\mathsf{p}} M' \text{ and } N \rightsquigarrow_{\mathsf{p}} N'$$

We must show $H \vdash (x{:}M') \to N' : \star$.

By IH we immediately get $H \vdash M' : \star$ and $H, x : M \vdash N' : \star$. Since $M \rightsquigarrow_{\mathsf{p}} M'$ we also have $M \curlyvee M'$, so applying ET_JOIN we get $H \vdash \mathsf{join} : M = M'$. Then by context conversion (lemma 29) we get $H, x : M' \vdash N' : \star$. We conclude by re-applying ET_PI.

Case ET_IPI Similar to the previous case.

Case ET_ABS . The rule looks like

$$\frac{H, x : M \vdash n : N}{H \vdash \lambda x.n : (x{:}M) \to N}\text{ET\_ABS}$$

The only non-trivial way the expression $\lambda x.n$ can step is by SP_ABS to $\lambda x.n'$ when $n \rightsquigarrow_{\mathsf{p}} n'$. By IH we get $H, x : M \vdash n' : N$. So re-applying ET_ABS we get $H \vdash \lambda x.n' : (x{:}M) \to N$ as required.

Cases ET_IABS, ET_REC.

These are similar to the previous case. For IABS, note that the free variable condition is preserved by lemma 25.

Case ET_TCON. The rule looks like

$$\frac{\begin{array}{l}\mathsf{data}\, D\, \Xi\, \mathsf{where}\, \{\overline{d_i : \Xi_i \to\ D\,\Xi}^{i \in 1..j}\} \in H \\ H \vdash \overline{M_i} : \Xi^{+}\end{array}}{H \vdash D\overline{M_i} : \star}\text{ET\_TCON}$$

The only way the expression can step is by SP_TCON, so $\forall i.\ M_i \rightsquigarrow_{\mathsf{p}} M_i'$. By the mutual IH, we get $H \vdash \overline{M_i}' : \Xi^{+}$. So by re-applying ET_TCON we have $H \vdash D\overline{M_i} : \star$ as required.

Case ET_ABSTCON. Similar to the previous case.

Case ET_DCON. The rule looks like

$$\frac{\begin{array}{l}\mathsf{data}\, D\, \Xi\, \mathsf{where}\, \{\overline{d_i : \Xi_i \to\ D\,\Xi}^{i \in 1..j}\} \in H \\ H \vdash \overline{M_i} : \Xi^{+} \\ H \vdash \overline{m_i} : [\overline{M_i}/\Xi]\Xi_i\end{array}}{H \vdash d_k\, \overline{m_i} : D\overline{M_i}}\text{ET\_DCON}$$

By the mutual induction hypothesis (with an empty substitution) we get $H \vdash \overline{M_i} : \Xi$ and $H \vdash \overline{m_i} : [\overline{M_i}/\Xi]\Xi_i$. Conclude by re-applying ET_DCON.

Case ET_APP. The rule looks like

$$\frac{\begin{array}{l} H \vdash m : (x{:}M) \to N \\ H \vdash n : M \\ H \vdash [n/x]N : \star \end{array}}{H \vdash m\, n : [n/x]N}\text{ET\_APP}$$

We consider how the expression $m\,n$ may step.

- To $m'\,n'$ by SP_APP if $m \rightsquigarrow_{\mathsf{p}} m'$ and $n \rightsquigarrow_{\mathsf{p}} n'$.
  By IH we have $H \vdash m' : (x{:}M) \to N$ and $H \vdash n' : M$. By lemma 23 we know $[n/x]N \rightsquigarrow_{\mathsf{p}} [n'/x]N$, so also by IH we have $H \vdash [n'/x]N : \star$. So re-applying ET_APP we get $H \vdash m'\,n' : [n'/x]N$.
  Finally, by ET_JOIN we have $H \vdash \mathsf{join} : [n/x]N = [n'/x]N$, and hence by ET_CONV we get $H \vdash m'\,n' : [n/x]N$ as required.
- To $[u'/x]m_1'$ by SP_APPBETA if m is $\lambda x.m_1$ and n is u, and $m_1 \rightsquigarrow_{\mathsf{p}} m_1'$ and $u \rightsquigarrow_{\mathsf{p}} u'$.
  By IH we have $H \vdash u' : M$. Also, $\lambda x.m_1 \rightsquigarrow_{\mathsf{p}} \lambda x.m_1'$ so by IH we have $H \vdash \lambda x.m_1' : (x{:}M) \to N$. By inversion (lemma 36) we know that $H, x : M_1 \vdash m_1 : N_1$ for some M_1, N_1 such that $H \vdash \mathsf{join} : (x{:}M_1) \to N_1 = (x{:}M) \to N$. By ET_INJDOM we have $H \vdash \mathsf{join} : M_1 = M$, and byregularity (lemma 28) we have $H \vdash M_1 : \star$, so by ET_CONV we get $H \vdash u' : M_1$. Now by substitution (lemma 30) we get

  $$H \vdash [u'/x]m_1' : [u'/x]N_1.$$

  Now, by ET_INJRNG we have $H \vdash \mathsf{join} : [u'/x]N_1 = [u'/x]N$. Also, by lemma 23 we know $[u/x]N \rightsquigarrow_{\mathsf{p}} [u'/x]N$, and we noted above that $H \vdash [u'/x]N : \star$, so by ET_JOIN we have $H \vdash \mathsf{join} : [u/x]N = [u'/x]N$. By symmetry and transitivity (lemma 35) this yields $H \vdash \mathsf{join} : [u'/x]N_1 = [u/x]N$.
  Finally, we had $H \vdash [u/x]N : \star$ as a premise to the rule. So by ET_CONV we get the required

  $$H \vdash [u'/x]m_1 : [u/x]N.$$

- To $([\mathsf{rec} f.u_1'/f]u_1')\; u_2'$ by SP_APPREC if m is $\mathsf{rec} f.u_1$, n is u_2, and $u_1 \rightsquigarrow_{\mathsf{p}} u_1'$ and $u_2 \rightsquigarrow_{\mathsf{p}} u_2'$.
  By IH we have $H \vdash u_2 : M$. Also, since $\mathsf{rec} f.u_1 \rightsquigarrow_{\mathsf{p}} \mathsf{rec} f.u_1'$, by IH we have $H \vdash \mathsf{rec} f.u_1' : (x{:}M) \to N$. And since by lemma 23 $[u_2/x]N \rightsquigarrow_{\mathsf{p}} [u_2'/x]N$, by IH we get $H \vdash [u_2'/x]N : \star$.
  By inversion (lemma 38 we know $H, f : M_1 \vdash u_1' : M_1$ for some M_1 such that $H \vdash \mathsf{join} : M_1 = (x{:}M) \to N$ and $H \vdash M_1 : \star$ and such that M_1 is an arrow type.
  So by the ET_REC rule, we get $H \vdash \mathsf{rec} f.u_1' : M_1$. Then by substitution (lemma 30) we have $H \vdash [\mathsf{rec} f.u_1'/f]u_1' : M_1$.
  By regularity (lemma 31) on the original premise of the rule we know $H \vdash (x{:}M) \to N : \star$, so by ET_CONV we have $H \vdash [\mathsf{rec} f.u_1'/f]u_1' : (x{:}M) \to N$. Then re-apply ET_APP to get

  $$H \vdash ([\mathsf{rec} f.u_1'/f]u_1')\; u_2' : [u_2'/x]N.$$

  As we noted above $[u_2/x]N \rightsquigarrow_{\mathsf{p}} [u_2'/x]N$, and both expressions are well-kinded, so by ET_JOIN we know $H \vdash \mathsf{join} : [u_2'/x]N = [u_2/x]N$. So by finally applying ET_CONV we get the required

  $$H \vdash ([\mathsf{rec} f.u_1'/f]u_1')\; u_2' : [u_2/x]N.$$

- To abort by SP_ABORT. By regularity (lemma 31) on the original premise we know $H \vdash [n/x]N : \star$. So by ET_ABORT we have $H \vdash \mathsf{abort} : [n/x]N$ as required.

Case ET_IAPP. The typing rule looks like

$$\frac{\begin{array}{l} H \vdash m : [x{:}M] \to N \\ H \vdash u : M \end{array}}{H \vdash m[] : [u/x]N}\ \text{ET\_IAPP}$$

We consider how the expression $m[]$ may step:

- To $m'[]$ by SP_IAPP if $m \rightsquigarrow_{\mathsf{p}} m'$. By IH we know $H \vdash m' : [x{:}M] \to N$, so by re-applying ET_IAPP we get $H \vdash m'[] : [u/x]N$ as required.
- To m_1' by SP_IAPPBETA if m is $\lambda[].m_1$ and $m_1 \rightsquigarrow_{\mathsf{p}} m_1'$.
  Note that $\lambda[].m_1 \rightsquigarrow_{\mathsf{p}} \lambda[].m_1'$, so by IH we get $H \vdash \lambda[].m' : [x{:}M] \to N$. Then by inversion (lemma 37) we know $H, x : M_1 \vdash n : N_1$ for some M_1 and N_1 with $H \vdash \mathsf{join} : ([x{:}M_1] \to N_1) = ([x{:}M] \to N)$, and $x \notin \mathsf{FV}(m_1')$.
  Now, we have $H \vdash u : M$ as an assumption to the rule. By regularity (lemma 31) on that assumption we get $H \vdash M : \star$, and by ET_IINJDOM we have $H \vdash \mathsf{join} : M_1 = M$. So by ET_CONV we get $H \vdash u : M_1$. Then by substitution (lemma 30) we get

  $$H \vdash [u/x]n : [u/x]N_1.$$

  Since we know x is not free in n this is the same as saying $H \vdash n : [u/x]N_1$. Furthermore, by ET_IINJDOM we get $H \vdash \mathsf{join} : [u/x]N_1 = [u/x]N$, and by regularity on the original derivation we have $H \vdash [u/x]N : \star$. So by ET_CONV we get the required

  $$H \vdash m' : [u/x]N.$$

- To $([\mathsf{rec} f.u_1'/f]u_1')[]$ by SP_IAPPREC if m is $\mathsf{rec} f.u_1$ and $u_1 \rightsquigarrow_{\mathsf{p}} u_1'$.
  Note that $\mathsf{rec} f.u_1 \rightsquigarrow_{\mathsf{p}} \mathsf{rec} f.u_1'$, so by IH we know $H \vdash \mathsf{rec} f.u_1' : [x{:}M] \to N$. By inversion (lemma 38) we get that $H, f : M_1 \vdash u_1' : M_1$ for some arrow type M_1 such that $H \vdash \mathsf{join} : M_1 = [x{:}M] \to N$ and $H \vdash M_1 : \star$. By ET_REC we then have $H \vdash \mathsf{rec} f.u_1' : M_1$, hence by substitution (lemma 30) we have

  $$H \vdash [\mathsf{rec} f.u_1'/f]u_1' : M_1.$$

  By regularity (lemma 31) applied to the original typing rule we know $H \vdash [x{:}M] \to N : \star$, so by ET_CONV we then have

  $$H \vdash [\mathsf{rec} f.u_1'/f]u_1' : [x{:}M] \to N.$$

  So re-applying ET_IAPP we get the required

  $$H \vdash ([\mathsf{rec} f.u_1'/f]u_1')[] : [u/x]N.$$

- To abort by SP_ABORT.
  By regularity (lemma 31) applied to the original type rule we know $H \vdash [u/x]N : \star$, so by ET_ABORT we have

  $$H \vdash \mathsf{abort} : [u/x]N$$

  as required.

Case ET_EQ. The rule looks like

$$\frac{H \vdash m : M \quad H \vdash n : N}{H \vdash m = n : \star}\text{ET\_EQ}$$

The only non-trivial way the expression $m = n$ can step is by SP_EQ to $m' = n'$, when $m \rightsquigarrow_{\mathsf{p}} m'$ and $n \rightsquigarrow_{\mathsf{p}} n'$. By IH we immediatly get $H \vdash m : M$ and $H \vdash n : N$, and we conclude by re-applying SP_EQ.

Case ET_CONV. The rule looks like

$$\frac{\begin{array}{l} H \vdash u_1 : M_1 = N_1 \quad \dots \quad H \vdash u_i : M_i = N_i \\ H \vdash m : [M_1/x_1]\dots[M_i/x_i]M \\ H \vdash [N_1/x_1]\dots[N_i/x_i]M : \star \end{array}}{H \vdash m : [N_1/x_1]\dots[N_i/x_i]M}\text{ET\_CONV}$$

and we know that $m \rightsquigarrow_{\mathsf{p}} m'$. Directly by IH we get $H \vdash m' : [M_1/x_1]\dots[M_i/x_i]M$, and conclude by re-applying ET_CONV.

The cases for $H \vdash \overline{m_i} : \Xi$ are:

Case ETL_EMPTY. Trivial.

Case ETL_CONS. After pushing in the substitution, the rule looks like:

$$\frac{\begin{array}{l} H \vdash m : [n_1/y_1]\dots[n_i/y_i]M \\ H \vdash [n_1/y_1]\dots[n_i/y_i]M : \star \\ H \vdash \overline{m_i} : [m/x][n_1/y_1]\dots[n_i/y_i]\Xi \end{array}}{H \vdash m\,\overline{m_i} : (x : [n_1/y_1]\dots[n_i/y_i]M)[n_1/y_1]\dots[n_i/y_i]\Xi}\text{ETL\_CONS}$$

By mutual IH we have $H \vdash m' : [n_1/y_1]\dots[n_i/y_i]M$.

By repeatedly applying lemma 23 we know $[n_1/y_1]\dots[n_i/y_i]M \rightsquigarrow_{\mathsf{p}} [n'_1/y_1]\dots[n'_i/y_i]M$, so by mutual IH we get $H \vdash [n'_1/y_1]\dots[n'_i/y_i]M : \star$.

By ET_JOIN we then have $H \vdash \mathsf{join} : [n_1/y_1]\dots[n_i/y_i]M = [n'_1/y_1]\dots[n'_i/y_i]M$, so by ET_CONV we get $H \vdash m : [n'_1/y_1]\dots[n'_i/y_i]M$.

Finally, by the IH (using that $m \rightsquigarrow_{\mathsf{p}} m'$) we have $H \vdash \overline{m_i}' : [m'/x][n'_1/y_1]\dots[n'_i/y_i]\Xi$. So re-applying ETL_CONS we get the required

$$H \vdash m'\,\overline{m_i}' : [n'_1/y_1]\dots[n'_i/y_i]\Xi.$$

Case ETL_ICONS. After pushing in the substitution, the rule looks like:

$$\frac{\begin{array}{l} H \vdash u : [n_1/y_1]\dots[n_i/y_i]M \\ H \vdash [n_1/y_1]\dots[n_i/y_i]M : \star \\ H \vdash \overline{m_i} : [u/x][n_1/y_1]\dots[n_i/y_i]\Xi \end{array}}{H \vdash [\,]\,\overline{m_i} : [x : [n_1/y_1]\dots[n_i/y_i]M][n_1/y_1]\dots[n_i/y_i]\Xi}\text{ETL\_ICONS}$$

By repeatedly applying lemma 23 we know $[n_1/y_1]\dots[n_i/y_i]M \rightsquigarrow_{\mathsf{p}} [n'_1/y_1]\dots[n'_i/y_i]M$, so by mutual IH we get $H \vdash [n'_1/y_1]\dots[n'_i/y_i]M : \star$.

By ET_JOIN we then have $H \vdash \mathsf{join} : [n_1/y_1]\dots[n_i/y_i]M = [n'_1/y_1]\dots[n'_i/y_i]M$, so by ET_CONV we get $H \vdash u : [n'_1/y_1]\dots[n'_i/y_i]M$.

Finally, by the IH (using that $u \rightsquigarrow_{\mathsf{p}} u$, reflexively) we have $H \vdash \overline{m_i}' : [u/x][n'_1/y_1]\dots[n'_i/y_i]\Xi$. So re-applying ETL_CONS we get the required

$$H \vdash [\,]\,\overline{m_i}' : [n'_1/y_1]\dots[n'_i/y_i]\Xi.$$

□

B.6 Progress

Lemma 42 (Soundness of equality). *If $H_D \vdash u : M$ and $M \curlyvee (m_1 = n_1)$, then $m_1 \curlyvee n_1$.*

Proof. **Cases ET_TYPE, ET_PI, ET_IPI, ET_TCON, ET_ABSTCON, ET_DCON, ET_ABS,ET_IABS, ET_REC, ET_EQ.** The M in the conclusion of these rules have a defined head constructor, which is not $=$. So by lemma 16, M cannot be joinable with $m_1 = n_1$.

Cases ET_CASE, ET_APP, ET_IAPP, ET_ABORT. These expressions are not values.

Case ET_VAR. This is impossible in an H_D context, since it doesn't contain any variable declarations.

Case ET_JOIN. The rule looks like

$$\frac{\begin{array}{l} m \curlyvee n \\ H \vdash m = n : \star \end{array}}{H \vdash \mathsf{join} : m = n}\text{ET\_JOIN}$$

By injectivity (lemma 17) we have have $m \curlyvee m_1$ and $n \curlyvee n_1$. And by assumption we have $m \curlyvee n$. So by symmetry and transitivity (lemma 22) we have $m_1 \curlyvee n_1$ as required.

Case ET_CONV. The rule looks like

$$\frac{\begin{array}{l} H \vdash u_1 : M_1 = N_1 \quad \dots \quad H \vdash u_i : M_i = N_i \\ H \vdash m : [M_1/x_1]\dots[M_i/x_i]M \\ H \vdash [N_1/x_1]\dots[N_i/x_i]M : \star \end{array}}{H \vdash m : [N_1/x_1]\dots[N_i/x_i]M}\text{ET\_CONV}$$

and we are given that that $[N_1/x_1]\dots[N_i/x_i]M \curlyvee (m_1 = n_1)$. It suffices to show that $[M_1/x_1]\dots[M_i/x_i]M \curlyvee (m_1 = n_1)$, and then the result follows by the IH for m.

But by the IH for u_i we know $M_i \curlyvee N_i$, so we get this by repeatedly applying lemma 24.

Case ET_INJRNG. The rule looks like

$$\frac{\begin{array}{l} H \vdash u_1 : (x{:}M) \to N_1 = (x{:}M) \to N_2 \\ H \vdash u : M \end{array}}{H \vdash \mathsf{join} : [u/x]N_1 = [u/x]N_2}\text{ET\_INJRNG}$$

and we are given that $([u/x]N_1 = [u/x]N_2) \curlyvee (m_1 = n_1)$.

By IH we get $(x{:}M) \to N_1 \curlyvee (x{:}M) \to N_2$, so by injectivity (lemma 17) we know $N_1 \curlyvee N_2$. Then by lemma 19 we get $[u/x]N_1 \curlyvee [u/x]N_2$ as required.

Case ET_INJDOM, ET_IINJDOM, ET_IINJRNG, ET_INJTCON. Similar to the previous case.

□

Lemma 43 (Canonical forms). *Suppose $H_D \vdash u : M$. Then:*

1. *If $M \curlyvee (x{:}M_1) \to M_2$, then u is either $\lambda x.u_1$ or $\mathsf{rec}\, f.u_1$.*
2. *If $M \curlyvee [x{:}M_1] \to M_2$, then u is either $\lambda[].u_1$ or $\mathsf{rec}\, f.u_1$.*
3. *If $M \curlyvee D\overline{M_i}$ then u is $d\overline{u_i}$, where $\mathsf{data}\, D\, \Xi\, \mathsf{where}\, \{\overline{d_i : \Xi_i \to D\, \Xi}^{i \in 1..j}\} \in H_D$ and d is one of the d_j.*

Proof. By induction on $H_D \vdash u : M$. The cases are:

Cases ET_TYPE, ET_PI, ET_IPI, ET_TCON, ET_ABSTCON, ET_EQ, ET_JOIN, ET_INJRNG, ET_INJDOM, ET_IINJDOM, ET_IINJRNG, ET_INJTCON.

The M in the conclusion of these rules have a defined head constructor, which is not one of the interesting ones. So by lemma 16, M cannot be joinable with one of the interesting types.

Cases ET_CASE, ET_APP, ET_IAPP, ET_ABORT. These expressions are not values.

Case ET_VAR. This is impossible in an H_D context, since it doesn't contain any variable declarations.

Cases ET_DCON,ET_ABS,ET_IABS. The type in these expressions is joinable with one of the interesting ones, and by lemma 16 it can be joinable with at most one of them. The expression in the rule does indeed have the required form.

Case ET_REC. The rule looks like

$$\frac{\begin{array}{l} H, f : M \vdash u : M \\ H \vdash M : \star \\ M \text{ is } (x{:}M_1) \to M_2 \text{ or } [x{:}M_1] \to M_2 \end{array}}{H \vdash \mathsf{rec}\, f.u : M}\text{ET\_REC}$$

We know from the side condition to the rule that M is a relevant or irrelevant arrow. Then the expression does indeed have the required form.

Case ET_CONV. The rule looks like

$$\frac{\begin{array}{l} H \vdash u_1 : M_1 = N_1 \quad \dots \quad H \vdash u_i : M_i = N_i \\ H \vdash m : [M_1/x_1] \dots [M_i/x_i]M \\ H \vdash [N_1/x_1] \dots [N_i/x_i]M : \star \end{array}}{H \vdash m : [N_1/x_1] \dots [N_i/x_i]M}\text{ET\_CONV}$$

Suppose, for example, that $[N_1/x_1] \dots [N_i/x_i]M \curlyvee (x{:}M_1) \to N_1$. It suffices to show that $[M_1/x_1] \dots [M_i/x_i]M \curlyvee (x{:}M_1) \to N_1$, and then the result follows by the IH for m.

But by soundness of equality (lemma 42), for each u_i we know $M_i \curlyvee N_i$, so we get this by repeatedly applying lemma 24.

□

Theorem 44 (Progress). *If $H_D \vdash m : M$, then either m is a value, m is* abort*, or $m \rightsquigarrow_{\mathsf{cbv}} m'$ for some m'.*

Proof. By induction on $H_D \vdash m : M$. The cases are:

Cases ET_TYPE, ET_VAR, ET_PI, ET_IPI ET_TCON ET_ABSTCON, ET_EQ, ET_JOIN, ET_INJDOM, ET_INJRNG, ET_IINJDOM, ET_IINJRNG, ET_INJTCON, ET_ABS, ET_IABS, ET_REC.

These rules have a value as a subject.

Case ET_CASE. The typing rule looks like

$$\frac{\begin{array}{l} H \vdash n : D\overline{n_i} \\ H \vdash M : \star \\ \mathsf{data}\, D\, \Xi\, \mathsf{where}\, \{ \overline{d_i : \Xi_i \to D\, \Xi}^{i \in 1..l} \} \in H \\ \forall i.\ H, [\overline{n_i}/\Xi]\Xi_i, y : n = d_i\, \Xi_i \vdash m_i : M \\ \forall i.\ \{y\} \cup \mathsf{dom}^-(\Xi_i) \;\#\; \mathsf{FV}(m_i) \\ \overline{x_i}_i \text{ is } \mathsf{dom}^+(\Xi_i) \end{array}}{H \vdash \mathsf{case}\, n\, \mathsf{of}\, \{ \overline{d_i \overline{x_i}_i \Rightarrow m_i}^{i \in 1..l} \} : M}\text{ET\_CASE}$$

By IH, we have that n is either a value, is abort, or steps. If it steps, the entire expression steps by SC_CTX. If it is abort, the entire expression steps to abort by SC_ABORT.

Finally, suppose n is a value. By canonical forms (lemma 43) we know that it must be of the form $d\,\overline{u_i}$, and defined by a datatype declaration for D in the context. Since datatype declarations are unique (lemma 32), it must be the same datatype declaration that is mentioned in the typing rule above. So the case expression has a branch for d, and can step by SC_CASEBETA.

Case ET_DCON. The expression is $d\,\overline{m_i}$. By IH, each of the m_i is a values, is abort, or steps. If they are all values, the entire expression is a value. Otherwise, if the first non-value is abort the entire expression steps by SC_ABORT, and if it steps the expression steps by SC_CTX.

Case ET_APP. The rule looks like

$$\frac{\begin{array}{l} H \vdash m : (x{:}M) \to N \\ H \vdash n : M \\ H \vdash [n/x]N : \star \end{array}}{H \vdash m\,n : [n/x]N}\text{ET\_APP}$$

By IH, m and n either, step, are abort is are values. If m steps, the entire expression steps by SC_CTX. If it is abort the entire expression steps by SC_ABORT. So in the following we can assume it is a value.

By similar reasoning, we can assume n is a value.

Now, by canonical forms (lemma 43) we know that m is either $\lambda x.m_1$ or $\mathsf{rec}\,f.u_1$. If it is $\lambda x.m_1$ the entire expression steps to $[n/x]m_1$ by SC_APPBETA, while if it is $\mathsf{rec}\,f.u_1$ the entire expression steps to $([\mathsf{rec}\,f.u_1/f]u_1)\ n$ by SC_APPREC.

Case ET_IAPP. The rule looks like

$$\frac{\begin{array}{l} H \vdash m : [x{:}M] \to N \\ H \vdash u : M \end{array}}{H \vdash m[] : [u/x]N}\text{ET\_IAPP}$$

By the IH, m either steps, is abort or is a value. If m steps, then the entire expresions steps by SC_CTX and the context $\bullet[]$. If it is abort, the entire expression steps to abort by SC_ABORT and the same context.

Finally, m may be a value. In that case, by canonical forms (lemma 43), m is either $\lambda[].m_1$ or $\mathsf{rec}\,f.u_1$. If it is $\lambda[].m_1$, the entire expression steps to m_1 by SC_IAPPBETA. If it is $\mathsf{rec}\,f.u_1$, then the entire expression steps to $([\mathsf{rec}\,f.u_1/f]u_1)[]$ by SC_IAPPREC.

Case ET_ABORT. The subject of the typing rule is abort.

Case ET_CONV. Follows directly by IH.

□