2012

Irrelevance, Heterogeneous Equity, and Call-by-value Dependent Type Systems

Vilhelm Sjöberg
University of Pennsylvania

Chris Casinghino
University of Pennsylvania

Nathan Collins
Portland State University

Ki Yung Ahn
Portland State University

Tim Sheard
Portland State University

See next page for additional authors

Follow this and additional works at: http://repository.upenn.edu/cis_papers

Part of the Computer Sciences Commons

Recommended Citation


Copyright held by the authors.

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_papers/627
For more information, please contact libraryrepository@pobox.upenn.edu.
Irrelevance, Heterogeneous Equity, and Call-by-value Dependent Type Systems

Abstract
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 heterogeneous, completely erased elimination form.

Disciplines
Computer Sciences

Comments

Copyright held by the authors.

Author(s)
Vilhelm Sjoberg, Chris Casinghino, Nathan Collins, Ki Yung Ahn, Tim Sheard, Harley D. Eades III, Peng Fu, Garrin Kimmell, Aaron Stump, and Stephanie Weirich

This conference paper is available at ScholarlyCommons: http://repository.upenn.edu/cis_papers/627
Irrelevance, Heterogeneous 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

Ki Yung Ahn
University of Pennsylvania
kya@cs.pdx.edu

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

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 heterogeneous, completely erased elimination form.

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 work particularly well for nonterminating dependent languages (as we explain in section 1.1).

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” (section 2.4), and we design a new variation of the elimination rule, “n-ary conv”, to better exploit this feature (section 2.5). We also discuss how to add type annotations to this rule (section 2.6).
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

\[ \text{div} : \text{Nat} \to \text{Nat} \to \text{Nat} \]

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 \text{div}, which requires a proof of \text{div}'s precondition that the denominator be non-zero:

\[ \text{safediv} : \text{Nat} \to (y: \text{Nat}) \to (p: \text{isZero } y = \text{false}) \to \text{Nat} \]
\[ \text{safediv} = \lambda x: \text{Nat}. \lambda y: \text{Nat}. \lambda p: (\text{isZero } y = \text{false}). \text{div} x y \]

Programs written using \text{safediv} are guaranteed to not divide by zero, even though our language is inconsistent as a logic. This works because \lambda-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 `Bool = 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 [28].

While strict λ-abstractions give preconditions, strict data constructors can be used to express post-conditions. 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

```haskell
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

```haskell
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

```haskell
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 to 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 [6], but by convention we use uppercase metavariables `A`, `B` for expressions that are types. Some of the expressions are standard: the type of types ⋆ [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.

The typing judgment is written `Γ ⊢ 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]):

```haskell
Γ ::= · | Γ, x : A | Γ, data D Δ⁺ where \{ d_i : Δ_i \rightarrow D Δ⁺ \}_{i \in 1..j}
```
\( x, y, f, p \) \( \in \) variables
\( D \in \) data types, including \( \text{Nat} \)
\( d \in \) constructors, including \( 0 \) and \( S \)
\( i, j \in \) natural numbers

expressions

\[
a, b, A, B ::= \star \mid x \mid \text{rec } f : A.a \mid \text{abort}_A \\
| (x: A) \rightarrow B \mid \lambda x : A.a \mid a \ b \\
| [x: A] \rightarrow B \mid \lambda [x : A].a \mid a \ b \\
| D \bar{A}_i \mid d [\bar{A}_i] \bar{a}_i \mid \text{case } a \text{ as } [y] \text{ of } \{ d_j \Delta_j \Rightarrow b_j \} \\
| a = b \mid \text{join}_{a=b} \ i \ j \mid \text{injdom} \ a \mid \text{injrng} \ a \mid \text{injtcon} \ a \\
| \text{conv } a \text{ at } [(\sim P_1/x_1)] \ldots [(\sim P_i/x_i)] A
\]

telescopes

\[
\Delta ::= \cdot \mid (x : A) \Delta \mid [x : A] \Delta
\]

expression lists

\[
\bar{a}_i ::= \cdot \mid a \bar{a}_i \mid [a] \bar{a}_i
\]

conv proofs

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

values

\[
v ::= \star \mid x \mid \text{rec } f : A.v \\
| (x: A) \rightarrow B \mid \lambda x : A.a \\
| [x: A] \rightarrow B \mid \lambda [x : A].a \\
| D \bar{A}_i \mid d [\bar{A}_i] \bar{v}_i \\
| a = b \mid \text{join}_{a=b} \ i \ j \mid \text{injdom} \ a \mid \text{injrng} \ a \mid \text{injtcon} \ a \\
| \text{conv } v \text{ at } [(\sim P_1/x_1)] \ldots [(\sim P_i/x_i)] A
\]

Figure 1: Syntax of the annotated language
expressions \[m, n, M, N ::= \ast | x | \text{rec } f.u | \text{abort} | (x : M) \rightarrow N | \lambda x.m | m \cdot n | [x : M] \rightarrow N | \lambda [] . m | m [] | D \overrightarrow{M_i} \mid d \overrightarrow{m_i} \mid \text{case } n \text{ of } \{ d_j \overrightarrow{x_i_j} \Rightarrow m_j \}_{j \in 1..k} \mid m = n \mid \text{join} \]

telescopes \[\Xi ::= \cdot | (x : M) \Xi | [x : M] \Xi\]

expression lists \[\overrightarrow{m_i} ::= \cdot | m \cdot \overrightarrow{m_i} | [] \overrightarrow{m_i}\]

values \[u ::= \ast | x | \text{rec } f.u | (x : M) \rightarrow N | \lambda x.m | [x : M] \rightarrow N | \lambda [] . m | D \overrightarrow{M_i} \mid d \overrightarrow{m_i} \mid m = n \mid \text{join}\]

evaluation contexts \[\mathcal{E} ::= \cdot | \cdot m | u \cdot | \cdot [] | d \overrightarrow{m_i} \cdot \overrightarrow{m_i} | \text{case } \bullet \text{ of } \{ d_j \overrightarrow{x_i_j} \Rightarrow m_j \}\]

\[m \rightsquigarrow \text{cbv } m'\]

\[
\begin{align*}
(\lambda x.m) u & \rightsquigarrow \text{cbv } [u/x]m & \text{SC-APPBETA} \\
\text{(rec } f.u) u_2 & \rightsquigarrow \text{cbv } ([\text{rec } f.u/f] u_1) u_2 & \text{SC-APPREC}
\end{align*}
\]

\[
\begin{align*}
\overrightarrow{\lambda [] . m} & \rightsquigarrow \text{cbv } m & \text{SC-IAPPBETA} \\
\overrightarrow{\text{(rec } f.u)[[]]} & \rightsquigarrow \text{cbv } ([\text{rec } f.u/f] [[]]) & \text{SC-IAPPREC}
\end{align*}
\]

\[
\begin{align*}
\text{case } (d_i \overrightarrow{m_i}) \text{ of } \{ d_j \overrightarrow{x_i_j} \Rightarrow m_j \}_{j \in 1..k} & \rightsquigarrow \text{cbv } [\overrightarrow{x_i_j}/\overrightarrow{m_i}]m_i & \text{SC-CASEBETA}
\end{align*}
\]

\[
\begin{align*}
m \rightsquigarrow \text{cbv } n & & \text{SC-CTX} \\
\mathcal{E}[m] & \rightsquigarrow \text{cbv } \mathcal{E}[n] & \text{SC-ABORT}
\end{align*}
\]

Figure 2: Syntax and operational semantics of the unannotated language
In order to study computational irrelevance and erasure, we define a separate language of unannotated expressions ranged over by metavariabes $m, M$. 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 $\rightsquigarrow_{\text{cbv}}$ 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 $\rightsquigarrow_{\text{cbv}}$ models runtime evaluation. However, in the specification of the type system we use a more liberal notion of parallel reduction, denoted $\rightsquigarrow_{\text{p}}$. The difference is that $\rightsquigarrow_{\text{p}}$ allows reducing under binders, e.g. $(\lambda x.1 + 1) \rightsquigarrow_{\text{p}} (\lambda x.2)$ even though $(\lambda x.1 + 1)$ is already a CBV value. The main reason for introducing $\rightsquigarrow_{\text{p}}$ in addition to $\rightsquigarrow_{\text{cbv}}$ is for the metatheory: in order to characterize when two expressions are provably equal (lemma 13) we need a notion of reduction that satisfies the substitution properties in section 3.2, and we defined $\rightsquigarrow_{\text{p}}$ accordingly. But because $\rightsquigarrow_{\text{p}}$ allows strictly more reductions than $\rightsquigarrow_{\text{cbv}}$, defining the type system in terms of $\rightsquigarrow_{\text{p}}$ 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:

- $\Gamma \vdash a : A$ Typing of annotated expressions
- $H \vdash m : M$ Typing of unannotated expressions
- $m \rightsquigarrow_{\text{cbv}} m'$ (Runtime, deterministic CBV) evaluation
- $m \rightsquigarrow_{\text{p}} m'$ (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 rec $f : A.a$ form, with the typing rule

$$
\frac{\Gamma, f : A \vdash v : A \quad \Gamma \vdash A : \star}{\Gamma \vdash \text{rec } f : A.a : A} \quad \text{T.REC}
$$

With this rule the body of a well-typed rec-expression is always a value, but we leave it a general expression $a$ in the syntax so that substitution $[a/x]b$ is always defined. 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 $\text{rec } f : (x:A) \to A_2 \mid [x:A_1] \to A_2$. Rec-expressions are values, and a rec-expression in an evaluation context steps by the rule $(\text{rec } f . u)_{u_2} \rightsquigarrow_{\text{cbv}} ([\text{rec } f . u/f]_{u_1})_{u_2}$. This maintains the invariant that CBV evaluation only substitutes values for variables.

In addition to nonterminating expressions, we include explicit error terms $\text{abort}_A$, which can be given any well-formed type.

$$
\frac{\Gamma \vdash A : \star}{\Gamma \vdash \text{abort}_A : A} \quad \text{T.ABORT}
$$

An abort expression propagates past any evaluation context by the rule $\varepsilon'[\text{abort}] \rightsquigarrow_{\text{cbv}} \text{abort}$. 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 \( \beta \)-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 \((\lambda x.3) \ abort = abort\) and \((\lambda x.3) \ abort = 3\) 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 be proven equal if they have a common reduct under CBV parallel reduction \(\sim_p\). This relation is similar to \(\sim_{cbv}\), except that it permits evaluation under binders and subexpressions can be evaluated in parallel. The rules for \(\lambda\)-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 \(\beta\)-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 \(\text{Vec Nat} (0 + x)\) and \(\text{Vec Nat} x\) to be equal types.

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

\[
\frac{\Gamma \vdash a : (x:A) \rightarrow B \quad \Gamma \vdash b : A}{\Gamma \vdash a \ b : [b/x]B}
\]

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 (lemma 2) that \((x:A) \rightarrow B\) is well-formed, the derivation of \(\Gamma \vdash (x:A) \rightarrow B : \star\) may involve reductions, and substituting a non-value \(b\) for \(x\) may block a \(\beta\)-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...
Irrelevance, Heterogeneous Equality, and CBV

\[
\begin{align*}
\Gamma &\vdash a : (x:A) \rightarrow B \\
\Gamma &\vdash b : A \\
\Gamma &\vdash [b/x]B : * \\
\Gamma &\vdash a b : [b/x]B
\end{align*}
\]

This rule is simple, yet expressive. Previous work \cite{29,12,27} 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.

Because our annotated type system satisfies substitution of values, both of these rules are special cases of our rule above (proofs are in appendix B.7):

**Lemma 1** (Substitution for the annotated language). If \(\Gamma_1, x_1 : A_1, \Gamma_2 \vdash a : A\), then \(\Gamma_1, [v_1/x_1]\Gamma_2 \vdash [v_1/x_1]a : [v_1/x_1]A\).

**Lemma 2** (Regularity for the annotated language). If \(\Gamma \vdash a : A\), then \(\Gamma \vdash A : *\).

**Lemma 3.** The following rules are admissible.

\[
\begin{align*}
\Gamma &\vdash a : (x:A) \rightarrow B \\
\Gamma &\vdash v : A \\
\Gamma &\vdash a v : [v/x]B
\end{align*}
\]

\[
\begin{align*}
\Gamma &\vdash a : A \rightarrow B \\
\Gamma &\vdash b : A \\
\Gamma &\vdash a b : B
\end{align*}
\]

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. \cite{12} 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 \(\text{conv } a \text{ at } \sim b\).

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 \(\text{conv}\) is to first evaluate \(b\) until it reaches \(\text{refl}_\text{eq}\), the only constructor of the equality type, and then step by \(\text{conv } a \text{ at } \sim \text{refl}_\text{eq} \Rightarrow a\).

However, relevance can get in the way of reasoning about programs. Equations involving \(\text{conv}\) such as \((\text{conv } a \text{ at } \sim b) = a\) are not easily provable in Coq unless \(b\) is \(\text{refl}_\text{eq}\). 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 \cite{24}. 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 \(\text{conv } a \text{ at } \sim \text{refl}_\text{eq} \sim a\) into just \(a\). But the Coq typechecker does not know about extraction.

Our language integrates extraction into the type-system, similarly to ICC* \cite{7}. Specifically, we define an erasure function \(|·|\) which takes an annotated expression \(a\) and produces an unannotated expression \(m \equiv |a|\). The definition of \(|·|\) 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.

\footnote{Some authors reserve the word “conversion” for definitional equality. Our type system does not have a definitional equality judgment, so we hope our use of the word does not cause confusion.}
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 \( \text{Nat} = (\text{Nat} \to \text{Nat}) \) it is possible to type the unannotated looping term \( (\lambda x.x.x) \) 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 \textit{definitional} and \textit{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 an 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 [6]), 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, \( m n \rightsquigarrow_{\text{cbv}} m' n' \), 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 unannotated 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

\[
\begin{align*}
\text{Vec} & \, (a : *) \, : \, \text{Nat} \rightarrow * \, \text{where} \\
nil & \, : \, \text{Vec} \, a \, 0 \\
\text{cons} & \, : \, (n : \text{Nat}) \rightarrow \text{Vec} \, a \, n \rightarrow \text{Vec} \, a \, (S \, n)
\end{align*}
\]

\[
\begin{align*}
\text{app} & \, : \, (n1 \, n2 : \text{Nat}) \rightarrow (a : *) \rightarrow \text{Vec} \, a \, n1 \rightarrow \text{Vec} \, a \, n2 \rightarrow \text{Vec} \, a \, (n1+n2) \\
\text{app} \, n1 \, n2 \, a \, xs \, ys & \, = \\
\text{case} \, xs \, \text{of} \\
\text{nil} & \Rightarrow \text{ys} \\
(\text{cons} \, n \, x \, xs) & \Rightarrow \text{cons} \, a \, (n+n2) \, x \, (\text{app} \, n \, n2 \, a \, xs \, ys)
\end{align*}
\]

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

\[
\text{app-assoc} : (n1 \, n2 \, n3 : \text{Nat}) \rightarrow (v1 : \text{Vec} \, a \, n1) \rightarrow (v2 : \text{Vec} \, a \, n2) \rightarrow (v3 : \text{Vec} \, a \, n3) \rightarrow (\text{app} \, a \, n1 \, (n2+n3) \, v1 \, (\text{app} \, a \, n2 \, n3 \, v2 \, v3)) \rightarrow (\text{app} \, a \, (n1+n2) \, n3 \, (\text{app} \, a \, n1 \, n2 \, v1 \, v2) \, v3)
\]

If we proceed by pattern-matching on \( v1 \), then when \( v1 = \text{cons} \, n \, x \, v \) we have to show, after reducing the RHS, that

\[
(\text{cons} \, a \, (n + (n2 + n3)) \, x \, (\text{app} \, a \, n \, (n2 + n3) \, v \, (\text{app} \, a \, n2 \, n3 \, v2 \, v3))) = (\text{cons} \, a \, ((n + n2) + n3) \, x \, (\text{app} \, a \, (n + n2) \, n3 \, (\text{app} \, a \, n \, n2 \, v2) \, v3))
\]

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 \(\texttt{Vec}\) to syntactically indicate that \(n\) is irrelevant by surrounding it with square brackets.

\[
\text{data } \texttt{Vec'} (a:\ast) : \texttt{Nat} \rightarrow \ast \text{ where } \\
\text{nil'} : \texttt{Vec'} a 0 \\
\text{cons'} : [n:\texttt{Nat}] \rightarrow a \rightarrow \texttt{Vec'} a n \rightarrow \texttt{Vec'} a (\texttt{S} n)
\]

Irrelevant constructor arguments are not represented in memory at run-time, and equations between irrelevant arguments are trivially true since our \(T_{\text{JOIN}}\) rule is stated using erasure.

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

\[
\Gamma, x : A \vdash b : B \\
x \notin \text{FV}([b]) \\
\Gamma \vdash \lambda [x : A].b : [x : A] \rightarrow B \quad \text{\(T_{\text{IABS}}\)}
\]

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 \(\lbrack a [b]\) goes to \(\lbrack 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 \(23\). For example, using a hypothetical equality we can type the term

\[
\lambda [p : \texttt{Bool} = \texttt{Nat}].1 + \texttt{conv true at} \sim p : [p : \texttt{Bool} = \texttt{Nat}] \rightarrow \texttt{Nat}.
\]

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

Irrelevant arguments are very useful in dependently typed programming. In addition to datatype indices, they can be used for type arguments of polymorphic functions (we could make the argument \(a\) of the \texttt{app} function irrelevant), and for proofs of preconditions (we could make the argument \(p\) of the \texttt{safe div} function in section \(1.1\) irrelevant).

**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 \((\lambda x : (\texttt{Bool} = \texttt{Nat}).a) \texttt{(loop)}\) will not lead to type error thanks to our CBV semantics, the same reasoning clearly does not work for \((\lambda [x : \texttt{Bool} = \texttt{Nat}].a) [\texttt{loop}]\). To maintain the invariant that variables always stand for values, we restrict the irrelevant application rule to only allow values in the
argument position:

\[\Gamma \vdash a : [x : A] \rightarrow B\]
\[\Gamma \vdash v : A\]
\[\Gamma \vdash a[v] : [v/x]B\]

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 \(\text{Bool} = \text{Nat}\) (as above). The following counter-example shows that we can get in trouble even by erasing an \textit{abort} of type \(\text{Nat}\). First, note that since the reduction relation treats variables as values, \((\lambda x.\text{Bool}) x \Rightarrow p\ \text{Bool}\). So we have \(\text{join}: ((\lambda x : \text{Nat.} \text{Bool}) x = (\lambda x : \text{Nat.} \text{Nat}) x) = (\text{Bool} = \text{Nat})\). Then the following term typechecks:

\[
\lambda [x : \text{Nat}].\lambda p : ((\lambda x : \text{Nat.} \text{Bool}) x) = ((\lambda x : \text{Nat.} \text{Nat}) x).\text{conv} p \text{at} \sim \text{join} : [x : \text{Nat}] \rightarrow (p : (\lambda x : \text{Nat.} \text{Bool}) x = (\lambda x : \text{Nat.} \text{Nat}) x) \rightarrow (\text{Bool} = \text{Nat})
\]

On the other hand, by our reduction rule for error terms, \((\lambda x.\text{Bool}) \text{abort} \Rightarrow p\ \text{abort}\), so \(\text{join}: ((\lambda x : \text{Nat.} \text{Bool}) \text{abort}_{\text{Nat}}) = ((\lambda x : \text{Nat.} \text{Nat}) \text{abort}_{\text{Nat}})\).

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

**The need for termination checking** The value restriction is a severe limitation on the practical use of irrelevant arguments. For example, even if we make the length argument to \text{cons'} irrelevant, we cannot make the length arguments to \text{app} irrelevant. The problem is that in the recursive case we would want to return

\[\text{cons'} a \ [n+n2] x' \ (\text{app} a \ [n] \ x\text{'} [n2] \ y)\]

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. A first attempt would look like

\[
\text{app} : (n1 \ n2 : \text{Nat}) \rightarrow (a : \text{+}) \rightarrow \text{Vec'} n1 a \rightarrow \text{Vec'} n2 a \rightarrow \text{Vec'} (n1+n2) a
\]

\[
\text{app} n1 n2 a x_{s} y_{s} =
\text{case} x_{s} \text{ of}
\text{nil'} \Rightarrow y_{s}
(\text{cons'} [n] x_{s}) \Rightarrow \text{let} m = n1-1+n2 \text{ in}
\text{cons'} a \ [m] x_{s} (\text{app} (n1-1) n2 a x_{s} y_{s})
\]

This carries out the addition at runtime, so the application of \text{cons'} is accepted. But the program still does not typecheck, due to the mismatch between \(m+1\) and \(n1 + n2\). To make it check, we need to insert type conversions. Even worse, the conversions rely on the fact that \(n1 - 1 + n2 + 1 = n1 + n2\). The proof of this uses induction on \(n2\), i.e., a call to a recursive function, so the proof also can not be erased and has to be evaluated at runtime. The rewritten \text{app} function is more complicated, and because of the proof even asymptotically slower, which is quite unsatisfying.

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). To allow the desired definition of app the termination analysis only has to prove that addition terminates, which is not hard.

Value-restricted irrelevance already has uses: for example, except for type-level computation all types are values, so we could compile ML into our language erasing all types. But to support precisely-typed programs without a performance penalty it is essential to also be able to erase proofs, and as we demonstrated above this is not possible without some form of termination analysis. Therefore, we consider this language as only a step towards a practical design.

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

\[
\text{data } D \Delta^+ \text{ where } \{ d_i : \Delta_i \rightarrow D \Delta^+_{i\in \{1..j\}} \}
\]

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 \( \Delta \) range over lists of relevance-annotated variable declarations like \((x : A)[y : B](z : C)\), also known as telescopes, while overlined metavariables \( \overline{\alpha} \) range over lists of terms. Metavariables \( \Delta^+ \) range over telescopes that have only relevant declarations. 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 \( \Delta \) is \((x : A)[y : B](z : C)\) and \( \overline{\alpha} \) is a \([b] \ c\), then...

\[
...\text{this:} \quad \text{is shorthand for this:}
\]

\[
a_1 \Delta \quad a_1 x \ [y] z
\]

\[
\Delta \rightarrow A_1 \quad (x : A) \rightarrow [y : B] \rightarrow (z : C) \rightarrow A_1
\]

\[
[\overline{\alpha}] / \Delta \ a_1 \quad a/x \ [b/y] \ [c/z] a_1
\]

\[
\Gamma, \Delta \quad \Gamma, x : A_1, y : B, z : C
\]

\[
\Gamma \vdash \overline{\alpha} : \Delta \quad \Gamma \vdash a : A \land \Gamma \vdash b : [a/x]B \land \Gamma \vdash c : [b/y] [a/x]C
\]

We also simplify the rules by having **parameters but not indices**. Each datatype has a list of parameters \( \Delta \), 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 [26]). For example, the declaration of Vec’ above can be reformulated without indices as

\[
data \text{Vec’ (a:*)(n:Nat)} \text{ where}
\]

\[
nil’ : [p:n=0] \rightarrow \text{Vec’ a n}
\]

\[
\text{cons’} : [m:\text{Nat}] \rightarrow [p:n=S m] \rightarrow a \rightarrow \text{Vec’ a m} \rightarrow \text{Vec’ a n}
\]

To make the statement of the canonical forms lemma simpler (see lemma [4] 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 \( d [\overline{\alpha}] \overline{\beta} \).

In the corresponding elimination form (the case expression case \( b \) as \([y]\) of \( \{ d_i \Delta_i \Rightarrow a_i \}_{i \in \{1..j\}} \)) 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 \( \Gamma \vdash b : D \overline{B} \), then for each case we check

\[
\Gamma, [\overline{B} / \Delta^+] \Delta_i, y : b = d_i \Delta_i \vdash a_i : A
\]

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: \( d \left[ A_i \right] \) is \( d \left[ a_i \right] \). On the other hand, it never makes sense for parameters to datatype constructors to be irrelevant. For example, if the parameters to \text{Vec} were made irrelevant, the join rule would validate \( \Gamma \vdash \text{join} : (\text{Vec} \left[ \text{Nat} \right] [1]) = (\text{Vec} \left[ \text{Bool} \right] [2]) \), which would defeat the point of having the parameters in the first place. This is reflected in our syntax for datatypes, which requires that the list of parameters is a \( \Delta^+ \) (i.e. contains no irrelevant declarations). In order to typecheck a datatype constructor, we look up the corresponding datatype declaration in the context and check that the provided parameters have the right type.

\[
\text{data } D \Delta^+ \text{ where } \{ d_i : \Delta_i \rightarrow D \Delta^+ \}_{i \in 1..j} \in \Gamma \\
\Gamma \vdash \text{join} : * \\
\Gamma \vdash D \Delta_i : * \\
\text{TCON}
\]

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

\[
\text{cons'} \left[ \text{Bool} \right] [1] [0] [\text{join}] \text{true} \left( \text{Nil'} \left[ \text{Bool} \right] [0] [\text{join}] \right)
\]

is well-typed and erases to \( \text{cons'} \left[ \right] [\text{Nil'} \left[ \right] ] \text{true} \left( \text{Nil}' [\text{Nil'}] \right) \). 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

\[
\text{cons'} \left[ \text{m} : \text{Nat} \right] [p : n = S m] (x : a) (xs : \text{Vec'} a m) \Rightarrow \ldots \text{body} \ldots
\]

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 \( \text{Vec} \) type we could write a constant-time length function by projecting out \( m \), but that is not possible with \( \text{Vec'} \).

### 2.4 Very heterogeneous equality

The \text{app-assoc} example also illustrates a different problem with indexed data: the two sides of the equation have different types (namely \( \text{Vec} a (n_1 + (n_2 + n_3)) \) versus \( \text{Vec} a ((n_1 + n_2) + n_3) \)—so, e.g., the usual equality in Coq does not even allow writing down the equation! We need some form of heterogeneous equality. The most popular choice for this is \( \text{JMeq} \) [15], which allows you to state any equality, but only use them if both sides have (definitionally) 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 \( \text{JMeq} \), we allow stating any equation as long as the two sides are well-typed. Our formation rule for the equality type is

\[
\Gamma \vdash a : A \quad \Gamma \vdash b : B \\
\Gamma \vdash a = b : * \\
\text{T_EQ}
\]

Unlike \( \text{JMeq} \), 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 [25], although the details differ.

We showed a simplified version (\( \text{VCONV} \)) of our conversion rule on page 9; we present the full rule (\( \text{T_CONV} \)) in section 2.6. We now build-up the full rule from the simplified rule, step-by-step, motivating
each addition along the way. First, in order to be able to change only parts of a type, we phrase the rule in terms of substituting into a template $A$.

$$
\Gamma \vdash a : [B_1/x]A \quad \Gamma \vdash v : B_1 = B_2 \\
\Gamma \vdash [B_2/x]A : * \\
\Gamma \vdash \text{conv at } [\sim v/x]A : [B_2/x]A \quad \text{CONV_SUBST}
$$

For example, given a proof $\Gamma \vdash v : y = 0$, we can convert the type Vec Nat $(y + y)$ to Vec Nat $(y + 0)$ using the template substitution VecNat $(y + \sim v)$.

We need the premise $\Gamma \vdash [B_2/x]A : *$ 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 possible to write a function that takes a proof of 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.

### 2.5 Multiple simultaneous conversions

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

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

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.

$$
\Gamma \vdash v_1 : A_1 = B_1 \quad \ldots \quad \Gamma \vdash v_i : A_i = B_i \\
\Gamma \vdash a : [A_1/x_1] \ldots [A_i/x_i]A \\
\Gamma \vdash [B_1/x_1] \ldots [B_i/x_i]A : * \\
\Gamma \vdash \text{conv at } [\sim v_1/x_1] \ldots [\sim v_i/x_i]A : [B_1/x_1] \ldots [B_i/x_i]A \quad \text{CONV_MULTI_SUBST}
$$

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 a $(n + (n2 + n3))$ and $A'$ be Vec a $(n(n2 + n3))$. In such an example, $A$ and $A'$ are indeed provably equal, but our $n$-ary conversion rule obviates the need to provide 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 $(\lambda x : (1 = 0).1) = (\lambda x : \text{Nat}.1)$, so by transitivity we would get $(\lambda x : \text{Nat}.1) = (\lambda x : \text{Nat}.0)$, and from there to $1 = 0$.

### 2.6 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 $\lambda$-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.

The last step is to understand how nontermination and irrelevance affect the final annotated conv and join rules, $\text{T\_CONV}$ and $\text{T\_JOIN}$ below. The conv rule in the erased language, including $n$-ary substitution, looks like

$$
H \vdash u_1 : M_1 = N_1 \quad \ldots \quad H \vdash u_i : M_i = N_i \\
H \vdash m : [M_1/x_1] \ldots [M_i/x_i]M \\
H \vdash [N_1/x_1] \ldots [N_i/x_i]M : * \\
\hline
H \vdash m : [N_1/x_1] \ldots [N_i/x_i]M \quad \text{ET\_CONV}
$$

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 $\text{CONV\_MULTISUBST}$ rule we showed above.

However, $\text{CONV\_MULTISUBST}$ needs one more 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 $P_j$ interleaved with the template, marking it with a tilde. So our final annotated rule looks like this:

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

$$
\forall j.\ ((P_j \text{ is } v_j \text{ and } \Gamma \vdash v_j : A_j = B_j) \text{ or } (P_j \text{ is } [A_j = B_j] \text{ and } x_j \notin \text{FV}(|A|) ))
\Gamma \vdash a : [A_1/x_1] \ldots [A_i/x_i]A
\Gamma \vdash [B_1/x_1] \ldots [B_i/x_i]A : *
\hline
\Gamma \vdash \text{conv at} \ [\sim P_1/x_1] \ldots [\sim P_i/x_i]A : [B_1/x_1] \ldots [B_i/x_i]A \
\text{T\_CONV}
$$

For example, if $a : \text{Vec} \ A \ x$ and $y : x = 3$, then $\text{conv a at Vec A ~y}$ has type $\text{Vec} \ A \ 3$.

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

$$
m_1 \sim_P n \quad m_2 \sim_P n \\
H \vdash m_1 = m_2 : * \\
\hline
H \vdash \text{join} : m_1 = m_2 \quad \text{ET\_JOIN}
$$

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 type-checker 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 final annotated join rule looks like

$$\Gamma \vdash \text{join}_{a=b}^i j : a = b_n$$

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 \text{join} as an abbreviation for \text{join} 100 100. When we want to prove equations between terms that are already values, we can use \text{conv} to change subterms of them. For example, to prove the equality $\text{Vec} A (1 + 1) = \text{Vec} A 2$ we write

\text{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, with this choice of annotations we cannot show an equation like $(\lambda x. (\lambda y. y) x) = (\lambda x. x)$. So far, 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 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 two type systems were designed so that there are enough annotations to make typechecking the annotated language decidable, and to make erasure into the unannotated system preserve well-typedness:

**Lemma 4 (Decidability of type checking).** There is an algorithm which given $\Gamma$ and $a$ computes the unique $A$ such that $\Gamma \vdash a : A$, or reports that there is no such $A$.

**Lemma 5 (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 gone through erasure. As an example, compare the annotated and unannotated versions of the IABS rule:

$$\Gamma, x : A \vdash b : B \quad \Gamma \vdash \lambda [x : A] b : [x : A] \rightarrow B \quad H, x : M \vdash n : N \quad H \vdash \lambda [x : M] n : [x : M] \rightarrow N$$

**T.IABS**

**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. One could ask whether it would be possible to define an operational semantics for the annotated terms and then prove preservation for the annotated language. The main complication of doing that is that as terms steps extra type conversions must be added, which would complicate the step relation.

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 6** (Substitution of \(\sim_p\)). If \(N \sim_p N'\), then \([N/x]M \sim_p [N'/x]M\).

**Lemma 7** (Substitution into \(\sim_p\)). If \(u \sim_p u'\) and \(m \sim_p m'\), then \([u/x]m \sim_p [u'/x]m'\).

**Lemma 8** (Confluence of \(\sim_p\)). If \(m \sim_p^* m_1\) and \(m \sim_p^* m_2\), then there exists some \(m'\) such that \(m_1 \sim_p^* m'\) and \(m_2 \sim_p^* m'\).

**Definition 9** (Joinability). We write \(m_1 \lesssim m_2\) if there exists some \(n\) such that \(m_1 \sim_p^* n\) and \(m_2 \sim_p^* n\).

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

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 10** (Substitution). If \(H, x_1 : M_1, H_2 \vdash m : M\) and \(H \vdash u_1 : M_1\), then \(H, [u_1/x_1]H_2 \vdash [u_1/x_1]m : [u_1/x_1]M\).

Preservation also needs an inversion lemmas for \(\lambda\)-s, irrelevant \(\lambda\)-s, rec, and data constructors. They are similar, and we show the one for \(\lambda\)-abstractions as an example.

**Lemma 11** (Inversion for \(\lambda\)-s). If \(H \vdash \lambda x.n : M\), then there exists \(m_1, M_1, N_1\), such that \(H \vdash m_1 : M = (x : M_1) \rightarrow N_1\) 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 \(\beta\)-convertible to the type \((x : M_1) \rightarrow N_1\), not just provably equal to it. But in our language, if the context contained the equality \((\text{Nat} \rightarrow \text{Nat}) = \text{Nat}\), then we could show \(H \vdash \lambda x.x : \text{Nat}\) 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 prove the preservation theorem. For type safety we are only interested in preservation for \(\sim_{cbv}\), but it is convenient to generalize the theorem to \(\sim_p\).

**Theorem 12** (Preservation).

If \(H \vdash m : M\) and \(m \sim_p m'\), 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

\[
H \vdash u_1 : (x : M_1) \rightarrow N_1 = (x : M) \rightarrow N_2 \quad \text{ET_INJDOM}
\]

\[
H \vdash u : M
\]

\[
H \vdash \text{join} : [u/x]N_1 = [u/x]N_2 \quad \text{ET_INJREG}
\]

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

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

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

Second, consider the case when an application steps by \(\beta\)-reduction, \((\lambda x.m_0) u \sim_p [u/x]m_0\), 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 \(H \vdash \text{join} : (x : M) \rightarrow N = (x : M_1) \rightarrow N_1\). 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).

### 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 \(N = (\text{Nat} \rightarrow \text{Nat})\)? 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 13** (Soundness of equality). If \(H_D \vdash u : M\) and \(M \not\sim (m_1 = n_1)\), then \(m_1 \not\sim n_1\).

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 \(\lambda\)-abstraction, we need to know that it is never the case that \((x : M) \rightarrow N \not\sim (m_1 = n_1),\)
which follows because \( \sim_p \) preserves the top-level constructor of a term. To handle the injectivity rules, we need to know that \( \gamma \) is injective in the sense that \( (x : M_1) \rightarrow N_1 \ \gamma \ (x : M_2) \rightarrow N_2 \) implies \( M_1 \ \gamma \ M_2 \); again this follows by reasoning about \( \sim_p \). Finally, consider the conversion rule. The case looks like

\[
\begin{align*}
H_D \vdash u_1 : M_1 &= N_1 & \ldots & H_D \vdash u_i : M_i = N_i \\
H_D \vdash u : [M_1/x_1] \ldots [M_i/x_i]M \\
H_D \vdash [N_1/x_1] \ldots [N_i/x_i]M : * \\
\hline
H_D \vdash u : [N_1/x_1] \ldots [N_i/x_i]M 
\end{align*}
\]

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

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

**Lemma 14** (Canonical forms). Suppose \( H_D \vdash u : M \).

1. If \( M \ \gamma \ (x : M_1) \rightarrow M_2 \), then \( u \) is either \( \lambda x. u_1 \) or \( \text{rec } f. u \).
2. If \( M \ \gamma \ [x : M_1] \rightarrow M_2 \), then \( u \) is either \( \lambda [] . u_1 \) or \( \text{rec } f. u \).
3. If \( M \ \gamma \ D \ M \), then \( u \) is \( d \bar{u} \), where \( D \bar{\Sigma}^+ \) where \( \{ d_i : \overline{\Sigma}_i \rightarrow D \bar{\Sigma}^+ | i \in \{1, \ldots, l\} \} \in H_D \) and \( d \) is one of the \( d_i \).

**Theorem 15** (Progress). If \( H_D \vdash m : M \), then either \( m \) is a value, \( m \) is abort, or \( m \sim_{\text{cbv}} m' \) 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 [31], ATS [30], Omega [22], Haskell with GADTs [19]), or by restricting dependent application to values or “pure” expressions (e.g. DML [14], F* [27], 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 \( \Pi \Sigma \) [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.

\( \lambda^\Xi \) [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. \( \lambda^\Xi \) 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], EPTS [17], and Abel’s system [1]. One of the key differences between the systems is whether the variable \( x \) in an irrelevant arrow type \( [x : A] \rightarrow B \) is allowed to occur freely in \( B \) (“Miquel [16]-style irrelevance”, our choice) or only in irrelevant positions in \( B \) (“Pfenning [20]-style”, see also [21]). 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 \((\text{conv} \ a \ \text{at} \sim \ b) = a\), 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 \[25\], 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 \text{conv} rule.

GHC Core \[26, 28\] 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 evaluated 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 the 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’ previously unseen combination of features.

Acknowledgments

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


A Full Language Specification

A.1 Syntax

\[ \text{tele}, \Delta \quad ::= \quad \cdot \quad | \quad (x : A)\Delta \quad | \quad [x : A]\Delta \quad | \quad \text{telescope} \quad | \quad \text{empty telescope} \quad | \quad \text{relevant binding} \quad | \quad \text{irrelevant binding} \]

\[ \text{teleplus}, \Delta^+ \quad ::= \quad \cdot \quad | \quad (x : A)\Delta^+ \quad | \quad \text{(relevant) telescope} \quad | \quad \text{empty telescope} \quad | \quad \text{relevant binding} \]

\[ \text{env}, \Gamma \quad ::= \quad \cdot \quad | \quad \Gamma,\text{decl} \quad | \quad \text{typing environment} \quad | \quad \text{empty} \]

\[ \text{decl} \quad ::= \quad x : A \quad | \quad \text{variable} \quad | \quad \text{data}D\Delta^+ \quad | \quad \text{datatype} \quad | \quad \text{data}D\Delta^+ \quad | \quad \text{abstract datatype name} \]

\[ \text{exp}, a, b, A, B \quad ::= \quad * \quad | \quad x \quad | \quad D \bar{x} \quad | \quad \text{type} \quad | \quad \text{variable} \quad | \quad \text{data} \quad | \quad \text{recursive definition} \quad | \quad \lambda x : A.a \quad | \quad \lambda \text{-abstraction} \quad | \quad \lambda\bar{x} : A.a \quad | \quad \lambda \text{-abstraction} \quad | \quad a b \quad | \quad \text{application} \quad | \quad a [b] \quad | \quad \text{implicit application} \quad | \quad (x : A) \rightarrow B \quad | \quad \text{function type} \quad | \quad [x : A] \rightarrow B \quad | \quad \text{irrelevant function type} \quad | \quad \text{case } a \text{ as } y \text{ of } \{ d_j : \Delta_j \rightarrow D \Delta^+ \}_{j \in 1..k} \quad | \quad \text{pattern matching} \quad | \quad a = b \quad | \quad \text{equality proposition} \quad | \quad \text{join}_{a=b} i j \quad | \quad \text{equality proof} \quad | \quad \text{injdom } a \quad | \quad \text{equality proof} \quad | \quad \text{injrng } a b \quad | \quad \text{equality proof} \quad | \quad \text{injcon} a \quad | \quad \text{equality proof} \quad | \quad \text{conv } a \text{ at } [\sim P_1/x_1]...[\sim P_i/x_i]A \quad | \quad \text{type conversion} \quad | \quad \text{abort}_A \quad | \quad \text{failure} \]
\[
P ::= \begin{cases} v \\ [a = b] \end{cases} \quad \text{Proofs used in conv rule}
\]
\[
\text{val, } v ::= \begin{cases} x \\ * \\ (x: A) \rightarrow B \\ [x: A] \rightarrow B \\ a = b \\ \text{conv at } [\sim P_1/x_1] \ldots [\sim P_i/x_i] A \\ \text{join}_{a=b} i j \\ D A_i \\ d [A_i] v_i \\ \lambda x : A. a \\ \lambda [x : A]. a \\ \text{rec } f : A.a \end{cases} \quad \text{Values}
\]
\[
\text{explist, } \overline{a_i}, \overline{b_i}, \overline{A_i}, \overline{B_i} ::= \begin{cases} \cdot \\ a \overline{a_i} \\ [a] \overline{a_i} \end{cases} \quad \text{list of expressions}
\]
\[
\text{vallist, } \overline{v_i} ::= \begin{cases} \cdot \\ v \overline{v_i} \\ [v] \overline{v_i} \end{cases} \quad \text{list of values}
\]
\[
\text{etele, } \Xi ::= \begin{cases} \cdot \\ (x : M) \Xi \\ [x : M] \Xi \end{cases} \quad \text{unannotated telescope}
\]
\[
\text{eteleplus, } \Xi^+ ::= \begin{cases} \cdot \\ (x : M) \Xi^+ \end{cases} \quad \text{unannotated (relevant) telescope}
\]
\[
\text{eenv, } H ::= \begin{cases} \cdot \\ H, \text{edecl} \end{cases} \quad \text{typing environment}
\]
Irrelevance, Heterogeneous Equality, and CBV

decl ::=
  \( x : M \)  
  \( \text{typing env declaration} \)
  \( \text{variable} \)
  \( \text{data} \)
  \( D \Xi^+ \)  
  \( \text{datatype} \)
  \( \text{abstract datatype name} \)

\( \text{eexp, m, n, M, N} \) ::=
  \( \ast \)  
  \( x \)  
  \( \text{type} \)
  \( \text{variable} \)
  \( \text{datatype} \)
  \( D \Xi^+ \)  
  \( \text{data} \)
  \( d \Xi^+ \)  
  \( \text{recursive definition} \)
  \( \lambda x. m \)  
  \( \lambda \text{-abstraction} \)
  \( \lambda [] . m \)  
  \( \lambda -abstraction \)
  \( m n \)  
  \( \text{application} \)
  \( m[] \)  
  \( \text{irrelevant application} \)
  \( (x : M) \rightarrow N \)  
  \( \text{function type} \)
  \( [x : M] \rightarrow N \)  
  \( \text{irrelevant function type} \)
  \( \text{case n of \{ d_{i,j} : \Xi_{i,j} \rightarrow D \Xi^+ | i \in 1..j \} } \)  
  \( \text{pattern matching} \)
  \( m = n \)  
  \( \text{equality proposition} \)
  \( \text{equality proof} \)
  \( \text{failure} \)

\( \text{eval, u} \) ::=
  \( x \)  
  \( \ast \)  
  \( (x : M) \rightarrow N \)  
  \( [x : M] \rightarrow N \)  
  \( m = n \)  
  \( \text{join} \)
  \( D \Xi^+ \)  
  \( d \Xi^+ \)  
  \( \text{recursive definition} \)
  \( \lambda x. m \)  
  \( \lambda [] . m \)  
  \( \text{values} \)
\[\textit{explist}, \overrightarrow{m}, \pi_i, \overrightarrow{M}_i, \overrightarrow{N}_i ::=} \begin{array}{c}
  \text{list of expressions} \\
  \cdot \\
  m \overrightarrow{m}_i \\
  \overrightarrow{m}_i \\
\end{array}
\]

\[\textit{evallist}, \overrightarrow{u}_i ::=} \begin{array}{c}
  \cdot \\
  u \overrightarrow{u}_i \\
  \overrightarrow{u}_i \\
\end{array}
\]

\[\textit{evaletx}, \mathcal{E} ::=} \begin{array}{c}
  \mathcal{E} \\
  \cdot \\
  \cdot \overrightarrow{m} \\
  \cdot u \cdot \\
  \cdot \overrightarrow{u}_i \\
  \text{case } \cdot \text{ of } \{d_j \overrightarrow{x}_j \Rightarrow m_j\} \\
  d \overrightarrow{u}_i \cdot \overrightarrow{m}_i \\
\end{array}
\]

\[\text{Evaluation contexts}\]
A.2 Erasure function

The erasure function \(|a|\) is defined by:

\[
\begin{align*}
|*| &= * \\
|x| &= x \\
|DA_i| &= |A_i| \\
|d| &= d \\
|recf : A.\nu| &= recf.\nu \\
|\lambda x : A.a| &= \lambda x.a \\
|a b| &= |a| |b| \\
|a| &= |a| [] \\
|(x:A) \rightarrow B| &= (x:A) \rightarrow |B| \\
|[x:A] \rightarrow B| &= [x:A] \rightarrow |B| \\
|a = b| &= |a| = |b| \\
|join_{a=b} ij| &= join \\
|injdom a| &= join \\
|injrng a b| &= join \\
|injcon a| &= join \\
|case a as [x] of \{ d_j \Delta_j \Rightarrow b_j \} \| &= case |a| of \{ d_j \Delta_j \Rightarrow b_j \} \\
|\text{where } x_{ij} \text{ are the relevant variables of } \Delta_j| \\
|\text{conv at } [\sim P_1/x_1] \ldots [\sim P_i/x_i]A| &= |a| \\
|abort_A| &= abort \\
|\bullet| &= \bullet \\
|a| &= |a| |\bar{a}| \\
|[a]| &= [] |\bar{a}|
\end{align*}
\]

A.3 CBV evaluation

\[ m \rightsquigarrow_{\text{cbv}} n \]

\[
\begin{align*}
(\lambda x.m) u &\rightsquigarrow_{\text{cbv}} [u/x]m \quad \text{SC_APPBETA} \\
(recf.u) u_2 &\rightsquigarrow_{\text{cbv}} ([recf.u/f]u_1) u_2 \quad \text{SC_APPREC} \\
(\lambda []).m &\rightsquigarrow_{\text{cbv}} m \quad \text{SC_IAPPBETA} \\
(recf.u[]) &\rightsquigarrow_{\text{cbv}} ([recf.u/f]u_1[]) \quad \text{SC_IAPPREC} \\
\text{case } (d_i \bar{m}_i) \text{ of } \{ d_j \bar{x}_{ij} \Rightarrow b_j \} &\rightsquigarrow_{\text{cbv}} [\bar{m}_i/\bar{x}_{ij}]m_i \quad \text{SC_CASEBETA} \\
\text{E}[\text{abort}] &\rightsquigarrow_{\text{cbv}} \text{abort} \quad \text{SC_ABORT} \\
\text{E}[m] &\rightsquigarrow_{\text{cbv}} E[n] \quad \text{SC_CTX}
\end{align*}
\]
A.4 Parallel reduction

\[ m \leadsto_p n \]

\[
\begin{array}{c}
m \leadsto_p m' \\
\text{SP_REFL}
\end{array}
\]

\[
\begin{array}{c}
u \leadsto_p u' \\
\text{rec.f.u} \leadsto_p \text{rec.f.u} \\
\text{SP_REC}
\end{array}
\]

\[
\begin{array}{c}
m \leadsto_p m' \\
\lambda.x.m \leadsto_p \lambda.x.m' \\
\text{SP_ABS}
\end{array}
\]

\[
\begin{array}{c}
M \leadsto_p M' \\
N \leadsto_p N' \\
(x : M) \rightarrow N \leadsto_p (x : M') \rightarrow N' \ 	ext{SP_PI}
\end{array}
\]

\[
\begin{array}{c}
M \leadsto_p M' \\
N \leadsto_p N' \\
[x : M] \rightarrow N \leadsto_p [x : M'] \rightarrow N' \ 	ext{SP_IPI}
\end{array}
\]

\[
\begin{array}{c}
m \leadsto_p m' \\
n \leadsto_p n' \\
m = n \leadsto_p m' = n' \ 	ext{SP_EQ}
\end{array}
\]

\[
\begin{array}{c}
m \leadsto_p m' \\
n \leadsto_p n' \\
m n \leadsto_p m' n' \ 	ext{SP_APP}
\end{array}
\]

\[
\begin{array}{c}
m \leadsto_p m' \\
u \leadsto_p u' \\
(\lambda.x.m) u \leadsto_p [u'/x]m' \ 	ext{SP_APPBETA}
\end{array}
\]

\[
\begin{array}{c}
u_1 \leadsto_p u'_1 \\
u_2 \leadsto_p u'_2 \\
(\text{rec.f.u}) u_2 \leadsto_p ([\text{rec.f.u}/f]u'_1) u'_2 \ 	ext{SP_APPREC}
\end{array}
\]

\[
\begin{array}{c}
m \leadsto_p m' \\
[\text{rec.f.u}] \leadsto_p [\text{rec.f.u}/f]u'_1 \\
\text{SP_JAPP}
\end{array}
\]

\[
\begin{array}{c}
m \leadsto_p m' \\
(\lambda [[]] . m) [] \leadsto_p m' [] \\
\text{SP_JAPPBETA}
\end{array}
\]

\[
\begin{array}{c}
u_1 \leadsto_p u'_1 \\
(\text{rec.f.u}) [] \leadsto_p ([\text{rec.f.u}/f]u'_1) [] \\
\text{SP_JAPPREC}
\end{array}
\]

\[
\begin{array}{c}
\forall i. M_i \leadsto_p M'_i \ 	ext{SP_TCON}
\end{array}
\]

\[
\begin{array}{c}
\forall i. m_i \leadsto_p m'_i \\
\text{SP_DCON}
\end{array}
\]
Irrelevance, Heterogeneous Equality, and CBV

\[ m \leadsto_p m' \]
\[ \forall j. \ m_j \leadsto_p m_j' \]
\[ \text{case} \{ \bar{d}_j \bar{x}_j \Rightarrow m_j'_{\bar{j} \in 1..k} \} \]
\[ \leadsto_p \text{case'} \{ \bar{d}_j \bar{x}_j \Rightarrow m_j'_{\bar{j} \in 1..k} \} \]
\[ \forall i. \ u_i \leadsto_p u_i' \]
\[ m_i \leadsto_p m_i' \]
\[ \text{case} (d_i \bar{x}_i) \{ \bar{d}_j \bar{x}_j \Rightarrow m_j''_{\bar{j} \in 1..k} \} \]
\[ \leadsto_p [x_i''/x_i'] \]
\[ \delta' \text{[abort]} \leadsto_p \text{abort} \]

\[ n \not\gamma m \]

\[ m_1 \leadsto_p^* n \]
\[ m_2 \leadsto_p^* n \]
\[ m_2 \not\gamma m_2 \]
\[ \text{JJOIN} \]

A.5 Annotated type system

\[ \Gamma \vdash a : A \]

\[ \vdash \Gamma \]
\[ \Gamma \vdash * : \star \text{T_TYPE} \]
\[ \Gamma \vdash b : D \bar{B}_i \]
\[ \Gamma \vdash A : * \]
\[ \text{data} D \Delta^+ \text{where} \{ \bar{d}_i : \Delta_i \rightarrow D \Delta^+_{\bar{i} \in 1..l} \} \in \Gamma \]
\[ \forall i. \ \Gamma, [\bar{B}_i/\Delta^+] \Delta_i, y : b = d_i \Delta_i \vdash a_i : A \]
\[ \forall i. \ \{ y \} \cup \text{dom} - (\Delta_i) \# \text{FV} ([a_i]) \]
\[ \Gamma \vdash \text{case } b \text{ as } [y] \text{ of} \{ \bar{d}_i \Delta_i \Rightarrow a_i_{\bar{i} \in 1..l} \} : A \]

\[ x : A \in \Gamma \]
\[ \vdash \Gamma \]
\[ \Gamma \vdash x : A \text{T_VAR} \]
\[ \\]
\[ \Gamma \vdash A : * \quad \Gamma, x : A \vdash B : * \]
\[ \Gamma \vdash (x:A) \rightarrow B : * \text{T_PI} \]
\[ \\]
\[ \Gamma \vdash A : * \quad \Gamma, x : A \vdash B : * \]
\[ \Gamma \vdash [x:A] \rightarrow B : * \text{T_IPI} \]
\[ \\]
\[ \text{data} D \Delta^+ \text{where} \{ \bar{d}_i : \Delta_i \rightarrow D \Delta^+_{\bar{i} \in 1..l} \} \in \Gamma \]
\[ \Gamma \vdash \bar{A}_i : \Delta^+ \]
\[ \\]
\[ \text{T_TCON} \]
\[ \Gamma \vdash D \bar{A}_i : * \]
\[ \\]
\[ \text{data} D \Delta^+ \in \Gamma \]
\[ \Gamma \vdash \bar{A}_i : \Delta^+ \]
\[ \Gamma \vdash D \bar{A}_i : * \text{T_ABSTCON} \]
\[ \text{data} D^\Delta \text{ where } \{ \delta_i : \Delta_i \rightarrow D^\Delta_{i \in [j]} \} \in \Gamma \]
\[ \Gamma \vdash A_i : \Delta^+ \]
\[ \Gamma \vdash \overline{a}_i : \overline{A}_i/\Delta \Delta_i \]
\[ \Gamma \vdash d_k[\overline{A}_i] \overline{a}_i : D \overline{A}_i \]
\[ \Gamma, x : A \vdash b : B \]
\[ \Gamma \vdash \lambda x : A.b : (x:A) \rightarrow B \]
\[ \Gamma, x : A \vdash b : B \]
\[ x \notin \text{FV}(|b|) \]
\[ \Gamma \vdash \lambda x : A,b : [x:A] \rightarrow B \]
\[ \Gamma, f : A \vdash v : A \quad \Gamma \vdash A : * \quad A \text{ is } (x:A_1) \rightarrow A_2 \text{ or } [x:A_1] \rightarrow A_2 \]
\[ \Gamma \vdash \text{rec } f : A.v : A \]
\[ \Gamma \vdash a : (x:A) \rightarrow B \]
\[ \Gamma \vdash b : A \]
\[ \Gamma \vdash [b/x]B : * \]
\[ \Gamma \vdash a b : [b/x]B \]
\[ \Gamma \vdash a : [x:A] \rightarrow B \]
\[ \Gamma \vdash v : A \]
\[ \Gamma \vdash a[v] : [v/x]B \]
\[ \Gamma \vdash A : * \]
\[ \Gamma \vdash \text{abort}_A : A \]
\[ \Gamma \vdash a : A \quad \Gamma \vdash b : B \]
\[ \Gamma \vdash a = b : * \]
\[ |a| \sim^{i}_{\text{cbv}} n \quad |b| \sim^{j}_{\text{cbv}} n \]
\[ \Gamma \vdash a = b : * \]
\[ \Gamma \vdash \text{join}_{a=b} i j : a = b \]
\[ \forall j. \ ((P_j \text{ is } v_j \text{ and } \Gamma \vdash v_j : A_j = B_j) \text{ or } (P_j \text{ is } [A_j = B_j] \text{ and } x_j \notin \text{FV}(|A|))) \]
\[ \Gamma \vdash a : [A_1/x_1] \ldots [A_i/x_i]A \]
\[ \Gamma \vdash [B_1/x_1] \ldots [B_i/x_i]A : \ast \]
\[ \Gamma \vdash \text{conv at } [\sim P_1/x_1] \ldots [\sim P_i/x_i]A : [B_1/x_1] \ldots [B_i/x_i]A \]
\[ \Gamma \vdash v_1 : ((x:A_1) \rightarrow B_1) = ((x:A_2) \rightarrow B_2) \]
\[ \Gamma \vdash \text{injdom } v_1 : A_1 = A_2 \]
\[ \Gamma \vdash v_1 : ((x:A) \rightarrow B_1) = ((x:A) \rightarrow B_2) \quad \Gamma \vdash v : A \]
\[ \Gamma \vdash \text{injrng } v_1 : [v/x]B_1 = [v/x]B_2 \]
\[ \Gamma \vdash v_1 : ([x:A_1] \rightarrow B_1) = ([x:A_2] \rightarrow B_2) \quad \Gamma \vdash v : A \]
\[ \Gamma \vdash \text{injdom } v_1 : A_1 = A_2 \]
\[ \Gamma \vdash v_1 : ([x:A] \rightarrow B_1) = ([x:A] \rightarrow B_2) \quad \Gamma \vdash v : A \]
\[ \Gamma \vdash \text{injrng } v_1 : [v/x]B_1 = [v/x]B_2 \]
\[\Gamma \vdash v_1 : D \overline{A_i} = D \overline{A'_i}^{T_{\text{INJCON}}}\]

\[\Gamma \vdash \text{injtcon}_k v_1 : A_k = A'_k \]

\[\Gamma \vdash \text{ENV}_\text{WF}_\text{EMPTY}\]

\[\Gamma \vdash A : \star \quad \Gamma \vdash x : A\]

\[\Gamma \vdash \text{ENV}_\text{WF}_\text{VAR}\]

\[\Gamma, \Delta \vdash D \notin \text{dom}(H) \quad \Gamma, \text{data} D \Delta^+, \Delta \vdash \Delta_i \rightarrow D \Delta^+: \star \quad \{d_i : \Delta_i \rightarrow D \Delta^+: \star \}_{i \in 1..j} \]

\[\Gamma \vdash \Delta, \text{data} D \Delta^+ \quad \{d_i : \Delta_i \rightarrow D \Delta^+: \star \}_{i \in 1..j} \]

\[\Gamma \vdash \text{ENV}_\text{WF}_\text{ABSDTYPE}\]

\[\Gamma \vdash \overline{a_i} : \Delta\]

\[\Gamma \vdash \overline{a_i} : \Delta \quad \Gamma \vdash a : A \quad \Gamma \vdash A : \star \quad \Gamma \vdash \overline{a_i} : [a/x] \Delta \quad \Gamma \vdash a \overline{a_i} : (x : A) \Delta \]

\[\Gamma \vdash \text{TL}_\text{CONS}\]

\[\Gamma \vdash \text{TL}_\text{EMPTY}\]

\[\Gamma \vdash H : M\]

\[\Gamma \vdash \text{ET}_\text{TYPE}\]

\[\Gamma \vdash n : D \overline{a_i}\]

\[\Gamma \vdash M : \star\]

\[\text{data} D \overline{\Xi}^+ \quad \{d_i : \overline{\Xi}_i \rightarrow D \overline{\Xi}^+: \star \}_{i \in 1..j} \in H\]

\[\forall i. \ H, [\overline{a_i} / \overline{\Xi}^+] \overline{\Xi}_i, y : n = d_i \overline{\Xi}_i \vdash m_i : M\]

\[\forall i. \ \{y\} \cup \text{dom}^-(\overline{\Xi}_i) \notin \text{FV}(m_i)\]

\[\overline{a_i} \in \text{dom}^+(\overline{\Xi}_i) \]

\[\Gamma \vdash \text{case} n \text{ of } \{d_i \overline{a_i} \Rightarrow m_i \}_{i \in 1..j} : M\]

\[\Gamma \vdash \text{ET}_\text{CASE}\]

\[\Gamma \vdash x : M \in H\]

\[\Gamma \vdash H\]

\[\Gamma \vdash x : M \quad \text{ET}_\text{VAR}\]

---

**A.6 Unannotated type system**

\[H \vdash m : M\]

\[H \vdash H : \star \quad \text{ET}_\text{TYPE}\]

\[H \vdash n : D \overline{a_i}\]

\[H \vdash M : \star \quad \text{ET}_\text{VAR}\]
\[
H \vdash M : \star \quad H, x : M \vdash N : \star \\
\text{ET}_{\text{PI}}
\]

\[
H \vdash (x : M) \to N : \star \\
\text{ET}_{\text{PI}}
\]

\[
H \vdash M : \star \quad H, x : M \vdash N : \star \\
H \vdash [x : M] \to N : \star \\
\text{ET}_{\text{PI}}
\]

\[
data D \Xi^+ \text{ where } \{ d_i : \Xi_i \to D \Xi^+_i \}_{i \in 1..j} \in H \\
H \vdash M_i : \Xi^+ \\
\text{ET}_{\text{TCON}}
\]

\[
data D \Xi^+ \in H \\
H \vdash M_i : \Xi^+ \\
\text{ET}_{\text{ABSTCON}}
\]

\[
data D \Xi^+ \text{ where } \{ d_i : \Xi_i \to D \Xi^+_i \}_{i \in 1..j} \in H \\
H \vdash M_i : \Xi^+ \\
H \vdash m_i : [M_i/\Xi] \Xi_i \\
\text{ET}_{\text{DCON}}
\]

\[
H, x : M \vdash n : N \\
H \vdash \lambda x.n : (x : M) \to N \\
\text{ET}_{\text{ABS}}
\]

\[
H, x : M \vdash n : N \\
x \notin \text{FV}(n) \\
H \vdash \lambda [\_].n : [x : M] \to N \\
\text{ET}_{\text{IABS}}
\]

\[
H, f : M \vdash u : M \\
H \vdash M : \star \\
M \text{ is } (x : M_1) \to M_2 \text{ or } [x : M_1] \to M_2 \\
H \vdash \text{rec } f.u : M \\
\text{ET}_{\text{REC}}
\]

\[
H \vdash m : (x : M) \to N \\
H \vdash n : M \\
H \vdash [n/x]N : \star \\
H \vdash m : [n/x]N \\
\text{ET}_{\text{APP}}
\]

\[
H \vdash m : [x : M] \to N \\
H \vdash u : M \\
H \vdash m[\_]: [u/x]N \\
\text{ET}_{\text{IAPP}}
\]

\[
H \vdash M : \star \\
H \vdash \text{abort} : M \\
\text{ET}_{\text{ABORT}}
\]

\[
H \vdash m : M \\
H \vdash n : N \\
H \vdash m = n : \star \\
\text{ET}_{\text{EQ}}
\]

\[
m \not\in n \\
H \vdash m = n : \star \\
\text{ET}_{\text{JOIN}}
\]
\[
\begin{align*}
H \vdash u_1 : M_1 = N_1 & \quad \ldots \quad H \vdash u_i : M_i = N_i \\
H \vdash m : [M_1/x_1] \ldots [M_i/x_i]M \\
H \vdash [N_1/x_1] \ldots [N_i/x_i]M : * & \quad \text{ET_CONV} \\
H \vdash m : [N_1/x_1] \ldots [N_i/x_i]M \\
H \vdash u_1 : (x : M_1) \rightarrow N_1 = (x : M_2) \rightarrow N_2 & \quad \text{ET_INJDOM} \\
H \vdash u_1 : (x : M) \rightarrow N_1 = (x : M) \rightarrow N_2 \\
H \vdash u : M & \quad \text{ET_INJDOM} \\
H \vdash m : [M_1/x]N_1 = [u/x]N_2 & \quad \text{ET_INJRNG} \\
H \vdash u_1 : [x : M_1] \rightarrow N_1 = [x : M_2] \rightarrow N_2 & \quad \text{ET_INJDOM} \\
H \vdash u_1 : [x : M] \rightarrow N_1 = [x : M] \rightarrow N_2 \\
H \vdash u : M & \quad \text{ET_INJRNG} \\
H \vdash m : [u/x]N_1 = [u/x]N_2 & \quad \text{ET_INJCON} \\
\vdash H & \quad \text{H is a well-formed environment} \\
\vdash \text{EENV._WF_EMPTY} \\
\vdash H \quad x \notin \text{dom}(H) \\
H \vdash M : * & \quad \text{EENV._WF_VAR} \\
\vdash H, x : M \\
\vdash H, \Xi \\
D \notin \text{dom}(H) \\
\forall i. \quad d_i \notin \text{dom}(H) \\
\forall i. \quad H, \text{data} D \Xi^+ \vdash \Xi_i \rightarrow D \Xi^+ : * & \quad \text{EENV._WF_DTYPE} \\
\vdash H, \text{data} D \Xi^+ \text{ where } \{ d_i : \Xi_i \rightarrow D \Xi^+ \}_{i \in 1..j} \\
\vdash H, \Xi \quad D \notin \text{dom}(H) & \quad \text{EENV._WF_ABSDTYPE} \\
\vdash H, \text{data} D \Xi^+ \\
\vdash H, m_i : \Xi & \quad \text{ETL._EMPTY} \\
H \vdash m : M \\
H \vdash M : * \\
H \vdash m_i : [m/x]\Xi & \quad \text{ETL._CONS} \\
\vdash H, m m_i : (x : M)\Xi \\
\vdash H, u : M \\
H \vdash M : * \\
H \vdash m_i : [u/x]\Xi & \quad \text{ETL._CONS} \\
\vdash H, \big| m_i : [x : M]\Xi
\end{align*}
\]
B Proofs

B.1 Correctness of annotated system

Lemma 16 (Decidability of type checking). There is an algorithm which given $\Gamma$ and $a$ computes the unique $A$ such that $\Gamma \vdash a : A$, or reports that there is no such $A$.

Proof. The algorithm follows the structure of $a$—for each syntactic form we see that only one typing rule could apply, and that the premises of that rule are uniquely determined. □

Lemma 17 (Correctness of erasure). If $\Gamma \vdash a : A$, then $|\Gamma| \vdash |a| : |A|$.

Proof. Easy induction—each annotated rule corresponds directly to an unannotated rule where all terms have gone through erasure. □

B.2 Facts about parallel reduction

Definition 18. The head constructor of an expression is defined as follows:

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

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

Lemma 19. If $m \rightsquigarrow_p m'$ and $\text{hd}(m)$ is defined, then $\text{hd}(m) = \text{hd}(m')$.

Proof. By inspecting the definition of $\rightsquigarrow_p$ we see that it always preserves the head constructor of a term. □

Lemma 20. If $m \not\rightsquigarrow m'$, then $m$ and $m'$ do not have different head constructors.

Proof. Expanding the definition of $\not\rightsquigarrow$ we know that $m \rightsquigarrow_p n$ and $m' \rightsquigarrow_p n$ for some $n$. If $m$ and $m'$ had (defined and) different head constructors, then by repeatedly applying Lemma 19 we would get that $n$ had two different head constructors, which is impossible. □

Lemma 21 (Injectivity of $\not\rightsquigarrow$).

- If $m_1 = n_1 \not\rightsquigarrow m_2 = n_2$, then $m_1 \not\rightsquigarrow m_2$ and $n_1 \not\rightsquigarrow n_2$.
- If $D M_1 \not\rightsquigarrow D M_2$, then $M_1 \not\rightsquigarrow M_2$.
- If $(x : M_1) \rightarrow N_1 \not\rightsquigarrow (x : M_2) \rightarrow N_2$ then $M_1 \not\rightsquigarrow M_2$ and $N_1 \not\rightsquigarrow N_2$.
- If $[x : M_1] \rightarrow N_1 \not\rightsquigarrow [x : M_2] \rightarrow N_2$ then $M_1 \not\rightsquigarrow M_2$ and $N_1 \not\rightsquigarrow N_2$.
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 $\gamma$, we have that $m_1 = n_1 \leadsto_p N$ and $m_2 = n_2 \leadsto_p N$ for some $N$.

By lemma 20 we know that $N$ has the shape $n = m$. So it suffices to prove that, for any $n_1,m_1$, if $n_1 = m_1 \leadsto_p n = m$, then $n_1 \leadsto_p n$. 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 22. If $u \leadsto_p u'$ and $m \leadsto_p m'$, then $[u/x]m \leadsto_p [u'/x]m'$.

Proof. By induction on $m \leadsto_p m'$

Lemma 23. If $M \gamma M'$, then $[u/x]M \gamma [u/x]M'$.

Proof. Expanding the definition of $\gamma$ we get $M \leadsto_p M_1$ and $M' \leadsto_p M_1$ for some $M_1$. Repeatedly applying Lemma 22 we then get $[u/x]M \leadsto_p [u/x]M_1$ and $[u/x]M' \leadsto_p [u/x]M_1$ as required.

Lemma 24 (One-step diamond property for $\leadsto_p$). If $m \leadsto_p m_1$ and $m \leadsto_p m_2$, then there exists some $m'$ such that $m_1 \leadsto_p m'$ and $m_2 \leadsto_p m'$.

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, $m_1 m_2 \leadsto_p m_11 m_21$ and $m_1 m_2 \leadsto_p m_12 m_22$, where $m_1 \leadsto_p m_11$, $m_1 \leadsto_p m_21$, $m_2 \leadsto_p m_21$ and $m_2 \leadsto_p m_22$.
  By the induction hypothesis for $m_1$, there exists $m'_1$, such that $m_11 \leadsto_p m'_1$ and $m_12 \leadsto_p m'_2$. Similarly for $m_2$. So by SC_APP we have $m_11 m_21 \leadsto_p m'_1 m'_2$ and $m_12 m_22 \leadsto_p m'_1 m'_2$ 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 $m_0 \leadsto_p m_02$. So we have:

$$
(\lambda x.m_0) u \leadsto_p [u_1/x]m_01 \quad \text{where } m_0 \leadsto_p m_01 \text{ and } u \leadsto_p u_1.
$$

$$
(\lambda x.m_0) u \leadsto_p (\lambda x.m_02) u_2 \quad \text{where } m_0 \leadsto_p m_02 \text{ and } u \leadsto_p u_2.
$$

Now by IH we get $m'_0$ and $u'$. By substitution (lemma 22) we get $[u_1/x]m_01 \leadsto_p [u'/x]m'_0$, while by SC_APPBETA we get $(\lambda x.m_02) u_2 \leadsto_p [u'/x]m'_0$. So the terms are joinable as required.

On the other hand, if both the reductions are by SC_APPREC, then we have

$$
(\lambda x.m_0) u \leadsto_p [u_1/x]m_01 \quad \text{where } m_0 \leadsto_p m_01 \text{ and } u \leadsto_p u_1.
$$

$$
(\lambda x.m_0) u \leadsto_p [u_2/x]m_02 \quad \text{where } m_0 \leadsto_p m_02 \text{ and } u \leadsto_p u_2.
$$

Then by IH we again get $m'_0$ and $u'$, and by substitution (twice), the two terms are again joinable at $[u'/x]m'_0$.
\begin{itemize}
  \item One reduction is SC_APPREC. So \( m_1 m_2 \) must be \((\text{rec} \ f \ u) \ u_2 \). 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 \( \text{rec} \ f \ u \) can step is by congruence to \( \text{rec} \ f \ u \sim_p \text{rec} \ f \ u \), so we have
  \[
  (\text{rec} \ f \ u) \ u_2 \sim_p (\text{rec} \ f \ u)[u_1] \ u_21 \quad \text{where} \quad u_1 \sim_p u_11 \text{ and } u_2 \sim_p u_21
  
  (\text{rec} \ f \ u) \ u_2 \sim_p (\text{rec} \ f \ u) \ u_22 \quad \text{where} \quad u_1 \sim_p u_12 \text{ and } u_2 \sim_p u_22
  \]
  
  Now, by IH we have \( u_1' \) and \( u_2' \). By congruence, \( \text{rec} \ f \ u \sim_p \text{rec} \ f \ u \), so by substitution (lemma 22) we get \([\text{rec} \ f \ u][u_11] \sim_p [\text{rec} \ f \ u][u_1]' \), and then by congruence ((\( \text{rec} \ f \ u \))[u_1] \ u_21 \sim_p ([\text{rec} \ f \ u][u_1]' \ u_2') \). Meanwhile, by SC_APPREC we have \((\text{rec} \ f \ u) \ u_22 \sim_p (\text{rec} \ f \ u)[u_1]' \ u_2' \) 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.

  \item One reduction is SC_ABORT. So \( m_1 m_2 \) must be \( \text{abort} \ m_2 \) or \( u_1 \) \( \text{abort} \). Then by considering possible cases, we see that the other reduction must be SC_ABORT or SC_APP (the \( \beta \)-rules cannot match because \( \text{abort} \) is not a value). If the other rule is SC_ABORT we are trivially done, if it is SC_APP then the term steps to \( u_1' \) \( \text{abort} \), which can step to \( \text{abort} \) as required.
\end{itemize}

\[\blacksquare\]

\textbf{Lemma 25} (Confluence of \( \sim_p \)). If \( m \sim_p^* m_1 \) and \( m \sim_p^* m_2 \), then there exists some \( m' \) such that \( m_1 \sim_p^* m' \) and \( m_2 \sim_p^* m' \).

\textit{Proof.} This is a simple corollary of the 1-step version (lemma 24), by “diagram-chasing to fill in the rectangle” (see e.g. \cite{5}, lemma 3.2.2).

\[\blacksquare\]

\textbf{Lemma 26} (\( \sim \) is an equivalence relation).

1. For any \( m, m \sim m \).

2. If \( m \not\sim n \) then \( n \not\sim m \).

3. If \( m_1 \sim m_2 \) and \( m_2 \sim m_3 \), then \( m_1 \sim m_3 \).

\textit{Proof.} (1) and (2) are immediate just by expanding the definition of \( m \sim n \) and \( m \sim_p n \).

For (3), by expanding the definition we have some \( n_1 \) and \( n_2 \) such that \( m_1 \sim_p n_1, m_2 \sim_p n_1, m_2 \sim_p n_2 \) and \( m_3 \sim_p n_2 \). So by confluence (lemma 25) applied to the two middle ones, there exists some \( n \) such that \( n_1 \sim_p n \) and \( n_2 \sim_p n \). Then we have \( m_1 \sim_p n \) and \( m_3 \sim_p n \) as required.

\[\blacksquare\]

\textbf{Lemma 27.} If \( N \sim_p N' \), then \([N/x]M \sim_p [N'/x]M \).

\textbf{Lemma 28.} If \( N \not\sim N' \), then \([N/x]M \not\sim [N'/x]M \).

\textit{Proof.} Expanding the definition of \( \sim \) we have \( N \sim_p N_1 \) and \( N' \sim_p N_1 \) for some \( N_1 \). Now repeatedly apply Lemma 27 to get \([N/x]M \sim_p [N_1/x]M \) and \([N'/x]M \sim_p [N_1/x]M \).

\[\blacksquare\]

\textbf{Lemma 29.} If \( m \sim_p m' \), then \( \text{FV}(m') \subseteq \text{FV}(m) \).
B.3 Structural properties

Lemma 30 (Free variables in typing judgments). If \( H \vdash m : M \), then \( \text{FV}(m) \subseteq \text{dom}(H) \) and \( \text{FV}(M) \subseteq \text{dom}(H) \).

Lemma 31 (Regularity for contexts). If \( H \vdash m : M \) then \( \vdash H \).

Lemma 32 (Regularity for variable lookup). If \( H_1, x : M, H_2 \vdash n : N \), then \( H_1 \vdash M : \star \).

Lemma 33 (Context conversion). If \( H, x : M, H' \vdash n : N \) and \( H \vdash \text{join} : M = M' \) and \( H \vdash M' : \star \), then \( H, x : M', H' \vdash n : N \).

Lemma 34 (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 35 (Regularity). If \( H \vdash m : M \), then \( \vdash M : \star \).

Lemma 36 (Data constructors are unique in the environment). If \( \vdash H \), and

\[
\text{data } D \Xi^{+} \text{ where } \{ d_i : \Xi_i \rightarrow D \Xi^{+}_{i \in 1..j} \} \in H
\]

and

\[
\text{data } D' \Xi'^{+} \text{ where } \{ d'_i : \Xi'_i \rightarrow D' \Xi'^{+}_{i \in 1..j} \} \in H,
\]

and \( d_k = d'_i \), then \( D = D' \) and \( \Xi^{+} = \Xi'^{+} \) and \( \Xi_k = \Xi'_i \).

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 \( \lambda \).

We first need some basic facts about equality.

Lemma 37 (Inversion for equality). If \( H \vdash m = n : M_0 \) then, \( H \vdash m : M \) and \( H \vdash n : N \).

Proof. Induction of \( H \vdash m = n : M_0 \). The only cases where the subject of the conclusion of the rule is an equality are \text{ET.EQ} \text{ (where we get the result as a premise to the rule) and ET.CONV \text{ (direct by induction).}}

Lemma 38 (Proof irrelevance for equality proofs). If \( H \vdash u : M = N \), then \( H \vdash \text{join} : M = N \)

Proof. By regularity (lemma 35) we know \( H \vdash M = N : \star \), so by inversion (lemma 37) we have \( H \vdash M : \star \).

So by ET.TJOIN, \( H \vdash \text{join} : M = M \).

Now by ET.TCONV we get \( H \vdash \text{join} : M = N \) by using the assumed proof \( u \) to change \( M \) to \( N \).

Lemma 39 (Propositional equality is an equivalence relation).

- If \( H \vdash m : M \), then \( H \vdash \text{join} : m = m \).
- If \( H \vdash u : m_1 = m_2 \), then \( H \vdash \text{join} : m_2 = m_1 \).
- If \( H \vdash u : m_1 = m_2 \) and \( H \vdash u' : m_2 = m_3 \), then \( H \vdash \text{join} : m_1 = m_3 \).

Proof. (1) is just a special case of ET.JOIN.

(2) We have \( H \vdash \text{join} : m_1 = m_1 \), 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 40 (Inversion for \( \lambda \)). If \( H \vdash \lambda x. n : M \), then \( H \vdash \text{join} : (x : M_1) \rightarrow N_1 = M \) 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 \( \lambda \) as the subject of the conclusion.

Case **ET_ABS** The rule looks like

\[
\frac{H, x : M \vdash n : N}{H \vdash \lambda x. n : (x : M) \rightarrow N}^{\text{ET_ABS}}
\]

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

Case **ET_CONV** The rule looks like

\[
\begin{align*}
H \vdash u_1 : M_1 = N_1 & \quad \ldots \quad H \vdash u_i : M_i = N_i \\
H \vdash m : [M_1/x_1] \ldots [M_i/x_i]M & \\
H \vdash [N_1/x_1] \ldots [N_i/x_i]M : \ast \\
\hline
H \vdash m : [N_1/x_1] \ldots [N_i/x_i]M
\end{align*}
\]

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 \([35][37][39]\)) we have \( H \vdash \text{join} : [M_1/x_1] \ldots [M_i/x_i]M = [M_1/x_1] \ldots [M_i/x_i]M \). By applying ET_CONV using the proof \( u_1 \ldots u_i \) we get \( H \vdash \text{join} : [M_1/x_1] \ldots [M_i/x_i]M = [N_1/x_1] \ldots [N_i/x_i]M \). Then by transitivity (lemma \([39]\)) 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 41 (Inversion for irrelevant \( \lambda \)). If \( H \vdash \lambda [] n : M \), then \( H \vdash \text{join} : [x : M_1] \rightarrow N_1 = M \) for some \( M_1 \) and \( N_1 \), and \( H, x : M_1 \vdash n : N_1 \) where \( x \notin \text{FV}(n) \).

Lemma 42 (Inversion for rec). If \( H \vdash \text{rec} f, u : M \), then \( H \vdash \text{join} : M = M_1 \) and \( H, f : M_1 \vdash u : M_1 \) for some \( M_1 \) such that \( H \vdash M_1 : \ast \) and \( M_1 \) is an relevant or irrelevant arrow type.

Lemma 43 (Inversion for dcon). If \( H \vdash d \overline{m} : M \), then \( H \vdash \text{join} : D \overline{N}_i = M \) for some \( \overline{N}_i \) such that:

- data \( D \Xi^+ \) where \( \{ \overline{d}_i : \overline{E}_i \rightarrow D \Xi^+ \}_{i \in 1..j} \in H \) and \( d \) is \( d_i \) for one of the constructors in the declaration.
- \( H \vdash \overline{N}_i : \Xi \)
- \( H \vdash \overline{m}_i : [\overline{N}_i / \Xi] \Xi_j \)

B.5 Preservation

Lemma 44 (A conversion rule for value lists). If \( H \vdash \overline{u} : [\overline{M}_i / \overline{N}_i] \Xi \) and \( \forall i. H \vdash \text{join} : M_i = N_i \) and \( H, [\overline{N}_i / \Xi] \Xi \), then \( H \vdash \overline{u} : [\overline{N}_i / \Xi] \Xi \).

Proof. We proceed by induction on the structure of \( \Xi \).

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

\[
H \vdash u : [\bar{M}_i/\bar{\gamma}]M \\
H \vdash [\bar{M}_i/\bar{\gamma}]M : \star \\
H \vdash \bar{u}_i : [u/x][\bar{M}_i/\bar{\gamma}]\Xi \\
\overline{H \vdash u \bar{u}_i : (x : [\bar{M}_i/\bar{\gamma}]M)[\bar{M}_i/\bar{\gamma}]\Xi } \text{ ETL_CONS}
\]

and

\[
\vdash H, x : [\bar{N}_i/\bar{\gamma}]M, [\bar{N}_i/\bar{\gamma}]\Xi.
\]

By inversion on this, we have \(H \vdash [\bar{N}_i/\bar{\gamma}]M : \star\).

Now since we know \(\forall i. H \vdash \text{join} : M_i = N_i\) and \(H \vdash [\bar{N}_i/\bar{\gamma}]M : \star\), then by ETL_CONV we have \(H \vdash u : [\bar{N}_i/\bar{\gamma}]M\).

By substitution (lemma 34) we get \(\vdash H, [u/x][\bar{N}_i/\bar{\gamma}]\Xi\). We know \(u\) is well-typed, so by ET_JOIN we have \(H \vdash \text{join} : u = u\). Then by IH, taking the multi-substitution to be \([u/x][\bar{M}_i/\bar{\gamma}]\), we get \(H \vdash \bar{u}_i : [u/x][\bar{N}_i/\bar{\gamma}]\Xi\). So re-applying ETL_CONS we get

\[
H \vdash u \bar{u}_i : (x : [\bar{N}_i/\bar{\gamma}]M)[\bar{N}_i/\bar{\gamma}]\Xi
\]

as required.

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

\[
H \vdash u : [\bar{M}_i/\bar{\gamma}]M \\
H \vdash [\bar{M}_i/\bar{\gamma}]M : \star \\
H \vdash \bar{u}_i : [u/x][\bar{M}_i/\bar{\gamma}]\Xi \\
\overline{H \vdash \top \bar{u}_i : (x : [\bar{M}_i/\bar{\gamma}]M)[\bar{M}_i/\bar{\gamma}]\Xi } \text{ ETL_CONS}
\]

By reasoning as in the previous case we get \(H \vdash u : [\bar{N}_i/\bar{\gamma}]M\) and \(H \vdash [\bar{N}_i/\bar{\gamma}]M : \star\) and \(H \vdash \bar{u}_i : [u/x][\bar{N}_i/\bar{\gamma}]\Xi\). Then re-apply ETL_CONS.

\qed

Theorem 45 (Preservation).

1. If \(H \vdash m : M\) and \(m \leadsto^p m'\), then \(H \vdash m' : M\).

2. If \(H \vdash \bar{m}_i : [n_1/y_1] \ldots [n_i/y_i]\Xi\) and \(\forall i. m_i \leadsto^p m'_i\) and \(\forall j. n_j \leadsto^p n'_j\), then \(H \vdash \bar{m}_i' : [n'_1/y_1] \ldots [n'_i/y_i]\Xi\).

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

Cases ETL_TYPE, ETL_VAR, ETL_ABORT, ETL_JOIN, ETL_INJDOM, ETL_INJRNG, ET_INJDOM, ET_INJRNG, ET_INJTCON.

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

Case ETL_CASE. The rule looks like

\[
\Gamma \vdash b : D \bar{B}_i \\
\Gamma \vdash A : \star \\
\text{data } D \Delta^+ \text{ where } \{ \bar{d}_i : \Delta_i \rightarrow D \Delta^+ \}_{i \in [1..]} \in \Gamma \\
\forall i. \Gamma, [\bar{B}_i/\Delta^+] \Delta_i : y \vdash b = d_i \Delta_i \vdash a_i : A \\
\forall i. \{y\} \cup \text{dom } (\Delta_i) \# \text{FV}(\{a_j\}) \\
\Gamma \vdash \text{case } b \text{ as } [y] \text{ of } \{ d_i \Delta_i \Rightarrow a_i \}_{i \in [1..]} : A \text{ T_CASE}
\]

We consider the ways the expression case of \(\{ d_i \bar{x}_{ij} \Rightarrow m_{ij} \}_{i \in [1..]}\) may step:
To case \( n' \) of \( \{ d_j \xi_j \Rightarrow m_j^{i \in 1..k} \} \) by SP.CASE when \( n \rightsquigarrow_p n' \) and \( \forall j, m_j \rightsquigarrow_p m_j' \).

By IH we get \( H \vdash n : D \overline{\eta} \). Also by IH, for each \( j \) we have

\[
H, [\overline{\eta}/\Xi_j], y : n = d_j \xi_j \vdash m_j' : M.
\]

Now by regularity (lemma 32) and inversion (lemma 37) we know that \( d_j \xi_j \) is well-typed in the context \( H, [\overline{\eta}/\Xi^+] \Xi_j \). And we already observed that \( n \) and \( n' \) are well-typed. So \( n = d_j \xi_j \) and \( n' = d_j \xi_j \) are well-formed equations. Since \( n = d_j \xi_j \rightsquigarrow_p n' = d_j \xi_j \), by ET_JOIN we have \( H, [\overline{\eta}/\Xi^+] \Xi_j \vdash \text{join} : (n = d_j \xi_j) = (n' = d_j \xi_j) \). So by context conversion (lemma 33) we have

\[
H, [\overline{\eta}/\Xi^+] \Xi_j, y : n' = d_j \xi_j \vdash m_j' : M.
\]

Then we can re-apply ET_CASE to get the required

\[
H \vdash \text{case } n' \text{ of } \{ d_j \xi_j \Rightarrow m_j^{i \in 1..k} \} : M.
\]

To \( [\overline{\eta}'/\xi_i]m_i' \) by SP.CASEBETA when \( n = d_i \overline{\eta}_i \), and \( \forall i. u_i \rightsquigarrow_p u_i' \) and \( m_i \rightsquigarrow_p m_i' \). Notice that the step rule in particular requires that \( d_i \) is one of the branches of the case expression.

By inversion (lemma 43) on the premise \( H \vdash d_i \overline{\eta}_i : D \overline{\eta}_i \), we know that \( H \vdash n : D \overline{\eta}_i \) with \( H \vdash \text{join} : D \overline{\eta}_i = D \overline{\eta}_i \) and \( H \vdash \overline{\eta}_i : \Xi^+ \) and \( H \vdash \overline{\eta}_i : [\overline{\eta}_i/\Xi^+] \Xi_i \). (We know that the \( D \), \( \Xi \) and \( \Xi_i \) 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 36)).

By the rule INJTCON we get \( H \vdash \text{join} : N_i = n_i \) for each \( i \). So by value-list conversion (lemma 44) we have \( H \vdash \overline{\eta}_i : [\overline{\eta}_i/\Xi^+] \Xi_i \).

We next claim that \( H, y : d_i \overline{\eta}_i = d_i \overline{\eta}_i' \vdash [\overline{\eta}_i'/\Xi_i]m_i' : M \). To show this we prove a more general claim: for any prefix \( u'_1 \ldots u'_k \) of \( \overline{\eta}_i \), and supposing \( \Xi_i \) has the form \( (x_1 : M_1) \ldots (x_k : M_k) \Xi_0 \), we have

\[
H, [u'_1/x_1] \ldots [u'_k/x_k] [\overline{\eta}_i'/\Xi^+] \Xi_0, y : [u'_1/x_1] \ldots [u'_k/x_k](d_i \overline{\eta}_i = d_i \overline{\eta}_i) \vdash [u'_1/x_1] \ldots [u'_k/x_k]m_i : [u'_1/x_1] \ldots [u'_k/x_k]M
\]

This follows by induction on \( k \) (by applying substitution, lemma 34 \( k \) times). So in particular, we have

\[
H, y : [\overline{\eta}_i'/\Xi_i](d_i \overline{\eta}_i = d_i \overline{\eta}_i) \vdash [\overline{\eta}_i'/\Xi_i]m_i : [\overline{\eta}_i'/\Xi_i]M
\]

But by the premises \( H \vdash M : * \) and \( H \vdash d_i \overline{\eta}_i : D \overline{\eta}_i \) together with lemma 30 we know that \( x_1 \) are not free in \( d_i \overline{\eta}_i \) or \( M \), so this simplifies to

\[
H, y : d_i \overline{\eta}_i = d_i \overline{\eta}_i' \vdash [\overline{\eta}_i'/\Xi_i]m_i' : M
\]

as we claimed.

Next, we know \( d_i \overline{\eta}_i \) is well-typed because that is a premise to the rule. By the mutual IH we have that \( H \vdash d_i \overline{\eta}_i' : D \overline{\eta}_i \), so \( d_i \overline{\eta}_i' \) is well-typed too. So by ET_JOIN we have \( H \vdash \text{join} : d_i \overline{\eta}_i = d_i \overline{\eta}_i' \). Then by substitution (lemma 34) again we have

\[
H \vdash \text{join}[y] [\overline{\eta}_i'/\Xi_i]m_i' : [\text{join}[y]]M.
\]

But as a side-condition to the rule (plus lemma 29) we know that \( y \not\in \text{FV}(m_i') \), and \( y \) is a bound variable which we can pic so that \( y \not\in \text{FV}(M) \). So we have in fact show the required

\[
H \vdash [\overline{\eta}_i'/\Xi_i]m_i' : M.
\]
• To abort by SP_ABORT. By regularity (lemma 35) on the original typing derivation we know that $H \vdash M : \star$, so by ET_ABORT we have $H \vdash \text{abort} : M$ as required.

Case ET.PI. The rule looks like

$$
\begin{align*}
H \vdash M : \star & \quad H, x : M \vdash N : \star \\
\hline
H \vdash (x : M) \rightarrow N : \star
\end{align*}
$$

ET.PI

The only way $(x : M) \rightarrow N$ can step (except trivially by SP.REFL) is by SP.PI:

$$(x : M) \rightarrow N \rightsquigarrow_p (x : M') \rightarrow N'$$

where $M \rightsquigarrow_p M'$ and $N \rightsquigarrow_p N'$

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

By IH we immediately get $H \vdash M' : \star$ and $H, x : M \vdash N' : \star$. Since $M \rightsquigarrow_p M'$ we also have $M \not\succcurlyeq M'$, so applying ET.Join we get $H \vdash \text{join} : M = M'$. Then by context conversion (lemma 33) 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

$$
\begin{align*}
H & \vdash \xi_i \rightarrow D \\
\hline
H \vdash \lambda x. n : (x : M) \rightarrow N
\end{align*}
$$

ET.ABS

The only non-trivial way the expression $\lambda x. n$ can step is by SP.ABS to $\lambda x. n'$ when $n \rightsquigarrow_p n'$. By IH we get $H, x : M \vdash n' : N$. So re-applying ET.ABS we get $H \vdash \lambda x. n' : (x : M) \rightarrow N$ 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 29.

Case ET.TCON. The rule looks like

$$
\begin{align*}
\text{data } & D \Xi^+ \text{ where } \{ d_i : \Xi_i \rightarrow D \Xi^+_i \text{ } i \in 1..j \} \in H \\
H & \vdash M_i : \Xi^+ \\
\hline
H \vdash DM_i : \star
\end{align*}
$$

ET.TCON

The only way the expression can step is by SP.TCON, so $\forall i. M_i \rightsquigarrow_p M_i'$. By the mutual IH, we get $H \vdash M_i' : \Xi^+$. So by re-applying ET.TCON we have $H \vdash DM_i : \star$ as required.

Case ET.ABSTCON. Similar to the previous case.

Case ET.DCON. The rule looks like

$$
\begin{align*}
\text{data } & D \Xi^+ \text{ where } \{ d_i : \Xi_i \rightarrow D \Xi^+_i \text{ } i \in 1..j \} \in H \\
H & \vdash M_i : \Xi^+ \\
\hline
H \vdash \overline{m_i} : [M_i/\Xi]\Xi_i \\
\hline
H \vdash d_k \overline{m_i} : DM_i
\end{align*}
$$

ET.DCON

By the mutual induction hypothesis (with an empty substitution) we get $H \vdash M_i : \Xi$ and $H \vdash \overline{m_i} : [M_i/\Xi]\Xi_i$. Conclude by re-applying ET.DCON.
We consider how the expression $m n$ may step.

- To $m' n'$ by SP_APP if $m \sim_p m'$ and $n \sim_p n'$.
  By IH we have $H \vdash m' : (x : M) \to N$ and $H \vdash n' : M$. By lemma \ref{app:27} we know $[n/x]N \sim_p [n'/x]N$, so also by IH we have $H \vdash [n'/x]N : \ast$. So re-applying ET_APP we get $H \vdash m' n' : [n'/x]N$.
  Finally, by ET_JOIN we have $H \vdash \text{join} : [n/x]N = [n'/x]N$, 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 $m_1 \sim_p m'_1$ and $u \sim_p u'$.
  By IH we have $H \vdash u' : M$. Also, $\lambda x. m_1 \sim_p \lambda x. m'_1$ so by IH we have $H \vdash \lambda x. m'_1 : (x : M) \to N$. By inversion (lemma \ref{app:40}) we know that $H, x : M_1 \vdash m_1 : N_1$ for some $M_1, N_1$ such that $H \vdash \text{join} : (x : M_1) \to N_1 = (x : M) \to N$. By ET_INJDOM we have $H \vdash \text{join} : M_1 = M$, and by regularity (lemma \ref{app:32}) we have $H \vdash M_1 : \ast$, so by ET_CONV we get $H \vdash u' : M_1$. Now by substitution (lemma \ref{app:34}) we get

$$H \vdash [u'/x]m'_1 : [u'/x]N_1.$$ 

Now, by ET_INJRNG we have $H \vdash \text{join} : [u'/x]N_1 = [u'/x]N$. Also, by lemma \ref{app:27} we know $[u'/x]N \sim_p [u'/x]N$, and we noted above that $H \vdash [u'/x]N : \ast$, so by ET_JOIN we have $H \vdash \text{join} : [u'/x]N = [u'/x]N$. By symmetry and transitivity (lemma \ref{app:39}) this yields $H \vdash \text{join} : [u'/x]N_1 = [u'/x]N$.

Finally, we had $H \vdash [u'/x]N : \ast$ as a premise to the rule. So by ET_CONV we get the required

$$H \vdash [u'/x]m'_1 : [u'/x]N.$$ 

- To $([\text{rec} f. u]/f) u'_1$ by SP_APPREC if $m$ is $\text{rec} f. u$, $n$ is $u_2$, and $u_1 \sim_p u'_1$ and $u_2 \sim_p u'_2$.
  By IH we have $H \vdash u_2 : M$. Also, since $\text{rec} f. u \sim_p \text{rec} f. u$, by IH we have $H \vdash \text{rec} f. u : (x : M) \to N$, and since by lemma \ref{app:27} $[u_2/x]N \sim_p [u'_2/x]N$, by IH we get $H \vdash [u'_2/x]N : \ast$.
  By inversion (lemma \ref{app:42}) we know $H, f : M_1 \vdash u'_1 : M_1$ for some $M_1$ such that $H \vdash \text{join} : M_1 = (x : M) \to N$ and $H \vdash M_1 : \ast$ and such that $M_1$ is an arrow type.
  So by the ET_REC rule, we get $H \vdash \text{rec} f. u : M_1$. Then by substitution (lemma \ref{app:34}) we have $H \vdash [\text{rec} f. u]/f u'_1 : M_1$.
  By regularity (lemma \ref{app:35}) on the original premise of the rule we know $H \vdash (x : M) \to N : \ast$, so by ET_CONV we have $H \vdash [\text{rec} f. u]/f u'_1 : (x : M) \to N$. Then re-apply ET_APP to get

$$H \vdash ([\text{rec} f. u]/f u'_1) u'_2 : [u'_2/x]N.$$ 

As we noted above $[u_2/x]N \sim_p [u'_2/x]N$, and both expressions are well-kindred, so by ET_JOIN we know $H \vdash \text{join} : [u'_2/x]N = [u_2/x]N$. So by finally applying ET_CONV we get the required

$$H \vdash ([\text{rec} f. u]/f u'_1) u'_2 : [u_2/x]N.$$
To abort by \texttt{SP\_ABORT}. By regularity (lemma \texttt{35}) on the original premise we know \( H \vdash [n/x]N : \star \). So by ET\_ABORT we have \( H \vdash \text{abort} : [n/x]N \) as required.

**Case ET\_IAPP.** The typing rule looks like

\[
\frac{H \vdash m : [x:M] \to N \quad H \vdash u : M}{H \vdash m[u/x] : [u/x]N} \text{ET\_IAPP}
\]

We consider how the expression \( m[] \) may step:

- To \( m'[1] \) by \texttt{SP\_IAPP} if \( m \sim_p m' \). By IH we know \( H \vdash m' : [x:M] \to N \), so by re-applying ET\_IAPP we get \( H \vdash m'[1] : [u/x]N \) as required.

- To \( m'_1 \) by \texttt{SP\_IAPPBETA} if \( m \) is \( \lambda \cdot m_1 \) and \( m_1 \sim_p m'_1 \).

  Note that \( \lambda \cdot m_1 \sim_p \lambda \cdot m'_1 \), so by IH we get \( H \vdash \lambda \cdot : m_1' : [x:M] \to N \). Then by inversion (lemma \texttt{41}) we know \( H, x : M_1 \vdash n : N_1 \) for some \( M_1 \) and \( N_1 \) with \( H \vdash \text{join} : ([x:M_1] \to N_1) = ([x:M] \to N) \), and \( x \notin \text{FV}(m'_1) \).

  Now, we have \( H \vdash u : M \) as an assumption to the rule. By regularity (lemma \texttt{35}) on that assumption we get \( H \vdash M : \star \), and by ET\_INJDOM we have \( H \vdash \text{join} : M_1 = M \). So by ET\_CONV we get \( H \vdash u : M_1 \). Then by substitution (lemma \texttt{34}) 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\_INJDOM we get \( H \vdash \text{join} : [u/x]N_1 = [u/x]N \), and by regularity on the original derivation we have \( H \vdash [u/x]N : \star \). So by ET\_CONV we get the required

\[
H \vdash [u/x]N.
\]

- To \( ([\text{rec}\cdot f.u]/f')[] \) by \texttt{SP\_IAPPREC} if \( m \) is \( \text{rec}\cdot f.u \) and \( u_1 \sim_p u_1' \).

  Note that \( \text{rec}\cdot f.u \sim_p \text{rec}\cdot f.u \), so by IH we know \( H \vdash \text{rec}\cdot f.u : [x:M] \to N \). By inversion (lemma \texttt{42}) we get \( H, f : M_1 \vdash u' : M_1 \) for some arrow type \( M_1 \) such that \( H \vdash \text{join} : M_1 = [x:M] \to N \) and \( H \vdash M_1 : \star \). By ET\_REC we then have \( H \vdash \text{rec}\cdot f.u : M_1 \), hence by substitution (lemma \texttt{34}) we have

\[
H \vdash [\text{rec}\cdot f.u]/f'u_1' : M_1.
\]

By regularity (lemma \texttt{35}) applied to the original typing rule we know \( H \vdash [x:M] \to N : \star \), so by ET\_CONV we then have

\[
H \vdash [\text{rec}\cdot f.u]/f'u_1' : [x:M] \to N.
\]

So re-applying ET\_IAPP we get the required

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

- To abort by \texttt{SP\_ABORT}.

  By regularity (lemma \texttt{35}) applied to the original type rule we know \( H \vdash [u/x]N : \star \), so by ET\_ABORT we have

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

as required.
Case **ET_EQ.** The rule looks like

\[
\frac{H \vdash m : M \quad H \vdash n : N}{H \vdash m = n : \ast} \quad \text{ET_EQ}
\]

The only non-trivial way the expression \( m = n \) can step is by **SP_EQ** to \( m' = n' \), when \( m \sim_p m' \) and \( n \sim_p n' \). By IH we immediately 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

\[
\frac{H \vdash u_1 : M_1 = N_1 \quad \ldots \quad H \vdash u_i : M_i = N_i}{H \vdash m : [M_1/x_1] \ldots [M_i/x_i]M} \quad \text{ETL}
\]

\[
\frac{H \vdash [N_1/x_1] \ldots [N_i/x_i]M : \ast}{H \vdash m : [N_1/x_1] \ldots [N_i/x_i]M} \quad \text{ETL}
\]

and we know that \( m \sim_p m' \). Directly by IH we get \( H \vdash m' : [M_1/x_1] \ldots [M_i/x_i]M \), and conclude by re-applying **ET_CONV**.

The cases for \( H \vdash m_\rightarrow : \Xi \) are:

**Case ETL_EMPTY.** Trivial.

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

\[
\frac{H \vdash m : [n_1/y_1] \ldots [n_i/y_i]M}{H \vdash [n_1/y_1] \ldots [n_i/y_i]M} \quad \text{ETL}
\]

\[
\frac{H \vdash [n_1/y_1] \ldots [n_i/y_i]M : \ast}{H \vdash m : [n_1/y_1] \ldots [n_i/y_i]M} \quad \text{ETL}
\]

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

By repeatedly applying lemma \[27\] we know \([n_1/y_1] \ldots [n_i/y_i]M \sim_p [n'_1/y_1] \ldots [n'_i/y_i]M \), so by mutual IH we get \( H \vdash [n'_1/y_1] \ldots [n'_i/y_i]M : \ast \).

By **ET_Join** we then have \( H \vdash \text{join} : [n_1/y_1] \ldots [n_i/y_i]M = [n'_1/y_1] \ldots [n'_i/y_i]M \), so by **ET_CONV** we get \( H \vdash m : [n'_1/y_1] \ldots [n'_i/y_i]M \).

Finally, by the IH (using that \( m \sim_p m' \)) we have \( H \vdash m' : [m'/x][n'_1/y_1] \ldots [n'_i/y_i]\Xi \). So re-applying **ETL_CONS** we get the required

\[
H \vdash m' \overline{m}' : [n'_1/y_1] \ldots [n'_i/y_i]\Xi.
\]

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

\[
\frac{H \vdash u : [n_1/y_1] \ldots [n_i/y_i]M}{H \vdash [n_1/y_1] \ldots [n_i/y_i]M} \quad \text{ETL}
\]

\[
\frac{H \vdash [n_1/y_1] \ldots [n_i/y_i]M : \ast}{H \vdash m : [n_1/y_1] \ldots [n_i/y_i]M} \quad \text{ETL}
\]

By repeatedly applying lemma \[27\] we know \([n_1/y_1] \ldots [n_i/y_i]M \sim_p [n'_1/y_1] \ldots [n'_i/y_i]M \), so by mutual IH we get \( H \vdash [n'_1/y_1] \ldots [n'_i/y_i]M : \ast \).

By **ET_Join** we then have \( H \vdash \text{join} : [n_1/y_1] \ldots [n_i/y_i]M = [n'_1/y_1] \ldots [n'_i/y_i]M \), so by **ET_CONV** we get \( H \vdash u : [n'_1/y_1] \ldots [n'_i/y_i]M \).

Finally, by the IH (using that \( u \sim_p u \), reflexively) we have \( H \vdash \overline{m}' : [u/x][n'_1/y_1] \ldots [n'_i/y_i]\Xi \). So re-applying **ETL_CONS** we get the required

\[
H \vdash u \overline{m}' : [n'_1/y_1] \ldots [n'_i/y_i]\Xi.
\]
B.6 Progress

Lemma 46 (Soundness of equality). If \( H_D \vdash u : M \) and \( M \not\gamma \ (m_1 = n_1) \), then \( m_1 \not\gamma n_1 \).


The \( M \) in the conclusion of these rules have a defined head constructor, which is not \( = \). So by lemma 20 \( M \) 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

\[
\begin{align*}
  m & \not\gamma n \\
  H & \vdash m = n : \star \\
  H & \vdash \text{join} : m = n \text{ ET_JOIN}
\end{align*}
\]

By injectivity (lemma 21) we have have \( m \not\gamma m_1 \) and \( n \not\gamma n_1 \). And by assumption we have \( m \not\gamma n \).

So by symmetry and transitivity (lemma 26) we have \( m_1 \not\gamma n_1 \) as required.

Case ET_CONV. The rule looks like

\[
\begin{align*}
  H & \vdash u_1 : M_1 = N_1 \quad \ldots \quad H \vdash u_i : M_i = N_i \\
  H & \vdash m : [M_1/x_1] \ldots [M_i/x_i]M \\
  H & \vdash [N_1/x_1] \ldots [N_i/x_i]M : \star \\
  H & \vdash m : [N_1/x_1] \ldots [N_i/x_i]M \text{ ET_CONV}
\end{align*}
\]

and we are given that \([N_1/x_1] \ldots [N_i/x_i]M \not\gamma (m_1 = n_1)\). By the IH for \( m \) it suffices to show that \([M_1/x_1] \ldots [M_i/x_i]M \not\gamma (m_1 = n_1)\).

But by the IH for \( u_i \) we know \( M_i \not\gamma N_i \), so we get this by repeatedly applying lemma 28.

Case ET_INJRN. The rule looks like

\[
\begin{align*}
  H & \vdash u_1 : (x : M) \rightarrow N_1 = (x : M) \rightarrow N_2 \\
  H & \vdash u : M \\
  H & \vdash \text{join} : [u/x]N_1 = [u/x]N_2 \text{ ET_INJRN}
\end{align*}
\]

and we are given that \([(u/x)N_1 = [u/x]N_2] \not\gamma (m_1 = n_1)\).

By IH we get \((x : M) \rightarrow N_1 \not\gamma (x : M) \rightarrow N_2\), so by injectivity (lemma 21) we know \( N_1 \not\gamma N_2 \). Then by lemma 23 we get \([u/x]N_1 \not\gamma [u/x]N_2\) as required.

Case ET_INJDOM, ET_INJDOM, ET_INJRN, ET_INJTCO. Similar to the previous case.

Lemma 47 (Canonical forms). Suppose \( H_D \vdash u : M \). Then:

1. If \( M \not\gamma (x : M_1) \rightarrow M_2 \), then \( u \) is either \( \lambda x. u_1 \) or \( \text{rec } f. u \).

2. If \( M \not\gamma \ [x : M_1] \rightarrow M_2 \), then \( u \) is either \( \lambda [x]. u_1 \) or \( \text{rec } f. u \).

3. If \( M \not\gamma \ D \bar{M} \) then \( u \) is \( d \bar{u} \), where data \( D \bar{e} \) where \( \{ \overline{d_i} : \bar{e} \rightarrow D \bar{e} \} \in H_D \) and \( d \) is one of the \( d_i \).
Proof. By induction on \(H_D \vdash u : M\). The cases are:

**Cases** ET\_TYPE, ET\_PI, ET\_JPI, ET\_TCON, ET\_ABSTCON, ET\_EQ, ET\_JOIN, ET\_INJRNG, ET\_INJDOM, ET\_INJDOM, ET\_INJRNG, 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 20 \(M\) 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 20 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

\[
\begin{align*}
H, f : M & \vdash u : M \\
H & \vdash M : \star \\
M & \text{is } (x : M_1) \to M_2 \text{ or } [x : M_1] \to M_2 \\
H & \vdash \text{rec } f . u : M
\end{align*}
\]

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

\[
\begin{align*}
H & \vdash u_1 : M_1 = N_1 \\
& \ldots \\
H & \vdash u_i : M_i = N_i \\
H & \vdash m : [M_1/x_1] \ldots [M_i/x_i]M \\
H & \vdash [N_1/x_1] \ldots [N_i/x_i]M : \star \\
H & \vdash m : [N_1/x_1] \ldots [N_i/x_i]M
\end{align*}
\]

Suppose, for example, that \([N_1/x_1] \ldots [N_i/x_i]M \uparrow (x : M_1) \to N_1\). By the IH for \(m\) it suffices to show that \([M_1/x_1] \ldots [M_i/x_i]M \uparrow (x : M_1) \to N_1\).

But by soundness of equality (lemma 46), for each \(u_i\) we know \(M_i \uparrow N_i\), so we get this by repeatedly applying lemma 28.

\[\square\]

**Theorem 48** (Progress). If \(H_D \vdash m : M\), then either \(m\) is a value, \(m\) is abort, or \(m \sim \text{cbv } m'\) for some \(m'\).

**Proof.** By induction on \(H_D \vdash m : M\). The cases are:

**Cases** ET\_TYPE, ET\_VAR, ET\_PI, ET\_JPI, ET\_TCON, ET\_ABSTCON, ET\_EQ, ET\_JOIN, ET\_INJDOM, ET\_INJRNG, ET\_INJDOM, ET\_INJRNG, ET\_INJTCON, ET\_ABS, ET\_IABS, ET\_REC.

These rules have a value as a subject.

**Case** ET\_CASE. The typing rule looks like

\[
\begin{align*}
H & \vdash n : D\overline{m_i} \\
H & \vdash M : \star \\
\text{data } D & \Xi^+ \text{ where } \{d_i : \Xi_i \to D \Xi^+ \mid i \in 1 \ldots l \} \in H \\
\forall i. & \ H, [m_i/\Xi_i]^+ \Xi_i, y : n = d_i \Xi_i \vdash m_i : M \\
\forall i. & \ H, y \cup \text{dom }^-(\Xi_i) \# \text{FV } (m_i) \\
x_i & \text{ is } \text{dom }^+(\Xi_i) \\
H & \vdash \text{case } \{d_i \overline{x_i} \Rightarrow m_i \mid i \in 1 \ldots l\} : M
\end{align*}
\]
By IH, we have that $n$ is either a value, is abort, or steps. If it steps, the entire expression steps by $\textsc{sc\_ctx}$. If it is abort, the entire expression steps by $\textsc{sc\_abort}$. Finally, suppose $n$ is a value. By canonical forms (lemma 47) we know that it must be of the form $d\overline{m}$, and defined by a datatype declaration for $D$ in the context. Since datatype declarations are unique (lemma 36), 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 $\textsc{sc\_casebeta}$.

**Case ET_DCON.** The expression is $d\overline{m}$. 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 $\textsc{sc\_abort}$, and if it steps the expression steps by $\textsc{sc\_ctx}$.

**Case ET_APP.** The rule looks like

$$
\begin{align*}
H \vdash m &: (x:M) \rightarrow N \\
H \vdash n &: M \\
H \vdash [n/x]N : \star \\
\hline
H \vdash m n &: [n/x]N
\end{align*}
$$

By IH, $m$ and $n$ either, step, are abort is are values. If $m$ steps, the entire expression steps by $\textsc{sc\_ctx}$. If it is abort, the entire expression steps by $\textsc{sc\_abort}$. 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 47) we know that $m$ is either $\lambda x.m_1$ or $\text{rec}\hspace{1pt}f.u$. If it is $\lambda x.m_1$ the entire expression steps to $[n/x]m_1$ by $\textsc{sc\_appbeta}$, while if it is $\text{rec}\hspace{1pt}f.u$ the entire expression steps to $((\text{rec}\hspace{1pt}f.u/[f]u_1)n$ by $\textsc{sc\_apprec}$.

**Case ET_IAPP.** The rule looks like

$$
\begin{align*}
H \vdash m &: [x:M] \rightarrow N \\
H \vdash u &: M \\
\hline
H \vdash m[] &: [u/x]N
\end{align*}
$$

By the IH, $m$ either steps, is abort or is a value. If $m$ steps, then the entire expression steps by $\textsc{sc\_ctx}$ and the context $\bullet\hspace{1pt}[]$. If it is abort, the entire expression steps to abort by $\textsc{sc\_abort}$ and the same context.

Finally, $m$ may be a value. In that case, by canonical forms (lemma 47), $m$ is either $\lambda []m_1$ or $\text{rec}\hspace{1pt}f.u$. If it is $\lambda []m_1$, the entire expression steps to $m_1$ by $\textsc{sc\_iappbeta}$. If it is $\text{rec}\hspace{1pt}f.u$, then the entire expression steps to $((\text{rec}\hspace{1pt}f.u/[f]u_1)]$ by $\textsc{sc\_iapprec}$.

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

**Case ET_CONV.** Follows directly by IH.

}\square

### B.7 Regularity and substitution for the annotated language

While not needed for the type safety proof, in this section we supply proofs of regularity and substitution for the *annotated* language. This is of interest because it proves that the “value-dependent” application rule is admissible in our system.

**Lemma 49.** If $\Gamma \vdash a : A$, then $\text{FV}(a) \subseteq \text{dom}(\Gamma)$ and $\text{FV}(A) \subseteq \text{dom}(\Gamma)$. 
Lemma 50 (Weakening for ⊨ Γ.). If ⊨ Γ, Γ′ then ⊨ Γ.

Lemma 51 (Weakening for the annotated language). If Γ ⊨ a : A and ⊨ Γ, Γ′ then Γ, Γ′ ⊨ a : A.

Lemma 52 (Substitution commutes with erasure). We always have [[a/x]b] = [[a]/[x]]b.

Proof. By induction on b.

Lemma 53. If m ⊢ cbv m′, then [u_0/x_0]m ⊢ cbv [u_0/x_0]m′.

Proof. By induction on m ⊢ cbv m′. The cases are:

SC_APPBETA. The assumed step is (λ.x.m) u ⊢ cbv [u/x]m, and we must show [u_0/x_0]((λ.x.m) u) ⊢ cbv [u_0/x_0][u/x]m.

Pushing the substitution down we know [u_0/x_0]((λ.x.m) u) = (λ.x.[u_0/x_0]m) ([u_0/x_0]u), which steps to [[u_0/x_0]u/x][u_0/x_0]m. Since x is a bound variable we can pick it so that x ∉ FV (u_0) . Then [[u_0/x_0]u/x][u_0/x_0]m = [u_0/x_0][u/x]m as required.

SC_CASEBETA. Similar to the previous case.

SC_APPREC. The assumed step is (rec.f.u) u_2 ⊢ cbv ([rec.f.u/f]u_1) u_2, and we must show

[u_0/x_0](([rec.f.u] u_2) ⊢ cbv [u_0/x_0](([rec.f.u/f]u_1) u_2).

Pushing down the substitution we know [u_0/x_0](([rec.f.u] u_2) = (rec.f.u) ([u_0/x_0]u_2), which steps to (([rec.f.u/f]) [u_0/x_0]u_1) ([u_0/x_0]u_2).

By picking the bound variable f so that f ∉ FV (u_0) we have [rec.f.u/f] [u_0/x_0]u_1 = [u_0/x_0][rec.f.u/f]u_1 as required.

SC_IAPPREC. Similar to the previous case.

SC_IAPPBETA, SC_ABORT, SC_CTX. Immediate by just pushing in the substitution.


Proof. By commuting the substitution (lemma 52) we know [[v/x]a] = [[v]/[x]]a. Then apply lemma 53 to each step of the reduction sequence [[a] ⊢ cbv n.

Lemma 55 (Substitution for the annotated language). Suppose Γ ⊨ v_1 : A_1. Then,

1. If Γ_1, x_1 : A_1, Γ_2 ⊨ a : A, then Γ_1, [v_1/x_1]Γ_2 ⊨ [v_1/x_1]a : [v_1/x_1]A.

2. If ⊨ Γ_1, x_1 : A_1, Γ_2, then ⊨ Γ_1, [v_1/x_1]Γ_2.

Proof. By mutual induction on Γ_1, x_1 : A_1, Γ_2 ⊨ a : A and ⊨ Γ_1, x_1 : A_1, Γ_2. Most cases follow directly by IH. Two interesting cases are:

Case T_VAR. We get ⊨ Γ_1, [v_1/x_1]Γ_2 by the mutual IH. Then do a case-split on where in the context x occurs:

- If x : A ∈ Γ_1, then by T_VAR we have Γ_1, [v_1/x_1]Γ_2 ⊨ x : A.
  By ⊨ Γ_1, x_1 : A_1, Γ_2 we know Γ_0 ⊨ A : * for some prefix Γ_0 of Γ_1, so in particular by lemma 49 we know FV (A) ∩ dom (Γ_0), so x_1 ∉ FV (A).
  Also, by ⊨ Γ_1, x_1 : A_1, Γ_2 we know x ∉ x_1. So [v_1/x_1]x = x and [v_1/x_1]A = A, so we have showed Γ, [v_1/x_1]Γ_2 ⊨ [v_1/x_1]x : [v_1/x_1]A as required.
If \( x = x_1 \), then \([v_1/x_1]x = v_1\), so by assumption we have \( \Gamma \vdash [v_1/x_1]x : A_1 \). By the assumption \( \vdash \Gamma_1, x_1 : A_1, \Gamma_2 \) we know that \( x_1 \) is not free in \( A_1 \), so \([v_1/x_1]A = A_1\) and so we have shown \( \Gamma_1 \vdash [v_1/x_1]x : [v_1/x_1]A \). Finally by weakening (lemma 51) we have \( \Gamma_1, [v_1/x_1] \Gamma_2 \vdash [v_1/x_1]x : [v_1/x_1]A \) as required.

If \( x : A \in \Gamma_2 \), then \( x : [v_1/x_1]A \in [v_1/x_1] \Gamma_2 \), so we have \( \Gamma_1, [v_1/x_1] \Gamma_2 \vdash x : [v_1/x_1]A \) by T_VAR. By the same reasoning as above we know \( x_1 \neq x \), so this shows \( \Gamma_1, [v_1/x_1] \Gamma_2 \vdash [v_1/x_1]x : [v_1/x_1]A \) as required.

**Case T_JOIN.** The typing rule looks like

\[
\frac{|a| \sim_{cbv}^i n \quad |b| \sim_{cbv}^i n}{\Gamma \vdash a = b: \ast} \quad \text{T_JOIN}
\]

By lemma 54 we get \([v_1/x_1]a \sim_{cbv}^i n\) and \([v_1/x_1]b \sim_{cbv}^i n\). By IH we have \( \Gamma_1, [v_1/x_1] \Gamma_2 \vdash [v_1/x_1](a = b): \ast \). Then re-apply T_JOIN.

\[\square\]

**Lemma 56** (Regularity inversion for the annotated language). If \( \Gamma \vdash x:A \rightarrow B : A_0 \) for some \( A_0 \), then \( \Gamma \vdash A : \ast \) and \( \Gamma, x:A \vdash B: \ast \).

**Proof.** By induction on the assumed typing derivation. The only rules that can apply are the intro rule, which has the required statements as assumptions, and conversion, which goes directly by IH.

\[\square\]

**Lemma 57** (Regularity for the annotated language). If \( \Gamma \vdash a:A \), then \( \Gamma \vdash A : \ast \) and \( \vdash \Gamma \).

**Proof.** Induction on \( \Gamma \vdash a:A \). The cases are:

**Cases** T_TYPE, T_PI, T_PPI, T_TCON, T_ABSTCON, T_EQ.

By T_TYPE we have \( \Gamma \vdash \ast : \ast \) as required. We get \( \vdash \Gamma \) by IH (or assumption in the TYPE case).

**Cases** T_CASE, T_REC, T_ABORT, T_APP, T_JOIN, T_CONV.

We have \( \Gamma \vdash A : \ast \) as a premise to the rule, and \( \vdash \Gamma \) by IH.

**Case T_VAR.** By inversion on \( \vdash \Gamma \) plus weakening (lemma 51).

**Case T_DCON.** By T_TCON, using the premise \( \Gamma \vdash \overline{A}_1 : \Delta^+ \).

**Case T_ABS.** By IH we get \( \Gamma, x:A \vdash B : \ast \) and \( \vdash \Gamma, x:A \). Inversion on the latter gives \( \Gamma \vdash A : \ast \) so by T_PI we get \( \Gamma \vdash (x:A) \rightarrow B : \ast \) as required.

Meanwhile, weakening (lemma 50) on \( \vdash \Gamma, x:A \) gives \( \vdash \Gamma \) as required.

**Case T_ABS.** Similar to the previous case.

**Case T_IAPP.** By the IH we have \( \Gamma \vdash [x:A] \rightarrow B : \ast \) and \( \vdash \Gamma \).

Now by inversion on \( \vdash \Gamma \vdash [x:A] \rightarrow B : \ast \) (lemma 56) we get \( \Gamma, x:A \vdash B : \ast \). Then by substitution (lemma 55) we have \( \Gamma \vdash [v/x]B : \ast \) as required.
Case **T_INJDOM**. By IH we have $\vdash \Gamma$. Also by IH we have $\Gamma \vdash ((x:A_1) \to B_1) = ((x:A_2) \to B_2) : \star$, so by applying inversion (lemma 56) twice we get $\Gamma \vdash A_1 : \star$ and $\Gamma \vdash A_2 : \star$. Then by $T_{\text{EQ}}$ we have $\Gamma \vdash A_1 = A_2 : \star$ as required.

Case **T_INJRNG**. By similar reasoning to the previous case we get $\Gamma, x : A \vdash B_1 : \star$ and $\Gamma, x : A \vdash B_2 : \star$.

Then by substitution (lemma 55) we have $\Gamma \vdash [v/x]B_1 : \star$ and $\Gamma \vdash [v/x]B_2 : \star$, so by $T_{\text{EQ}}$ we have $\Gamma \vdash [v/x]B_1 = [v/x]B_2 : \star$ as required.

Case **T_INJDOM, T_INJRNG**. Similar to the previous two cases.

Case **T_INJCON**. By IH we have $\Gamma \vdash D \tilde{A_i} = D \tilde{A_i} : \star$, so by applying inversion (lemma 56) twice we have $\Gamma \vdash \tilde{A_i} : \Delta$ for some $\Delta$. By inversion on that judgment we get $\Gamma \vdash A_k : \star$, and similarly $\Gamma \vdash A_k' : \star$. So by $T_{\text{EQ}}$ we have $\Gamma \vdash A_k = A_k' : \star$ as required.

Lemma 58 (Strengthening for the annotated language). If $\Gamma_1, x_1 : A_1, \Gamma_2 \vdash a : A$ and $x_1$ is not free in $\Gamma_2$, $a$ or $A$, then $\Gamma_1, \Gamma_2 \vdash a : A$.

**Proof.** By induction on $\Gamma_1, x_1 : A_1, \Gamma_2 \vdash a : A$. □

Lemma 59 (Value application). The following rule is admissible.

\[
\begin{array}{c}
\Gamma \vdash a : (x:A) \to B \\
\Gamma \vdash v : A \\
\hline
\Gamma \vdash a \ v : [v/x]B
\end{array}
\]

**Proof.** By regularity (lemma 57) we have $\Gamma \vdash (x:A) \to B : \star$. So by inversion (lemma 56) we know $\Gamma, x : A \vdash B : \star$. Then by substitution (lemma 55) we have $\Gamma \vdash [v/x]B : \star$, so we can apply $T_{\text{APP}}$. □

Lemma 60 (Nondependent application). The following rule is admissible.

\[
\begin{array}{c}
\Gamma \vdash a : A \to B \\
\Gamma \vdash b : A \\
\hline
\Gamma \vdash a \ b : B
\end{array}
\]

**Proof.** By regularity (lemma 57) we have $\Gamma \vdash A \to B : \star$, so by inversion (lemma 56) we have $\Gamma, x : A \vdash B : \star$. By strengthening (lemma 58) we have $\Gamma \vdash B : \star$. Since $x$ is not free we know $[b/x]B = B$, so this also shows $\Gamma \vdash [b/x]B : \star$, and we can apply $T_{\text{APP}}$. □