Refinement Kinds
A Theory of Type-Safe Meta-Programming

LUÍS CAIRES, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa and NOVA-LINCS, Portugal
BERNARDO TONINHO, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa and NOVA-LINCS, Portugal

This work introduces the novel concept of kind refinement, which we technically develop in the context of an explicitly polymorphic ML-like language with type-level computation. As type refinements embed rich specifications by means of comprehension principles expressed by predicates over values in the type domain, kind refinements provide rich kind specifications by means of predicates over types in the kind domain. By leveraging our powerful refinement kind discipline, types in our language are not just used to statically classify program expressions and values, but also conveniently manipulated as tree-like data structures, with their kinds refined by logical constraints on such structures. Remarkably, the resulting typing and kinding disciplines allow for powerful forms of type reflection, ad-hoc polymorphism, and type-safe type meta-programming which are common in modern software development, but hardly expressible in extant type theories.

CCS Concepts: • Theory of computation → Type theory; • Software and its engineering → Functional languages; Domain specific languages;

Additional Key Words and Phrases: Refinement Kinds, Typed Meta-Programming, Type Theory

1 INTRODUCTION

Current software development ecosystems increasingly rely on automation, often based on tools that generate code from various types of specifications, leveraging the various reflection and meta-programming facilities that modern languages provide: an example of such a tool could be a generator that given as input a XML database schema, produces the complete code of a web application. Automated code generation, domain specific languages, and meta-programming are increasingly becoming productivity drivers for the software industry, while also making bringing programming more accessible to non-experts, and, more generally, increasing the level of abstraction of languages and tools for program construction.

These concepts are more commonly supported by so-called dynamic languages and related frameworks, such as Ruby and Ruby on Rails, JavaScript and Node.js, but are also present in static languages such as Java, Scala, Go and F#, that provide support for reflection and general meta-programming facilities, allowing code, and more frequently types, to be manipulated as data by programs. Unfortunately, meta-programming constructs and idioms aggressively challenge the safety guarantees of static typing, which becomes especially problematic given that meta-programs are notoriously hard to test for correctness.

This paper introduces for the first time the concept of refinement kinds and illustrates how the associated discipline cleanly supports static type checking of type-level reflection, parametric and ad-hoc polymorphism, which can all be combined to implement interesting meta-programming idioms. Refinement kinds, presented for the first time in this work, are a natural transposition of the well-known concept of refinement types (of values) [Bengtson et al. 2011; Rondon et al. 2008; Vazou et al. 2013] to the realm of kinds (of types). Several systems of refinement types

Authors’ addresses: Luís Caires, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa and NOVA-LINCS, Portugal, lcaires@fct.unl.pt; Bernardo Toninho, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa and NOVA-LINCS, Portugal, btoninho@fct.unl.pt.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/
have been proposed in the literature, generally motivated as a pragmatic compromise between usability and the expressiveness of full-fledged dependent types, which require proof objects to be explicitly constructed by programmers. Our work aims to show that the simple and arguably natural notion of introducing refinements in the kind structure allows us to cleanly support sophisticated statically typed meta-programming concepts, which we illustrate in the context of a higher-order polymorphic λ-calculus with imperative constructs, chosen as a convenient representative for languages with higher-order store.

Just as refinement types support expressive type specifications by comprehension principles expressed by \textit{predicates over values} in the type domains (typically implemented by SMT decidable Floyd-Hoare assertions [Rushby et al. 1998]), refinement kinds support rich and flexible kind specifications by means of comprehension principles expressed by \textit{predicates over types} in the kind domains. They also naturally support a natural notion of subkinding by entailment in the refinement logic. For example, we introduce in our language one least upper bound kind for each small type kinds, from which more concrete kinds and types may be defined by refinement, adding an unusual degree of plasticity to subkinding.

Crucially, types in our language may be reflectively manipulated as first-class (abstract-syntax) labelled trees (cf. XML data), both statically and at runtime. We expect that the deduction of relevant structural properties of such tree representations of types to be amenable to rather efficient implementation, unlike typical value domains (e.g., integers, arrays) manipulated by mainstream programming languages, and easier to automate using off-the-shelf SMT solvers (e.g. [de Moura and Bjørner 2008]). Remarkably, even if types in our system can be essentially manipulated by type-level functions and operators as abstract-syntax trees, our system statically ensures the sound inhabitation of the outcomes of type-level computations by the associated program-level terms, enforcing type safety. This allows our language to express challenging reflection idioms in a type-safe way, that we have no clear perspective on how to cleanly and effectively embed in extant (dependent) type theories.

To make the design of our framework more concrete, we briefly detail our treatment of record types. Usually, a record type is represented by a tuple of label-and-type pairs, subject to the constraint that all the labels must be pairwise distinct (e.g. see [Harper and Pierce 1991]). In order to support more effective manipulation of record types by type-level functions, record types in our theory are represented by values of a list-like data structure: the record type constructors are the type of empty records \{\} and the “cons” cell \((L : T)@R\), which constructs the record type obtained by adding a field declaration \((L : T)\) to the record type \(R\).

The record type constructors are functions \texttt{headLabel}(\(R\)), \texttt{headType}(\(R\)) and \texttt{tail}(\(R\)), which apply to any non-empty record type \(R\). As will be shown latter, the more usual record field projection operator \(r.L\) and record type field projection operator \(T.L\) turn out to be definable in our language using suitable meta-programs. In our system, record labels (cf. names) are type and term-level first-class values of kind \(Nm\). Record types also have their own kind, dubbed \(Rec\). As we will see, our theory provides a range of basic kinds that specialize the kind of all (small) types \(Type\) via subkinding, which can be further specialized via kind refinement.

For example, we may define the record type \texttt{Person} \(\equiv (\text{name} : \text{String})@\langle \text{age} : \text{Int} \rangle@\langle \rangle\), which we conveniently abbreviate by \((\text{name} : \text{String}; \text{age} : \text{Int})\). We then have that \texttt{headLabel(Person)} = \text{name}, \texttt{headType(Person)} = \text{String} and \texttt{tail(Person)} = \langle \text{age} : \text{Int} \rangle. The kinding of the \((L : T)@R\) type constructor may be clarified in the following type-level function \texttt{addFieldType}:

\[
\text{addFieldType} :: \Pi l :: \text{Nm}. \Pi t :: \text{Type}. \Pi r :: \text{Rec}. \Pi s :: \text{Rec} | l \notin \text{lab}(s), \text{Rec}, \lambda l :: \Pi s, \lambda t :: \text{Type}. \lambda r :: \text{Rec} | l \notin \text{lab}(s), \langle l : t \rangle@r
\]
The addFieldType type-level function takes a label \( l \), a type \( t \) and any record type \( r \) that does not contain label \( l \), and returns the expected extended record type of kind \( \text{Rec} \). Notice that the kind of all record types that do not contain label \( l \) is represented by the refinement kind \( \{ s : \text{Rec} \mid l \notin \text{lab}(s) \} \).

A refinement kind in our system is noted \( \{ t : \mathcal{K} \mid \varphi(t) \} \), where \( \mathcal{K} \) is a (small) kind, and the logical formula \( \varphi(t) \) expresses a constraint on the form of the type \( t \) that inhabits \( \mathcal{K} \). As expected in refinement type systems [Bengtson et al. 2011; Swamy et al. 2011; Vazou et al. 2014], we expect our underlying logic of refinements to include a decidable theory for the various finite tree-like data types used to schematically represent type specifications, as is the case of our record-types-as-lists, function-types-as-pairs (i.e. a pair of a domain and an image type), and so on. The kind refinement rule is thus expressed

\[
\frac{\Gamma \vdash \varphi(T/t) \quad \Gamma \vdash r : \mathcal{K}}{\Gamma \vdash t : \{ t : \mathcal{K} \mid \varphi \}} \quad (\text{KREF})
\]

where \( \Gamma \vdash \varphi \) denotes entailment in the refinement logic. Basic formulas of our refinement logic include propositional logic, equality, and some useful predicates and functions on types, including the primitive type constructors and destructors, such as \( \text{lab}(R) \) (record label set), \( L \in S \) (label membership), \( S\#S' \) (label set apartness), \( R@S \) (concatenation), \( \text{dom}(F) \) (function domain selector).

Interestingly, given the presence of equality in refinements, it is always possible to define for any type \( T \) of kind \( \mathcal{K} \) a precise singleton kind \( S(T) \) of the form \( \{ t : \mathcal{K} \mid t \equiv T : \mathcal{K} \} \). As another simple example, consider the kind \( \text{Auto} \) of automorphisms, defined as \( \{ t : \text{Fun} \mid \text{dom}(t) \equiv \text{img}(t) : \text{Type} \} \).

A use of the type-level function addFieldType given above is, for instance, the definition of the following term-level polymorphic record extension function

\[
\begin{align*}
\text{addField} & : \forall l : \text{Nm}. \forall t : \text{Type}. \forall r : \{ s : \text{Rec} \mid l \notin \text{lab}(s) \}. t \to r \to \text{addFieldType} l t r \\
\text{addField} & \equiv \forall l : \text{Nm}. \forall t : \text{Type}. \forall r : \{ s : \text{Rec} \mid l \notin \text{lab}(s) \}. \lambda x : t. \lambda y : r. (l = x) @ y
\end{align*}
\]

The addField function takes a label \( l \), a type \( t \), a record type \( r \) that does not contain label \( l \), and values of types \( t \) and \( r \), respectively, returning a record of type addFieldType \( l t r \).

The type-level and term-level functions addFieldType and addField respectively illustrate some of the key insights of our type theory, namely the use of types and their refined kinds as specifications that can be manipulated as tree-like structures by programs in a fully type-safe way. For instance, the following judgment, expressing the correspondence between the term-level computation addField \( l t r x y \) and the type-level computation addFieldType \( l t r \), is derivable:

\[
\vdash \text{addField} l t r x y : \text{addFieldType} l t r
\]

An instance of this judgement yields:

\[
\vdash \text{addField} \text{name} \text{String} \langle \text{age} : \text{Int} \rangle \text{ "jack"} \langle \text{age} = 20 \rangle : \text{addFieldType} \text{name} \text{String} \langle \text{age} : \text{Int} \rangle
\]

Noting that \( \langle \text{age} : \text{Int} \rangle : \{ s : \text{Rec} \mid \text{name} \notin \text{lab}(s) \} \) is derivable since \( \text{name} \notin \text{lab}(\langle \text{age} : \text{Int} \rangle) \) is provable in the refinement logic, we have the following term and type-level evaluations:

\[
\begin{align*}
(\text{addField} \text{name} \text{String} \langle \text{age} : \text{Int} \rangle \text{ "jack"} \langle \text{age} = 20 \rangle) & \to^* \langle \text{name} = \text{ "jack"}; \text{age} = 20 \rangle \\
(\text{addFieldType} \text{name} \text{String} \langle \text{age} : \text{Int} \rangle) & \equiv \langle \text{name} : \text{String}; \text{age} : \text{Int} \rangle
\end{align*}
\]

Using the available refinement principles, our system can also derive the following more precise kinding for the type addFieldType \( l t r \):

\[
\vdash \text{addField} l t r : \{ s : \text{Rec} \mid s \equiv (l : t)@r : \text{Rec} \}
\]
Contributions. We summarise the main contributions of this work: First, we motivate for the
first time the concept of refinement kinds, showing how it supports the flexible and clean definition
of statically typed meta-programs through several examples (Section 2). Second, we technically
develop our refinement kind system (Section 3), using as core language a ML-like polymorphic
λ-calculus (Section 4) with records, references and collections, supporting type-level computation.
Third, we establish the key meta-theoretical result (Section 5) of type safety through type unicity,
type preservation and progress (Theorems 5.8, 5.9 and 5.11, respectively).

We conclude with an overview of key related work (Section 6), and offer some concluding
remarks and discussion on the pragmatics of the language (7). Appendices A, B and C list omitted
definitions of the type theory, its semantics and proof outlines, respectively.

2 PROGRAMMING WITH REFINEMENT KINDS

Before delving into the technical intricacies of our theory in Section 3 and beyond, we illustrate
the various features and expressiveness of our theory through a series of examples that showcase
how our language supports (in a perhaps surprisingly clean way) challenging (from a static typing
perspective) meta-programming idioms.

Generating Mutable Records. We begin with a simple higher-order meta-program that computes a "generator" for mutable records from a specification of its representation type, expressed as an arbitrary record type. Consider the following definition of the (recursive) function genConstr:

\[
\text{genConstr} \triangleq \ \Lambda S::[r::\text{Rec} | \text{nonEmpty}(r)]. \lambda V::[v::\text{Rec} | \ellab(v)^{\#}\ellab(S)]. \lambda v:V.
\]

\[
\lambda x:\text{headType}(S) . \text{if nonEmpty}(\text{tail}(S)) \text{ then}
\]

\[
\text{genConstr} \ \text{tail}(S) \ (\text{headLabel}(S) : \text{ref headType}(S)) @V \ (\text{headLabel}(S) = \text{ref} \ x) @v
\]

\[
\text{else} \ (\text{headLabel}(S) = \text{ref} \ x) @v
\]

Given a non-empty record type \(S\), function genConstr returns a constructor function for a mutable record whose fields are specified by \(S\). We use an informal notation to express recursive definitions, which in our formal core language is represented by an explicit structural recursion construct. Parameters \(V\) and \(v\) are accumulating parameters that track intermediate types, values and a
disjointness invariant on those types during computation (for simplicity, we generate the record
fields in reverse order).

Intuitively, and recovering the record type Person from above, genConstr Person \(\ell\) \(\ell\) computes
to a value equivalent to \(\lambda x:\text{String}. \lambda y: \text{Int}. \text{(age = ref} y; \text{name = ref} x)\).

Notice that function genConstr accepts any non-empty record type \(S\), and proceeds by recursion
on the structure on type \(S\), as a list of label-type pairs. The parameter \(S\) holds the types of the fields
still pending for addition to the final record type, parameter \(V\) holds the types of the fields
already added to the final record type, and \(v\) holds the already built mutable record value. To
properly call genConstr, we “initialize” \(V\) with \(\ell\) (i.e. the empty record type), and \(v\) to \(\ell\). Moreover,
the refined kind of \(V\) specifies the label apartness constraint needed to type check the recursive
call of genConstr, in particular, given \(\ellab(V)^{\#}\ellab(S)\), the refinement logic deduces \(\text{headLabel}(S) \neq \ellab(V)\), needed to kind check \(\ellab(\text{headLabel}(S) : \text{ref headType}(S)) @V\); and \(\ellab(\ellab(\text{headLabel}(S) : \text{ref headType}(S)) @V)^{\#}\ellab(\text{tail}(S))\), required to kind and type check the recursive call. In our
language, genConstr can be typed as follows:

\[
\text{genConstr} : \forall V::[r::\text{Rec} | \text{nonEmpty}(r)]. \forall V::[v::\text{Rec} | \ellab(v)^{\#}\ellab(S)]. \text{(GType} S V)
\]
where \texttt{GType} is the (recursive) type-level function such that

\[
\text{GType} :: \Pi S::\{r::\text{Rec} \mid \text{nonEmpty}(r)\}. \Pi V::\{v::\text{Rec} \mid \text{lab}(v)\#\text{lab}(S)\}. \text{Fun}
\]

\[
\text{GType} \triangleq \\
\lambda S::\{r::\text{Rec} \mid \text{nonEmpty}(r)\}.
\lambda V::\{v::\text{Rec} \mid \text{lab}(v)\#\text{lab}(S)\}.
\text{headType}(S) \rightarrow \text{if \ nonEmpty}(\text{tail}(S)) \text{ then}
\enspace \text{GType tail}(S) \langle \text{headLabel}(S) :: \text{ref headType}(S) \rangle @ V \text{ else } V
\]

We can see that, in general, the type-level application \texttt{GType} \{L_1 : T_1; \ldots; L_n : T_n\} \langle \rangle computes the type \(T_1 \rightarrow \ldots \rightarrow T_n \rightarrow \langle L_n : \text{ref } T_n; \ldots; L_1 : \text{ref } T_1\rangle\). In particular, we have

\[
\text{genConstr Person} \langle \rangle : \text{String} \rightarrow \text{Int} \rightarrow \langle \text{age = ref } \text{Int}; \text{name = ref } \text{String} \rangle
\]

**From Record Types to XML Tables.** As a second example, we develop a generic function \texttt{MkTable} that generates and formats an XML table for any record type, inspired by the example in Section 2.2 of [Chlipala 2010]. We start by introducing an auxiliary type-level Map function, that returns the record type obtained from a record type \texttt{R} by applying a type transformation \texttt{G} (of higher-order kind) to the type of each field of \texttt{R}.

\[
\text{Map} :: \Pi G::(\Pi X :: \text{Type. Type}). \Pi R::\text{Rec.} \{r :: \text{Rec} \mid \text{lab}(r) = \text{lab}(R)\}
\]

\[
\lambda G::(\Pi X :: \text{Type. Type}). \lambda R::\text{Rec.}
\text{if \ nonEmpty}(R) \text{ then } (\text{headLabel}(R) :: G \text{ headType}(R)) @ (\text{Map } G \text{ tail}(R)) \text{ else } \langle \rangle
\]

The logical constraint \text{lab}(r) = \text{lab}(R) expresses that the result of \text{Map } G \text{ R} has exactly the same labels as record type \texttt{R}. This implies that \text{headLabel}(R) \notin \text{lab}(\text{Map } G \text{ tail}(R)) in the recursive call, thus allowing the “cons” to be well-kindled. We now define:

\[
\text{XForm} :: \Pi t :: \text{Type. Type}
\]

\[
\text{XForm} \triangleq \lambda t::\text{Type.} \langle \text{tag : String; toStr : } t \rightarrow \text{String} \rangle
\]

\[
\text{MkTableType} :: \lambda r::\text{Rec.} \{r :: \text{Rec} \mid \text{lab}(r) = \text{lab}(R)\}
\]

\[
\text{MkTableType} \triangleq \lambda r::\text{Rec.} \text{Map } \text{XForm } r
\]

\[
\text{MkTable} :: \forall R::\text{Rec.} (\text{MkTableType } R) \rightarrow R \rightarrow \text{String}
\]

\[
\text{MkTable} \triangleq \lambda R::\text{Rec.} \lambda M::\text{MkTableType } R \lambda r::R.
\text{if \ nonEmpty}(R) \text{ then}
\enspace "\langle \text{tr}>\langle \text{th}>" + M.\text{recHeadLabel}(M).\text{tag} + "\langle /\text{th}>" +
M.\text{recHeadLabel}(M).\text{toStr } r.\text{recHeadLabel}(M) + "\langle /\text{tr}>\langle /\text{tr}>"
\text{MkTable } \text{tail}(R) \text{ recTail}(M) \text{ recTail}(r)
\text{else }"
\]

It is instructive to discuss why and how this code is well-typed, witnessing the expressiveness of refinement kinds, despite their conceptual simplicity (which can be judged by the arguably parsimonious nature of the definitions above). Let us first consider the expression \texttt{M.recHeadLabel(M).tag}. Notice that, by declaration, \texttt{R:Rec} and \texttt{r:R}. However, the expression under consideration is to be typed under the assumption that \text{nonEmpty}(R), which is added to the current set of refinement assumptions while typng the then branch. Using \text{TT} for the type of \texttt{M}, Since \text{MkTableType } R :: \{r::\text{Rec} \mid \text{lab}(r) = \text{lab}(R)\}, by refinement we have that \text{lab}(TT) = \text{lab}(R) and thus \text{nonEmpty}(TT),
allowing recHeadLabel(M) to be defined. Since M : MkTableType R we have

\[(\text{MkTableType } R) \equiv (\text{Map XForm } R) \equiv
\langle \text{headLabel}(R) : XForm \text{headType}(R) \rangle \@ (\text{Map G tail}(R))\]

We thus derive headLabel(TT) \equiv headLabel(R). Then

\text{headType}(\text{MkTableType } R) \equiv
\text{XForm headType}(R) \equiv \langle \text{tag : String; toString : headType}(R) \to \text{String} \rangle

Hence M.headLabel(M).tag : String. By a similar reasoning, we conclude r.recHeadLabel(M) : headType(R). In Section 4.1, we will see more precisely how refinements augment the simple type-level function applications in order to make precise the reasoning sketched above.

**Generating Getters and Setters.** As a final introductory example, we develop a generic function MkMut that generates a getter/setter wrapper for any mutable record (i.e. a record where all its fields are of reference type). We first define the auxiliary type-level MutableRec function, that returns the mutable record type obtained from a record type R in terms of Map:

\[
\text{MutableRec} \::= \Pi R :: \text{Rec.} \{ r :: \text{Rec | lab} = \text{lab}(R) \}
\]

\[
\text{MutableRec} \triangleq \text{Map (} \lambda r :: \text{Type.} \text{ref r)}
\]

We then define the auxiliary type-level SetGet function, that returns the record type that exposes the getter/setter interface generated from record type R:

\[
\text{SetGetRec} \::= \Pi R :: \text{Rec.} \{ r :: \text{Rec | lab} = \text{set++lab}(r) \cup \text{get++lab}(R) \}
\]

\[
\text{SetGetRec} \triangleq \lambda R :: \text{Rec.}
\]

if nonEmpty(R) then

\[
\langle \text{get++headLabel}(R) : 1 \to \text{headType}(R) \rangle \@
\]

\[
\langle \text{set++headLabel}(R) : \text{headType}(R) \to 1 \rangle \@
\]

\[
\text{SetGetRec. tail}(R)
\]

else \{

Here, n++m denotes the name obtained by appending n to m, and n++S denotes the label set obtained from S by prefixing every label in S with name n. The function SetGet is well kinded since the refinement kind constraints imply that the resulting getter/setter interface type is well formed (i.e. all labels distinct). We can finally depict the type and code of the MkMut function:

\[
\text{MkMut} :: \forall R :: \text{Rec.} \text{MutableRec } R \to \text{SetGetRec } R
\]

\[
\text{MkMut} \triangleq \Delta R :: \text{Rec.}
\]

\[
\lambda r :: \text{MutableRec } R
\]

if nonEmpty(R) then

\[
\langle \text{get++headLabel}(R) = \lambda x :: 1.!(r.recHeadLabel(R)) \rangle \@
\]

\[
\langle \text{set++headLabel}(R) = \lambda x :: \text{headType}(R).r.recHeadLabel(R) := x \rangle \@
\]

\[
\text{MkMut. tail}(R) \text{ recTail(r)}
\]

else \{

For example, assuming r : MutableRec Person we have that MkMut Person r computes a record equivalent to:

\[
\langle \text{getname} = \lambda x :: 1.!(r.name); \text{setname} = \lambda x :: \text{String}.r.name := x; \text{getname} = \lambda x :: 1.!(r.name); \text{setage} = \lambda x :: \text{Int}.r.age := x \rangle
\]

where (MkMut Person r) : SetGetRec Person.
We thus have the type of empty records ⟨⟩ with a higher-order store and the appropriate reference types, collections (i.e. lists) and records.

The typing and kinding systems rely on type-level functions (from types to types) and a novel form of pairs of labels and types which maintains the invariant that all labels in a record must be distinct.

Our notion of record type, as explored in Section 2, is essentially a type-level list (⟨⟩, and the constructor ⟨L : T⟩@R, which given a record type R that does not contain the label L, generates a record type that is an extension of R with the

3 A TYPE THEORY WITH KIND REFINEMENTS

Having given an informal overview of the various features and expressiveness of our theory, we now formally develop our theory of refinement kinds, targeting an ML-like functional language with a higher-order store and the appropriate reference types, collections (i.e. lists) and records. The typing and kinding systems rely on type-level functions (from types to types) and a novel form of subkinding and kind refinements. We first address our particular form of (sub)kinding and the type-level operations enabled by this fine-grained view of kinds, addressing kind refinements and their interaction types and type-level functions in Section 3.1.

Given that kinds are classifiers for types, we introduce a separate kind for each of the key type constructs of the language. Thus, we have a kind for records, Rec, which classifies record types; a kind Col, for collection types; a kind Fun, for function types; a kind Ref, for reference types; a kind GenK for polymorphic function types (whose type parameter must be of kind K); and, a kind Nm for labels in record types (and records). All of these are specialisations (i.e. subkinds) of the kind of all (small) types, Type. We write K for any such kind. The language of types (a type-level λ-calculus) provides the expected constructors for the types described above, but crucially also introduces type destructors that allow us to inspect the structure of types of a given kind and, in combination with type-level functions and structural recursion, enable a form of typed meta-programming. Indeed, our type language is essentially one of (inductive) structures and their various constructors and destructors (and basic data types Bool and 1). The syntax of types and kinds is given in Figure 1.

Record Types. Our notion of record type, as explored in Section 2, is essentially a type-level list of pairs of labels and types which maintains the invariant that all labels in a record must be distinct. We thus have the type of empty records ⟨⟩, and the constructor ⟨L : T⟩@R, which given a record type R that does not contain the label L, generates a record type that is an extension of R with the

Fig. 1. Syntax of Kinds, Types and Refinements
label $L$ associated with type $T$. Record types are associated with three destructors: $\text{headLabel}(T)$, which projects the label of the head of the record $T$ (when seen as a list); $\text{headType}(T)$ which projects the type at the head of the record $T$; and $\text{tail}(T)$ which produces the tail of the record $T$ (i.e. drops its first label and type pair). As we will see (Example 3.1), since our type-level $\lambda$-calculus allows for (structural) recursion, we can define a suitable record projection type construct in terms of these lower-level primitives.

**Function Types and Polymorphism.** Functions between terms of type $T$ and $S$ are typed by the usual $T \to S$. Given a function type $T$, we can inspect its domain and image via the destructors $\text{dom}(T)$ and $\text{img}(T)$, respectively.

Polymorphic function types are represented by $\forall t::K. T$ (with $t$ bound in $T$, as usual). Note that the kind annotation for the type variable $t$ allows us to express not only general parametric polymorphic functions (by specifying the kind as Type) but also some form of subkinding polymorphism, since we can restrict the kind of $t$ to a specialized basic kind such as Ref or Fun.

For instance, we can specify the type $\forall t::\text{Fun}. t \to \text{dom}(t) \to \text{img}(t)$ of functions that, given a function type $t$, a function of such a type and a value in its domain produce a value in its image (i.e. the type of function application). The destructor for such a type, $\text{tmap}(T) S$, takes a polymorphic function type $T$ (of functions from types of kind $K$ to some type $T'$) and a type $S$ of kind $K$ and constructs the appropriately instantiated type $T'(S/t)$.

**Collections and References.** The type of collections of elements of type $T$ is written as $T^*$, with the associated type destructor $\text{colOf}(T)$, which projects out the type of the collection elements. Similarly, reference types $\text{ref} T$ are bundled with a destructor $\text{refOf}(T)$ which determines the type of of the referenced elements.

**Kind Test.** Just as many programming languages have a type case construct [Abadi et al. 1991] that allows for the runtime testing of the type of a given expression, our $\lambda$-calculus of types has a kind case construct, if $T::\mathcal{K}$ as $t \Rightarrow S$ else $U$, which checks the kind of type $T$ against kind $\mathcal{K}$, computing to type $S$ if the kinds match and to $U$ otherwise. Combined with a term-level analogue, such constructs enable ad-hoc polymorphism, insofar as we can express non-parametric function types.

### 3.1 Type-level Functions and Refinements

The language of types that we have introduced up to this point consists essentially of a language of tree-like structures and their various constructors and destructors. As we have mentioned, our type language is actually a $\lambda$-calculus for the manipulation of such structures and so includes functions from types to types, $\lambda t::K.T$, and their respective application, written $T S$. We also include a type-level structural recursion operator $\mu F : (\Pi t::K. K'). \lambda t::K. T$, which allows us to define recursive type functions from kind $K$ to $K'$. While written as a fixpoint operator, we syntactically enforce that recursive calls must always take structurally smaller arguments to ensure well-foundedness.

Type-level functions are dependently kinded, with kind $\Pi t::K. K'$ (i.e. the kind of $T$ in a type $\lambda$-abstraction can refer to the type of its argument), where the dependencies manifest themselves in kind refinements. Just as the concept of type refinements allow for rich type specifications through the integration of predicates over values of a given type in the type structure, our notion of kind refinements integrate predicates over types in the kind structure, enabling for the kinding system to specify and enforce logical constraints on the structure of types. A kind refinement, written $\{ t::\mathcal{K} \mid \varphi \}$, where $\mathcal{K}$ is a basic kind, and $\varphi$ is a logical formula (with $t$ bound in $\varphi$), characterises types $T$ of kind $\mathcal{K}$ such that the property $\varphi$ holds of $T$ (i.e. $\varphi(T/t)$ is true). The language of properties
\( \phi \) consists of (type) predicates, propositional logic connectives and type equality, providing a form of equational reasoning on types.

Such a seemingly simple extension already provides a significant boost in expressiveness. For instance, by using equality in the refinement formula we can encode singleton-like patterns such as \( \{ t :: \text{Fun} \mid \text{img}(t) \equiv \text{Bool} :: \text{Type} \} \), the kind of function types whose image is a Bool. Moreover, by combining kind refinements and type-level functions, we can express non-trivial type transformations in a fully typed (or kinded) way. For instance consider the following:

\[
dropField \triangleq \lambda l \colon \text{Nm} . \mu F : (\Pi r :: \text{Rec} \mid l \in \text{lab}(r)) . \{ r :: \text{Rec} \mid l \notin \text{lab}(r) \} . \lambda t :: (r :: \text{Rec} \mid l \in \text{lab}(r)) .
\]

\[
\text{if headLabel}(t) \equiv l :: \text{Nm} \text{ then } \text{tail}(t) \text{ else } \langle \text{headLabel}(t) : \text{headType}(t) \rangle@(F(\text{tail}(t)))
\]

The function \( \text{dropField} \) above takes label \( l \) and a record type with a field labelled by \( l \) and removes the corresponding field and type pair from the record type (recall that \( \text{lab}(r) \) denotes the refinement-level set of labels of \( r \)). Such a function combines structural recursion (where \( \text{tail}(t) \) is correctly deemed as structurally smaller than \( t \)) with our type-level refinement test, if \( \varphi \) then \( T \) else \( S \). We note that the well-kindness of such a function relies crucially on the ability to derive that, when the record label \( \text{headLabel}(t) \) is not \( l \), since we know that \( l \) must be in \( t \), then \( \text{tail}(t) \) is still a record type containing \( l \) (we make this kind of reasoning precise in Section 4.1).

### 3.2 Kinding and Type Equality

Having formally introduced the key components of our kind and type language, we now detail the kinding and type equality of our theory, making precise the intuitions of the previous sections.

The kinding judgment is written \( \Gamma \vdash t :: K \), denoting that type \( T \) has kind \( K \) under the assumptions in the structural context \( \Gamma \). Contexts contain assumptions of the form \( t :: K, x :: T \) and \( \varphi :: t \) stands for a type of kind \( K, x \) stands for a term of type \( T \) and refinement \( \varphi \) is assumed to hold, respectively. Kinding relies on a context well-formedness judgment, written \( \Gamma \vdash \), a kind well-formedness judgment \( \Gamma \vdash K \), subkinding judgment \( \Gamma \vdash K \leq K' \) and the refinement well-formedness and entailment judgments, \( \Gamma \vdash \varphi \) and \( \Gamma \models \varphi \). Context well-formedness simply checks that all types, kinds and refinements in \( \Gamma \) are well-formed. Kind well-formedness is defined in the standard way, relying on refinement well-formedness (see Appendix A.1), which requires that formulae and types in refinements must be well-formed. Subkinding codifies the informal reasoning from the beginning of this section, specifying that all basic kinds are a specialization of \( \text{Type} \); and captures equality of kinds. Kind equality, written \( \Gamma \vdash K \equiv K' \), identifies definitionally equal kinds, which due to the presence of kind refinements requires reasoning about equivalent refinements (and the types that may appear therein).

Kinding (and typing) presupposes the existence of a signature \( \Sigma \) that specifies the arities and kindings of all type predicates, as well as any extensions to the reasoning principles of definitional equality. Moreover, we assume the signature also contains the constants (and kinding) of Figure 2, which is a form of “pre-kinding” for all the type destructors, indicating that they expect arguments of the appropriate kinds and produce types of kind \( \text{Type} \). We note that the three record type destructors are only well-kindred when applied to a non-empty record type. As we will see, this basic kinding can be further specialized by the kinding rules through kind refinements.

We now introduce the key kinding rules for the various types in our theory and their associated definitional equality rules. The type equality judgment is written \( \Gamma \models T \equiv S :: K \), denoting that \( T \) and \( S \) are equal types of kind \( K \).

**Refinements, Type Properties and Destructors.** A kind refinement is introduced by the rule

\[
\frac{\Gamma \models \varphi(T/t) \quad \Gamma \vdash T :: K}{\Gamma \vdash t :: \{ T :: K \mid \varphi \}} \quad \text{(kRef)}
\]
headLabel :: \Pi r:Rec | nonEmpty(r).Nm
colOf :: \Pi t:Col.Type

headType :: \Pi r:Rec | nonEmpty(r).Type
dom :: \Pi t:Fun.Type
tail :: \Pi r:Rec | nonEmpty(r).Rec
img :: \Pi t:Fun.Type
refOf :: \Pi t:Ref.Type
tmap :: \Pi t:Gen_K.\Pi t:K.Type

Fig. 2. Simple Kinding for Type Destructors

Given a type \( T \) of kind \( \mathcal{K} \) and a valid property \( \varphi \) of \( T \), then we are justified in stating that \( T \) is of kind \( \{t:\mathcal{K} \mid \varphi\} \). Crucially, since equality can be reflected in refinements, the rule above may be used to derive refinements that specify the shape of the refined types, for instance, the expected \( \beta \)-like equational reasoning for records allows us to derive \( \langle\ell : \text{Bool} \Rightarrow \text{Bool}\rangle() :: \{t:\text{Rec} \mid \text{headType}(t) \equiv \text{Bool} \Rightarrow \text{Bool} :: \text{Type}\} \). In general, we provide a form of equality elimination rule in refinements, stating that (for a well-formed property \( \varphi \)) the validity of a property \( \varphi \) of some type \( T \) is closed under type equality:

\[
\frac{\Gamma \vdash T \equiv S :: K \quad \Gamma, x : K \vdash \varphi \quad \Gamma \vdash \varphi[T/x]}{\Gamma \vdash \varphi[S/x]} \quad (\text{R-EQELIM})
\]

As we have previously illustrated, properties can also be tested for validity in types through a conditional construct if \( \varphi \) then \( T \) else \( S \). Provided that the property \( \varphi \) is well-formed, if \( T \) is of kind \( K \) assuming \( \varphi \) and \( S \) of kind \( K \) assuming \( \neg \varphi \), then the conditional test is well-kind, as specified by the rule (K-ITE). The equality principals for the property test rely of validity of the specified property, as expected (with a degenerate case where both branches are equal types).

\[
\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \varphi \vdash T :: K \quad \Gamma, \neg \varphi \vdash S :: K}{\Gamma \vdash \text{if } \varphi \text{ then } T \text{ else } S :: K} \quad (\text{K-ITE})
\]

\[
\frac{\Gamma \vdash \neg \varphi \quad \Gamma, \varphi \vdash T_1 :: K \quad \Gamma, \neg \varphi \vdash T_2 :: K}{\Gamma \vdash \text{if } \varphi \text{ then } T_1 \text{ else } T_2 :: K} \quad (\text{K-ITEE})
\]

Given the basic kinding for type destructors that is present in the base signature \( \Sigma \), we further generalise the kinding of type destructors (and their associated equality principles) via kind refinement. For conciseness, we write \( \text{elim}_K \) to stand for any destructor for kind \( \mathcal{K} \) (e.g. if \( \mathcal{K} \) is \( \text{Gen}_K \) then \( \text{elim}_K \) is \( \text{tmap} \), if \( \mathcal{K} \) is \( \text{Rec} \) then \( \text{elim}_K \) can be \( \text{headLabel} \), \( \text{headType} \) or \( \text{tail} \), and so on):

\[
\frac{\Gamma \vdash T :: \{t:\mathcal{K} \mid \text{elim}_K(t) \equiv T' :: \mathcal{K}'\} \quad \Gamma \vdash T'(T/t) :: \mathcal{K}'(T/t)}{\Gamma \vdash \text{elim}_K(T) :: \mathcal{K}'(T/t)} \quad (\text{K-ELIM})
\]

\[
\frac{\Gamma \vdash T \equiv S :: \{t:\mathcal{K} \mid \text{elim}_K(T) \equiv T' :: \mathcal{K}'\} \quad \Gamma \vdash T'(T/t) :: \mathcal{K}'(T/t)}{\Gamma \vdash \text{elim}_K(T) :: T'(T/t) :: \mathcal{K}'(T/t)} \quad (\text{EQ-ELIM})
\]

The kinding and corresponding equality rules above allow for equalities in refinements that mention destructors to be reflected in the kinding (and equalities) of the given destructor (the instantiation of \( t \) with \( T \) is required to ensure well-formedness of kinds and types outside the refinement). These principles become particularly interesting when reasoning from refinements that appear in type variables. For instance, the type \( \forall t:\{f: \text{Fun} \mid \text{dom}(f) \equiv \text{Bool} \land \text{img}(f) \equiv \text{Bool} :: \text{Type}\}.t \rightarrow \text{Bool} \) can be used to type the term \( \Lambda t:\{f: \text{Fun} \mid \text{dom}(f) \equiv \text{Bool} :: \text{Type} \land \text{img}(f) \equiv \text{Bool} :: \text{Type}\}.\lambda f:t.(f \, \text{true}) \), where \( \Lambda \) is the binder for polymorphic functions, as usual. Crucially,
typing (and kinding) exploits not only the fact that we know that the type variable $t$ stands for a function type, but also the fact that the domain and codomain are the type Bool, which then warrants the application of $f$ to a boolean in order to produce a boolean, despite the basic kinding information only specifying that $f$ is a function.

**Type Functions and Function Types.** The rules that govern the kinding of type-level functions are the standard kinding rules from a suitable type theory (to streamline the presentation, we omit the congruence rules for equality):

$$ \Gamma \vdash \lambda t : K. T :: \Pi t : K. K' \quad (K-FUN) \quad \Gamma \vdash T :: \Pi t : K. K' \quad \Gamma \vdash S :: K \quad (K-APP) \quad \Gamma \vdash t \in \Gamma \quad \Gamma \vdash t :: K \quad (K-VAR) $$

$$ \Gamma \vdash \mu F : (\Pi t : K. K'). \lambda t : K. T :: \Pi t : K. K' \quad (K-FIX) $$

Structural recursive functions, defined via a fixpoint construct, are defined by the following rules:

$$ \Gamma, F : \Pi t : K. K', t : K + T :: K' \quad \text{structural}(T, F, t) \quad \Gamma \vdash \mu F : (\Pi t : K. K'). \lambda t : K. T :: \Pi t : K. K' $$

$$ \Gamma \vdash (\mu F : (\Pi t : K_1. K_2). \lambda t : K_1. T) S \equiv T[S/t] \quad \Gamma \vdash \Gamma \vdash t : K_2 \quad \text{structural}(T, F, t) \quad \Gamma \vdash S :: K_1 \quad \text{structural}(T, F, t) \quad \Gamma \vdash (\mu F : (\Pi t : K_1. K_2). \lambda t : K_1. T)/F :: K_2[S/t] \quad (EQ-FIXUNF) $$

The predicate structural$(T, F, t)$ enforces that calls of $F$ in $T$ must take arguments that are structurally smaller than $t$ (i.e. the arguments must be syntactically equal to $t$ applied to a destructor). More precisely, the predicate structural$(T, F, t)$ holds iff all occurrences of $F$ in $T$ are applied to terms smaller than $t$, where the notion of size is given by $\text{elim}_K(t) < t$, with $\mathcal{K}$ is any basic kind, with the exception of $\text{Gen}_K$, for any $K$.

The equality rule allows for the appropriate unfolding of the recursion to take place. Polymorphic function types are assigned kind $\text{Gen}_K$, as expected, and the $\beta$-like equality principle for the elimination form $\text{tmap}(\forall t : K. T) S$ performs the appropriate instantiation of $t$ with $S$ in $T$.

$$ \Gamma \vdash K \quad \Gamma, t : K + T :: \mathcal{K} \quad (K-VAR) \quad \Gamma \vdash \forall t : K. T :: \text{Gen}_K \quad \Gamma \vdash t : K + T :: \mathcal{K} \quad \Gamma \vdash S :: K \quad \Gamma \vdash \text{tmap}(\forall t : K. T) S \equiv T[S/t] :: \text{Type} \quad (EQ-TMAP) $$

Our manipulation of function types as essentially a pair of types (a domain type and an image type) gives rise to the following natural equalities:

$$ \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \mathcal{K}' \quad (EQ-DOM) \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \mathcal{K}' \quad (EQ-IMG) $$

**Records and Labels.** The kinding rules the govern record type constructors and field labels are:

$$ \Gamma \vdash \{ \} :: \text{Rec} \quad (K-RECNIL) \quad \Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{ t : \text{Rec} \mid L \notin \text{lab}(t) \} \quad (K-RECONS) \quad \Gamma \vdash \ell \in \mathcal{N} \quad \Gamma \vdash \{ L : T @ S :: \text{Rec} \} \quad (K-LABEL) \quad \Gamma \vdash \ell :: \text{Nm} $$

The rule for non-empty records crucially requires that the tail $S$ of the record type must not contain the field label $L$. The equality principles for the three destructors are fairly straightforward,
projecting out the appropriate record type component, provided the record is well-kindled.

\[
\frac{\Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \text{Rec} | L \not\equiv \text{lab}(t)\}}{\Gamma \models \text{headLabel}(\langle L : T \rangle @ S) \equiv L :: \text{Nm}}
\]

\[
\frac{\Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \text{Rec} | L \not\equiv \text{lab}(t)\}}{\Gamma \models \text{headType}(\langle L : T \rangle @ S) \equiv T :: \text{Type}}
\]

\[
\frac{\Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \text{Rec} | L \not\equiv \text{lab}(t)\}}{\Gamma \models \text{tail}(\langle L : T \rangle @ S) \equiv S :: \text{Rec}}
\]

**Collections and Reference Types.** At the level of kinding, there is little difference between a collection type and a reference type. They both denote a structure that “wraps” a single type (the type of the collection elements for the former and the type of the referenced values in the latter). Thus, the respective destructor simply unwraps the underlying type.

\[
\frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \text{Col}} \quad \frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash \text{ref} T :: \text{Ref}} \quad \frac{\Gamma \models \text{colOf}(T^*) \equiv T :: \text{Type}}{\Gamma \models \text{refOf}(\text{ref} T) \equiv T :: \text{Type}}
\]

**Conversion and Subkinding.** As we have informally described earlier, our theory of kinds is predicated on the idea that we can distinguish between the different types of our language at the kind level, such that given a general kind \(\text{Type}\), the kind of record types \(\text{Rec}\) is a specialisation of \(\text{Type}\), and similarly for the other type-level base constructs of the theory. We formalise this idea through a subkinding relation, which also internalises kind equality, and the corresponding subsumption rule:

\[
\frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash K \leq K' \quad \Gamma \vdash \mathcal{K} \equiv K'}{\Gamma \vdash T :: K'} \quad \frac{\Gamma \vdash \mathcal{K} \equiv K'}{\Gamma \vdash K \leq K'} \quad \frac{\Gamma \vdash \mathcal{K} \leq K' \quad \Gamma, t : \mathcal{K} \vdash \varphi}{\Gamma \vdash \{t : \mathcal{K} \mid \varphi\} \leq \{t : \mathcal{K}' \mid \varphi'\}}
\]

Rule (sub-refkind) specifies that a refined kind is always a subkind of its unrefined variant. Rule (sub-ref) allows for subkinding between refined kinds, by requiring that the basic kind respects subkinding and that the refinements are equivalent (i.e. equi-provable).

**Kind Case and Bottom.** The kind case type-level mechanism is kinded in a natural way (rule (k-kcase)), accounting for the case where the kind of type \(T\) matches the specified kind \(\mathcal{K}'\) with type \(S\) and with type \(U\) otherwise.

\[
\frac{\Gamma \vdash \mathcal{K} \quad \Gamma \vdash T :: \mathcal{K}'' \quad \Gamma, t : \mathcal{K} :: S :: K' \quad \Gamma \vdash U :: K'}{\Gamma \models \text{if } T :: \mathcal{K} \text{ as } t \Rightarrow S \text{ else } U :: K'} \quad \frac{\Gamma \models \bot}{\Gamma \vdash \bot :: \mathcal{K}}
\]

Our treatment of \(\bot\) allows for \(\bot\) to be of *any* (well-formed) kind, provided one can conclude \(\bot\) is valid. The associated equality principles implement the kind case by testing the specified kind against the derivable kind of type \(T\). When \(\bot\) is provable from \(\Gamma\) then we can derive any equality via rule (eq-bot).
\[
\frac{\Gamma \vdash \varphi[T/t]}{\Gamma \vdash T :: K} \quad (\text{KREF})
\]

\[
\frac{\Gamma \vdash \varphi \quad \Gamma, \varphi \vdash T :: K \quad \Gamma, \neg \varphi \vdash S :: K}{\Gamma \vdash \text{if } \varphi \text{ then } T \text{ else } S :: K} \quad (\text{K-ITE})
\]

\[
\frac{\Gamma \vdash \varphi \quad \Gamma, \varphi \vdash T :: K \quad \Gamma, \neg \varphi \vdash T_1 :: K \quad \Gamma, \neg \varphi \vdash T_2 :: K}{\Gamma \vdash \text{if } \varphi \text{ then } T_1 \text{ else } T_2 :: K} \quad (\text{EQ-ITE}\text{T})
\]

\[
\frac{\Gamma \vdash \neg \varphi \quad \Gamma, \varphi \vdash T_1 :: K \quad \Gamma, \neg \varphi \vdash T_2 :: K}{\Gamma \vdash \text{if } \varphi \text{ then } T_1 \text{ else } T_2 :: K} \quad (\text{EQ-ITE}\text{E})
\]

\[
\frac{\Gamma \vdash T :: \{ t : K \mid \text{elim}_K(t) \equiv T' :: K' \}}{\Gamma \vdash \text{elim}_K(T) :: K'(T/t)} \quad (\text{K-ELIM})
\]

\[
\frac{\Gamma \vdash S :: \{ t : K \mid \text{elim}_K(T) \equiv T' :: K' \}}{\Gamma \vdash T'(T/t) :: K'(T/t)} \quad (\text{EQ-ELIM})
\]

\[
\frac{\Gamma \vdash K \quad \Gamma, t : K \vdash T :: K'}{\Gamma \vdash \lambda t : K.T :: \Pi t : K.K'} \quad (\text{K-FUN})
\]

\[
\frac{\Gamma \vdash T :: \Pi t : K.K' \quad \Gamma \vdash S :: K}{\Gamma \vdash T \cdot S :: K'(S/t)} \quad (\text{K-APP})
\]

\[
\frac{\Gamma \vdash (\lambda t : K.T) \cdot S \equiv T[S/t] :: K'(S/t)}{\Gamma \vdash \text{structural}(T, F, t)} \quad (\text{K-FIX})
\]

\[
\frac{\Gamma, t : K_1 \vdash K_2 \quad \Gamma, F : \Pi t : K_1.K_2, t : K_1 \vdash T :: K_2 \quad \Gamma \vdash S :: K_1}{\Gamma \vdash (\mu F : (\Pi t : K_1.K_2).\lambda t : K_1.T) \cdot S :: K_2\{S/t\}} \quad (\text{EQ-FIXUNF})
\]

\[
\frac{\Gamma \vdash T :: K \quad \Gamma, t : K \vdash S :: K' \quad \Gamma \vdash U :: K'}{\Gamma \vdash \text{if } T :: K \text{ as } t \Rightarrow S \text{ else } U :: K'} \quad (\text{EQ-KCASE}\text{T})
\]

\[
\frac{\Gamma \vdash \bot}{\Gamma \vdash \bot :: K} \quad (\text{EQ-BOT})
\]

\[
\frac{\Gamma \vdash T :: K_0 \quad \Gamma, K_0 \not\equiv K \quad \Gamma, t : K \vdash S :: K' \quad \Gamma \vdash U :: K'}{\Gamma \vdash \text{if } T :: K_0 \text{ as } t \Rightarrow S \text{ else } U :: K'} \quad (\text{EQ-KCASE}\text{F})
\]

A summary of the kinding and type equality rules is given in Figures 3 and 4.

**Example 3.1 (Representing Record Field Selection in types and values).** With the development presented up to this point we can already implement the more usual record selection operator $T.L$, where $T$ is a record type and $L$ is a field label of $T$. We represent such a construct as a type-level function that given some $L :: \text{Nm}$ produces a recursive type-function that essentially iterates over
\[\begin{align*}
\Gamma &\vdash t: \mathcal{K} & \Gamma &\vdash S :: K \\
\forall t: \mathcal{K}.T :: \text{Gen}_K & \quad (\text{K-\forall}) & \Gamma &\vdash \text{tmap}(\forall t: \mathcal{K}.T) \equiv T(S/t) :: \text{Type} \\
\Gamma &\vdash T :: \mathcal{K} & \Gamma &\vdash S :: \mathcal{K}' \\
\text{dom}(T \to S) &\equiv T :: \text{Type} & \Gamma &\vdash T :: \mathcal{K} & \Gamma &\vdash S :: \mathcal{K}' \\
\text{img}(T \to S) &\equiv S :: \text{Type} & \Gamma &\vdash T :: \mathcal{K} & \Gamma &\vdash S :: \mathcal{K}' \\
\end{align*}\]

Fig. 4. Kinding and Type Equality rules – 2 (Excerpt)

a type record of kind \(\{r::\text{Rec} \mid \ell \in \text{lab}(r)\}::\text{Type}\):

\[\lambda L::\text{Nm}.\ell F : (\Pi t::\{r::\text{Rec} \mid L \in \text{lab}(r)\}::\text{Type}).\lambda \ell::\{r::\text{Rec} \mid L \in \text{lab}(r)\}::\text{Type}.\text{headLabel}(t) \equiv L::\text{Nm} \quad \text{then} \quad \text{headType}(t) \equiv \ell::\text{Nm} \quad \text{else} \quad F(\text{tail}(t))\]

The function iteratively tests the label at the head of the record against \(L\), producing the type at the head of the record when the labels match and recursing otherwise. It is instructive to consider the kinding for the property test construct (let \(\Gamma_0\) be \(L::\text{Nm}, F::\{r::\text{Rec} \mid L \in \text{lab}(r)\}::\text{Type}, t::\{r::\text{Rec} \mid L \in \text{lab}(r)\}::\text{Type}\)):

\[\begin{align*}
\Gamma_0 \vdash \text{headLabel}(t) &\equiv L :: \text{Nm} \\
\text{if} \quad \Gamma_0 \vdash \text{headLabel}(t) \equiv L :: \text{Nm} \quad \text{then} \quad \text{headType}(t) \equiv \ell :: \text{Nm} \quad \text{else} \quad F(\text{tail}(t)) :: \text{Type} \\
\end{align*}\]
We allow for recursive terms via a fixpoint construct which is achieved via the reasoning principles built into our theory of refinements (see Section 4.1). Where \( D \) (dubbed \( \epsilon \)) and the concatenation of an element \( t \) with a collection \( N \) is well-formed we must be able to derive \( \Gamma \vdash t \equiv N \), which is also achieved via logical refinement reasoning.

### 4 A PROGRAMMING LANGUAGE WITH KIND REFINEMENTS

Having covered the key details of the kinding system and how type equality captures the appropriate type-level computations induced by our type manipulation constructs, we finally introduce the syntax and typing for our programming language per se.

The syntax of terms is given in Figure 5. Most constructs are standard. We highlight our treatment of records, mirroring that of record types, as (heterogeneous) lists of pairings of field labels and values of memory locations. Dependencies from terms in types, non-termination in the term language does not affect the overall soundness of the development. We also mirror the type-level property test and kind case constructs in the term language as \( \text{if } \varphi \text{ then } M \text{ else } N \) and \( \text{if } T :: K \text{ as } t \Rightarrow M \text{ else } N \), respectively. As we have initially stated, our language has general higher-order references, represented with the constructs \( \text{ref } M, \! M \) and \( M := N \), which create a reference to \( M \), dereference a reference \( M \) and assign \( N \) to the reference \( M \), respectively. As usual in languages with a store, we use \( l \) to stand for the runtime values of memory locations.

The typing rules for the language are given in Figure 6. The typing judgment is written as \( \Gamma \vdash_S M : T \), where \( S \) is a location typing environment. We write \( \Gamma ; S \vdash \) to state that \( S \) is a valid mapping from locations to well-kindred types, according to the typing context \( \Gamma \).
\begin{align*}
\Gamma \vdash \text{(recterm)} & \quad \Gamma \vdash \text{(recTail)} \\
\Gamma \vdash \text{(true)} & \quad \Gamma \vdash \text{(false)} \\
\Gamma \vdash \text{(emp)} & \quad \Gamma \vdash \text{(cons)} \\
\Gamma \vdash \text{(ref)} & \quad \Gamma \vdash \text{(derref)} \\
\Gamma \vdash \text{(prop-ite)} & \quad \Gamma \vdash \text{(kindcase)} \\
\Gamma \vdash \text{(conv)} & \quad \Gamma \vdash \text{(fix)} \\
\Gamma \vdash \text{(fl)} & \quad \Gamma \vdash \text{(ye)} \\
\Gamma \vdash \text{(reclabel)} & \quad \Gamma \vdash \text{(rectail)}
\end{align*}

Fig. 6. Typing Rules
The main difference with respect to the standard rules for a language of this nature appears in the rules for the various elimination forms. Consider the function application rule:

\[
\Gamma \vdash T_1 :: \{ t :: \text{Fun} \mid \text{dom}(t) \equiv T_2 :: \mathcal{K} \land \text{img}(t) \equiv U :: \mathcal{K}' \}
\]

\[
\Gamma \vdash S M : T_1 \quad \Gamma \vdash S N : T_2
\]

\[
\Gamma \vdash S M N : U[T_1/t]
\]

Instead of stating that \( M \) is of type \( T_2 \rightarrow U \), we use the refinement kind information to specify that \( M \) is of some type \( T_1 \) whose kind is Fun with domain type \( T_2 \) and image type \( U \). The formulation via kind refinement subsumes the standard formulation, since (assuming \( T_2 \) and \( U \) are well-formed) we can trivially derive that \( T_2 \rightarrow U :: \{ f :: \text{Fun} \mid \text{dom}(t) \equiv T_2 :: \mathcal{K} \land \text{img}(t) \equiv U :: \mathcal{K}' \} \) from the equality principles of the function type destructors. The key advantage in our presentation is that it allows us to derive typings of the form

\[
\vdash \Lambda s :: \text{Type}. \Lambda t :: \{ f :: \text{Fun} \mid \text{dom}(f) \equiv s :: \text{Type} \land \text{img}(f) \equiv \text{Bool} :: \text{Type} \}.
\]

\[
\lambda x :: t. \lambda y :: (x y) :: \forall s :: \text{Type}. \forall t :: \{ f :: \text{Fun} \mid \text{dom}(f) \equiv s :: \text{Type} \land \text{img}(f) \equiv \text{Bool} :: \text{Type} \}. \text{Bool}
\]

Despite not knowing the exact form of the function type that is to be instantiated for \( t \), by specifying its domain and image types we can type applications of terms of type \( t \) correctly. This is in contrast with what happens in existing type theories (even those with sophisticated dependent types such as Agda [Norell 2007] or that of Coq [CoqDevelopmentTeam 2004]), where the leveraging of dependent types, explicit equality proofs and equality elimination would be needed to provide an “equivalently” typed term. Thus, all our elimination rules follow this general pattern, where we exploit the kind of the type of the term being deconstructed to inform the typing. We also highlight the typing of the property test term construct,

\[
\text{(prop-ite)}
\]

\[
\Gamma \vdash \varphi \quad \Gamma, \varphi \vdash S M : T_1 \quad \Gamma, \neg \varphi \vdash S N : T_2
\]

\[
\Gamma \vdash S \text{if } \varphi \text{ then } M \text{ else } N : \text{if } \varphi \text{ then } T_1 \text{ else } T_2
\]

which types the term \( \text{if } \varphi \text{ then } M \text{ else } N \) with the type \( \text{if } \varphi \text{ then } T_1 \text{ else } T_2 \) and thus allows for a conditional branching where the types of the branches differ. Rule (\text{kindcase}) mirrors the equivalent rule for the type-level kind case, typing the term if \( T :: \mathcal{K} \text{ as } t \Rightarrow M \text{ else } N \) with the type \( U \) of both \( M \) and \( N \) but testing the kind of type \( T \) against \( \mathcal{K} \). Such a construct enables us to define non-parametric polymorphic functions, and introduce forms of ad-hoc polymorphism. For instance, we can derive the following:

\[
\Lambda s :: \text{Type}. \lambda x :: s. \text{if } s :: \text{Ref} \text{ as } t \Rightarrow (\text{if } \text{refOf}(t) \equiv \text{Int} :: \text{Type} \text{ then } !x \text{ else } 0) \text{ else } 0 :: \forall s :: \text{Type}. s \rightarrow \text{Int}
\]

The function above takes a type \( s \), a term \( x \) of that type and, if \( s \) is of kind \text{Ref} such that \( s \) is a reference type for integers (note the use of reflection using destructor \text{refOf}(-) on type \( s \), returns \(!x\), otherwise simply returns \( 0 \). The typing exploits the equality rule for the property test where both branches are the same type.

Finally, as expected, the type conversion rule (\text{conv}) allows us to coerce between equal types of a basic kind, allowing for type-level computation to manifest itself in the typing of terms.

\textbf{Example 4.1 (Record Selection).} Using the record selection type of Example 3.1 we can construct a term-level analogue of record selection. Given a label \( L \) and a term \( M \) of type \( T \) of kind \( \{ r :: \text{Rec} \mid L \in r \} \), we define the record selection construct \( M.L \) as (for conciseness, let \( \mathcal{R} = \{ r :: \text{Rec} \mid L \in \text{lab}(r) \} \)):

\[
M.L \triangleq (\mu F :: t :: \mathcal{R}. t \rightarrow (t.L)L). \Lambda t :: \mathcal{R}. \lambda x :: t.
\]

\[
\text{if headLabel}(t) \equiv L :: \text{Nm} \text{ then recHeadTerm}(x) \text{ else } F[t\text{tail}(t)](\text{tail}(x)) \Rightarrow T M
\]
such that $M.L : T.L$. The typing requires crucial use of type conversion to allow for the unfolding of the recursive type function to take place (let $\Gamma_0$ be $F:\forall t :: R.t \rightarrow (t.L), x:T$):

```
| (conv) | $\Gamma_0 \vdash$ if headLabel($T$) $\equiv$ $L :: Nm$ then headType($T$) else tail($T$).$L$ $\equiv$ $T.L :: Type$
```

\[ \Gamma_0 \vdash$ if headLabel($T$) $\equiv$ $L :: Nm$ then recHeadTerm($x$) else $F[tail(T)](tail(x)) : T.L \]

with $\mathcal{D}$ a derivation of

\[ \Gamma_0 \vdash$ if (headLabel($T$) $\equiv$ $L :: Nm)$ then recHeadTerm($x$) else $F[tail(T)](tail(x)) : T_0 \]

where $T_0$ is if (headLabel($T$) $\equiv$ $L :: Nm$) then headType($T$) else tail($T$).$L$, requiring a similar extended equational reasoning to that of Example 3.1. Specifically, in the then branch we must show that $\Gamma_0$, headLabel($T$) $\equiv$ $L :: Nm$ $\vdash$ recHeadTerm($x$) : headType($T$), which is derivable from $x:T$ and $T :: \{r::Rec \mid headType(r) :: headType(r) :: Type\}$ -- the latter following from refinement and reflexivity of equality -- via typing rule (recterm).

The else branch requires showing that $\Gamma_0$, headLabel($T$) $\equiv$ $L :: Nm$ $\vdash$ headType($T$) $\equiv$ $L :: Nm$ $\vdash$ recHeadTerm($x$) $\equiv$ headType($T$) $\equiv$ Type} -- the latter following from refinement and reflexivity of equality -- via typing rule (recterm).

The else branch requires showing that $\Gamma_0$, $\neg$headLabel($T$) $\equiv$ $L :: Nm$ $\vdash$ $F[tail(T)](tail(x)) : tail(T).L$, which is derivable from $F : \forall t :: R.t \rightarrow (t.L)$ and $x:T$ as follows: tail($T$) $:: R$ is follows from $\neg$headLabel($T$) $\equiv$ $L$ and $T :: R$ (see Section 4.1), thus $F[tail(T)] : tail(T) \rightarrow tail(T).L$. Since tail($x$) $:: tail(T)$ from $x : T$ and $T :: \{r::Rec \mid tail(t) :: tail(t) :: Rec\}$ via rule (rectail), we conclude using the application rule.

Thus, combining the type and term-level record projection constructs we have that the following is admissible:

\[ \Gamma \vdash L :: Nm \quad \Gamma \vdash M : T \quad \Gamma \vdash T :: \{r::Rec \mid L \in lab(r)\} \]

\[ \Gamma \vdash M.L : T.L \]

### 4.1 Reasoning in Refinements

In the various examples and code snippets throughout this paper we have used reasoning principles on refinements (and the equalities present therein) that go beyond the standard definitional equality principles of $\beta$-conversion of types (i.e. type-level computation combined with congruence principles).

From a foundational point of view, enriching the type-theoretic definitional equality (i.e. the internal equality of the theory that does not require the explicit construction of proof objects) beyond the simple principles of $\beta$-conversion and related computation principles can easily make type-checking undecidable. The tension between the power and decidability of definitional equality is essentially the major design choice of any type theory. Broadly speaking, type theories either have a very powerful and undecidable definitional equality (i.e. extensional type theories) or a limited but decidable definitional equality (i.e. intensional type theories) [Hofmann 1997]. For instance, the theories underlying Coq and Agda fall under the latter category, whereas the theory underlying a system such as those in the NuPRL family [Constable et al. 1986] are of the former variety.

Languages with refinement types such as Liquid Haskell [Vazou et al. 2014], F-Star [Swamy et al. 2011] (or with constrained forms of dependent types such as Dependent ML [Xi 2007]) live somewhere in the middle of the spectrum, effectively equipping types with a richer notion of equality (via the automated reasoning associated with the logic of refinements) but disallowing the full power of extensional theories in order to preserve decidability of type-checking. Our approach follows in this tradition, and so we allow for limited forms of additional logical reasoning on refinements, extending equality with axiom schemas that pertain to the manipulation of type-level records and finite sets of record labels, as well as (decidable) predicates on types which are left unspecified since they can be defined according to the specific domain-specific needs. Thus, the full
logic of refinements consists of (classical) propositional logic, conversion of types and the reasoning that follows from type predicates and the axiom schemas of Figure 7.

We adopt the following notational conventions: capital letters $R, S, L, N$ stand for universally quantified objects of the appropriate kind (omitted for conciseness); as mentioned in Section 2, $\text{lab}(R)$ stands for a refinement level operation that given a record $R$ produces a finite set containing all the field labels of $R$; field labels can be concatenated using operation $N++L$, appending $L$ to $N$, which is overloaded on finite sets of labels (e.g. $N++\text{lab}(R)$, denoting the set obtained by prefixing $N$ to all labels in $\text{lab}(R)$). The (label) set operations of membership test $L \in S$, apartness $S \# S'$, equality $S = S$ and union $S \cup S'$ have the obvious meanings and their axiomatization is omitted for conciseness. Finally, the predicate $\text{nonEmpty}(R)$ is defined as notation for $(R \equiv \emptyset)$.

Thus, axiom $(\text{Rec-EmpOrCons})$ characterizes the fact that a record type must be the empty record or the concatenation of its head elements to its tail; axiom $(\text{Rec-DisjointLabels})$ codifies the disjointness principle of record field labels, where in all but the empty record, the label at the head of a record cannot be in the label set of its tail; Axioms $(\text{Lab-NotInEmpty})$ and $(\text{Lab-InHeadTail})$ specify that no label is in the label set of the empty record and moreover, a label is in the label set of $R$ if it is the label at the head of the record or in the label set of the tail of $R$; axiom $(\text{LabConcatEq})$ specifies label or name concatenation; axiom $(\text{LabelSet-InEq})$ allows for combined reasoning of inclusion and label set equality; finally, the axioms $(\text{LabelConcat-SetConcat})$ and $(\text{LabelSet-Concat})$ deal with field or name concatenation, respectively specifying that a label $L$ being a member of the label set of $S$ is equivalent to the prefixing of $N$ to $L$ being a member of the (set-level) concatenation on $N$ to the set of labels of $S$, and that labels sets are closed under prefixing.

All the various examples throughout the paper are derivable via the reasoning principles codified above. For instance, as mentioned in Example 4.1, given $L \in \text{lab}(T)$ and $\neg(\text{headLabel}(T) \equiv L)$ we can derive that $L \in \text{lab}(\text{tail}(T))$ through axiom $(\text{Lab-InHeadTail})$ and some basic propositional reasoning. Similarly, in Example 3.1 we derive $\text{nonEmpty}(t)$ from $L \in \text{lab}(t)$ via axiom $(\text{Lab-NotInEmpty})$ and propositional reasoning. In the XML table example of Section 2, we derive $\text{nonEmpty}(TT)$ from $\text{nonEmpty}(R)$ and $\text{lab}(TT) = \text{lab}(R)$ via $(\text{LabelSet-InEq})$. 

Fig. 7. Axiom Schemas for Record Types and Labels
5 OPERATIONAL SEMANTICS AND METATHEORY

We now formulate the operational semantics of our language and develop the standard type safety results in terms of uniqueness of types, type preservation and progress.

Since the programming language includes a higher-order store, we formulate its semantics in a (small-step) store-based reduction semantics. Recalling that the syntax of the language includes the runtime representation of store locations \( l \), we represent the store \((H, H')\) as a finite map from labels \( l \) to values \( v \). Given that kinding and refinement information is needed at runtime for the property and kind test constructs, we tacitly thread a typing environment in the reduction semantics.

Moreover, since types in our language are themselves structured objects with computational significance, we make use of a type reduction relation, written \( T \rightarrow T' \), defined as a call-by-value reduction semantics on types given by orienting the type equality rules of Figures 3 and 4, excluding rule \((\text{EQ-ELIM})\), left-to-right, plus congruence rules (for the sake of brevity, and due to its straightforward nature, we omit a complete definition of type reduction). It is convenient to define a notion of type value, denoted by \( T_v, S_v \), and given by the following grammar:

\[
T_v, S_v ::= \lambda\ell::K.T | \forall l::K.T | \ell | \emptyset | (\ell : T_v)@S_v | T_v^* | \text{ref } T_v | T_v \rightarrow S_v | \bot | \text{Bool} | 1 | t
\]

We note that it follows naturally that type reduction is strongly normalizing. The values of the term language are defined by the grammar:

\[
v, v' ::= \text{true} | \text{false} | \emptyset | (\ell = v)@v' | \lambda x:T.M | \Lambda::K.M | v :: v' | \varepsilon | l
\]

Values consist of the booleans true and false (extensions to other basic data types are straightforward as usual); the empty record \( \emptyset \); the non-empty record that assigns fields to values, \( (\ell = v)@v' \); the empty collection, \( \varepsilon \), and the non-empty collection of values, \( v :: v' \); as well as type and \( \lambda \)-abstraction.

For convenience of notation we write \( (\ell_1 : T_1, \ldots, \ell_n : T_n) \) for \( (\ell_1 : T_1)\@\ldots\@(\ell_n : T_n)\@\emptyset \), and similarly \( (\ell_1 = M_1, \ldots, \ell_n = M_n) \) for \( (\ell_1 = M_1)\@\ldots\@(\ell_n = M_n)\@\emptyset \).

The operational semantics are defined in terms of the judgment \( (H; M) \rightarrow (H'; M') \), indicating that term \( M \) with store \( H \) reduces to \( M' \), resulting in the store \( H' \). For conciseness, we omit congruence rules such as:

\[
\begin{align*}
\text{(R-RecConsL)} & : \quad (H; M) \rightarrow (H'; M') \\
\text{where the record field labelled by } \ell \text{ is evaluated (and the resulting modifications in store } H \text{ to } H' \\
\text{are propagated accordingly). The reduction rules enforce a call-by-value, left-to-right evaluation order and are listed in Figure 8 (note that we require types occurring in an active position to be first reduced to a type value, following the call-by-value discipline). We refer the reader to Appendix B for the complete set of rules.}
\end{align*}
\]

The three rules for the record destructors project the appropriate record element as needed. The treatment of references also standard, with rule \((\text{R-RefV})\) creating a new location \( l \) in the store which then stores value \( v \); rule \((\text{R-DerefV})\) querying the store for the contents of location \( l \); and rule \((\text{R-AssignV})\) replacing the contents of location \( l \) with \( v \) and returning \( v \). Rules \((\text{R-PropT})\) and \((\text{R-PropP})\) are the only ones that appeal to the entailment relation for refinements, making use of the running environment \( \Gamma \) which is threaded through the reduction rules straightforwardly. Similarly, rules \((\text{R-KindL})\) and \((\text{R-KindR})\) mimic the equality rules of the kind case construct, testing the kind of type \( T \) against \( K \).
We now develop the main metatheoretical results for our theory of type preservation, progress and uniqueness of kinding and typing. We begin by noting that types and their kinding system are not significantly more complex than a minimal type theory such as LF [Harper et al. 1993; Harper and Pfenning 2005], given that types form a $\lambda$-calculus that is then “dependently typed” by kinds and kind refinements (plus the additional equational reasoning on refinements). Without refinements, the type level constructs are essentially those of $F_\omega$ [Girard 1986] augmented with our primitives to...
manipulate types as data and conditional types. Further, when we consider terms and their typing
there is no significant additional complexity since types occur in terms but not vice-versa.

In the remainder of this section we write $\Gamma \vdash \mathcal{J}$ to stand for a typing, kinding, entailment or
equality judgment as appropriate. Since the refinement language is not fully specified, we must
assume some basic properties of (non-equality) refinements, which we summarise in Proposition 5.1
below, where we use refinements $\varphi$ and $\psi$ to stand for refinements that are not derived using the
equality rules of Section 3.2 – for those we develop the necessary properties by appealing to these
basic principles of the incompletely specified refinement language.

Postulate 5.1 (Assumed Properties of Refinements).

Substitution: If $\Gamma \vdash T :: K$ and $\Gamma, t : K \vdash \varphi$ then $\Gamma, \Gamma'(T/k) \models \varphi(T/t)$;

Weakening: If $\Gamma \models \varphi$ then $\Gamma' \models \varphi$ where $\Gamma \subseteq \Gamma'$;

Cut: If $\Gamma \models \varphi$ and $\Gamma, \varphi \models \psi$ then $\Gamma \models \psi$;

Identity: $\Gamma, \varphi, \Gamma' \models \varphi$, for any $\varphi$;

Functionality: If $\Gamma \models T \equiv S :: K$ and $\Gamma, t : K \vdash \varphi$ then $\Gamma \models \varphi(T/t) \equiv \varphi(S/t)$.

Decidability: $\Gamma \models \varphi$ is decidable.

The general structure of the development is as follows: we first establish basic structural properties
of substitution (Lemma 5.1) and weakening, which we can then use to show that we can apply
type and kind conversion inside contexts (Lemma 5.2), which then can be used to show a so-called
validity property for equality (Theorem 5.3), stating that equality derivations only manipulate
well-formed objects (from which kind preservation – Corollary 5.4 – follows immediately).

Lemma 5.1 (Substitution).

(a) $\Gamma \vdash T :: K$ and $\Gamma, t : K \vdash \mathcal{J}$ then $\Gamma, \Gamma'(T/t) \vdash \mathcal{J}(T/t)$;

(b) $\Gamma \vdash M : T$ and $\Gamma, x : T, \Gamma' \vdash N : S$ then $\Gamma, \Gamma' \vdash N[M/x] : S$.

Lemma 5.2 (Context Conversion).

(a) Let $\Gamma, x : T \vdash \mathcal{J}$ and $\Gamma \vdash T' :: K$.

(b) Let $\Gamma, t : K \vdash \mathcal{J}$ and $\Gamma \vdash K'$.

Theorem 5.3 (Validity for Equality).

(a) If $\Gamma \vdash K \equiv K'$ then $\Gamma \vdash K$ and $\Gamma \vdash K'$.

(b) If $\Gamma \vdash T \equiv T' :: K$ then $\Gamma \vdash T \rightarrow K$, $\Gamma \vdash T :: K$ and $\Gamma \vdash T' :: K$.

(c) If $\Gamma \vdash \varphi \equiv \psi$ then $\Gamma \vdash \varphi$ and $\Gamma \vdash \psi$

Corollary 5.4 (Kind Preservation). If $\Gamma \vdash T :: K$ and $T \rightarrow T'$ then $\Gamma \vdash T' :: K$.

This setup then allows us to show functionality properties of kinding and equality (Lemmas 5.5
and 5.6). Lemma 5.5 essentially states that substitution is consistent with the theory’s internal
equality judgment (i.e. substituting an object $X$ in some $Y$ is equal to substituting any object $X'$,
equal to $X$, in $Y$). Similarly, Lemma 5.6 shows that equality is compatible with substitution of equals.

Lemma 5.5 (Functionality of Kinding and Refinements).

Assume $\Gamma \vdash T \equiv S :: K$, $\Gamma \vdash T :: K$ and $\Gamma \vdash S :: K$:

(a) If $\Gamma, t : K, \Gamma' \vdash T' :: K'$ then $\Gamma, \Gamma'(T/t) \models T'(T/t) \equiv T'(S/t) :: K'(T/t)$

(b) If $\Gamma, t : K, \Gamma' \vdash K'$ then $\Gamma, \Gamma'(T/t) \vdash K(T/t) \models K(S/t)$.

(c) If $\Gamma, t : K, \Gamma' \vdash \varphi$ then $\Gamma, \Gamma'(T/t) \models \varphi(T/t) \equiv \varphi(S/t)$

Lemma 5.6 (Functionality of Equality). Assume $\Gamma \vdash T_0 \equiv S_0 :: K$:
To the best of our knowledge, ours is the first work to explore the concept of refinement kinds and

\[ S \subseteq \] The inversion principles can be found in Appendix C.

programming, [Chlipala 2010] developed a type system that supports type-level record computations

\[ \text{[Swamy et al. 2011], JavaScript [Vekris et al. 2016].} \]

\[ \text{[Bengtson et al. 2011; Rondon et al. 2008; Vazou et al. 2013], which effectively extends type} \]

\[ \text{statically typed meta-programming features such as type reflection, ad-hoc polymorphism, and} \]

\[ \text{type-level computation.} \]

\[ \text{Finally, progress can be established in a fairly direct manner (relying on a straightforward} \]

\[ \text{notion of progress for the type reduction relation). The main interesting aspect is that progress} \]

\[ \text{relies crucially on the decidability of entailment due to the term-level and type-level predicate test} \]

\[ \text{construct.} \]

\[ \text{LEMMA 5.10 (TYPE PROGRESS).} \]

\[ \text{THEOREM 5.11 (PROGRESS).} \]

\[ \text{RELATED WORK} \]

\[ \text{To the best of our knowledge, ours is the first work to explore the concept of refinement kinds and} \]

\[ \text{illustrate their expressiveness as a convenient programming language feature that cleanly integrates} \]

\[ \text{statically typed meta-programming features such as type reflection, ad-hoc polymorphism, and} \]

\[ \text{type-level computation.} \]

\[ \text{The concept of refinement kind is a natural extension of the well-known notion of refinement} \]

\[ \text{type [Bengtson et al. 2011; Rondon et al. 2008; Vazou et al. 2013], which effectively extends type} \]

\[ \text{specifications with (SMT decidable) logical assertions. Refinement types have been applied to} \]

\[ \text{various verification domains such as security [Bengtson et al. 2011] or the verification of data-} \]

\[ \text{structures [Kawaguchi et al. 2009; Xi and Pfenning 1998], and are being incorporated in full-fledged} \]

\[ \text{programming languages, e.g., ML [Freeman and Pfenning 1991] Haskell [Vazou et al. 2014], F*} \]

\[ \text{[Swamy et al. 2011].} \]

\[ \text{With the aim of supporting common meta-programming idioms in the domain of web pro-} \]

\[ \text{gramming, [Chipala 2010] developed a type system that supports type-level record computations} \]
with similar aims as ours, fully avoiding type dependency. In our case, we generalise type-level
computations to other types as data, and rely on more amenable explicit type dependency, in
the style of System-F polymorphism. Therefore, we still avoid the need to pollute programs with
explicit proof terms, but through our development of a principled theory of kind refinements.

Our extension of the concept of refinements to kinds, together with the introduction of primitives
to reflectively manipulate types as data (cf. ASTs) and express constraints on those data also
highlights how kind refinements match fairly well with the programming practice of our time (e.g.,
interface reflection in Java-like languages), contrasting the focus of our work with the goals of
other approaches to meta-programming such as [Altenkirch and McBride 2002; Calcagno et al.
2003]. The work of [Weirich et al. 2013] studies an extension to the core language (System FC)
of the Glasgow Haskell Compiler (GHC) with a notion of kind equality proofs, in order to allow
type-level computation in Haskell to refer to kind-level functions. Their development, being based
on System FC, is designed to manipulate explicit type (and kind) coercions as part of the core
language itself, which have a non-trivial structure (as required by the various type features of GHC),
and thus differs significantly from our work which is designed to keep type and kind conversion as
implicit as possible. However, their work can be seen as a stepping stone towards the integration of
refinement kinds and related constructs in a general purpose functional language such as Haskell.

The relationship between refinement types and dependent types through proof irrelevance,
allowing the programmer to avoid explicitly writing proof witnesses for refinements, was clarified
by [Freeman and Pfenning 1991]. The idea of expressing constraints (e.g., disjointness) on record
labels with predicates goes back to [Harper and Pierce 1991], although in our system we admit in
the refinement logic convenient predicates and operators applicable to not just record types, but
also to other kinds of types such as function types, collections types and even polymorphic function
types. The basic concept of a statically checked type-case construct was introduced in [Abadi et al.
1991]; however, our refinement kind checking of dynamic type conditionals on types and kinds
if $\varphi$ then $e_1$ else $e_2$ and if $T :: K$ as $t$ ⇒ $e_1$ else $e_2$ greatly extends the precision of type and kind
checking, and naturally supports very flexible forms of statically checked ad-hoc polymorphism, as
we have shown.

Some works [Fähndrich et al. 2006; Huang and Smaragdakis 2008; Smaragdakis et al. 2015]
have addressed the challenge of typing specific meta-programming idioms in concrete languages
such as Java and C#. Our work shows instead how the fundamental concept of refinement kinds
suggests itself as a general type-theoretic principle towards statically checked typeful [Cardelli
1991] meta-programming, including programs that manipulate types as data, or build types and
programs from data (e.g., as the type providers of F# [Petricek et al. 2016]) which seems to be
out of reach for existing static type systems. Our language conveniently expresses programs that
automatically generate types and operations from data specifications, while statically ensuring that
generated types satisfy the intended invariants, as expressed by refinements.

7 CONCLUDING REMARKS

We have introduced the concept refinement kinds and developed the associated type theory, in
the context of higher-order polymorphic $\lambda$-calculus with imperative constructs, several kinds of
datatypes, and type-level computation. The resulting programming language cleanly supports static
typing of sophisticated features such as type-level reflection, ad-hoc and parametric polymorphism
which can be elegantly combined in order to provide non-trivial meta-programming idioms, which
we have illustrated with several examples.

While the full development of an algorithmic formulation of our type system is under development
(together with an implementation) implementation we note that, given that the type derivations rely
on the entailment for refinements (which include type equalities in general), it is crucial that such a
judgment remain decidable. While the interaction of type equality and logical kind refinements can be non-trivial, the type equality principles defined in Section 3.2 essentially amount to normalising (which can require deciding logical refinement) the types and comparing normal forms. Kinding, typing and refinements also require reasoning about equality up-to type predicates and the axiom schemas of Section 4.1. However, just as modern refinement type systems make extensive use of SMT solvers to offload the reasoning about refinement properties (which can refer to data values and thus make the reasoning significantly more complex than our manipulation of types as simple tree-like structures), a reasonable algorithmic development of our theory relies on a combination of type normalisation and SMT solvers to derive the necessary refinements.

There are many interesting avenues of exploration that have been opened by this work. From a theoretical point-of-view, it would be instructive to study the tension imposed on shallow embeddings of our system in general dependent type theories such as Coq. After including existential types, variant types and higher-type imperative state (e.g., the ability to introduce references storing types at the term-level), which have been left out of this presentation for the sake of focus, it would be relevant to further investigate limited forms of dependent types or refinement types. It would be also interesting to investigate how refinement kinds and stateful types (e.g., tystate or other forms of behavioral types) may be used to express and type-check invariants on meta-programs with challenging scenarios of strong updates, e.g., involving changes in representation of abstract data types.

REFERENCES


Appendix

Refinement Kinds
A Theory of Type-Safe Meta-Programming

Additional definitions and proofs of the main materials.
We define the syntax of kinds $K, K'$, refinements $\phi, \phi'$, types $T, S, R$, and terms $M, N$ below. We assume countably infinite sets of type variables $X$, names $N$ and term variables $V$. We range over type variables with $t, t', s, s'$, name variables with $n, m$ and term variables with $x, y, z$.

Kinds

\[
K, K' ::= \mathcal{K} \mid \{t::K \mid \phi\} \mid \Pi t::K.
\]

Revised and Dependent Kinds

Base Kinds

Types

\[
T, S, R ::= t \mid \lambda t::K.T \mid TS
\]

Type-level Functions

Structural Recursion

Polymorphism

Refinements $\phi, \psi$

\[
\phi, \psi ::= P(T_1, \ldots, T_n)
\]

Type Predicates

Propositional Logic

Terms

\[
M, N ::= x \mid \lambda x:T.M \mid M N
\]

Functions

Type Abstraction and Application

Type Equality

Logs

true | false

Booleans

if $\phi$ then $M$ else $N$

Property Test

if $T :: K$ as $t$ ⇒ $M$ else $N$

Kind Case

$\epsilon \mid M :: N$

References

$\mu F:T.M$

Recursion
A.1 Kinding and Typing

Our type theory is defined by the following judgments:

\( \Gamma \vdash K \) is a well-formed kind under the assumptions in \( \Gamma \)

\( \Gamma \vdash \phi \) Refinement \( \phi \) is well-formed under the assumptions in \( \Gamma \)

\( \Gamma \vdash T :: K \) Type \( T \) is a (well-formed) type of kind \( K \) under the assumptions in \( \Gamma \)

\( \Gamma \vdash S M : T \) Term \( M \) has type \( T \) under the assumptions in \( \Gamma \) and store typing \( S \)

\( \Gamma \vdash \phi \) Refinement \( \phi \) holds under the assumptions in \( \Gamma \)

\( \Gamma \vdash \phi \equiv \psi \) Refinements \( \phi \) and \( \psi \) are equal

\( \Gamma \vdash K \equiv K' \) Kinds \( K \) and \( K' \) are equal

\( \Gamma \vdash K \leq K' \) Kind \( K \) is a sub-kind of \( K' \)

\( \Gamma \vdash T \equiv T' :: K \) Types \( T \) and \( T' \) of kind \( K \) are equal

We also parameterize typing by a signature of type-level constants that specify basic well-formedness constraints on the various type destructors:

\[
\begin{align*}
\text{headLabel} & :: \Pi t : \{r :: \text{Rec} \mid \text{notEmpty}(r)\}.\text{Nm} \\
\text{headType} & :: \Pi t : \{r :: \text{Rec} \mid \text{notEmpty}(r)\}.\text{Type} \\
n\text{tail} & :: \Pi t : \{r :: \text{Rec} \mid \text{notEmpty}(r)\}.\text{Rec} \\
\text{refOf} & :: \Pi t : \text{Ref}.\text{Type} \\
\text{colOf} & :: \Pi t : \text{Col}.\text{Type} \\
\text{dom} & :: \Pi t : \text{Fun}.\text{Type} \\
\text{img} & :: \Pi t : \text{Fun}.\text{Type} \\
\text{tmap} & :: \Pi t : \text{Gen}_K.\Pi s : K.\text{Type}
\end{align*}
\]

We write \( \text{elim}_K(T) \) to range over elimination forms for a given (base) kind \( K \) applied to type \( T \).

**Context Well-formedness.**

\[
\begin{array}{llllll}
\Gamma \vdash K & \Gamma \vdash T :: K & \Gamma \vdash \phi & \Gamma \vdash S \vdash \Gamma \vdash \Gamma \vdash T :: K \\
\Gamma, t : K \vdash & \Gamma, x : T \vdash & \Gamma, \varphi \vdash & \Gamma ; S ; I : T \vdash & \cdot \vdash \\
\Gamma ; \cdot \vdash & \\
\end{array}
\]

**Kind well-formedness.**

\[
\begin{align*}
\Gamma \vdash K & \in \{\text{Rec}, \text{Col}, \text{Fun}, \text{Ref}, \text{Nm}, \text{Type}\} \\
\Gamma \vdash K & \quad \Gamma \vdash t : K \vdash K' \\
\Gamma \vdash K & \quad \Gamma \vdash \Pi t : K. K' \\
\Gamma \vdash K & \quad \Gamma \vdash K & \quad \Gamma \vdash K & \quad \Gamma \vdash t : K \vdash \varphi \\
\Gamma \vdash \text{Gen}_K & & \Gamma \vdash \{t :: K | \varphi\}
\end{align*}
\]

**Refinement Well-formedness.** We presuppose a signature \( \Sigma \) that specifies predicates, their arities and the kinds of their type arguments. We assume that kinds occurring in a signature have been checked for well-formedness.

\( P : K_1, \ldots, K_n \in \Sigma \quad \forall i \in \{1, \ldots, n\}. \Gamma \vdash T_i :: K_i \)

+ Well-formedness of propositional logic formulas

\[ \Gamma \vdash P(T_1, \ldots, T_n) \]

\[ \Gamma \vdash T :: K \quad \Gamma \vdash S :: K \]

\[ \Gamma \vdash T \equiv S :: K \]

**Refinement Satisfiability.**

**Propositional Logic**

\[ \Gamma \models T \equiv S :: K \quad \Gamma, x : K \vdash \varphi \quad \Gamma \models \varphi[T/x] \]

\[ \Gamma \models \varphi[S/x] \] (EqElim)

**Kinding.**

\[ t : K \in \Gamma \quad \Gamma \vdash \quad \Gamma \vdash T :: K \quad \Gamma \vdash K \leq K' \quad \Gamma \vdash \quad \Gamma \vdash T :: K' \]

\[ \Gamma \vdash T :: \Pi t : K.K' \quad \Gamma \vdash S :: K \quad \Gamma \vdash K \quad \Gamma, t : K \vdash T :: K' \]

\[ \Gamma \vdash T S :: K'(S/t) \quad \Gamma \vdash \lambda t : K.T :: \Pi t : K.K' \]

\[ \Gamma \vdash K \quad \Gamma, t : K \vdash T :: K \quad \Gamma \vdash \ell : \mathbb{N} \quad \Gamma \vdash \quad \Gamma \vdash \quad \Gamma \vdash \]

\[ \Gamma \vdash \quad \Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: K \quad \Gamma \vdash S :: \{t :: \text{Rec} | L \not\in \text{lab}(t)\} \]

\[ \Gamma \vdash \emptyset :: \text{Rec} \quad \Gamma \vdash \langle L : T \rangle @ S :: \text{Rec} \]

\[ \Gamma \vdash T :: K \quad \Gamma \vdash S :: K' \quad \Gamma \vdash T :: K \quad \Gamma \vdash T :: K \]

\[ \Gamma \vdash T \rightarrow S :: \text{Fun} \quad \Gamma \vdash T^* :: \text{Col} \quad \Gamma \vdash \text{ref} T :: \text{Ref} \]

\[ \Gamma \vdash T :: \{t :: K \mid \text{elim}_K(t) \equiv T' :: K'\} \quad \Gamma \vdash T'(T/t) :: K'(T/t) \]

\[ \Gamma \vdash \text{elim}_K(T) :: K'(T/t) \]

\[ \Gamma \vdash \varphi \quad \Gamma, \varphi \vdash T :: K \quad \Gamma, \neg \varphi :: S :: K \]

\[ \Gamma \vdash K \quad \Gamma \vdash T :: K' \quad \Gamma \vdash t : K \vdash S :: K' \quad \Gamma \vdash U :: K' \]

\[ \Gamma \vdash \text{if } \varphi \text{ then } T \text{ else } S :: K \]

\[ \Gamma \vdash \text{if } T :: K \text{ as } t \Rightarrow S \text{ else } U :: K' \]

\[ \Gamma, F : \Pi t : K.K', t : K \vdash T :: K' \quad \text{structural}(T, F, t) \]

\[ \Gamma \vdash \mu F : (\Pi t : K.K').\lambda t : K.T :: \Pi t : K.K' \]

\[ \Gamma \models \bot \quad \Gamma \vdash \quad \Gamma \vdash \varphi[T/t] \quad \Gamma \vdash T :: K \]

\[ \Gamma \vdash \bot :: K \quad \Gamma \vdash T :: \{t :: K \mid \varphi\} \]

**Sub-kinding.**

\[ \Gamma \vdash K \equiv K' \]

\[ \Gamma \vdash K \leq K' \quad \Gamma \vdash \quad \Gamma \vdash K \leq K' \]

\[ \Gamma \vdash K \quad \Gamma, t : K \vdash \varphi \quad \Gamma \vdash K \leq K' \quad \Gamma \vdash t : K' \models \varphi \equiv \varphi' \]

\[ \Gamma \vdash \{t :: K \mid \varphi\} \leq K \quad \Gamma \vdash \{t : K \mid \varphi\} \leq \{t : K' \mid \varphi'\} \]
Refinement Kinds

Typing. For readability we omit the store typing environment from all rules except in the location typing rule. In all other rules the store typing is just propagated unchanged.

\[
\begin{align*}
\text{(VAR)} & \quad (x:T) \in \Gamma \quad \Gamma ; S \vdash \quad \Gamma \vdash \\
\quad \Gamma \vdash x : T & \quad \Gamma \vdash S x : T \quad \Gamma \vdash \cdot : I \\
\text{(→I)} & \quad \Gamma ; S T : \text{Type} \quad \Gamma , x:T \vdash M : U \\
\text{(→-E)} & \quad \Gamma \vdash T_1 :: t : \text{Fun} \mid \text{dom}(t) \equiv T_2 \equiv \mathcal{K} \land \text{img}(t) = U \equiv \mathcal{K}' \\
\quad \Gamma ; S M : T_1 & \quad \Gamma ; S N : T_2 \\
\quad \Gamma \vdash S M N : U[T_1/t] & \quad \Gamma \vdash \lambda x:T.M : T \to U \\
\text{(∀E)} & \quad \Gamma \vdash T' :: \{f : \text{Gen}_K \mid \text{tmap}(f) \equiv U \equiv \mathcal{K}\} \\
\quad \Gamma ; S M : T' & \quad \Gamma \vdash T : \mathcal{K} \quad \Gamma \vdash U : \mathcal{K} \\
\quad \Gamma \vdash S M[T'] : U & \quad \Gamma \vdash \cdot : \emptyset \\
\text{(∀I)} & \quad \Gamma \vdash \cdot : \emptyset \\
\text{(∃l)} & \quad \Gamma \vdash S L :: \text{Nm} \quad \Gamma \vdash S :: \{t : \text{Rec} \mid L \not\equiv \text{lab}(t)\} \quad \Gamma \vdash S M : T \quad \Gamma \vdash S N : U \\
\quad \Gamma \vdash S \langle L = M \rangle @ N : \langle L : T \rangle @ U & \quad \Gamma \vdash S \text{recHeadLabel}(M) : L[U/t] \\
\text{(RECTERM)} & \quad \Gamma \vdash S \text{recHeadTerm}(M) : T[U/t] \\
\text{(RECTAIL)} & \quad \Gamma \vdash S \text{tail}(M) : T[U/t] \\
\text{(TRUE)} & \quad \Gamma \vdash \cdot : \text{Bool} \\
\text{(FALSE)} & \quad \Gamma \vdash \cdot : \text{Bool} \\
\text{(BOOL-ITE)} & \quad \Gamma \vdash S \text{false} : \text{Bool} \\
\text{(CONS)} & \quad \Gamma \vdash \cdot : \text{colHead}(M) : T \\
\text{(DEREF)} & \quad \Gamma \vdash \cdot : \text{refHead}(M) : T \\
\text{(ASSIGN)} & \quad \Gamma \vdash \cdot : \text{refHead}(M) : T \\
\text{(PROP-ITE)} & \quad \Gamma \vdash \cdot : \text{refHead}(M) : T \\
\text{(KINDCASE)} & \quad \Gamma \vdash \cdot : \mathcal{K}' \\
\text{(CONV)} & \quad \Gamma \vdash \cdot : \mathcal{K} \\
\text{(FIX)} & \quad \Gamma \vdash \cdot : \mathcal{K} \\
\end{align*}
\]

Reflexivity, Transitivity, Symmetry + Congruence+

\[
\begin{align*}
\Gamma &\vdash \mathcal{K} \equiv \mathcal{K}' & \Gamma \vdash t: \mathcal{K} \vdash \varphi \equiv \psi \\
\Gamma &\vdash \{t: \mathcal{K} \mid \varphi\} \equiv \{t: \mathcal{K}' \mid \psi\} & \Gamma \vdash \varphi \equiv \psi \\
\Gamma &\vdash \{t: \mathcal{K} \mid \varphi\} \equiv \{t: \mathcal{K}' \mid \psi\} & \Gamma \vdash \varphi \equiv \psi
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash T \equiv S &:: \{r::\text{Rec} | \text{nonEmpty}(r)\} & \Gamma \vdash T \equiv S &:: \{r::\text{Rec} | \text{nonEmpty}(r)\} \\
\Gamma \vdash \text{headLabel}(T) \equiv \text{headLabel}(S) :: \text{Nm} & & \Gamma \vdash \text{headType}(T) \equiv \text{headType}(S) :: \text{Type} \\
\Gamma \vdash T \equiv S &:: \{r::\text{Rec} | \text{nonEmpty}(r)\} & \\
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash L :: \text{Nm} & & \Gamma \vdash T :: \mathcal{K} & & \Gamma \vdash S :: \{t::\text{Rec} | L \not\in \text{lab}(t)\} \\
\Gamma \vdash \text{headLabel}((L : T)@S) \equiv L :: \text{Nm} & & \Gamma \vdash \text{headType}((L : T)@S) \equiv T :: \text{Type} \\
\Gamma \vdash L :: \text{Nm} & & \Gamma \vdash T :: \mathcal{K} & & \Gamma \vdash S :: \{t::\text{Rec} | L \not\in \text{lab}(t)\} \\
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash \text{tail}(T) \equiv \text{tail}(S) :: \text{Rec} & & \Gamma \vdash T :: \mathcal{K} & & \Gamma \vdash T :: S \equiv \{t::\mathcal{K} | \text{elim}_{\mathcal{K}}(T) \equiv T' :: \mathcal{K}'\} & & \Gamma \vdash T'(T/t) :: K'(T/t) \\
\Gamma \vdash T :: \mathcal{K} & & \Gamma \vdash T \equiv S :: \text{Col} & & \Gamma \vdash \text{colOf}(T) \equiv \text{colOf}(S) :: \text{Type} \\
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash \text{colOf}(T') &\equiv T :: \text{Type} & & \Gamma \vdash \text{colOf}(T) \equiv \text{colOf}(S) :: \text{Type} \\
\Gamma \vdash T \equiv S :: \mathcal{K} & & \Gamma \vdash \text{refOf}(T) \equiv \text{refOf}(S) :: \text{Type} \\
\Gamma \vdash \text{refOf}(\text{ref}(T)) \equiv T :: \text{Type} & & \Gamma \vdash \text{refOf}(\text{ref}(S)) :: \text{Type} \\
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash T \equiv S :: \mathcal{K} & & \Gamma \vdash \text{refOf}(\text{ref}(T)) \equiv T :: \text{Type} \\
\Gamma \vdash T \equiv S :: \mathcal{K} & & \Gamma \vdash T' \equiv S' :: \mathcal{K} \\
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash \text{dom}(T) \equiv \text{dom}(S) :: \text{Type} & & \Gamma \vdash \text{img}(T) \equiv \text{img}(S) :: \text{Type} \\
\Gamma \vdash T :: \mathcal{K} & & \Gamma \vdash T :: \mathcal{K} & & \Gamma \vdash S :: \mathcal{K}' \\
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash \text{dom}(T \rightarrow S) \equiv T :: \text{Type} & & \Gamma \vdash \text{img}(T \rightarrow S) :: S :: \text{Type} \\
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash T \equiv T' :: \mathcal{K}_0 & & \Gamma \vdash \mathcal{K} \equiv \mathcal{K}' & & \Gamma, t::\mathcal{K} \vdash S :: \mathcal{S} :: \mathcal{K}'' & & \Gamma \vdash U :: U' :: \mathcal{K}'' \\
\Gamma \vdash \text{if } T :: \mathcal{K} \text{ as } t \Rightarrow S \equiv \text{if } T' :: \mathcal{K}' \text{ as } t \Rightarrow S' \equiv U' :: \mathcal{K}'' \\
\Gamma \vdash T :: \mathcal{K} & & \Gamma, t::\mathcal{K} \vdash S :: \mathcal{K}' & & \Gamma \vdash U :: \mathcal{K}' \\
\end{align*}
\]
\[
\begin{align*}
\Gamma \vdash \text{if } T :: \mathcal{K} \text{ as } t \Rightarrow S :: \mathcal{K} & & \Gamma, t::\mathcal{K} \vdash S :: \mathcal{K}' & & \Gamma \vdash U :: \mathcal{K}' \\
\end{align*}
\]
\[
\Gamma \vDash \varphi \equiv \psi \quad \Gamma, \varphi \vDash T_1 \equiv S_1 : K \quad \Gamma, \neg \varphi \vDash T_2 \equiv S_2 : K
\]

\[
\Gamma \vDash \text{if } \varphi \text{ then } T_1 \text{ else } T_2 \equiv \text{if } \psi \text{ then } S_1 \text{ else } S_2 : K
\]

\[
\Gamma \vDash \varphi \quad \Gamma, \varphi \vdash T : K \quad \Gamma, \neg \varphi \vdash T : K \quad \Gamma \vDash \neg \varphi \quad \Gamma, \varphi \vdash T : K \quad \Gamma, \neg \varphi \vdash T : K
\]

\[
\Gamma \vDash \text{if } \varphi \text{ then } T_1 \text{ else } T_2 \equiv T_1 : K \quad \Gamma \vDash \text{if } \varphi \text{ then } T_1 \text{ else } T_2 \equiv T_2 : K
\]

\[
\Gamma \vdash \phi \quad \Gamma \vdash \psi : T \vdash \psi
\]

\[
\Gamma \vdash \mu F : (\Pi t : K_1. K_2). \lambda t : K_1. T \equiv \mu F : (\Pi t : K_1'. K_2'). \lambda t : K_1'. S \vdash \Pi t : K_1. K_2
\]

\[
\Gamma, t : K_1 + K_2 \quad \Gamma, F : \Pi t : K_1. K_2, t : K_1 \vdash T : K_2 \quad \Gamma \vdash S : K_1 \quad \text{structural}(T, F, t)
\]

\[
\Gamma \vdash (\mu F : (\Pi t : K_1. K_2). \lambda t : K_1. T) S \equiv T[S/t]\{(\mu F : (\Pi t : K_1. K_2). \lambda t : K_1. T)[F] : K_2[S/t]\}
\]

**B FULL OPERATIONAL SEMANTICS**

The type reduction relation, \( T \rightarrow T' \) is defined as a call-by-value reduction semantics on types \( T \), obtained by orienting the computational rules of type equality from left to right (thus excluding rule (E\text{-}ELIM)) and enforcing the call-by-value discipline. Recalling that type values are denoted by \( T_v, S_v \) and given by the following grammar:

\[
T_v, S_v ::= \lambda t :: K. T \mid \forall t :: K. T \mid \ell \mid \langle \rangle \mid \langle \ell : T_v \rangle @ S_v \mid T_v^* \mid \text{ref } T_v \mid T_v \rightarrow S_v \mid \bot \mid \text{Bool } \mid 1 \mid t
\]

The type reduction rules are:

\[
T \rightarrow T' \quad S \rightarrow S' \quad (\lambda t :: K. T) S \rightarrow (\lambda t :: K. T) S' \quad (\lambda t :: K. T) S_v \rightarrow T[S_v/t]\}
\]

\[
(\mu F : (\Pi t :: K. K'). \lambda t :: K. T) S_v \rightarrow T[S_v/t]\{(\mu F : (\Pi t :: K. K'). \lambda t :: K. T)[F] \}
\]

\[
L \rightarrow L' \quad T \rightarrow T' \quad S \rightarrow S' \quad (L : T) @ S \rightarrow (L' : T) @ S \quad (\ell : T_v) @ S \rightarrow (\ell : T_v') @ S \quad (\ell : T_v) @ S \rightarrow (\ell : T_v') @ S'
\]

\[
\text{headLabel}(T) \rightarrow \text{headLabel}(T') \quad \text{headType}(T) \rightarrow \text{headType}(T') \quad \text{tail}(T) \rightarrow \text{tail}(T')
\]

\[
\text{headLabel}(\langle \ell : T_v \rangle @ S_v) \rightarrow \ell \quad \text{headType}(\langle \ell : T_v \rangle @ S_v) \rightarrow T_v \quad \text{tail}(\langle \ell : T_v \rangle @ S_v) \rightarrow S_v
\]
\[
\begin{align*}
T \to T' & \quad T^* \to T'^* \\
\colOf(T) \to \colOf(T') & \quad \colOf(T_v^*) \to T_v \\
T \to T' & \quad \ref{T} \to \ref{T'} \\
\refOf(T) \to \refOf(T') & \quad \refOf(ref T_v) \to T_v \\
T \to T' & \quad S \to S' \\
(T \to S) \to (T' \to S) & \quad (T_v \to S) \to (T_v \to S') \\
\dom(T) \to \dom(T') & \quad \img(T) \to \img(T') \\
\dom(T_v \to S_v) \to T_v & \quad \img(T_v \to S_v) \to S_v \\
\Gamma \models \varphi & \quad \Gamma \models \neg \varphi & \quad T \to T' \\
\text{if } \varphi \text{ then } T \text{ else } S \to T & \quad \text{if } \varphi \text{ then } T \text{ else } S \to S \\
\text{if } T :: \mathcal{K} \text{ as } t \Rightarrow S \text{ else } U & \quad \text{if } T' :: \mathcal{K} \text{ as } t \Rightarrow S \text{ else } U \\
\Gamma + T_v :: \mathcal{K} & \quad \Gamma + T_v :: \mathcal{K}' \quad \Gamma + \mathcal{K}' \neq \mathcal{K} \\
\text{if } T_v :: \mathcal{K} \text{ as } t \Rightarrow S \text{ else } U \to S[T_v/t] & \quad \text{if } T_v :: \mathcal{K} \text{ as } t \Rightarrow S \text{ else } U \to U
\end{align*}
\]
The rules of our operational semantics are as follows:

\[
\begin{align*}
\text{R-RecConsLab} & : (H; L) \rightarrow (H'; L') \\
\text{R-RecConsL} & : (H; M) \rightarrow (H'; M') \\
\text{R-RecConsR} & : (H; M) \rightarrow (H'; M') \\
\text{R-RecHdLab} & : (H; M) \rightarrow (H'; M') \\
\text{R-RecHdLabV} & : (H; \text{recHeadLabel}(M)) \rightarrow (H'; \text{recHeadLabel}(M')) \\
\text{R-RecHdVal} & : (H; \text{recHeadTerm}(M)) \rightarrow (H'; \text{recHeadTerm}(M')) \\
\text{R-RecHdValV} & : (H; \text{recHeadTerm}(\ell)@v') \rightarrow (H; \ell) \\
\text{R-RecTail} & : (H; M) \rightarrow (H'; M') \\
\text{R-RecTailV} & : (H; \text{recTail}(M)) \rightarrow (H'; \text{recTail}(M')) \\
\text{R-Ref} & : (H; \text{ref } M) \rightarrow (H'; \text{ref } M') \\
\text{R-RefV} & : (H; \text{ref } v) \rightarrow (H[\ell \mapsto v]; \ell) \\
\text{R-Deref} & : (H; M) \rightarrow (H'; M') \\
\text{R-DerefV} & : (H; \text{deref } \ell) = v \\
\text{R-AssignL} & : (H; M := N) \rightarrow (H'; M' := N) \\
\text{R-AssignR} & : (H; M := M') \rightarrow (H'; M := M') \\
\text{R-AssignV} & : (H; \ell := v) \rightarrow (H[\ell \mapsto v]; v) \\
\text{R-PropT} & : \Gamma \models \phi \rightarrow (H; \text{if } \phi \text{ then } M \text{ else } N) \rightarrow (H; M) \\
\text{R-PropR} & : \Gamma \models \neg \phi \\
\text{R-PropF} & : (H; \text{if } \phi \text{ then } M \text{ else } N) \rightarrow (H; N) \\
\text{R-IfT} & : (H; \text{if } \text{true} \text{ then } M \text{ else } N) \rightarrow (H; M)
\end{align*}
\]

Refinement Kinds

1765 R-IfF  
1766 \langle H; \text{if } \text{false } M \text{ else } N \rangle \rightarrow \langle H; N \rangle  
1767 \langle H; \text{if } M \text{ then } N_1 \text{ else } N_2 \rangle \rightarrow \langle H'; \text{if } M' \text{ then } N_1 \text{ else } N_2 \rangle  
1768
1769 R-TAppTRed  
1770 T \rightarrow T'  
1771 \langle H; (\lambda t:K. M)[T] \rangle \rightarrow \langle H; (\lambda t:K. M)[T'] \rangle  
1772
1773 R-Fix  
1774 \langle H; \mu F:T.M \rangle \rightarrow \langle H; M[\mu F:T.M/F] \rangle  
1775 \langle H; (\lambda t:K. M)[T_u] \rangle \rightarrow \langle H; M[T_u/t] \rangle  
1776
1777 R-TAppL  
1778 \langle H; M \rangle \rightarrow \langle H'; M' \rangle  
1779 \langle H; M[T] \rangle \rightarrow \langle H'; M'[T] \rangle  
1780
1781 R-AppL  
1782 \langle H; M \rangle \rightarrow \langle H'; M' \rangle  
1783 \langle H; M N \rangle \rightarrow \langle H'; M' N \rangle  
1784
1785 R-ColConsL  
1786 \langle H; M \rangle \rightarrow \langle H'; M' \rangle  
1787 \langle H; M :: N \rangle \rightarrow \langle H'; M' :: N \rangle  
1788
1789 R-ColHd  
1790 \langle H; M \rangle \rightarrow \langle H'; M' \rangle  
1791 \langle H; \text{colHead}(M) \rangle \rightarrow \langle H'; \text{colHead}(M') \rangle  
1792
1793 R-ColTl  
1794 \langle H; M \rangle \rightarrow \langle H'; M' \rangle  
1795 \langle H; \text{colTail}(M) \rangle \rightarrow \langle H'; \text{colTail}(M') \rangle  
1796
1797 R-KindType  
1798 T \rightarrow T'  
1799 \langle H; \text{if } T :: \mathcal{K} \text{ as } t \Rightarrow M \text{ else } N \rangle \rightarrow \langle H; \text{if } T' :: \mathcal{K} \text{ as } t \Rightarrow M \text{ else } N \rangle  
1800
1801 R-KindL  
1802 \Gamma \vdash T :: \mathcal{K}  
1803 \langle H; \text{if } T :: \mathcal{K} \text{ as } t \Rightarrow M \text{ else } N \rangle \rightarrow \langle H; M[T/t] \rangle  
1804
1805 R-KindR  
1806 \Gamma \vdash T :: \mathcal{K}_0 \quad \Gamma \vdash \mathcal{K}_0 \neq \mathcal{K}  
1807 \langle H; \text{if } T :: \mathcal{K} \text{ as } t \Rightarrow M \text{ else } N \rangle \rightarrow \langle H; N \rangle  
1808
1809 C \ PROOFS  
1810 In the development below we pressupose the signature \( \Sigma \) has been checked for well-formedness.  
1811
1812 Lemma 5.1 (Substitution).

(a) If $\Gamma \vdash t :: K$ and $\Gamma, t : K, \Gamma' \vdash \mathcal{F}$ then $\Gamma, \Gamma'(T/t) \vdash \mathcal{F}(T/t)$.

(b) If $\Gamma \vdash M : T$ and $\Gamma, x : T, \Gamma' \vdash N : S$ then $\Gamma, \Gamma' \vdash N[M/x] : S$.

Proof. By induction on the derivation of the second given judgment. We show some illustrative cases.

(a)

$$
\begin{array}{l}
\text{Case:} \\
\Gamma, t : K, \Gamma' \vdash \mathcal{K}' \\
\quad \Gamma, t : K, \Gamma', s : \mathcal{K}' \vdash \varphi
\end{array}
$$

$$
\begin{array}{l}
\Gamma, \Gamma'(T/t) \vdash \mathcal{K}'(T/t) \\
\Gamma, \Gamma'(T/t), s : \mathcal{K}'(T/t) \vdash \varphi(T/t)
\end{array}
$$

by i.h.

$$
\begin{array}{l}
\Gamma, \Gamma'(T/t) \vdash \{s : \mathcal{K}'(T/t) \mid \varphi(T/t)\}
\end{array}
$$

by rule

$$
P : K_1, \ldots, K_n \in \Sigma \\
\forall i \in \{1, \ldots, n\}, \Gamma, t : K, \Gamma' \vdash T_i \vdash K_i
$$

by rule

$$
\begin{array}{l}
\Gamma, t : K, \Gamma' \vdash T_1 \equiv T_2 \vdash \mathcal{K}'
\end{array}
$$

by i.h.

$$
\begin{array}{l}
\Gamma, \Gamma'(T/t) \vdash T_1(T/t) \equiv T_2(T/t) \vdash \mathcal{K}'(T/t)
\end{array}
$$

by rule

$$
\begin{array}{l}
\Gamma, \Gamma'(T/t) \vdash x : \mathcal{K}'(T/t) \vdash \varphi(T/t)
\end{array}
$$

by i.h.

$$
\begin{array}{l}
\Gamma, \Gamma'(T/t) \vdash \varphi(T/t)(T_1(T/t)/x)
\end{array}
$$

by definition

$$
\begin{array}{l}
\Gamma, t : K, \Gamma' \vdash \varphi(T/t)(T_2(T/t)/x)
\end{array}
$$

by rule

$$
\begin{array}{l}
\Gamma, t : K, \Gamma' \vdash \mathcal{K}' \\
\Gamma, t : K, \Gamma', s : \mathcal{K}' \vdash T' \vdash \mathcal{K}
\end{array}
$$

by i.h.

$$
\begin{array}{l}
\Gamma, \Gamma'(T/t) \vdash \mathcal{K}'(T/t)
\end{array}
$$

by i.h.

$$
\begin{array}{l}
\Gamma, \Gamma'(T/t), s : \mathcal{K}'(T/t) \vdash T'(T/t) \vdash \mathcal{K}
\end{array}
$$

by rule

$$
\begin{array}{l}
\Gamma, t : K, \Gamma', \forall s : \mathcal{K}'(T/t).T'(T/t) \vdash \text{Gen}_{K'}(T/t)
\end{array}
$$

by rule

$$
\begin{array}{l}
\Gamma, t : K, \Gamma' \vdash L :: \text{Nm} \\
\quad \Gamma, t : K, \Gamma' \vdash T' :: \mathcal{K} \\
\quad \Gamma, t : K, \Gamma' \vdash S' :: \text{Rec}
\end{array}
$$

by i.h.

$$
\begin{array}{l}
\Gamma, \Gamma'(T/t) \vdash L[T(t)] :: \text{Nm}
\end{array}
$$

by i.h.

$$
\begin{array}{l}
\Gamma, \Gamma'(T/t) \vdash T'[T(t)] :: \mathcal{K}
\end{array}
$$

by i.h.

$$
\begin{array}{l}
\Gamma, \Gamma'(T/t) \vdash S'[T(t)] :: \{t : \text{Rec} \mid L[T(t)]\}
\end{array}
$$

by i.h.

$$
\begin{array}{l}
\Gamma, \Gamma'(T/t) \vdash T'(T/t) \vdash \text{Rec}
\end{array}
$$

by rule

$$
\begin{array}{l}
\Gamma, t : K, \Gamma' \vdash \{t : K' \mid \text{elim}_{K'}(t) \equiv T'' : K''\}
\end{array}
$$

by i.h.

$$
\begin{array}{l}
\Gamma, \Gamma'(T/t) \vdash T'[T(t)] :: \{t : K'[T(t)] \mid \text{elim}_{K'[T(t)]}(t) \equiv T''[T(t)] : K''[T(t)]\}
\end{array}
$$

by rule

$$
\begin{array}{l}
\Gamma, \Gamma'(T/t) \vdash \text{elim}_{K'}(T(t)) :: K''
\end{array}
$$

by i.h.

$$
\begin{array}{l}
\Gamma, \Gamma'(T/t) \vdash \{t : K'[T(t)] \mid \text{elim}_{K'[T(t)]}(t) \equiv T''[T(t)] : K''[T(t)]\}
\end{array}
$$

by rule
The remaining cases follow by similar reasoning, relying on type- and kind-preserving substitution in the language of refinements.

\[\begin{align*}
\text{Case:} & \quad \Gamma, \, t:K, \Gamma' + \varphi \quad \Gamma, \, t:K, \Gamma', \varphi + T' :: K' \quad \Gamma, \, t:K, \Gamma', \neg \varphi + S :: K' \\
\Gamma, \, t:K, \Gamma' + \text{if } \varphi \text{ then } T' \text{ else } S :: K' \\
\Gamma, \, \Gamma'[T/t] + \varphi[T/t] & \quad \text{by i.h.} \\
\Gamma, \, \Gamma'[T/t], \varphi[T/t] + T'[T/t] :: K'[T/t] & \quad \text{by i.h.} \\
\Gamma, \, \Gamma'[T/t], -\varphi[T/t] + S[T/t] :: K'[T/t] & \quad \text{by i.h.} \\
\Gamma, \, \Gamma'[T/t] + \text{if } \varphi[T/t] \text{ then } T'[T/t] \text{ else } S[T/t] :: K'[T/t] & \quad \text{by rule} \\
\Gamma, \, t:K, \Gamma' + S :: \{t : \text{Rec} \mid \ell \neq t\} & \quad \Gamma, \, t:K, \Gamma' + M : T' & \quad \Gamma, \, t:K, \Gamma' + N : S \\
\end{align*}\]

\[\begin{align*}
\text{Case:} & \quad \Gamma, \, t:K, \Gamma' + \text{recHeadLabel}(M) :: L[S/s] \\
\Gamma, \, \Gamma'[T/t] + M[T/t] :: S[T/t] & \quad \text{by i.h.} \\
\Gamma, \, \Gamma'[T/t] + S[T/t] :: \{s : \text{Rec} \mid \text{headLabel}(s) \equiv L[T/t] :: Nm\} & \quad \text{by i.h.} \\
\Gamma, \, \Gamma'[T/t] + \text{recHeadLabel}(M[T/t]) :: L[T/t][S[T/t]/s] & \quad \text{by rule} \\
\Gamma, \, t:K, \Gamma' + M :: S & \quad \Gamma, \, t:K, \Gamma' + S :: \{s : \text{Rec} \mid \text{headType}(s) \equiv K' :: K\} \\
\end{align*}\]

\[\begin{align*}
\text{Case:} & \quad \Gamma, \, t:K, \Gamma' + \text{recHeadTerm}(M) :: T'[S/s] \\
\Gamma, \, \Gamma'[T/t] + M[T/t] :: S[T/t] & \quad \text{by i.h.} \\
\Gamma, \, \Gamma'[T/t] + S[T/t] :: \{s : \text{Rec} \mid \text{headType}(s) \equiv T'[T/t] :: K'[T/t]\} & \quad \text{by i.h.} \\
\Gamma, \, \Gamma'[T/t] + \text{recHeadTerm}(M[T/t]) :: T'[T/t][S[T/t]/s] & \quad \text{by rule} \\
\end{align*}\]

\[\begin{align*}
\text{Case:} & \quad \Gamma, \, t':K, \Gamma', \, t:K_1 + K_2 & \quad \Gamma, \, t':K, \Gamma' + S :: K_1 \\
\Gamma, \, t':K, \Gamma', \, t:K_1 + K_2 & \quad \Gamma, \, t':K, \Gamma' + S :: K_1 \\
\end{align*}\]

\[\begin{align*}
\Gamma, \, t':K, \Gamma', \, t:K_1 + K_2 & \quad \Gamma, \, t':K, \Gamma' + S :: K_1 \\
\Gamma, \, t':K, \Gamma', \, t:K_1 + K_2 & \quad \Gamma, \, t':K, \Gamma' + S :: K_1 \\
\Gamma, \, t':K, \Gamma', \, t:K_1 + K_2 & \quad \Gamma, \, t':K, \Gamma' + S :: K_1 \\
\end{align*}\]
\[ \Gamma, x' : T \vdash T'[x'/x] \quad \text{alpha conversion, for fresh } x' \]

\[ \Gamma, x : T', x' : T \vdash T'[x'/x] \quad \text{by weakening} \]

\[ \Gamma, x' : T \vdash T'[x'/x][x'/x'] \quad \text{by substitution} \]

\[ \Gamma, x : T' \vdash T' \quad \text{by definition} \]

Statement (b) follows by the same reasoning.

\begin{enumerate}
\item \textbf{Lemma 5.5 (Functionality of Kinding and Refinements).}
\item Assume \( \Gamma \vdash T \equiv S \vdash K, \Gamma \vdash T \vdash K \) and \( \Gamma \vdash S \vdash K \).
\item (a) If \( \Gamma, t : K, \Gamma' \vdash T' :: K' \) then \( \Gamma, \Gamma'[T/t] \vdash T'[T/t] \equiv T'[S/t] :: K'[T/t] \).
\item (b) If \( \Gamma, t : K, \Gamma' \vdash K' \) then \( \Gamma, \Gamma'[T/t] \vdash K[T/t] \equiv K[S/t] \).
\item (c) If \( \Gamma, t : K, \Gamma' \vdash \phi \) then \( \Gamma, \Gamma'[T/t] \vdash \phi[T/t] \equiv \phi[S/t] \).
\end{enumerate}

\textbf{Proof.} By induction on the given kinding/kind well-formedness and entailment judgments.

Functionality follows by substitution and the congruence rules of definitional equality.

\[ \Gamma, t : K, \Gamma' \vdash \mathcal{K}' \quad \Gamma, t : K, \Gamma', t' \vdash \mathcal{K}' \vdash \phi \]

\textbf{Case:} \[ \Gamma, t : K, \Gamma' \vdash \{ t' : \mathcal{K}' \vdash \phi \} \]

\[ \Gamma, \Gamma'[T/t] \vdash \mathcal{K}'[T/t] \equiv \mathcal{K}'[S/t] \quad \text{by i.h.} \]

\[ \Gamma, \Gamma'[T/t], t' : \mathcal{K}'[T/t] \vdash \phi[T/t] \equiv \phi[S/t] \quad \text{by i.h.} \]

\[ \Gamma, \Gamma'[T/t] \vdash \{ t' : \mathcal{K}'[T/t] \vdash \phi[T/t] \} \equiv \{ t' : \mathcal{K}'[S/t] \vdash \phi[S/t] \} \quad \text{by kind ref. equality} \]

\[ \Gamma, t : K, \Gamma' \vdash T' \equiv S' :: \mathcal{K}' \quad \Gamma, t : K, t', x : \mathcal{K}' \vdash \phi \quad \Gamma, t : K, \Gamma' \vdash \phi[T'/x] \]

\textbf{Case:} \[ \Gamma, t : K, \Gamma' \vdash \phi[S'/x] \]

\[ \Gamma \vdash T \equiv S :: K, \Gamma \vdash T \vdash K \quad \text{by assumption} \]

\[ \Gamma, \Gamma'[T/t] \vdash \phi[S'/x][T/t] \quad \text{by substitution} \]

\[ \Gamma, \Gamma'[S/t] \vdash \phi[S'/x][S/t] \quad \text{by substitution} \]

\[ \Gamma, \Gamma'[T/t] \vdash \phi[S'/x][S/t] \quad \text{by context conversion} \]

\[ \Gamma, \Gamma'[T/t] \vdash \phi[S'/x][T/t] \equiv \phi[S'/x][S/t] \quad \text{by weakening and } \supset \text{I} \]

\[ \Gamma, \Gamma'[T/t] \vdash \phi[S'/x][S/t] \quad \text{by weakening and } \supset \text{I} \]

\[ \Gamma, \Gamma'[T/t] \vdash \phi[S'/x][S/t] \quad \text{by definition of refinement equivalence} \]

\[ \Gamma, t : K, \Gamma' \vdash K \quad \Gamma, t : K, \Gamma, s : \mathcal{K} \vdash T' :: \mathcal{K} \]

\textbf{Case:} \[ \Gamma, t : K, \Gamma \vdash \forall s : \mathcal{K}, T' :: \text{Gen}_K \]

\[ \Gamma, \Gamma'[T/t] \vdash K'[T/t] \equiv K'[S/t] \quad \text{by i.h.} \]

\[ \Gamma, \Gamma'[T/t], t' : K'[T/t] \vdash T'[T/t] \equiv T'[S/t] :: \mathcal{K} \quad \text{by i.h.} \]

\[ \Gamma, \Gamma'[T/t] \vdash \forall s : K'[T/t], T'[T/t] \equiv \forall s : K'[S/t], T'[S/t] :: \text{Gen}_K'[T/t] \quad \text{by } \forall \text{ Eq.} \]

\[ \Gamma, t : K, \Gamma' \vdash L :: \text{Nm} \quad \Gamma, t : K, \Gamma' \vdash T' :: \mathcal{K} \quad \Gamma, t : K, \Gamma' \vdash S' :: \{ t : \text{Rec} \mid L#t \} \]

\textbf{Case:} \[ \Gamma, t : K, \Gamma' \vdash \langle L : T' \rangle \vdash S' :: \text{Rec} \]

\[ \Gamma, \Gamma'[T/t] \vdash L[T/t] \equiv L[S/t] :: \text{Nm} \quad \text{by i.h.} \]

\[ \Gamma, \Gamma'[T/t] \vdash T'[T/t] \equiv T'[S/t] :: \mathcal{K} \quad \text{by i.h.} \]

\[ \Gamma, \Gamma'[T/t] \vdash S'[T/t] \equiv S'[S/t] :: \{ t : \text{Rec} \mid L[T/t]#t \} \quad \text{by i.h.} \]

\[ \Gamma, \Gamma'[T/t] \vdash \langle L[T/t] : T'[T/t]\rangle \equiv \langle L[S/t] : T'[S/t] \rangle :: \text{Rec} \quad \text{by Rec Eq.} \]

\[ \Gamma, t : K, \Gamma' \vdash T' :: \{ s : \mathcal{K}' \mid \text{elim}_{\mathcal{K}'}(s) \equiv T'' :: \mathcal{K}'' \} \]

\textbf{Case:} \[ \Gamma, t : K, \Gamma' \vdash \text{elim}_{\mathcal{K}'}(T') :: \mathcal{K}'' \]

\[ \Gamma, \Gamma'[T/t] \vdash T'[T/t] \equiv T'[S/t] :: \{ s : \mathcal{K}'[T/t] \mid \text{elim}_{\mathcal{K}'}(s) \equiv T''[T/t] :: K''[T/t] \} \quad \text{by i.h.} \]

\[ \Gamma, \Gamma'[T/t] \vdash \text{elim}_{\mathcal{K}'}(T'[T/t]) \equiv T''[T/t] :: K''[T/t] \quad \text{by } \text{elim}_{\mathcal{K}} \text{ eq. rule} \]
\(\Gamma, \Gamma'[T/t] \vdash \text{elim}_{K'}(T'(S/t)) \equiv T''(T/t) :: K''(T/t)\) by symmetry and \text{elim}_K\text{ eq. rule}

\(\Gamma, \Gamma'[T/t] \vdash \text{elim}_{K'}(T'(T/t)) \equiv \text{elim}_{K'}(T'(S/t)) :: K''(T/t)\) by sym. and transitivity

**Case:**
\(\Gamma, t : K, \Gamma' + \varphi, \Gamma, t : K, \Gamma', \varphi + T' :: K'\)
\(\Gamma, t : K, \Gamma', \neg \varphi + S' :: K'\)

\(\Gamma, \Gamma'[T/t], \varphi(T/t) + T'(T/t) \equiv T'(S/t) :: K'(T/t)\) by i.h.

\(\Gamma, \Gamma'[T/t], \neg \varphi(T/t) + S'(T/t) \equiv S'(S/t) :: K'(T/t)\) by i.h.

\(\Gamma, t : K, \Gamma', \varphi \models \varphi(T/t)\) tautology

\(\Gamma, \Gamma'[T/t], \varphi(T/t) \models \varphi(T/t)\) by substitution

\(\Gamma, \Gamma'[T/t], \varphi(S/t) \models \varphi(T/t)\) by ctxt. conversion

\(\Gamma, \Gamma'[T/t], \varphi(S/t) \models \varphi(S/t)\) by \(\supset\)I

\(\Gamma, \Gamma'[T/t] \models \varphi(T/t) \supset \varphi(S/t)\) by substitution

\(\Gamma, \Gamma'[T/t] \models \varphi(T/t) \supset \varphi(S/t)\) by ctxt. conversion

\(\Gamma, \Gamma'[T/t] \models \varphi(T/t) \equiv \varphi(S/t)\) by \(\supset\)I

\(\Gamma, \Gamma'[T/t] \models \text{if } \varphi(T/t) \text{ then } T'(T/t) \text{ else } S'(T/t) \equiv\)

\(\text{if } \varphi(S/t) \text{ then } T'(S/t) \text{ else } S'(S/t) :: K'(T/t)\) by rule

\(\Gamma, t : K, \Gamma' \models \varphi(T'/s)\)
\(\Gamma, t : K, \Gamma' + T' :: \mathcal{K}'\)

**Case:**
\(\Gamma, t : K, \Gamma' + T' :: \{s : \mathcal{K}' \mid \varphi\}\)

\(\Gamma, \Gamma'[T/t] \models \varphi(T'/s)[T/t] \equiv \varphi(T'/s)[S/t]\) by i.h.

\(\Gamma, \Gamma'[T/t] \models T'(T/t) \equiv T'(S/t) :: \mathcal{K}'(T/t)\) by i.h.

\(\Gamma, \Gamma'[T/t] \models T'(T/t) \equiv T'(S/t) :: \{s : \mathcal{K}'(T/t) \mid \varphi(T/t)\}\) by Eq Conversion

**Theorem 5.3 (Validity for Equality).**

(a) If \(\Gamma \vdash K \equiv K'\) then \(\Gamma \vdash K\) and \(\Gamma \vdash K'\).

(b) If \(\Gamma \models T \equiv T' :: K\) then \(\Gamma \vdash K\), \(\Gamma \vdash T :: K\) and \(\Gamma \vdash T' :: K\).

(c) If \(\Gamma \vdash \varphi \equiv \psi\) then \(\Gamma \vdash \varphi\) and \(\Gamma \vdash \psi\)

**Proof.** By induction on the given derivation.

**Case:**
\(\Gamma + \mathcal{K} \equiv \mathcal{K}'\)
\(\Gamma, t : \mathcal{K} + \varphi \equiv \psi\)

\(\Gamma + \{t : \mathcal{K} \mid \varphi\} \equiv \{t : \mathcal{K}' \mid \psi\}\) by i.h.

\(\Gamma \vdash \mathcal{K}\) and \(\Gamma \vdash \mathcal{K}'\) by i.h.

\(\Gamma, t : \mathcal{K} + \varphi\) and \(\Gamma, t : \mathcal{K} + \psi\) by i.h.

\(\Gamma \vdash \{t : \mathcal{K} \mid \varphi\}\) by refinement kind w.f.

\(\Gamma \vdash \{t : \mathcal{K}' \mid \psi\}\) by refinement kind w.f.

**Case:**
\(\Gamma \models T \equiv S :: \text{Gen}_K\)
\(\Gamma \models T_1 \equiv S_1 :: \text{Gen}_K\)

\(\Gamma \models \text{tmapping}(T_1) T_2 \equiv \text{tmapping}(S_1) S_2 :: \text{Type}\)

\(\Gamma \vdash t_1 :: \text{Gen}_K\) and \(\Gamma \vdash t_1 :: \text{Gen}_K\) by i.h.

\(\Gamma \vdash T_2 :: K\) and \(\Gamma \vdash S_2 :: K\) by i.h.

\(\Gamma \vdash \text{tmapping}(T_1) T_2 :: \text{Type}\) by kinding

\(\Gamma \vdash \text{tmapping}(S_1) S_2 :: \text{Type}\) by kinding

**Case:**
\(\Gamma \vdash t : K + t :: \mathcal{K}\)
\(\Gamma \vdash S :: K\)

\(\Gamma \models \text{tmapping}(\forall t : K.T) S \equiv T[S/t] :: \text{Type}\)
If \( \vdash t : T \vdash \) \( \mathcal{K} \)

Proof.

(a) Proof.

Lemma 5.6 (Functionality of Equality).

\[
\frac{\Gamma \mid \vdash t : \mathcal{K} \quad \text{elim}_\mathcal{K}(t) \equiv T' : K'}{\Gamma \mid \vdash T'(T/t) : K'(T/t)}
\]

Case:

\[
\Gamma \mid \vdash t : \mathcal{K} \quad \text{elim}_\mathcal{K}(t) \equiv T' : K'
\]

By i.h.

\[
\Gamma \mid \vdash T'(T/t) : K'(T/t)
\]

By subkinding

\[
\Gamma \mid \vdash T :: \mathcal{K}
\]

By assumption

\[
\Gamma \mid \vdash \text{colOf}(T^*) \equiv T :: \text{Type}
\]

By subkinding

Remaining cases follow by a similar reasoning.

□

**Corollary 5.4 (Kind Preservation).** If \( \Gamma \vdash T :: K \) and \( T \rightarrow T' \) then \( \Gamma \vdash T' :: K \).

Proof. Immediate from equality validity since \( T \rightarrow S \) implies \( T \equiv S \).

□

**Lemma 5.6 (Functionality of Equality).** Assume \( \Gamma \vdash T_0 \equiv S_0 :: K \):

(a) If \( \Gamma, t : K \vdash T \equiv S :: K' \) then \( \Gamma \vdash T_0(t) \equiv S_0(t) :: K'(T_0/t) \).

(b) If \( \Gamma, t : K \vdash K_1 \equiv K_2 \) then \( \Gamma \vdash T_0(t) \equiv K_2(t) \).

(c) If \( \Gamma, t : K \vdash \varphi \equiv \psi \) then \( \Gamma \vdash \varphi(T_0)/t) \equiv \psi(S_0/t) \).

Proof.

(a)

\[
\frac{\Gamma, t : K \vdash T \equiv S :: K'}{\Gamma \vdash T_0 \equiv S_0 :: K'} \quad \text{assumption}
\]

\[
\Gamma \vdash T_0 \equiv S_0 :: K \quad \text{assumption}
\]

\[
\Gamma \vdash T_0 :: K \quad \text{and} \quad \Gamma \vdash S_0 :: K 
\]

By eq. validity

\[
\Gamma, t : K \vdash T :: K' \quad \text{and} \quad \Gamma, t : K \vdash S :: K' 
\]

By eq. validity

\[
\frac{\Gamma \vdash S(T_0/t) \equiv S(T_0/t) :: K'(T_0/t)}{\Gamma \vdash T(T_0/t) \equiv S(T_0/t) :: K'(T_0/t)} \quad \text{by substitution}
\]

\[
\frac{\Gamma \vdash S(T_0/t) \equiv S(S_0/t) :: K'(T_0/t)}{\Gamma \vdash T(T_0/t) \equiv S(S_0/t) :: K'(T_0/t)} \quad \text{by functionality}
\]

\[
\Gamma \vdash T(T_0/t) \equiv S(T_0/t) :: K'(T_0/t)
\]

By transitivity

(b)

\[
\frac{\Gamma \vdash T_0 \equiv S_0 :: K}{\Gamma, t : K \vdash K_1 \equiv K_2} \quad \text{assumption}
\]

\[
\Gamma, t : K \vdash K_1 \equiv K_2 \quad \text{assumption}
\]

\[
\Gamma \vdash T_0 :: K \quad \text{and} \quad \Gamma \vdash S_0 :: K 
\]

By eq. validity

\[
\Gamma, t : K \vdash K_1 \quad \text{and} \quad \Gamma, t : K \vdash K_2 
\]

By eq. validity

\[
\frac{\Gamma \vdash K_1(T_0/t) \equiv K_2(T_0/t)}{\Gamma \vdash K_2(T_0/t) \equiv K_2(S_0/t)} \quad \text{by substitution}
\]

\[
\frac{\Gamma \vdash K_2(T_0/t) \equiv K_2(S_0/t)}{\Gamma \vdash K_2(T_0/t) \equiv K_2(S_0/t)} \quad \text{by functionality}
\]

\[
\Gamma \vdash K_1(T_0/t) \equiv K_2(S_0/t)
\]

By transitivity

(c)
Refinement Kinds

\[ \Gamma \vdash T_0 \equiv S_0 :: K \] assumption
\[ \Gamma, t : K \vdash \varphi \equiv \psi \] assumption
\[ \Gamma \vdash T_0 :: K \text{ and } \Gamma \vdash S_0 :: K \] by eq. validity
\[ \Gamma, t : K \vdash \varphi \text{ and } \Gamma, t : K \vdash \psi \] by eq. validity
\[ \Gamma \vdash \varphi[T_0/t] \equiv \psi[T_0/t] \] by substitution
\[ \Gamma \vdash \psi[T_0/t] \equiv \psi[S_0/t] \] by functionality
\[ \Gamma \vdash \varphi[T_0/t] \equiv \psi[S_0/t] \] by transitivity

\[ \Box \]

**Theorem 5.7 (Validity).**

(a) If \( \Gamma \vdash K \) then \( \Gamma \vdash \).
(b) If \( \Gamma \vdash T :: K \) then \( \Gamma \vdash K \).
(c) If \( \Gamma \vdash M : T \) then \( \Gamma \vdash T :: \text{Type} \).

**Proof.** Straightforward induction on the given derivation.

**Lemma C.1 (Injectivity).** If \( \Gamma \vdash \Pi t : K_1.K_2 \equiv \Pi t : K_1'.K_2' \) then \( \Gamma \vdash K_1 \equiv K_1' \) and \( \Gamma, t : K_1 \vdash K_2 \equiv K_2' \).

**Proof.** Straightforward induction on the given kind equality derivation.

**Lemma C.2 (Injectivity via Subkinding).** If \( \Gamma \vdash \Pi t : K_1.K_2 \leq K \) then \( \Gamma \vdash K \equiv \Pi t : K_1'.K_2' \) with \( \Gamma \vdash K_1 \equiv K_1' \) and \( \Gamma, t : K_1 \vdash K_2 \equiv K_2' \).

**Lemma C.3 (Inversion).**

(a) If \( \Gamma \vdash \lambda t : K. T :: K' \) then there is \( K_1 \) and \( K_2 \) such that \( \Gamma \vdash K' \equiv \Pi t : K_1.K_2, \Gamma \vdash K \equiv K_1 \) and \( \Gamma, t : K_1 \vdash T :: K_2 \).
(b) If \( \Gamma \vdash T S :: K \) then \( \Gamma \vdash T :: \Pi t : K_0.K_1, \Gamma \vdash S :: K_0 \) and \( \Gamma \vdash K \equiv K_1\{S/t\} \).
(c) If \( \Gamma \vdash \lambda x : T. M :: T' \) then there is \( T_1 \) and \( T_2 \) such that \( \Gamma \vdash T' \equiv T_1 \to T_2 :: \text{Type} \) and \( \Gamma, x : T_1 \vdash M :: T_2 \).
(d) If \( \Gamma \vdash (L : T)@S :: K \) then \( \Gamma \vdash L :: \text{Nm} \), \( \Gamma \vdash T :: \text{Type} \), \( \Gamma \vdash S :: \{t : \text{Rec} \mid L \neq t\} \) and \( \Gamma \vdash K \equiv \text{Rec} \).
(e) If \( \Gamma \vdash (L : M)@N : T \) then there is \( L', T_1, T_2 \) such that \( \Gamma \vdash L \equiv L' :: \text{Nm} \), \( \Gamma \vdash \langle L' : T_1 \rangle@T_2 :: \text{Rec} \), \( \Gamma \vdash T \equiv \langle L' : T_1 \rangle@T_2 \), \( \Gamma \vdash M :: T_1 \) and \( \Gamma \vdash N :: T_2 \).
(f) If \( \Gamma \vdash T :: \{t : K \mid \varphi\} \) then \( \Gamma \vdash \varphi[T/t] \), \( \Gamma \vdash T :: K \) and \( \Gamma, t : K \vdash \varphi \).
(g) If \( \Gamma \vdash \text{elim}_K(T) :: K \) then \( \Gamma \vdash T :: \{t : \mathcal{K} \mid \text{elim}_K(t) \equiv T' :: K'\} \) and \( \Gamma \vdash T'(T/t) :: K'(T/t) \) and \( \Gamma \vdash K \equiv K'(T/t) \).
(h) If \( \Gamma \vdash \varphi \text{ then } M \) else \( N : T \) then \( \Gamma \vdash T \equiv \text{if } \varphi \text{ then } T_1 \text{ else } T_2 :: K \) with \( \Gamma, \varphi \equiv M :: T_1 \) and \( \Gamma, \neg \varphi \equiv N :: T_2 \).
(i) If \( \Gamma \vdash \varphi \text{ then } T \) else \( S :: K \) then \( \Gamma \vdash \varphi, \varphi \equiv T :: K \) and \( \Gamma, \neg \varphi \equiv S :: K \).
(j) If \( \Gamma \vdash T \to S :: K \) then \( \Gamma \equiv \text{Fun} \), \( \Gamma \vdash T :: \mathcal{K} \) and \( \Gamma \vdash S :: \mathcal{K}' \), for some \( \mathcal{K}, \mathcal{K}' \).
(k) If \( \Gamma \vdash M :: N :: T \) then \( \Gamma \vdash T :: S :: \{t : \text{Col} \mid \text{colOf}(t) \equiv T' :: \mathcal{K}\} \), \( \Gamma \vdash N :: S \) and \( \Gamma \vdash M :: T'(S/t) \), for some \( T', \mathcal{K}, S, T' \).
(l) If \( \Gamma \vdash T^* :: K \) then \( \Gamma \vdash \text{Col} \) and \( \Gamma \vdash T' :: \mathcal{K}, \) for some \( \mathcal{K} \).
(m) If \( \Gamma \vdash \text{if } T' :: K \text{ as } t \Rightarrow M \text{ else } N : T \) then \( \Gamma \vdash T' :: \mathcal{K}, \Gamma, \Gamma, t : K :: M \) and \( \Gamma \vdash N :: S \), with \( \Gamma \vdash T :: S :: \mathcal{K} \), for some \( \mathcal{K}, \mathcal{K}', S \).
(n) If \( \Gamma \vdash \text{if } T' :: K \text{ as } t \Rightarrow S \text{ else } S' :: K' \) then \( \Gamma \vdash T' :: \mathcal{K}, \Gamma, \Gamma, t : K :: S :: K'' \), \( \Gamma \vdash \text{vdash} S' :: K'' \) and \( \Gamma \vdash K \equiv K'' \), for some \( \mathcal{K}, \mathcal{K}' \).
(o) If \( \Gamma \vdash \mu F.T.M : T \) then \( \Gamma, F :: T :: M : T \) and \( \text{structural}(F, M) \).
(p) If \( \Gamma \vdash \mu F :: (\Pi t : K_1.K_2).\lambda t : K_1. T' :: K \) then \( \Gamma, F : \Pi t : K_1.K_2, t : K_1 \vdash T' :: K_2 \), structural \( (T', F, t) \) and \( \Gamma \vdash K \equiv \Pi t : K_1.K_2 \).

(q) If \( \Gamma \vdash \text{recHeadLabel}(M) : T \) then \( \Gamma \vdash T \equiv L[S/t] :: \text{Nm}, \Gamma \vdash M : S \) and \( \Gamma \vdash S :: \{ t : \text{Rec} | \) 
headLabel(t) \equiv L :: \text{Nm} \)

(r) If \( \Gamma \vdash \text{recHeadTerm}(M) : T \) then \( \Gamma \vdash T \equiv T'[S/t] :: \mathcal{K}[S/t], \Gamma \vdash M : S \) and \( \Gamma \vdash S :: \{ t : \text{Rec} | \) 
headType(t) \equiv T' :: \mathcal{K} \)

(s) If \( \Gamma \vdash \text{tail}(M) : T \) then \( \Gamma \vdash T \equiv T'[S/t] :: \mathcal{K}[S/t], \Gamma \vdash M : S \) and \( \Gamma \vdash S :: \{ t : \text{Rec} | \) 
tail(t) \equiv T' :: \mathcal{K} \).

(t) If \( \Gamma \vdash \text{colHead}(M) : T \) then \( \Gamma \vdash M : T^* \)

(u) If \( \Gamma \vdash \text{colTail}(M) : T \) then \( \Gamma \vdash T \equiv T'[C/t] :: \mathcal{K}, \Gamma \vdash M : T_C \) and \( \Gamma \vdash C :: \{ t : \text{Col} | \) 
colOf(t) \equiv T' :: \mathcal{K} \).

(v) If \( \Gamma \vdash \text{ref}(M : T \) then \( \Gamma \vdash T \equiv T' \) and \( \Gamma \vdash M : T' \)

(w) If \( \Gamma \vdash !M : T \) then \( \Gamma \vdash T \equiv T'[S/t] :: \mathcal{K}, \Gamma \vdash M : S, \Gamma \vdash N : T' \) and \( \Gamma \vdash S :: \{ t : \text{Ref} | \) 
refOf(t) \equiv T' :: \mathcal{K} \).

(x) If \( \Gamma \vdash M := N : T \) then \( \Gamma \vdash T \equiv T'[S/t] :: \mathcal{K}, \Gamma \vdash M : S, \Gamma \vdash N : T, \Gamma \vdash S :: \{ t : \text{Ref} | \) 
refOf(t) \equiv T' :: \mathcal{K} \).

(y) If \( \Gamma \vdash M : N : T \) then \( \Gamma \vdash M : T_1, \Gamma \vdash N : T_2, \Gamma \vdash t : \text{Fundef}(t) \equiv T_2 :: \mathcal{K} \land \text{img}(t) = U :: \mathcal{K}' \) 
and \( \Gamma \vdash T = U(T_1/t) :: \mathcal{K}'[T_1/t] \).

(z) If \( \Gamma \vdash M[T] : S \) then \( \Gamma \vdash M : T', \Gamma \vdash T : K, \Gamma \vdash U :: \mathcal{K}, \Gamma \vdash T' :: \{ f : \text{GenK} | \) 
tmap(f)(T) \equiv U :: \mathcal{K}' \) and \( \Gamma \vdash S :: U :: \mathcal{K} \).

**Proof.** By induction on the structure of the given typing or kinding derivation, using validity.

(a) \[ \Gamma \vdash \lambda t : K.T :: K', \Gamma \vdash K' \leq K' \]

Case:

\[ \Gamma \vdash \lambda t : K.T :: K' \]

\( \Gamma \vdash K'' = \Pi t : K'_1.K'_2, \Gamma \vdash K = K'_1 \) and \( \Gamma, t : K'_1 + T :: K'_2 \)

by i.h.

\( \Gamma \vdash K'' \leq \Pi t : K_1.K_2, \Gamma \vdash t : K'_1 \leq K_1 \) and \( \Gamma, t : K'_1 + K'_2 \leq K_2 \)

by inversion

\( \Gamma, t : K_1 + T :: K'_2 \)

by contextual conversion

\( \Gamma, t : K_1 + T :: K_2 \)

by conversion

\( \Gamma \vdash K \equiv K_1 \)

by transitivity

Other cases follow by similar reasoning (or are immediate).

\[ \square \]

Below we do not list the (very) extensive list of all inversions. They follow the same pattern of the kinding inversion principle.

**Lemma C.4 (Equality Inversion).**

(1) If \( \Gamma \vdash T \equiv \lambda t : K_1.T_2 :: K' \) then \( \Gamma \vdash T \equiv \lambda t : K_0.T_2 :: \Pi t : K_0.K' \) with \( \Gamma \vdash K_0 \equiv K_1 \) and 
\( \Gamma, t : K_0 \vdash T_2 \equiv T_2' :: K'', \) for some \( K'' \).

(2) If \( \Gamma \vdash T \equiv T_0.S_0 :: K \) then \( \Gamma \vdash T \equiv T_1.S_1 :: K \) with \( \Gamma \vdash T_1 \equiv T_0 :: \Pi t : K_1.K_0.S_1 \equiv S_0 :: K_1 \) and 
\( K = K_0[S_1/t] \).

(3) If \( \Gamma \vdash \langle L : T \rangle @S :: K \) then \( \Gamma \vdash \langle L' : T' \rangle @S' :: K \) with \( \Gamma \vdash L \equiv L' :: \text{Nm}, \Gamma \vdash T' \equiv T : \mathcal{K}, \Gamma \vdash S' \equiv S :: \{ t : \text{Rec} | L \neq t \} \) and 
\( K = \text{Rec} \).

(4) If \( \Gamma \vdash K \equiv \{ t : \mathcal{K} \} \psi \) then \( \Gamma \vdash K \equiv \{ t : \mathcal{K}' \} \psi \) with \( \Gamma \vdash \mathcal{K} \equiv \mathcal{K}' \Gamma \equiv \psi \).

(5) If \( \Gamma \vdash T \equiv \text{elim}_K(S) :: K \) then \( \Gamma \vdash T \equiv \text{elim}_K(S') :: K \) with \( \Gamma \vdash S \equiv S' :: \{ t : \mathcal{K} | \) 
\text{elim}_K(t) \equiv T' :: K' \} \), \( \Gamma \vdash \mathcal{K} \equiv \mathcal{K}' \), \( \Gamma, T' :: \mathcal{K}'[S/t] \) and 
\( K = K'[S/t] \).

**Proof.** By induction on the given equality derivations, relying on validity, reflexivity, substitution, context conversion and inversion. We show two illustrative cases.
Case: Transitivity rule

\[ \Gamma \vdash T \equiv S' :: K \text{ and } \Gamma \vdash S' \equiv \text{elim}_K (S) :: K \]  
assumption

\[ \Gamma \vdash S' \equiv \text{elim}_K (S'') :: K \text{ with } \Gamma \vdash S \equiv S'' :: \{ t : K \mid \text{elim}_K (t) \equiv T' :: K' \}, \]

\[ \Gamma \vdash K' \equiv K, \Gamma \vdash T' :: K'(S/t) \text{ and } K = K'(S/t) \]
by i.h.

\[ \Gamma \vdash T \equiv \text{elim}_K (S'') :: K \]
by transitivity

\[ \Gamma, t : K_0 + T_1 :: K' \quad \Gamma + T_2 :: K_0 \]

Case:

\[ \Gamma \vdash (\lambda t : K_0 . T_1) T_2 \equiv T_1 [T_2 / t] :: K'[T_2 / t] \]

Subcase 1: \( T_1 = I, T_2 = \text{elim}_K (S) \)

\[ \Gamma_0 = K' = K \]
assumption

\[ \Gamma \vdash \text{elim}_K (S) :: K \]
assumption

\[ \Gamma + S :: \{ t : K \mid \text{elim}_K (t) \equiv T' :: K' \} \text{ and } \Gamma + T'(S/t) :: K'(S/t) \text{ with } K = K'(S/t) \]
by inversion

\[ \Gamma \vdash S :: \{ t : K \mid \text{elim}_K (t) \equiv T' :: K' \}
by reflexivity

\[ \Gamma \vdash \text{elim}_K (S)' :: K' \]
by substitution

\[ \Gamma \vdash S'(T_2 / t) :: \{ t : K \mid \text{elim}_K (t) \equiv T' :: K' \}
by reflexivity

\[ \Gamma + \text{elim}_K'(T_2 / t) (S'(T_2 / t)) = \text{elim}_K (S) 
by reflexivity

\[ \square \]

Lemma C.5 (Subkingding Inversion).

1. If \( \Gamma \vdash K \leq K' \text{ then } \Gamma + \vdash K \equiv K' \text{ or } \Gamma + \vdash K' \equiv Type. \)

2. If \( \Gamma + \vdash K \leq \{ t : K' \mid \phi \} \text{ with } \Gamma + \vdash K \equiv \text{elim}_K \leq K' \text{ and } \Gamma \vdash \psi \supset \phi. \)

3. If \( \Gamma \vdash \{ t : K' \mid \phi \} \leq K \text{ then } \Gamma + \vdash K \leq K \text{ and } \Gamma, t : K' \vdash \phi. \)

Proof. By induction on the given derivation, using equality inversion. \( \square \)

Lemma C.6. If \( \Gamma \vdash T :: K, \Gamma + T :: K' \text{ and } \Gamma + S :: K' \text{ and } \Gamma + K' \leq K \text{ then } \Gamma + T :: S :: K'. \)

Proof. By induction on the given equation derivation. \( \square \)

Theorem 5.8 (Unicity of Types and Kinds).

1. If \( \Gamma + M :: T \text{ and } \Gamma + M :: S \text{ then } \Gamma \vdash T :: S :: K \text{ and } \Gamma + T :: \leq Type. \)

2. If \( \Gamma + T :: K \text{ and } \Gamma + T :: K' \text{ then } \Gamma + K :: K' \text{ or } \Gamma + K' :: \leq K. \)

Proof. By induction on the structure of the given type/term.

Case: \( M \) is \( \{ \ell = M' \} @ N' \)

\[ \Gamma \vdash (\ell = M') @ N' :: T \text{ and } \Gamma + \{ \ell = M' \} @ N' :: S \]
assumption

\[ \Gamma + M' :: T_1, \Gamma + N' :: T_2, \Gamma + \ell :: L' :: Nm, \Gamma + \{ L' = T_1 \} @ T_2 :: \text{Rec} \]
and \( \Gamma \vdash T :: L' = T_1 @ T_2 :: \text{Rec} \)

\[ \Gamma + M' :: S_1, \Gamma + N' :: S_2, \Gamma + \ell :: L'' :: Nm, \Gamma + \{ L'' = S_1 \} @ S_2 :: \text{Rec} \]
and \( \Gamma \vdash S :: L'' = S_1 @ S_2 :: \text{Rec} \)

\[ \Gamma \vdash T_1 :: S_1 :: K_1 \text{ and } \Gamma + K_1 :: \leq Type \]
by i.h.

\[ \Gamma \vdash T_1 :: S_1 :: Type \]
by conversion
\begin{align*}
\Gamma \vdash T_2 \equiv S_2 :: K_2 \text{ and } \Gamma \vdash K_2 \leq \text{Type} & \quad \text{by i.h.} \\
\Gamma \vdash T_3 :: \text{Rec and } \Gamma \vdash T_2 :: \text{Rec} & \quad \text{by inversion and conversion} \\
\Gamma \vdash T_2 \equiv S_2 :: \text{Rec} & \quad \text{by Lemma C.6} \\
\text{Case: } T \equiv (L = S_1)@S_2 & \\
\Gamma \vdash (L = S_1)@S_2 :: K \text{ and } \Gamma \vdash (L = S_1)@S_2 :: K' & \quad \text{assumption} \\
\Gamma \vdash L :: \text{Nm, } \Gamma \vdash S_1 :: \text{Type, } \Gamma \vdash S_2 :: \{t: \text{Rec} \mid K \not \equiv t\} \text{ and } \Gamma \vdash K \equiv \text{Rec} & \quad \text{by inversion} \\
\Gamma \vdash L :: \text{Nm, } \Gamma \vdash S_1 :: \text{Type, } \Gamma \vdash S_2 :: \{t: \text{Rec} \mid K \not \equiv t\} \text{ and } \Gamma \vdash K' \equiv \text{Rec} & \quad \text{by inversion} \\
\Gamma \vdash \text{Rec} \leq \text{Rec} & \quad \text{by reflexivity} \\
\text{Case: } M \text{ is if } \varphi \text{ then } M' \text{ else } N' & \\
\Gamma \vdash \text{if } \varphi \text{ then } M' \text{ else } N' :: T \text{ and } \Gamma \vdash \text{if } \varphi \text{ then } M' \text{ else } N' :: S & \quad \text{assumption} \\
\Gamma, \varphi \vdash M' :: T_1, \Gamma, \neg \varphi \vdash N' :: T_2 \text{ and } \Gamma \vdash T \equiv \text{if } \varphi \text{ then } T_1 \text{ else } T_2 & \quad \text{by inversion} \\
\Gamma, \varphi \vdash M' :: S_1, \Gamma, \neg \varphi \vdash N' :: S_2 \text{ and } \Gamma \vdash S \equiv \text{if } \varphi \text{ then } S_1 \text{ else } S_2 & \quad \text{by inversion} \\
\Gamma, \varphi \vdash T_1 \equiv S_1 :: K_1 \text{ with } \Gamma \vdash K_1 \leq \text{Type} & \quad \text{by i.h.} \\
\Gamma, \neg \varphi \vdash T_2 \equiv S_2 :: K_2 \text{ with } \Gamma \vdash K_2 \leq \text{Type} & \quad \text{by i.h.} \\
\Gamma \vdash \text{if } \varphi \text{ then } T_1 \text{ else } T_2 \equiv \text{if } \varphi \text{ then } S_1 \text{ else } S_2 :: \text{Type} & \quad \text{by rule} \\
\text{Case: } M \text{ is if } T' :: K \text{ as } t \Rightarrow M' \text{ else } N' & \\
\Gamma \vdash \text{if } T' :: K \text{ as } t \Rightarrow M' \text{ else } N' :: T \text{ and } \Gamma \vdash \text{if } T' :: K \text{ as } t \Rightarrow M' \text{ else } N' :: S & \quad \text{assumption} \\
\Gamma \vdash T' :: K, \Gamma \vdash K, t :: K + M' :: T \text{ and } \Gamma \vdash N' :: T & \quad \text{by inversion} \\
\Gamma \vdash T' :: K, \Gamma \vdash K, t :: K + M' :: S \text{ and } \Gamma \vdash N' :: S & \quad \text{by inversion} \\
\Gamma \vdash T \equiv S :: K \text{ with } \Gamma \vdash K \leq \text{Type} & \quad \text{by i.h.} \\
\text{Case: } T \text{ is if } T' :: K \text{ as } t \Rightarrow S_1 \text{ else } S_2 & \\
\Gamma \vdash \text{if } T' :: K \text{ as } t \Rightarrow S_1 \text{ else } S_2 :: K & \quad \text{assumption} \\
\Gamma \vdash T' :: K, \Gamma \vdash K, t :: K + S_1 :: K \text{ and } \Gamma \vdash S_2 :: K & \quad \text{by inversion} \\
\Gamma \vdash T' :: K, \Gamma \vdash K, t :: K + S_1 :: K' \text{ and } \Gamma \vdash S_2 :: K' & \quad \text{by inversion} \\
\Gamma \vdash K \leq K' \text{ or } \Gamma \vdash K' \leq K & \quad \text{by i.h.} \\
\text{Case: } M \text{ is } \mu F \cdot T . M' & \\
\Gamma \vdash \mu F \cdot T . M' :: T \text{ and } \Gamma \vdash \mu F \cdot T . M' :: S & \quad \text{assumption} \\
\Gamma \vdash T \equiv T' :: K \text{ and } \Gamma, F :: T + M' :: T' & \quad \text{by inversion} \\
\Gamma \vdash S \equiv S' :: K' \text{ and } \Gamma, F :: T + M' :: S' & \quad \text{by inversion} \\
\Gamma, F :: T :: T' :: S' :: K \text{ with } \Gamma \vdash K \leq \text{Type} & \quad \text{by i.h.} \\
\Gamma, F :: T :: T \equiv T' :: K \text{ and } \Gamma, F :: T :: S \equiv S' :: K' & \quad \text{by weakening} \\
\Gamma, F :: T :: T \equiv S :: \text{Type} & \quad \text{by transitivity and conversion} \\
\Gamma \vdash T \equiv S :: \text{Type} & \quad \text{by strengthening} \\
\text{Case: } T \text{ is } \mu F :: (\Pi t : K. K'). \lambda t :: K. T' & \\
\Gamma \vdash \mu F :: (\Pi t : K_1. K_2). \lambda t : K_1. T' :: K \text{ and } \Gamma \vdash \mu F :: (\Pi t : K_1. K_2). \lambda t : K_1. T' :: K' & \quad \text{assumption} \\
\Gamma, F :: \Pi t : K_1. K_2, t :: K_1 + T' :: K_2, \text{structural}(T', F, t) \text{ and } \Gamma \vdash K \equiv \Pi t : K_1. K_2 & \quad \text{by inversion} \\
\Gamma, F :: \Pi t : K_1. K_2, t :: K_1 + T' :: K_2, \text{structural}(T', F, t) \text{ and } \Gamma \vdash K' \equiv \Pi t : K_1. K_2 & \quad \text{by inversion} \\
\Gamma \vdash K \leq K' & \quad \text{by transitivity} \\
\text{Case: } M \text{ is } \text{recHeadTerm}(M') & \\
\Gamma \vdash \text{recHeadTerm}(M') :: T \text{ and } \Gamma \vdash \text{recHeadTerm}(M') :: S & \quad \text{assumption} \\
\Gamma \vdash M :: T', \Gamma \vdash T' :: \{t: \text{Rec} \mid \text{headType}(t) \equiv T'' :: K\} \text{ and } \Gamma \vdash T \equiv T''/(T'/t) :: K[T'/t] & \quad \text{by inversion} \\
\Gamma \vdash M :: S', \Gamma \vdash S' :: \{t: \text{Rec} \mid \text{headType}(t) \equiv S'' :: K\} \text{ and } \Gamma \vdash S \equiv S''/(S'/t) :: K[S'/t] & \quad \text{by inversion} \\
\end{align*}
Γ ⊨ T’ ≡ S’ :: K with K ≤ Type by i.h.
Γ ⊨ T’ :: Rec and Γ ⊨ headType(T’) ≡ T’/(t/t) :: K{T’/t} by inversion
Γ ⊨ S’ :: Rec and Γ ⊨ headType(S’') ≡ S’/{S’/t} :: K{S’/t} by inversion
Γ ⊨ T’ :: {t:Rec | nonEmpty(t)} by conversion
Γ ⊨ S’ :: {t:Rec | nonEmpty(t)} by conversion
Γ ⊨ T’ ≡ S’ :: {t:Rec | nonEmpty(t)} by Lemma C.6
Γ ⊨ headType(T’) ≡ headType(S’’) :: Type by equality rule
Γ ⊨ T ≡ T’/{T’/t} :: Type by conversion
Γ ⊨ S ≡ S’/{S’/t} :: Type by conversion
Γ ⊨ T’/{T’/t} ≡ S’/{S’/t} :: Type by transitivity

Case: T is headType(T’)
Γ ⊨ headType(T’) :: K₁ and Γ ⊨ headType(T’) :: K₂ assumption
Γ ⊨ T’ :: {t:K | headType(t) ≡ T’’ :: K’}, Γ ⊨ T’’/{T’/t} :: K’/{T’/t} and Γ ⊨ K₁ ≡ K’/{T’/t}
by inversion
Γ ⊨ T’ :: {t:K’ | headType(t) ≡ T’’’ :: K’’}, Γ ⊨ T’’’/{T’’/t} :: K’’/{T’’/t} and Γ ⊨ K₂ ≡ K’’/{T’’/t}
by inversion
Γ ⊨ {t:K | headType(t) ≡ T’’ :: K’} ≤ {t:K’ | headType(t) ≡ T’’’ :: K’’}
or Γ ⊨ {t:K | headType(t) ≡ T’’ :: K’} ≥ {t:K’ | headType(t) ≡ T’’’ :: K’’} by i.h.
Subcase 1: Γ ⊨ {t:K | headType(t) ≡ T’’ :: K’} ≤ {t:K’ | headType(t) ≡ T’’’ :: K’’}
Γ ⊨ K’ ≤ K’’ and Γ, t:K’ ⊨ headType(t) ≡ T’’ :: K’ ≡ headType(t) ≡ T’’’ :: K’’ by inversion
Γ, t:K’ ⊨ K’ ≡ K’’ by entailment
Γ ⊨ K’/{T’/t} ≡ K’’/{T’/t} by substitution
Γ ⊨ K₁ ≤ K₂
Subcase 2 is symmetric.

Case: T is tmap(T₁) T₂
Γ ⊨ tmap(T₁) T₂ :: K and Γ ⊨ tmap(T₁) T₂ :: K’ assumption
Γ ⊨ T₁ :: Gen_K, Γ ⊨ T₂ :: K and Γ ⊨ K ≡ Type by inversion
Γ ⊨ T₁ :: Gen_K’, Γ ⊨ T₂ :: K’ and Γ ⊨ K’ ≡ Type by inversion
Γ ⊨ K · K’ ≤ K’’ since Γ ⊨ Type ≤ Type

\[ \text{Theorem 5.9 (Type Preservation). Let } \Gamma ⊢ S : T \text{ and } \Gamma ⊢ \Delta \text{. If } \langle H; M \rangle \rightarrow \langle H’; M’ \rangle \text{ then there exists } S’ \text{ such that } S ⊆ S’, \Gamma ⊢ S’ : H’ \text{ and } \Gamma ⊢ S’ : M’ : T. \]

Proof. By induction on the operational semantics and inversion on typing. We show the most significant cases.

Case: 
\[ \langle H; (\Delta t::K.M)[T₀] \rangle \rightarrow \langle H; (\Delta t::K.M)[T₀’] \rangle \]
Γ ⊨ T ≡ U :: K where Γ ⊨ Δ t::K.M : T₁, Γ ⊨ T₀ :: K, Γ ⊨ U :: K, by inversion
Γ ⊨ T₁ :: {f::Gen_K | tmap(f) T₀ ≡ U :: K’} by inversion
Γ ⊨ T₁ ≡ Vt::K.S :: Gen_K and Γ, t::K ⊨ S :: K by inversion
Γ ⊨ S{T₀/t} :: K by substitution
Γ ⊨ S{T₀/t} ≡ S{T₀’/t} :: K’ by definition
Γ ⊨ tmap(\langle Vt::K.S T₀ \rangle) ≡ tmap(\langle Vt::K.S T₀’ \rangle) :: K’ by functionality
Γ ⊨ U ≡ S{T₀/t} :: K by equality
Γ ⊨ U ≡ S{T₀’/t} :: K by transitivity
Γ ⊨ U ≡ S{T₀’/t} :: K by transitivity
\[ \Gamma \vdash (\Lambda t::K. M)[T'_0] : S(T'_0/t) \quad \text{by typing} \]
\[ \Gamma \vdash (\Lambda t::K. M)[T'_0] : U \quad \text{by conversion} \]

**Case:** \[ (H; M) \rightarrow (H'; M') \]
\[ (H; (\ell = M)@N) \rightarrow (H'; (\ell = M')@N) \]
\[ \Gamma \vdash S \triangleright T \equiv (L : T')@T'' \quad \text{by inversion} \]
\[ \exists S' \text{ such that } S \subseteq S', \Gamma \vdash_S H' \text{ and } \Gamma \vdash_S' M' : T' \quad \text{by i.h.} \]
\[ \Gamma \vdash_S (\ell = M')@N : (L : T')@T'' \quad \text{by RecCons rule} \]

**Case:** \[ (H; \text{recHeadTerm}(M)) \rightarrow (H'; \text{recHeadTerm}(M')) \]
\[ \Gamma \vdash S \triangleright T'(S'/t) : \mathcal{K}[S'/t], \Gamma \vdash_S M : S' \text{ and} \]
\[ \Gamma \vdash S' : \{ : \text{Rec} \mid \text{headType}(t) \equiv T' \equiv \mathcal{K} \} \quad \text{by inversion} \]
\[ \exists S_0 \text{ such that } S \subseteq S_0, \Gamma \vdash_{S_0} H' \text{ and } \Gamma \vdash_{S_0} M' : S' \quad \text{by i.h.} \]
\[ \Gamma \vdash_{S_0} \text{recHeadTerm}(M') : T'(S'/t) \quad \text{by typing rule} \]

**Case:** \[ (H; \text{recHeadTerm}((\ell = v)@v')) \rightarrow (H; v) \]
\[ \Gamma \vdash S \text{ recHeadTerm}((\ell = v)@v') : T' \text{ and } \Gamma \vdash_S v : T' \quad \text{by inversion} \]

**Case:** \[ \bar{\Gamma} \not\vdash \varphi \]
\[ \Gamma \models T \equiv \text{if } \varphi \text{ then } T_1 \text{ else } T_2 \equiv K \text{ with } \Gamma, \varphi \vdash_S M : T_1 \text{ and } \Gamma, \neg \varphi \vdash_S N : T_2 \quad \text{by inversion} \]
\[ \Gamma \models \text{if } \varphi \text{ then } T_1 \text{ else } T_2 \equiv T_1 \equiv K \quad \text{by eq. rule} \]
\[ \Gamma \models T \equiv T_1 \equiv K \quad \text{by transitivity} \]
\[ \Gamma \vdash_S M : T_1 \quad \text{by cut} \]

**Case:** \[ (H; \mu F.T.M) \rightarrow (H; M(\mu F.T.M/F)) \]
\[ \Gamma, F : T + M : T \text{ and structural}(F, M) \text{ by inversion } \Gamma \vdash M(\mu F.T.M/F) : T \quad \text{by substitution} \]
\[ \Gamma \vdash T : K \quad \text{by substitution} \]

**Case:** \[ (H; \text{if } T' : K \text{ as } t \Rightarrow M \text{ else } N) \rightarrow (H; M(T'/t)) \]
\[ \Gamma \vdash T' : K', \Gamma \vdash K, \Gamma, t : K \vdash M : T'' \text{ and } \Gamma \vdash N : T'' \quad \text{by inversion} \]
\[ \Gamma \vdash T : K \quad \text{assumption} \]
\[ \Gamma \vdash M(T'/t) : T'' \quad \text{by substitution} \]

**Lemma 5.10 (Type Progress).** \textit{If } \Gamma \vdash T : K \text{ then either } T \text{ is a type value or } T \rightarrow T', \text{ for some } T'.

**Proof.** Straightforward induction on kinding.

**Theorem 5.11 (Progress).** \textit{Let } \cdot \vdash_S M : T \text{ and } \cdot \vdash_S H. \text{ Then either } M \text{ is a value or there exists } S' \text{ and } M' \text{ such that } (H; M) \rightarrow (H'; M').

**Proof.** By induction on typing. Progress relies type progress and on the decidability of entailment due to the term-level and type-level predicate test construct.