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 *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 *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 $\langle\rangle$ and the "cons" cell $\langle L : T\rangle @R$, which constructs the record type obtained by adding a field declaration $\langle L : T\rangle$ to the record type R.

The record type destructors are functions $\mathbf{headLabel}(R)$, $\mathbf{headType}(R)$ and $\mathbf{tail}(R)$, 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 $\mathsf{Person} \triangleq \langle \mathit{name} : \mathsf{String}\rangle @\langle \mathit{age} : \mathsf{Int}\rangle @\langle\rangle$, which we conveniently abbreviate by $\langle \mathit{name} : \mathsf{String}; \mathit{age} : \mathsf{Int}\rangle$. We then have that $\mathbf{headLabel}(\mathsf{Person}) = \mathit{name}$, $\mathbf{headType}(\mathsf{Person}) = \mathsf{String}$ and $\mathbf{tail}(\mathsf{Person}) = \langle \mathit{age} : \mathsf{Int}\rangle$. The kinding of the $\langle L : T\rangle @R$ type constructor may be clarified in the following type-level function $\mathsf{addFieldType}$

$$\begin{aligned}
&\mathsf{addFieldType} :: \Pi l{::}\mathsf{Nm}.\, \Pi t{::}\mathsf{Type}.\, \Pi r{::}\{s{::}\mathsf{Rec} \mid l \notin \mathbf{lab}(s)\}.\, \mathsf{Rec}\\
&\mathsf{addFieldType} \triangleq \lambda l{::}\mathsf{Nm}.\lambda t{::}\mathsf{Type}.\lambda r{::}\{s{::}\mathsf{Rec} \mid l \notin \mathbf{lab}(s)\}.\langle l : t\rangle @r
\end{aligned}$$

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 Rec. Notice that the *kind* of all record types that do not contain label l is represented by the refinement kind $\{s{::}\mathsf{Rec} \mid l \notin \mathbf{lab}(s)\}$.

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

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}\ (\textsc{kref})$$

where $\Gamma \models \varphi$ 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 $\mathbf{lab}(R)$ (record label set), $L \in S$ (label membership), $S\#S'$ (label set apartness), $R@S$ (concatenation), $\mathbf{dom}(F)$ (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 $\mathsf{S}(T)$ of the form $\{t :: \mathcal{K} \mid t \equiv T :: \mathcal{K}\}$. As another simple example, consider the kind Auto of automorphisms, defined as $\{t :: \mathsf{Fun} \mid \mathbf{dom}(t) \equiv \mathbf{img}(t) :: \mathsf{Type}\}$.

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

$$\begin{array}{l}\mathsf{addField} : \forall l{::}\mathsf{Nm}.\forall t{::}\mathsf{Type}.\forall r{::}\{s{::}\mathsf{Rec} \mid l \notin \mathbf{lab}(s)\}.t \to r \to \mathsf{addFieldType}\ l\ t\ r \\ \mathsf{addField} \triangleq \Lambda l{::}\mathsf{Nm}.\Lambda t{::}\mathsf{Type}.\Lambda r{::}\{s{::}\mathsf{Rec} \mid l \notin \mathbf{lab}(s)\}.\lambda x{:}t.\lambda y{:}r.\langle l = x\rangle @ y\end{array}$$

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:

$$l{:}\mathsf{Nm}, t{:}\mathsf{Type}, r{:}\{s{::}\mathsf{Rec} \mid l \notin \mathbf{lab}(s)\}, x{:}t, y{:}r \vdash \mathsf{addField}\ l\ t\ r\ x\ y : \mathsf{addFieldType}\ l\ t\ r$$

An instance of this judgement yields:

$$\vdash \mathsf{addField}\ \mathit{name}\ \mathsf{String}\ \langle \mathit{age} : \mathsf{Int}\rangle\ \text{“jack”}\ \langle \mathit{age} = 20\rangle : \mathsf{addFieldType}\ \mathit{name}\ \mathsf{String}\ \langle \mathit{age} : \mathsf{Int}\rangle$$

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

$$\begin{array}{l}(\mathsf{addField}\ \mathit{name}\ \mathsf{String}\ \langle \mathit{age} : \mathsf{Int}\rangle\ \text{“jack”}\ \langle \mathit{age} = 20\rangle) \to^* \langle \mathit{name} = \text{“jack”}; \mathit{age} = 20\rangle \\ (\mathsf{addFieldType}\ \mathit{name}\ \mathsf{String}\ \langle \mathit{age} : \mathsf{Int}\rangle) \equiv \langle \mathit{name} : \mathsf{String}; \mathit{age} : \mathsf{Int}\rangle\end{array}$$

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

$$l{:}\mathsf{Nm}, t{:}\mathsf{Type}, r{:}\{s{::}\mathsf{Rec} \mid l \notin \mathbf{lab}(s)\} \vdash \mathsf{addFieldType}\ l\ t\ r :: \{s{::}\mathsf{Rec} \mid s \equiv \langle l : t\rangle @ r : \mathsf{Rec}\}$$

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:

$$
\begin{array}{l}
\mathsf{genConstr} \triangleq \Lambda S{::}\{r{::}\mathsf{Rec} \mid \mathsf{nonEmpty}(r)\}.\Lambda V{::}\{v{::}\mathsf{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.\lambda v{:}V.\\
\quad \lambda x{:}\mathbf{headType}(S).\mathbf{if}\ \mathsf{nonEmpty}(\mathbf{tail}(S))\ \mathbf{then}\\
\qquad\qquad \mathsf{genConstr}\ \mathbf{tail}(S)\ \boldsymbol{\langle}\mathbf{headLabel}(S) : \mathbf{ref}\ \mathbf{headType}(S)\boldsymbol{\rangle}@V\ \langle\mathbf{headLabel}(S) = \mathbf{ref}\ x\rangle@v\\
\qquad\qquad\qquad \mathbf{else}\ \langle\mathbf{headLabel}(S) = \mathbf{ref}\ x\rangle@v
\end{array}
$$

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 $\boldsymbol{\langle\rangle}$ $\langle\rangle$ computes to a value equivalent to $\lambda x{:}\mathsf{String}.\lambda y{:}\mathsf{Int}.\langle \mathit{age} = \mathbf{ref}\ y; \mathit{name} = \mathbf{ref}\ x\rangle$.

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 $\boldsymbol{\langle\rangle}$ (i.e. the empty record *type*), and v to $\langle\rangle$. Moreover, the refined kind of V specifies the label apartness constraint needed to type check the recursive call of genConstr, in particular, given $\mathbf{lab}(V)\#\mathbf{lab}(S)$, the refinement logic deduces $\mathbf{headLabel}(S) \notin \mathbf{lab}(V)$, needed to kind check $\boldsymbol{\langle}\mathbf{headLabel}(S) : \mathbf{ref}\ \mathbf{headType}(S)\boldsymbol{\rangle}@V$; and $\mathbf{lab}(\boldsymbol{\langle}\mathbf{headLabel}(S) : \mathbf{ref}\ \mathbf{headType}(S)\boldsymbol{\rangle}@V)\#\mathbf{lab}(\mathbf{tail}(S))$, required to kind and type check the recursive call. In our language, genConstr can be typed as follows:

$$
\mathsf{genConstr} : \forall S{::}\{r{::}\mathsf{Rec} \mid \mathsf{nonEmpty}(r)\}.\forall V{::}\{v{::}\mathsf{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.(\mathsf{GType}\ S\ V)
$$

where GType is the (recursive) type-level function such that

$$
\begin{array}{l}
\mathsf{GType} :: \Pi S{::}\{r{::}\mathsf{Rec} \mid \mathsf{nonEmpty}(r)\}.\, \Pi V{::}\{v{::}\mathsf{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}.\, \mathsf{Fun} \\
\mathsf{GType} \triangleq \\
\quad \lambda S{::}\{r{::}\mathsf{Rec} \mid \mathsf{nonEmpty}(r)\}. \\
\quad \lambda V{::}\{v{::}\mathsf{Rec} \mid \mathbf{lab}(v)\#\mathbf{lab}(S)\}. \\
\quad \mathbf{headType}(S) \to \mathbf{if}\ \mathsf{nonEmpty}(\mathbf{tail}(S))\ \mathbf{then} \\
\qquad\qquad\qquad\qquad \mathsf{GType}\ \mathbf{tail}(S)\ \langle \mathbf{headLabel}(S) : \mathbf{ref}\ \mathbf{headType}(S)\rangle @ V\ \mathbf{else}\ V
\end{array}
$$

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

$$
\mathsf{genConstr}\ \mathsf{Person}\ \langle\rangle\ \langle\rangle : \mathsf{String} \to \mathsf{Int} \to \langle \mathit{age} = \mathbf{ref}\ \mathsf{Int}; \mathit{name} = \mathbf{ref}\ \mathsf{String}\rangle
$$

From Record Types to XML Tables. As a second example, we develop a generic function 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 R by applying a type transformation G (of higher-order kind) to the type of each field of R.

$$
\begin{array}{l}
\mathsf{Map} :: \Pi G{::}(\Pi X :: \mathsf{Type}.\, \mathsf{Type}).\, \Pi R{::}\mathsf{Rec}.\, \{r :: \mathsf{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\mathsf{Map} \triangleq \\
\quad \lambda G{::}(\Pi X :: \mathsf{Type}.\, \mathsf{Type}).\lambda R{::}\mathsf{Rec}. \\
\qquad \mathbf{if}\ \mathsf{nonEmpty}(R)\ \mathbf{then}\ \langle \mathbf{headLabel}(R) : G\ \mathbf{headType}(R)\rangle @ (\mathsf{Map}\ G\ \mathbf{tail}(R))\ \mathbf{else}\ \langle\rangle
\end{array}
$$

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

$$
\begin{array}{ll}
\mathsf{XForm} & :: \Pi t :: \mathsf{Type}.\, \mathsf{Type} \\
\mathsf{XForm} & \triangleq \lambda t{::}\mathsf{Type}.\langle \mathit{tag} : \mathsf{String}; \mathit{toStr} : t \to \mathsf{String}\rangle \\
 & \\
\mathsf{MkTableType} & :: \lambda r{::}\mathsf{Rec}.\{r :: \mathsf{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\} \\
\mathsf{MkTableType} & \triangleq \lambda r{::}\mathsf{Rec}.\mathsf{Map}\ \mathsf{XForm}\ r \\
 & \\
\mathsf{MkTable} & : \forall R{::}\mathsf{Rec}.(\mathsf{MkTableType}\ R) \to R \to \mathsf{String} \\
\mathsf{MkTable} & \triangleq \Lambda R{::}\mathsf{Rec}.\lambda M{:}\mathsf{MkTableType}\ R.\lambda r{:}R. \\
 & \quad \mathbf{if}\ \mathsf{nonEmpty}(R)\ \mathbf{then} \\
 & \qquad \text{"<tr><th>"} + M.\mathbf{recHeadLabel}(M).\mathit{tag} + \text{"</th>"} + \\
 & \qquad M.\mathbf{recHeadLabel}(M).\mathit{toStr}\ r.\mathbf{recHeadLabel}(M) + \text{"</td></tr>"} \\
 & \qquad \mathsf{MkTable}\ \mathbf{tail}(R)\ \mathbf{recTail}(M)\ \mathbf{recTail}(r) \\
 & \quad \mathbf{else}\ \text{""}
\end{array}
$$

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 $M.\mathbf{recHeadLabel}(M).\mathit{tag}$. Notice that, by declaration, R::Rec and r:R. However, the expression under consideration is to be typed under the assumption that nonEmpty(R), which is added to the current set of refinement assumptions while typing the **then** branch. Using TT for the type of M, Since MkTableType R :: $\{r{::}\mathsf{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\}$, by refinement we have that $\mathbf{lab}(TT) = \mathbf{lab}(R)$ and thus nonEmpty(TT), allowing $\mathbf{recHeadLabel}(M)$ to be defined. Since $M : \mathsf{MkTableType}\ R$ we have

$$\begin{aligned}&(\mathsf{MkTableType}\ R) \equiv (\mathsf{Map}\ \mathsf{XForm}\ R) \equiv\\ &\quad \langle \mathbf{headLabel}(R) : \mathsf{XForm}\ \mathbf{headType}(R)\rangle @ (\mathsf{Map}\ G\ \mathbf{tail}(R))\end{aligned}$$

We thus derive $\mathbf{headLabel}(TT) \equiv \mathbf{headLabel}(R)$.Then

$$\begin{aligned}&\mathbf{headType}(\mathsf{MkTableType}\ R) \equiv\\ &\mathsf{XForm}\ \mathbf{headType}(R) \equiv \langle \mathit{tag} : \mathsf{String}; \mathit{toStr} : \mathbf{headType}(R) \to \mathsf{String}\rangle\end{aligned}$$

Hence $M.\mathbf{headLabel}(M).\mathit{tag} : \mathsf{String}$. By a similar reasoning, we conclude $r.\mathbf{recHeadLabel}(M) : \mathbf{headType}(R)$. 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:

$$\begin{aligned}&\mathsf{MutableRec} && :: \Pi R :: \mathsf{Rec}.\ \{r :: \mathsf{Rec} \mid \mathbf{lab}(r) = \mathbf{lab}(R)\}\\ &\mathsf{MutableRec} && \triangleq \mathsf{Map}\ (\lambda r{::}\mathsf{Type}.\mathbf{ref}\ r)\end{aligned}$$

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:

$$\begin{aligned}&\mathsf{SetGetRec} && :: \Pi R :: \mathsf{Rec}.\ \{r :: \mathsf{Rec} \mid \mathbf{lab}(r) = \mathit{set}{+}{+}\mathbf{lab}(R) \cup \mathit{get}{+}{+}\mathbf{lab}(R)\}\\ &\mathsf{SetGetRec} && \triangleq \lambda R{::}\mathsf{Rec}.\\ &&& \quad \mathbf{if}\ \mathsf{nonEmpty}(R)\ \mathbf{then}\\ &&& \qquad \langle \mathit{get}{+}{+}\mathbf{headLabel}(R) : \mathbf{1} \to \mathbf{headType}(R)\rangle @\\ &&& \qquad \langle \mathit{set}{+}{+}\mathbf{headLabel}(R) : \mathbf{headType}(R) \to \mathbf{1}\rangle @\\ &&& \qquad \mathsf{SetGetRec}\ \mathbf{tail}(R)\\ &&& \quad \mathbf{else}\ \langle\rangle\end{aligned}$$

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:

$$\begin{aligned}&\mathsf{MkMut} && :: \forall R :: \mathsf{Rec}.\mathsf{MutableRec}\ R \to \mathsf{SetGetRec}\ R\\ &\mathsf{MkMut} && \triangleq \Lambda R{::}\mathsf{Rec}.\\ &&& \quad \lambda r{:}\mathsf{MutableRec}\ R.\\ &&& \quad \mathbf{if}\ \mathsf{nonEmpty}(R)\ \mathbf{then}\\ &&& \qquad \langle \mathit{get}{+}{+}\mathbf{headLabel}(R) = \lambda x{:}\mathbf{1}.!(r.\mathbf{recHeadLabel}(R))\rangle @\\ &&& \qquad \langle \mathit{set}{+}{+}\mathbf{headLabel}(R) = \lambda x{:}\mathbf{headType}(R).r.\mathbf{recHeadLabel}(R) := x\rangle @\\ &&& \qquad \mathsf{MkMut}\ \mathbf{tail}(R)\ \mathbf{recTail}(r)\\ &&& \quad \mathbf{else}\ \langle\rangle\end{aligned}$$

For example, assuming $r : \mathsf{MutableRec}\ \mathsf{Person}$ we have that $\mathsf{MkMut}\ \mathsf{Person}\ r$ computes a record equivalent to:

$$\begin{aligned}&\langle \mathit{getname} = \lambda x{:}\mathbf{1}.!(r.\mathit{name});\\ &\ \mathit{setname} = \lambda x{:}\mathsf{String}.r.\mathit{name} := x;\\ &\ \mathit{getname} = \lambda x{:}\mathbf{1}.!(r.\mathit{name});\\ &\ \mathit{setage} = \lambda x{:}\mathsf{Int}.r.\mathit{age} := x\rangle\end{aligned}$$

where $(\mathsf{MkMut}\ \mathsf{Person}\ r) : \mathsf{SetGetRec}\ \mathsf{Person}$.

Kinds	K, K'	$::=$	$\mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K'$	Refined and Dependent Kinds
	$\mathcal{K}$	$::=$	$\mathsf{Rec} \mid \mathsf{Col} \mid \mathsf{Fun} \mid \mathsf{Ref} \mid \mathsf{Nm}$	Base Kinds
		$\mid$	$\mathsf{Type} \mid \mathsf{Gen}_K$	
Types	T, S, R	$::=$	$t \mid \lambda t{::}K.T \mid T\,S$	Type-level Functions
		$\mid$	$\boldsymbol{\mu} F : (\Pi t{:}K.\,K').\lambda t{::}K.\,T$	Structural Recursion
		$\mid$	$\forall t{::}K.T \mid \mathbf{tmap}(T)\,S$	Polymorphism
		$\mid$	$L \mid \langle\rangle \mid \langle L : T\rangle @S$	Record Type constructors
		$\mid$	$\mathbf{headLabel}(T) \mid \mathbf{headType}(T) \mid \mathbf{tail}(T)$	Record Type destructors
		$\mid$	$T^\star \mid \mathbf{colOf}(T)$	Collection Types
		$\mid$	$\mathbf{ref}\,T \mid \mathbf{refOf}(T)$	Reference Types
		$\mid$	$T \to S \mid \mathbf{dom}(T) \mid \mathbf{img}(T)$	Function Types
		$\mid$	$\mathbf{if}\,T :: \mathcal{K}\,\mathbf{as}\,t \Rightarrow S\,\mathbf{else}\,U$	Kind Case
		$\mid$	$\mathbf{if}\,\varphi\,\mathbf{then}\,T\,\mathbf{else}\,S$	Property Test
		$\mid$	$\bot \mid \top$	Empty and Top Types
		$\mid$	$\mathsf{Bool} \mid \mathbf{1} \mid \ldots$	Basic Data Types
Refinements	φ, ψ	$::=$	$P(T_1, \ldots, T_n)$	Type Predicates
		$\mid$	$\varphi \supset \psi \mid \varphi \wedge \psi \mid \ldots$	Propositional Logic
		$\mid$	$T \equiv S :: K$	Equality

Fig. 1. Syntax of Kinds, Types and Refinements

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 Gen_K 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 $\mathcal{K}$ 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 $\mathbf{1}$). 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 $\langle\rangle$, and the constructor $\langle L : T\rangle @R$, 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 label L associated with type T. Record types are associated with three destructors: **headLabel**(T), which projects the label of the head of the record T (when seen as a list); **headType**(T) which projects the type at the head of the record T; and **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 λ-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 $\mathbf{dom}(T)$ and $\mathbf{img}(T)$, 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 $\forall t{::}\mathsf{Fun}.t \to \mathbf{dom}(t) \to \mathbf{img}(t)$ 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, $\mathbf{tmap}(T)\,S$, 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 $T^\star$, with the associated type destructor $\mathbf{colOf}(T)$, which projects out the type of the collection elements. Similarly, reference types $\mathbf{ref}\,T$ are bundled with a destructor $\mathbf{refOf}(T)$ 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 λ-calculus of types has a *kind case* construct, $\mathbf{if}\ T :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow S\ \mathbf{else}\ U$, 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 λ-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 $\boldsymbol{\mu} F : (\Pi t{:}K.\,K').\lambda t{::}K.\,T$, 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 λ-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 φ is a logical formula (with t bound in φ), characterises types T of kind $\mathcal{K}$ such that the property φ holds of T (i.e. $\varphi\{T/t\}$ is true). The language of properties

φ 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 $\{t{::}\mathsf{Fun} \mid \mathbf{img}(t) \equiv \mathsf{Bool} :: \mathsf{Type}\}$, 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:

$$\begin{array}{l}\mathsf{dropField} \triangleq \lambda l{:}\mathsf{Nm}.\boldsymbol{\mu} F : (\Pi t{:}\{r{::}\mathsf{Rec} \mid l \in \mathbf{lab}(r)\}.\, \{r{::}\mathsf{Rec} \mid l \notin \mathbf{lab}(r)\}).\lambda t{::}\{r{::}\mathsf{Rec} \mid l \in \mathbf{lab}(r)\}. \\ \quad \mathbf{if}\ \mathbf{headLabel}(t) \equiv l :: \mathsf{Nm}\ \mathbf{then}\ \mathbf{tail}(t)\ \mathbf{else}\ \langle \mathbf{headLabel}(t) : \mathbf{headType}(t)\rangle @ (F\ (\mathbf{tail}(t)))\end{array}$$

The function 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 $\mathbf{lab}(r)$ denotes the refinement-level set of labels of r). Such a function combines structural recursion (where $\mathbf{tail}(t)$ is correctly deemed as structurally smaller than t) with our type-level refinement test, $\mathbf{if}\ \varphi\ \mathbf{then}\ T\ \mathbf{else}\ S$. We note that the well-kindedness of such a function relies crucially on the ability to derive that, when the record label $\mathbf{headLabel}(t)$ is not l, since we know that l must be in t, then $\mathbf{tail}(t)$ 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 Γ. Contexts contain assumptions of the form $t{:}K$, $x{:}T$ and φ – t stands for a type of kind K, x stands for a term of type T and refinement φ 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 Γ 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 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 Σ 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 Type. We note that the three record type destructors are only well-kinded 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

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}\ (\textsc{kref})$$

$$\begin{array}{llll@{\qquad}lll} \textbf{headLabel} & :: & \Pi t{:}\{r{::}\mathsf{Rec} \mid \mathsf{nonEmpty}(r)\}.\mathsf{Nm} & \textbf{colOf} & :: & \Pi t{:}\mathsf{Col}.\mathsf{Type} \\ \textbf{headType} & :: & \Pi t{:}\{r{::}\mathsf{Rec} \mid \mathsf{nonEmpty}(r)\}.\mathsf{Type} & \textbf{dom} & :: & \Pi t{:}\mathsf{Fun}.\mathsf{Type} \\ \textbf{tail} & :: & \Pi t{:}\{r{::}\mathsf{Rec} \mid \mathsf{nonEmpty}(r)\}.\mathsf{Rec} & \textbf{img} & :: & \Pi t{:}\mathsf{Fun}.\mathsf{Type} \\ \textbf{refOf} & :: & \Pi t{:}\mathsf{Ref}.\mathsf{Type} & \textbf{tmap} & :: & \Pi t{:}\mathsf{Gen}_K.\Pi s{:}K.\mathsf{Type} \end{array}$$

Fig. 2. Simple Kinding for Type Destructors

Given a type T of kind $\mathcal{K}$ and a *valid* property φ 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 β-like equational reasoning for records allows us to derive $\langle \ell : \mathsf{Bool} \to \mathsf{Bool}\rangle @\langle\rangle :: \{t{::}\mathsf{Rec} \mid \textbf{headType}(t) \equiv \mathsf{Bool} \to \mathsf{Bool} :: \mathsf{Type}\}$. In general, we provide a form of equality elimination rule in refinements, stating that (for a well-formed property φ) the validity of a property φ of some type T is closed under type equality:

$$\frac{\Gamma \models T \equiv S :: K \quad \Gamma, x : K \vdash \varphi \quad \Gamma \models \varphi\{T/x\}}{\Gamma \models \varphi\{S/x\}}\ (\textsc{r-eqElim})$$

As we have previously illustrated, properties can also be tested for validity in types through a conditional construct **if** φ **then** T **else** S. Provided that the property φ is well-formed, if T is of kind K assuming φ and S of kind K assuming $\neg\varphi$, then the conditional test is well-kinded, 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).

$$\frac{\Gamma \vdash \varphi \quad \Gamma, \varphi \vdash T :: K \quad \Gamma, \neg\varphi \vdash S :: K}{\Gamma \vdash \textbf{if } \varphi \textbf{ then } T \textbf{ else } S :: K}\ (\textsc{k-ite}) \qquad \frac{\Gamma \models \varphi \quad \Gamma, \varphi \vdash T_1 :: K \quad \Gamma, \neg\varphi \vdash T_2 :: K}{\Gamma \models \textbf{if } \varphi \textbf{ then } T_1 \textbf{ else } T_2 \equiv T_1 :: K}\ (\textsc{eq-iteT})$$

$$\frac{\Gamma \models \neg\varphi \quad \Gamma, \varphi \vdash T_1 :: K \quad \Gamma, \neg\varphi \vdash T_2 :: K}{\Gamma \models \textbf{if } \varphi \textbf{ then } T_1 \textbf{ else } T_2 \equiv T_2 :: K}\ (\textsc{eq-iteE}) \qquad \frac{\Gamma \vdash \varphi \quad \Gamma, \varphi \vdash T :: K \quad \Gamma, \neg\varphi \vdash T :: K}{\Gamma \models \textbf{if } \varphi \textbf{ then } T \textbf{ else } T \equiv T :: K}\ (\textsc{eq-iteEq})$$

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

$$\frac{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid elim_{\mathcal{K}}(t) \equiv T' :: K'\} \quad \Gamma \vdash T'\{T/t\} :: K'\{T/t\}}{\Gamma \vdash elim_{\mathcal{K}}(T) :: K'\{T/t\}}\ (\textsc{k-elim})$$

$$\frac{\Gamma \models T \equiv S :: \{t{::}\mathcal{K} \mid elim_{\mathcal{K}}(T) \equiv T' :: K'\} \quad \Gamma \vdash T'\{T/t\} :: K'\{T/t\}}{\Gamma \models elim_{\mathcal{K}}(T) \equiv T'\{T/t\} :: K'\{T/t\}}\ (\textsc{eq-elim})$$

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 $\forall t{::}\{f{:}\mathsf{Fun} \mid \textbf{dom}(f) \equiv \mathsf{Bool} :: \mathsf{Type} \wedge \textbf{img}(f) \equiv \mathsf{Bool} :: \mathsf{Type}\}.t \to \mathsf{Bool}$ can be used to type the term $\Lambda t{::}\{f{:}\mathsf{Fun} \mid \textbf{dom}(f) \equiv \mathsf{Bool} :: \mathsf{Type} \wedge \textbf{img}(f) \equiv \mathsf{Bool} :: \mathsf{Type}\}.\lambda f{:}t.(f\ \mathit{true})$, where Λ 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):

$$\frac{\Gamma \vdash K \quad \Gamma, t{:}K \vdash T :: K'}{\Gamma \vdash \lambda t{::}K.T :: \Pi t{:}K.K'}\ (\textsc{k-fun}) \qquad \frac{\Gamma \vdash T :: \Pi t{:}K.K' \quad \Gamma \vdash S :: K}{\Gamma \vdash T\,S :: K'\{S/t\}}\ (\textsc{k-app}) \qquad \frac{t{:}K \in \Gamma \quad \Gamma \vdash}{\Gamma \vdash t :: K}\ (\textsc{k-var})$$

$$\frac{\Gamma, t{:}K \vdash T :: K' \quad \Gamma \vdash S :: K}{\Gamma \models (\lambda t{::}K.T)\,S \equiv T\{S/t\} :: K'\{S/t\}}\ (\textsc{eq-funapp})$$

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

$$\frac{\Gamma, F{:}\Pi t{:}K.K', t{:}K \vdash T :: K' \quad \mathsf{structural}(T, F, t)}{\Gamma \vdash \boldsymbol{\mu} F : (\Pi t{:}K.\,K').\lambda t{::}K.\,T :: \Pi t{:}K.K'}\ (\textsc{k-fix})$$

$$\frac{\Gamma, t{:}K_1 \vdash K_2 \quad \Gamma, F{:}\Pi t{:}K_1.K_2, t{:}K_1 \vdash T :: K_2 \quad \Gamma \vdash S :: K_1 \quad \mathsf{structural}(T, F, t)}{\Gamma \models (\boldsymbol{\mu} F : (\Pi t{:}K_1.\,K_2).\lambda t{::}K_1.\,T)\,S \equiv T\{S/t\}\{(\boldsymbol{\mu} F : (\Pi t{:}K_1.\,K_2).\lambda t{::}K_1.\,T)/F\} :: K_2\{S/t\}}\ (\textsc{eq-fixunf})$$

The predicate $\mathsf{structural}(T, F, t)$ 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 $\mathsf{structural}(T, F, t)$ holds iff all occurrences of F in T are applied to terms smaller than t, where the notion of size is given by $elim_{\mathcal{K}}(t) < t$, with $\mathcal{K}$ is any basic kind, with the exception of 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 Gen_K, as expected, and the β-like equality principle for the elimination form $\mathbf{tmap}(\forall t{::}K.T)\,S$ performs the appropriate instantiation of t with S in T.

$$\frac{\Gamma \vdash K \quad \Gamma, t{:}K \vdash T :: \mathcal{K}}{\Gamma \vdash \forall t{::}K.T :: \mathsf{Gen}_K}\ (\textsc{k-}\forall) \qquad \frac{\Gamma, t{:}K \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: K}{\Gamma \models \mathbf{tmap}(\forall t{::}K.T)\,S \equiv T\{S/t\} :: \mathsf{Type}}\ (\textsc{eq-tmap})$$

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:

$$\frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \mathcal{K}'}{\Gamma \models \mathbf{dom}(T \to S) \equiv T :: \mathsf{Type}}\ (\textsc{eq-dom}) \qquad \frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \mathcal{K}'}{\Gamma \models \mathbf{img}(T \to S) \equiv S :: \mathsf{Type}}\ (\textsc{eq-img})$$

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

$$\overset{(\textsc{k-recnil})}{\frac{\Gamma \vdash}{\Gamma \vdash \langle\rangle :: \mathsf{Rec}}} \qquad \overset{(\textsc{k-reccons})}{\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \vdash \langle L : T\rangle @ S :: \mathsf{Rec}}} \qquad \overset{(\textsc{k-label})}{\frac{\Gamma \vdash \ell \in \mathcal{N}}{\Gamma \vdash \ell :: \mathsf{Nm}}}$$

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-kinded.

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \models \mathbf{headLabel}(\langle L : T\rangle @S) \equiv L :: \mathsf{Nm}}\ (\text{EQ-HEADLABEL})$$

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \models \mathbf{headType}(\langle L : T\rangle @S) \equiv T :: \mathsf{Type}}\ (\text{EQ-HEADTYPE})$$

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \models \mathbf{tail}(\langle L : T\rangle @S) \equiv S :: \mathsf{Rec}}\ (\text{EQ-TAIL})$$

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.

$$\frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T^{\star} :: \mathsf{Col}}\ (\text{K-COL}) \qquad \frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash \mathbf{ref}\, T :: \mathsf{Ref}}\ (\text{K-REF}) \qquad \frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \models \mathbf{colOf}(T^{\star}) \equiv T :: \mathsf{Type}}\ (\text{EQ-COL}) \qquad \frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \models \mathbf{refOf}(\mathbf{ref}\, T) \equiv T :: \mathsf{Type}}\ (\text{EQ-REF})$$

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 Type, the kind of record types Rec is a specialisation of 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:

$$\frac{\Gamma \vdash T :: K \quad \Gamma \vdash K \leq K'}{\Gamma \vdash T :: K'}\ (\text{K-SUB}) \qquad \frac{\Gamma \vdash K \equiv K'}{\Gamma \vdash K \leq K'}\ (\text{SUB-EQ}) \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathcal{K} \leq \mathsf{Type}}\ (\text{SUB-TYPE})$$

$$\frac{\Gamma \vdash \mathcal{K} \quad \Gamma, t{:}\mathcal{K} \vdash \varphi}{\Gamma \vdash \{t{::}\mathcal{K} \mid \varphi\} \leq \mathcal{K}}\ (\text{SUB-REFKIND}) \qquad \frac{\Gamma \vdash \mathcal{K} \leq \mathcal{K}' \quad \Gamma, t{:}\mathcal{K}' \models \varphi \equiv \varphi'}{\Gamma \vdash \{t{::}\mathcal{K} \mid \varphi\} \leq \{t : \mathcal{K}' \mid \varphi'\}}\ (\text{SUB-REF})$$

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.

$$\frac{\Gamma \vdash \mathcal{K} \quad \Gamma \vdash T :: \mathcal{K}'' \quad \Gamma, t{:}\mathcal{K} \vdash S :: K' \quad \Gamma \vdash U :: K'}{\Gamma \vdash \mathbf{if}\, T :: \mathcal{K}\ \mathbf{as}\, t \Rightarrow S\, \mathbf{else}\, U :: K'}\ (\text{K-KCASE}) \qquad \frac{\Gamma \models \bot \quad \Gamma \vdash K}{\Gamma \vdash \bot :: K}\ (\text{K-BOT})$$

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 Γ then we can derive any equality via rule (EQ-BOT).

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}\;(\textsc{kref}) \qquad \frac{\Gamma \models T \equiv S :: K \quad \Gamma, x : K \vdash \varphi \quad \Gamma \models \varphi\{T/x\}}{\Gamma \models \varphi\{S/x\}}\;(\textsc{r-eqelim})$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma, \varphi \vdash T :: K \quad \Gamma, \neg\varphi \vdash S :: K}{\Gamma \vdash \mathbf{if}\ \varphi\ \mathbf{then}\ T\ \mathbf{else}\ S :: K}\;(\textsc{k-ite}) \qquad \frac{\Gamma \models \varphi \quad \Gamma, \varphi \vdash T_1 :: K \quad \Gamma, \neg\varphi \vdash T_2 :: K}{\Gamma \models \mathbf{if}\ \varphi\ \mathbf{then}\ T_1\ \mathbf{else}\ T_2 \equiv T_1 :: K}\;(\textsc{eq-ite}\mathrm{T})$$

$$\frac{\Gamma \models \neg\varphi \quad \Gamma, \varphi \vdash T_1 :: K \quad \Gamma, \neg\varphi \vdash T_2 :: K}{\Gamma \models \mathbf{if}\ \varphi\ \mathbf{then}\ T_1\ \mathbf{else}\ T_2 \equiv T_2 :: K}\;(\textsc{eq-ite}\mathrm{E}) \qquad \frac{\Gamma \vdash \varphi \quad \Gamma, \varphi \vdash T :: K \quad \Gamma, \neg\varphi \vdash T :: K}{\Gamma \models \mathbf{if}\ \varphi\ \mathbf{then}\ T\ \mathbf{else}\ T \equiv T :: K}\;(\textsc{eq-ite}\mathrm{E}\textsc{q})$$

$$\frac{\Gamma \vdash T :: \{t : \mathcal{K} \mid elim_{\mathcal{K}}(t) \equiv T' :: K'\} \quad \Gamma \vdash T'\{T/t\} :: K'\{T/t\}}{\Gamma \vdash elim_{\mathcal{K}}(T) :: K'\{T/t\}}\;(\textsc{k-elim})$$

$$\frac{\Gamma \models T \equiv S :: \{t{::}\mathcal{K} \mid elim_{\mathcal{K}}(T) \equiv T' :: K'\} \quad \Gamma \vdash T'\{T/t\} :: K'\{T/t\}}{\Gamma \models elim_{\mathcal{K}}(T) \equiv T'\{T/t\} :: K'\{T/t\}}\;(\textsc{eq-elim})$$

$$\frac{\Gamma \vdash K \quad \Gamma, t{:}K \vdash T :: K'}{\Gamma \vdash \lambda t{:}K.T :: \Pi t{:}K.K'}\;(\textsc{k-fun}) \qquad \frac{\Gamma \vdash T :: \Pi t{:}K.K' \quad \Gamma \vdash S :: K}{\Gamma \vdash T\,S :: K'\{S/t\}}\;(\textsc{k-app}) \qquad \frac{t{:}K \in \Gamma \quad \Gamma \vdash}{\Gamma \vdash t :: K}\;(\textsc{k-var})$$

$$\frac{\Gamma, t{:}K \vdash T :: K' \quad \Gamma \vdash S :: K}{\Gamma \models (\lambda t{::}K.T)\,S \equiv T\{S/t\} :: K'\{S/t\}}\;(\textsc{eq-funapp}) \qquad \frac{\Gamma, F{:}\Pi t{:}K.K', t{:}K \vdash T :: K' \quad \mathsf{structural}(T, F, t)}{\Gamma \vdash \boldsymbol{\mu} F : (\Pi t{:}K.\,K').\lambda t{::}K.\,T :: \Pi t{:}K.K'}\;(\textsc{k-fix})$$

$$\frac{\Gamma, t{:}K_1 \vdash K_2 \quad \Gamma, F{:}\Pi t{:}K_1.K_2, t{:}K_1 \vdash T :: K_2 \quad \Gamma \vdash S :: K_1 \quad \mathsf{structural}(T, F, t)}{\Gamma \models (\boldsymbol{\mu} F : (\Pi t{:}K_1.\,K_2).\lambda t{::}K_1.\,T)\,S \equiv T\{S/t\}\{(\boldsymbol{\mu} F : (\Pi t{:}K_1.\,K_2).\lambda t{::}K_1.\,T)/F\} :: K_2\{S/t\}}\;(\textsc{eq-fixunf})$$

Fig. 3. Kinding and Type Equality rules – 1 (Excerpt)

$$\frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma, t{:}\mathcal{K} \vdash S :: K' \quad \Gamma \vdash U :: K'}{\Gamma \models \mathbf{if}\ T :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow S\ \mathbf{else}\ U \equiv S\{T/t\} :: K'}\;(\textsc{eq-kcase}\mathrm{T}) \qquad \frac{\Gamma \models \bot \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \models \bot \equiv T :: \mathcal{K}}\;(\textsc{eq-bot})$$

$$\frac{\Gamma \vdash T :: \mathcal{K}_0 \quad \Gamma \vdash \mathcal{K}_0 \not\equiv \mathcal{K} \quad \Gamma, t{:}\mathcal{K} \vdash S :: K' \quad \Gamma \vdash U :: K'}{\Gamma \models \mathbf{if}\ T :: K\ \mathbf{as}\ t \Rightarrow S\ \mathbf{else}\ U \equiv U :: K'}\;(\textsc{eq-kcase}\mathrm{F})$$

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 $L :: \mathsf{Nm}$ produces a recursive type-function that essentially iterates over

$$\frac{\Gamma \vdash K \quad \Gamma, t{:}K \vdash T :: \mathcal{K}}{\Gamma \vdash \forall t{:}K.T :: \mathsf{Gen}_K}\ (\textsc{k-}\forall) \qquad \frac{\Gamma, t{:}K \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: K}{\Gamma \models \mathbf{tmap}(\forall t{::}K.T)\, S \equiv T\{S/t\} :: \mathsf{Type}}\ (\textsc{eq-tmap})$$

$$\frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \mathcal{K}'}{\Gamma \models \mathbf{dom}(T \to S) \equiv T :: \mathsf{Type}}\ (\textsc{eq-dom}) \qquad \frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \mathcal{K}'}{\Gamma \models \mathbf{img}(T \to S) \equiv S :: \mathsf{Type}}\ (\textsc{eq-img})$$

$$\overset{(\textsc{k-recnil})}{\frac{\Gamma \vdash}{\Gamma \vdash \langle\rangle :: \mathsf{Rec}}} \qquad \overset{(\textsc{k-reccons})}{\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \vdash \langle L : T\rangle @ S :: \mathsf{Rec}}} \qquad \overset{(\textsc{k-label})}{\frac{\Gamma \vdash \ell \in \mathcal{N}}{\Gamma \vdash \ell :: \mathsf{Nm}}}$$

$$\overset{(\textsc{eq-headlabel})}{\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \models \mathbf{headLabel}(\langle L : T\rangle @ S) \equiv L :: \mathsf{Nm}}}$$

$$\overset{(\textsc{eq-headtype})}{\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \models \mathbf{headType}(\langle L : T\rangle @ S) \equiv T :: \mathsf{Type}}}$$

$$\overset{(\textsc{eq-tail})}{\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t : \mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \models \mathbf{tail}(\langle L : T\rangle @ S) \equiv S :: \mathsf{Rec}}}$$

$$\overset{(\textsc{k-col})}{\frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T^\star :: \mathsf{Col}}} \qquad \overset{(\textsc{k-ref})}{\frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash \mathbf{ref}\, T :: \mathsf{Ref}}} \qquad \overset{(\textsc{eq-col})}{\frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \models \mathbf{colOf}(T^\star) \equiv T :: \mathsf{Type}}} \qquad \overset{(\textsc{eq-ref})}{\frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \models \mathbf{refOf}(\mathbf{ref}\, T) \equiv T :: \mathsf{Type}}}$$

$$\frac{\Gamma \vdash T :: K \quad \Gamma \vdash K \leq K'}{\Gamma \vdash T :: K'}\ (\textsc{K-sub}) \qquad \frac{\Gamma \vdash K \equiv K'}{\Gamma \vdash K \leq K'}\ (\textsc{sub-eq}) \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathcal{K} \leq \mathsf{Type}}\ (\textsc{sub-type})$$

$$\frac{\Gamma \vdash \mathcal{K} \quad \Gamma, t{:}\mathcal{K} \vdash \varphi}{\Gamma \vdash \{t{::}\mathcal{K} \mid \varphi\} \leq \mathcal{K}}\ (\textsc{sub-refkind}) \qquad \frac{\Gamma \vdash \mathcal{K} \leq \mathcal{K}' \quad \Gamma, t{:}\mathcal{K}' \models \varphi \equiv \varphi'}{\Gamma \vdash \{t{::}\mathcal{K} \mid \varphi\} \leq \{t : \mathcal{K}' \mid \varphi'\}}\ (\textsc{sub-ref})$$

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

a type record of kind $\{r{::}\mathsf{Rec} \mid \ell \in \mathbf{lab}(r)\}$:

$$\begin{array}{c}\lambda L{::}\mathsf{Nm}.\boldsymbol{\mu} F : (\Pi t{:}\{r{::}\mathsf{Rec} \mid L \in \mathbf{lab}(r)\}.\,\mathsf{Type}).\lambda t{::}\{r{::}\mathsf{Rec} \mid L \in \mathbf{lab}(r)\}.\\ \mathbf{if}\ \mathbf{headLabel}(t) \equiv L :: \mathsf{Nm}\ \mathbf{then}\ \mathbf{headType}(t)\ \mathbf{else}\ F(\mathbf{tail}(t))\end{array}$$

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 Γ_0 be $L{:}\mathsf{Nm}, F{:}\Pi t{:}\{r{::}\mathsf{Rec} \mid L \in \mathbf{lab}(r)\}.\mathsf{Type}, t{:}\{r{:}\mathsf{Rec} \mid L \in \mathbf{lab}(r)\}$):

$$\frac{\Gamma_0 \vdash \mathbf{headLabel}(t) \equiv L :: \mathsf{Nm} \qquad \mathcal{D} \qquad \mathcal{E}}{\Gamma_0 \vdash \mathbf{if}\ \mathbf{headLabel}(t) \equiv L :: \mathsf{Nm}\ \mathbf{then}\ \mathbf{headType}(t)\ \mathbf{else}\ F(\mathbf{tail}(t)) :: \mathsf{Type}}\ (\textsc{k-ite})$$

Terms	M, N	::=	$x \mid \lambda x{:}T.M \mid M\,N$	Functions
		$\mid$	$\Lambda t{::}K.M \mid M[T]$	Type Abstraction and Application
		$\mid$	$\langle\rangle \mid \langle \ell = M\rangle @ N \mid \mathbf{recTail}(M)$	
		$\mid$	$\mathbf{recHeadLabel}(M) \mid \mathbf{recHeadTerm}(M)$	Records
		$\mid$	$\diamond$	Unit Element
		$\mid$	$\mathbf{if}\ M\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2$	
		$\mid$	$\mathit{true} \mid \mathit{false}$	Booleans
		$\mid$	$\mathbf{if}\ \varphi\ \mathbf{then}\ M\ \mathbf{else}\ N$	Property Test
		$\mid$	$\mathbf{if}\ T :: K\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N$	Kind Case
		$\mid$	$\varepsilon \mid M :: N$	
		$\mid$	$\mathbf{colHead}(M) \mid \mathbf{colTail}(M)$	Collections
		$\mid$	$\mathbf{ref}\ M \mid !M \mid M := N \mid l$	References
		$\mid$	$\boldsymbol{\mu} F{:}T.M$	Recursion

Fig. 5. Syntax of Terms

where $\mathcal{D}$ is a straightforward derivation of $\Gamma_0, \mathbf{headLabel}(t) \equiv L :: \mathsf{Nm} \vdash \mathbf{headType}(t) :: \mathsf{Type}$ and $\mathcal{E}$ is a derivation of $\Gamma_0, \neg\mathbf{headLabel}(t) \equiv L :: \mathsf{Nm} \vdash F(\mathbf{tail}(t)) :: \mathsf{Type}$. To show that $\mathbf{headLabel}(t) \equiv L$ is well-formed we must be able to derive $t :: \{r{::}\mathsf{Rec} \mid \mathsf{nonEmpty}(r)\}$ from $t :: \{r{::}\mathsf{Rec} \mid L \in \mathbf{lab}(r)\}$, which is achieved via the reasoning principles built into our theory of refinements (see Section 4.1). Similarly, the derivation $\mathcal{E}$ requires the ability to conclude that $\mathbf{tail}(t) :: \{r{::}\mathsf{Rec} \mid L \in \mathbf{lab}(r)\}$, using the information that $t :: \{r{::}\mathsf{Rec} \mid L \in \mathbf{lab}(r)\}$ and $\neg\mathbf{headLabel}(t) \equiv L :: \mathsf{Nm}$, 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 terms equipped with the appropriate destructors. Collections are built from the empty collection ε and the concatenation of an element M with a collection N, $M :: N$, with the usual destructors (dubbed $\mathbf{colHead}(M)$ and $\mathbf{colTail}(M)$) that project the head or the tail of such an homogeneous list. We allow for recursive terms via a fixpoint construct $\boldsymbol{\mu} F{:}T.M$, which we enforce to be structural (i.e. identical to the type-level recursion) to simplify the theory, noting that since there are no 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 $\mathbf{if}\ \varphi\ \mathbf{then}\ M\ \mathbf{else}\ N$ and $\mathbf{if}\ T :: K\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N$, respectively. As we have initially stated, our language has general higher-order references, represented with the constructs $\mathbf{ref}\ M$, $!M$ 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-kinded types, according to the typing context Γ.

$$\textsc{(var)}\ \frac{(x{:}T) \in \Gamma \quad \Gamma; S \vdash \quad \Gamma \vdash}{\Gamma \vdash_S x : T} \qquad \textsc{(1I)}\ \frac{\Gamma \vdash}{\Gamma \vdash \diamond : \mathbf{1}} \qquad \textsc{(}{\to}\textsc{I)}\ \frac{\Gamma \vdash_S T :: \mathsf{Type} \quad \Gamma, x{:}T \vdash_S M : U}{\Gamma \vdash_S \lambda x{:}T.M : T \to U}$$

$$\textsc{(}{\to}\textsc{E)}\ \frac{\begin{array}{c}\Gamma \vdash T_1 :: \{t{::}\mathsf{Fun} \mid \mathbf{dom}(t) \equiv T_2 :: \mathcal{K} \wedge \mathbf{img}(t) = U :: \mathcal{K}'\} \\ \Gamma \vdash_S M : T_1 \quad \Gamma \vdash_S N : T_2\end{array}}{\Gamma \vdash_S M\,N : U\{T_1/t\}} \qquad \textsc{(}\forall\textsc{I)}\ \frac{\Gamma \vdash K \quad \Gamma, t{:}K \vdash_S M : T}{\Gamma \vdash_S \Lambda t{::}K.M : \forall t{::}K.T}$$

$$\textsc{(}\forall\textsc{E)}\ \frac{\begin{array}{c}\Gamma \vdash T' :: \{f{::}\mathsf{Gen}_K \mid \mathbf{tmap}(f)\,T \equiv U :: \mathcal{K}\} \\ \Gamma \vdash_S M : T' \quad \Gamma \vdash T :: K \quad \Gamma \vdash U :: \mathcal{K}\end{array}}{\Gamma \vdash_S M[T] : U} \qquad (\langle\rangle I_1)\ \frac{\Gamma \vdash \quad \Gamma; S \vdash}{\Gamma \vdash_S \langle\rangle : \boldsymbol{\langle\rangle}}$$

$$(\langle\rangle I_2)\ \frac{\Gamma \vdash_S L :: \mathsf{Nm} \quad \Gamma \vdash S :: \{t{::}\mathsf{Rec} \mid L \notin \mathbf{lab}(t)\} \quad \Gamma \vdash_S M : T \quad \Gamma \vdash_S N : U}{\Gamma \vdash_S \langle L = M\rangle @N : \boldsymbol{\langle} L : T \boldsymbol{\rangle} @U}$$

$$\textsc{(reclabel)}\ \frac{\Gamma \vdash_S M : U \quad \Gamma \vdash U :: \{t{::}\mathsf{Rec} \mid \mathbf{headLabel}(t) \equiv L :: \mathsf{Nm}\}}{\Gamma \vdash_S \mathbf{recHeadLabel}(M) : L\{U/t\}}$$

$$\textsc{(recterm)}\ \frac{\Gamma \vdash_S M : U \quad \Gamma \vdash U :: \{t{::}\mathsf{Rec} \mid \mathbf{headType}(t) \equiv T :: \mathcal{K}\}}{\Gamma \vdash_S \mathbf{recHeadTerm}(M) : T\{U/t\}} \qquad \textsc{(rectail)}\ \frac{\Gamma \vdash_S M : U \quad \Gamma \vdash U :: \{t{::}\mathsf{Rec} \mid \mathbf{tail}(t) \equiv T :: \mathcal{K}\}}{\Gamma \vdash_S \mathbf{tail}(M) : T\{U/t\}}$$

$$\textsc{(true)}\ \frac{\Gamma \vdash \quad \Gamma; S \vdash}{\Gamma \vdash_S \mathit{true} : \mathsf{Bool}} \qquad \textsc{(false)}\ \frac{\Gamma \vdash \quad \Gamma; S \vdash}{\Gamma \vdash_S \mathit{false} : \mathsf{Bool}} \qquad \textsc{(bool-ite)}\ \frac{\Gamma \vdash_S M : \mathsf{Bool} \quad \Gamma \vdash_S N_1 : T \quad \Gamma \vdash_S N_2 : T}{\Gamma \vdash_S \mathbf{if}\ M\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2 : T}$$

$$\textsc{(emp)}\ \frac{\Gamma \vdash T :: \mathsf{Type} \quad \Gamma; S \vdash}{\Gamma \vdash_S \varepsilon : T^\star} \qquad \textsc{(cons)}\ \frac{\begin{array}{c}\Gamma \vdash U :: \{t{::}\mathsf{Col} \mid \mathbf{colOf}(t) \equiv T :: \mathcal{K}\} \\ \Gamma \vdash_S M : T\{U/t\} \quad \Gamma_S \vdash N : U\end{array}}{\Gamma \vdash_S M :: N : U} \qquad \textsc{(head)}\ \frac{\begin{array}{c}\Gamma \vdash T_c :: \{t{::}\mathsf{Col} \mid \mathbf{colOf}(t) \equiv T :: \mathcal{K}\} \\ \Gamma \vdash M : T_c\end{array}}{\Gamma \vdash \mathbf{colHead}(M) : T}$$

$$\textsc{(tail)}\ \frac{\Gamma \vdash M : T_c \quad \Gamma \vdash T_c :: \{t{::}\mathsf{Col} \mid \mathbf{colOf}(t) \equiv T :: \mathcal{K}\}}{\Gamma \vdash \mathbf{colTail}(M) : T\{T_c/t\}} \qquad \textsc{(loc)}\ \frac{\Gamma \vdash \quad \Gamma; S \vdash \quad S(l) = T}{\Gamma \vdash_S l : \mathbf{ref}\ T}$$

$$\textsc{(ref)}\ \frac{\Gamma \vdash_S M : T}{\Gamma \vdash_S \mathbf{ref}\ M : \mathbf{ref}\ T} \qquad \textsc{(deref)}\ \frac{\begin{array}{c}\Gamma \vdash U :: \{t{::}\mathsf{Ref} \mid \mathbf{refOf}(t) \equiv T :: \mathcal{K}\} \\ \Gamma \vdash_S M : U\end{array}}{\Gamma \vdash_S !M : T\{U/t\}} \qquad \textsc{(assign)}\ \frac{\begin{array}{c}\Gamma \vdash U :: \{t{::}\mathsf{Ref} \mid \mathbf{refOf}(t) \equiv T :: \mathcal{K}\} \\ \Gamma \vdash_S M : U \quad \Gamma \vdash_S N : T\end{array}}{\Gamma \vdash_S M := N : \mathbf{1}}$$

$$\textsc{(prop-ite)}\ \frac{\Gamma \vdash \varphi \quad \Gamma, \varphi \vdash_S M : T_1 \quad \Gamma, \neg\varphi \vdash_S N : T_2}{\Gamma \vdash_S \mathbf{if}\ \varphi\ \mathbf{then}\ M\ \mathbf{else}\ N : \mathbf{if}\ \varphi\ \mathbf{then}\ T_1\ \mathbf{else}\ T_2} \qquad \textsc{(kindcase)}\ \frac{\Gamma \vdash T :: \mathcal{K}' \quad \Gamma \vdash \mathcal{K} \quad \Gamma, t{:}\mathcal{K} \vdash_S M : U \quad \Gamma \vdash_S N : U}{\Gamma \vdash_S \mathbf{if}\ T :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N : U}$$

$$\textsc{(conv)}\ \frac{\Gamma \vdash_S M : U \quad \Gamma \models U \equiv T :: \mathcal{K}}{\Gamma \vdash_S M : T} \qquad \textsc{(fix)}\ \frac{\Gamma, F : T \vdash_S M : T \quad \mathsf{structural}(F, M)}{\Gamma \vdash_S \boldsymbol{\mu} F{:}T.M : T}$$

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:

$$\frac{\Gamma \vdash T_1 :: \{t{::}\mathsf{Fun} \mid \mathbf{dom}(t) \equiv T_2 :: \mathcal{K} \wedge \mathbf{img}(t) \equiv U :: \mathcal{K}'\} \quad \Gamma \vdash_S M : T_1 \quad \Gamma \vdash_S N : T_2}{\Gamma \vdash_S M\, N : U\{T_1/t\}} \ (\to\mathrm{E})$$

Instead of stating that M is of type $T_2 \to U$, 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 $T_2 \to U :: \{f{:}\mathsf{Fun} \mid \mathbf{dom}(t) \equiv T_2 :: \mathcal{K} \wedge \mathbf{img}(t) \equiv U :: \mathcal{K}'\}$ 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

$$\vdash \Lambda s{:}\mathsf{Type}.\Lambda t{:}\{f{::}\mathsf{Fun} \mid \mathbf{dom}(f) \equiv s :: \mathsf{Type} \wedge \mathbf{img}(f) \equiv \mathsf{Bool} :: \mathsf{Type}\}.$$
$$\lambda x{:}t.\lambda y{:}s.(x\, y) : \forall s{:}\mathsf{Type}.\forall t{::}\{f{::}\mathsf{Fun} \mid \mathbf{dom}(f) \equiv s :: \mathsf{Type} \wedge \mathbf{img}(f) \equiv \mathsf{Bool} :: \mathsf{Type}\}.\mathsf{Bool}$$

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,

$$\text{(PROP-ITE)}\ \frac{\Gamma \vdash \varphi \quad \Gamma, \varphi \vdash_S M : T_1 \quad \Gamma, \neg\varphi \vdash_S N : T_2}{\Gamma \vdash_S \mathbf{if}\ \varphi\ \mathbf{then}\ M\ \mathbf{else}\ N : \mathbf{if}\ \varphi\ \mathbf{then}\ T_1\ \mathbf{else}\ T_2}$$

which types the term $\mathbf{if}\ \varphi\ \mathbf{then}\ M\ \mathbf{else}\ N$ with the *type* $\mathbf{if}\ \varphi\ \mathbf{then}\ T_1\ \mathbf{else}\ T_2$ and thus allows for a conditional branching where the types of the branches differ. Rule (KINDCASE) mirrors the equivalent rule for the type-level kind case, typing the term $\mathbf{if}\ T :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N$ 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:

$$\Lambda s{::}\mathsf{Type}.\lambda x{:}s.\mathbf{if}\ s :: \mathsf{Ref}\ \mathbf{as}\ t \Rightarrow (\mathbf{if}\ \mathbf{refOf}(t) \equiv \mathsf{Int} :: \mathsf{Type}\ \mathbf{then}\ !x\ \mathbf{else}\ 0)\ \mathbf{else}\ 0 : \forall s{::}\mathsf{Type}.s \to \mathsf{Int}$$

The function above takes a type s, a term x of that type and, if s is of kind Ref such that s is a reference type for integers (note the use of reflection using destructor $\mathbf{refOf}(-)$ 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 (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.

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 $\{r{::}\mathsf{Rec} \mid L \in r\}$, we define the record selection construct $M.L$ as (for conciseness, let $\mathcal{R} = \{r{::}\mathsf{Rec} \mid L \in \mathbf{lab}(r)\}$):

$$M.L \triangleq (\boldsymbol{\mu} F{:}\forall t :: \mathcal{R}.t \to (t.L).\Lambda t :: \mathcal{R}.\lambda x{:}t.$$
$$\mathbf{if}\ \mathbf{headLabel}(t) \equiv L :: \mathsf{Nm}\ \mathbf{then}\ \mathbf{recHeadTerm}(x)\ \mathbf{else}\ F[\mathbf{tail}(t)](\mathbf{tail}(x)))\ T\ M$$

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 Γ_0 be $F{:}\forall t :: \mathcal{R}.t \to (t.L), x{:}T$):

$$\frac{\mathcal{D} \qquad \Gamma_0 \models \textbf{if headLabel}(T) \equiv L :: \mathsf{Nm}\ \textbf{then headType}(T)\ \textbf{else tail}(T).L \equiv T.L :: \mathsf{Type}}{\Gamma_0 \vdash \textbf{if headLabel}(T) \equiv L :: \mathsf{Nm}\ \textbf{then recHeadTerm}(x)\ \textbf{else}\ F[\textbf{tail}(T)](\textbf{tail}(x)) : T.L}\ (\textsc{conv})$$

with $\mathcal{D}$ a derivation of

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

where T_0 is **if** $(\textbf{headLabel}(T) \equiv L :: \mathsf{Nm})$ **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 $\Gamma_0, \textbf{headLabel}(T) \equiv L :: \mathsf{Nm} \vdash \textbf{recHeadTerm}(x) : \textbf{headType}(T)$, which is derivable from $x{:}T$ and $T :: \{r{::}\mathsf{Rec} \mid \textbf{headType}(r) \equiv \textbf{headType}(r) :: \mathsf{Type}\}$ – the latter following from refinement and reflexivity of equality – via typing rule (recterm).

The **else** branch requires showing that $\Gamma_0, \neg\textbf{headLabel}(T) \equiv L :: \mathsf{Nm} \vdash F[\textbf{tail}(T)](\textbf{tail}(x)) : \textbf{tail}(T).L$, which is derivable from $F : \forall t :: \mathcal{R}.t \to (t.L)$ and $x{:}T$ as follows: $\textbf{tail}(T) :: \mathcal{R}$ is follows from $\neg\textbf{headLabel}(T) \equiv L$ and $T :: \mathcal{R}$ (see Section 4.1), thus $F[\textbf{tail}(T)] : \textbf{tail}(T) \to \textbf{tail}(T).L$. Since $\textbf{tail}(x) : \textbf{tail}(T)$ from $x : T$ and $T :: \{r{::}\mathsf{Rec} \mid \textbf{tail}(t) \equiv \textbf{tail}(t) :: \mathsf{Rec}\}$ 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:

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash M : T \quad \Gamma \vdash T :: \{r{::}\mathsf{Rec} \mid L \in \textbf{lab}(r)\}}{\Gamma \vdash M.L : T.L}$$

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 β-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 β-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

$$R \equiv \langle\rangle \lor R \equiv \langle \mathbf{headLabel}(R) : \mathbf{headType}(R)\rangle @\mathbf{tail}(R) \qquad (\textsc{Rec-EmpOrCons})$$

$$R \equiv \langle\rangle \lor \mathbf{headLabel}(R) \notin \mathbf{lab}(\mathbf{tail}(R)) \qquad (\textsc{Rec-DisjointLabels})$$

$$L \notin \mathbf{lab}(\langle\rangle) \qquad (\textsc{Lab-NotInEmpty})$$

$$L \in \mathbf{lab}(R) \Leftrightarrow (L \equiv \mathbf{headLabel}(R) \lor L \in \mathbf{lab}(\mathbf{tail}(R))) \qquad (\textsc{Lab-InHeadTail})$$

$$L \equiv L' \Leftrightarrow N{+}{+}L \equiv N{+}{+}L' \qquad (\textsc{LabConcatEq})$$

$$\mathbf{lab}(R) = \mathbf{lab}(L) \land L \in \mathbf{lab}(R) \Rightarrow L \in \mathbf{lab}(L) \qquad (\textsc{LabelSet-InEq})$$

$$L \in \mathbf{lab}(S) \Leftrightarrow N{+}{+}L \in N{+}{+}\mathbf{lab}(S) \qquad (\textsc{LabConcat-SetConcat})$$

$$\mathbf{lab}(R) = \mathbf{lab}(L) \Leftrightarrow N{+}{+}\mathbf{lab}(R) = N{+}{+}\mathbf{lab}(L) \qquad (\textsc{LabelSet-Concat})$$

Fig. 7. Axiom Schemas for Record Types and Labels

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, $\mathbf{lab}(R)$ 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. $N{+}{+}\mathbf{lab}(R)$, denoting the set obtained by prefixing N to all labels in $\mathbf{lab}(R)$). 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 $\mathsf{nonEmpty}(R)$ is defined as notation for $\neg(R \equiv \langle\rangle)$.

Thus, axiom (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 (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 (Lab-NotInEmpty) and (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 iff it is the label at the head of the record or in the label set of the tail of R; axiom (LabConcatEq) specifies label or name concatenation; axiom (LabelSet-InEq) allows for combined reasoning of inclusion and label set equality; finally, the axioms (LabConcat-SetConcat) and (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 $L \in \mathbf{lab}(T)$ and $\neg(\mathbf{headLabel}(T) \equiv L)$ we can derive that $L \in \mathbf{lab}(\mathbf{tail}(T))$ through axiom (Lab-InHeadTail) and some basic propositional reasoning. Similarly, in Example 3.1 we derive $\mathsf{nonEmpty}(t)$ from $L \in \mathbf{lab}(t)$ via axiom (Lab-NotInEmpty) and propositional reasoning. In the XML table example of Section 2, we derive $\mathsf{nonEmpty}(TT)$ from $\mathsf{nonEmpty}(R)$ and $\mathbf{lab}(TT) = \mathbf{lab}(R)$ via (LabelSet-InEq).

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 (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:

$$T_v, S_v \quad ::= \quad \lambda t{::}K.T \mid \forall t{::}K.T \mid \ell \mid \langle\!\rangle \mid \langle\!\ell : T_v\rangle\!@S_v \mid T_v{}^{\star} \mid \mathbf{ref}\, T_v \mid T_v \rightarrow S_v \mid \bot \mid \mathsf{Bool} \mid \mathbf{1} \mid t$$

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

$$v, v' \quad ::= \quad \mathit{true} \mid \mathit{false} \mid \langle\rangle \mid \langle \ell = v\rangle@v' \mid \lambda x{:}T.\, M \mid \Lambda t{::}K.M \mid v :: v' \mid \varepsilon \mid l$$

Values consist of the booleans *true* and *false* (extensions to other basic data types are straightforward as usual); the empty record $\langle\rangle$; the non-empty record that assigns fields to values, $\langle \ell = v\rangle@v'$; the empty collection, ε, and the non-empty collection of values, $v :: v'$; as well as type and λ-abstraction. For convenience of notation we write $\langle\!\ell_1 : T_1, \ldots, \ell_n : T_n\!\rangle$ for $\langle\!\ell_1 : T_1\!\rangle@\ldots@\langle\!\ell_n : T_n\!\rangle@\langle\!\rangle$, and similarly $\langle \ell_1 = M_1, \ldots, \ell_n = M_n\rangle$ for $\langle \ell_1 = M_1\rangle@\cdots@\langle \ell_n = M_n\rangle@\langle\rangle$.

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

$$\text{(R-RecConsL)}\quad \frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; \langle \ell = M\rangle@N\rangle \longrightarrow \langle H'; \langle \ell = M'\rangle@N\rangle}$$

where the record field labelled by ℓ is evaluated (and the resulting modifications in store H to H' 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.

The three rules for the record destructors project the appropriate record element as needed. The treatment of references also standard, with rule (R-RefV) creating a new location l in the store which then stores value v; rule (R-DerefV) querying the store for the contents of location l; and rule for (R-AssignV) replacing the contents of location l with v and returning v. Rules (R-PropT) and (R-PropF) are the only ones that appeal to the entailment relation for refinements, making use of the running environment Γ which is threaded through the reduction rules straightforwardly. Similarly, rules (R-KindL) and (R-KindR) mimic the equality rules of the kind case construct, testing the kind of type T against $\mathcal{K}$.

$$\frac{}{\langle H; \mathbf{recHeadLabel}(\langle \ell = v \rangle @ v') \rangle \longrightarrow \langle H; \ell \rangle}\ (\text{R-RecHdLabV}) \qquad \frac{}{\langle H; \mathbf{recHeadTerm}(\langle \ell = v \rangle @ v') \rangle \longrightarrow \langle H; v \rangle}\ (\text{R-RecHdValV})$$

$$\frac{}{\langle H; \mathbf{recTail}(\langle \ell = v \rangle @ v') \rangle \longrightarrow \langle H; v' \rangle}\ (\text{R-RecTailV}) \qquad \frac{l \notin \mathsf{dom}(H)}{\langle H; \mathbf{ref}\ v \rangle \longrightarrow \langle H[l \mapsto v]; l \rangle}\ (\text{R-RefV}) \qquad \frac{H(l) = v}{\langle H; !l \rangle \longrightarrow \langle H; v \rangle}\ (\text{R-DerefV})$$

$$\frac{}{\langle H; l := v \rangle \longrightarrow \langle H[l \mapsto v]; \diamond \rangle}\ (\text{R-AssignV}) \qquad \frac{\Gamma \models \varphi}{\langle H; \mathbf{if}\ \varphi\ \mathbf{then}\ M\ \mathbf{else}\ N \rangle \longrightarrow \langle H; M \rangle}\ (\text{R-PropT})$$

$$\frac{\Gamma \models \neg\varphi}{\langle H; \mathbf{if}\ \varphi\ \mathbf{then}\ M\ \mathbf{else}\ N \rangle \longrightarrow \langle H; N \rangle}\ (\text{R-PropF}) \qquad \frac{}{\langle H; \mathbf{if}\ \mathit{true}\ \mathbf{then}\ M\ \mathbf{else}\ N \rangle \longrightarrow \langle H; M \rangle}\ (\text{R-IfT})$$

$$\frac{}{\langle H; \mathbf{if}\ \mathit{false}\ \mathbf{then}\ M\ \mathbf{else}\ N \rangle \longrightarrow \langle H; N \rangle}\ (\text{R-IfF}) \qquad \frac{}{\langle H; \boldsymbol{\mu} F{:}T.M \rangle \longrightarrow \langle H; M\{\boldsymbol{\mu} F{:}T.M/F\} \rangle}\ (\text{R-Fix})$$

$$\frac{T \rightarrow T'}{\langle H; (\Lambda t{::}K.M)[T] \rangle \longrightarrow \langle H; (\Lambda t{::}K.M)[T'] \rangle}\ (\text{R-TAppTRed})$$

$$\frac{}{\langle H; (\Lambda t{::}K.\,M)[T_v] \rangle \longrightarrow \langle H; M\{T_v/t\} \rangle}\ (\text{R-TApp}) \qquad \frac{}{\langle H; (\lambda x : T.M)\ v \rangle \longrightarrow \langle H; M\{v/x\} \rangle}\ (\text{R-AppV})$$

$$\frac{}{\langle H; \mathbf{colHead}(v :: v') \rangle \longrightarrow \langle H; v \rangle}\ (\text{R-ColHdV}) \qquad \frac{}{\langle H; \mathbf{colTail}(v :: v') \rangle \longrightarrow \langle H; v' \rangle}\ (\text{R-ColTlV})$$

$$\frac{T \longrightarrow T'}{\langle H; \mathbf{if}\ T :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N \rangle \longrightarrow \langle H; \mathbf{if}\ T' :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N \rangle}\ (\text{R-KindTRed})$$

$$\frac{\Gamma \vdash T_v :: \mathcal{K}}{\langle H; \mathbf{if}\ T_v :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N \rangle \longrightarrow \langle H; M\{T/t\} \rangle}\ (\text{R-KindL}) \qquad \frac{\Gamma \vdash T :: K_0 \quad \Gamma \vdash K_0 \not\equiv \mathcal{K}}{\langle H; \mathbf{if}\ T_v :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N \rangle \longrightarrow \langle H; N \rangle}\ (\text{R-KindR})$$

Fig. 8. Operational Semantics (Excerpt)

5.1 Metatheory

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 λ-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_ω [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 φ and ψ 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* $\Gamma, t{:}K, \Gamma' \models \varphi$ *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* φ*;*
Functionality: *If* $\Gamma \models T \equiv S :: K$ *and* $\Gamma, t : K, \Gamma' \vdash \varphi$ *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) *If* $\Gamma \vdash T :: K$ *and* $\Gamma, t{:}K, \Gamma' \vdash \mathcal{J}$ *then* $\Gamma, \Gamma'\{T/t\} \vdash \mathcal{J}\{T/t\}$*.*
(b) *If* $\Gamma \vdash M : T$ *and* $\Gamma, x{:}T, \Gamma' \vdash N : S$ *then* $\Gamma, \Gamma' \vdash N\{M/x\} : S$*.*

Lemma 5.2 (Context Conversion).

(a) *Let* $\Gamma, x{:}T \vdash$ *and* $\Gamma \vdash T' :: K$*. If* $\Gamma, x{:}T \vdash \mathcal{J}$ *and* $\Gamma \models T \equiv T' :: K$ *then* $\Gamma, x{:}T' \vdash \mathcal{J}$*.*
(b) *Let* $\Gamma, t{:}K \vdash$ *and* $\Gamma \vdash K'$*. If* $\Gamma, t{:}K \vdash \mathcal{J}$ *and* $\Gamma \vdash K \equiv K'$ *then* $\Gamma, t{:}K' \vdash \mathcal{J}$*.*

Theorem 5.3 (Validity for Eqality).

(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* $\Gamma \vdash \psi$

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

Lemma 5.6 (Functionality of Eqality). *Assume* $\Gamma \models T_0 \equiv S_0 :: K$*:*

(a) *If* $\Gamma, t{:}K \models T \equiv S :: K'$ *then* $\Gamma \models T\{T_0/t\} \equiv S\{S_0/t\} :: K'\{T_0/t\}$.
(b) *If* $\Gamma, t{:}K \vdash K_1 \equiv K_2$ *then* $\Gamma \vdash K_1\{T_0/t\} \equiv K_2\{S_0/t\}$.
(c) *If* $\Gamma, t{:}K \vdash \varphi \equiv \psi$ *then* $\Gamma \vdash \varphi\{T_0/t\} \equiv \psi\{S_0/t\}$.

With functionality and the previous properties we can then establish the so-called validity theorem (Theorem 5.7) for our theory, which is a general well-formedness property of the judgments of the language. Validity is crucial in establishing the various inversion principles (note that the inversion principles become non-trivial due to the closure of typing and kinding under equality) necessary to show uniqueness of types and kinds (Theorem 5.8) and type preservation (Theorem 5.9). The inversion principles can be found in Appendix C.

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* $\Gamma \vdash T :: \mathsf{Type}$.

Theorem 5.8 (Unicity of Types and Kinds).
(1) *If* $\Gamma \vdash M : T$ *and* $\Gamma \vdash M : S$ *then* $\Gamma \vdash T \equiv S :: K$ *and* $\Gamma \vdash K \leq \mathsf{Type}$.
(2) *If* $\Gamma \vdash T :: K$ *and* $\Gamma \vdash T :: K'$ *then* $\Gamma \vdash K \leq K'$ *or* $\Gamma \vdash K' \leq K$.

In order to state type preservation we first define the usual notion of well-typed store, written $\Gamma \vdash_S H$, denoting that for every l in $\mathsf{dom}(H)$ we have that $\Gamma \vdash_S l : \mathbf{ref}\, T$ with $\cdot \vdash H(l) : T$. We write $S \subseteq S'$ to denote that S' is an extension of S (i.e. it preserves the location typings of S).

Theorem 5.9 (Type Preservation). *Let* $\Gamma \vdash_S M : T$ *and* $\Gamma \vdash_s H$. *If* $\langle H; M\rangle \longrightarrow \langle H'; M'\rangle$ *then there exists* S' *such that* $S \subseteq S'$, $\Gamma \vdash_{S'} H'$ *and* $\Gamma \vdash_{S'} M' : T$.

Finally, progress can be established in a fairly direct manner (relying on a straightforward notion of progress for the type reduction relation). The main interesting aspect is that progress relies crucially on the decidability of entailment due to the term-level and type-level predicate test construct.

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

Theorem 5.11 (Progress). *Let* $\cdot \vdash_S M : T$ *and* $\cdot \vdash_S H$. *Then either* M *is a value or there exists* S' *and* M' *such that* $\langle H; M\rangle \longrightarrow \langle H'; M'\rangle$.

6 RELATED WORK

To the best of our knowledge, ours is the first work to explore the concept of refinement kinds and illustrate their expressiveness as a convenient programming language feature that cleanly integrates statically typed meta-programming features such as type reflection, ad-hoc polymorphism, and type-level computation.

The concept of refinement kind is a natural extension of the well-known notion of refinement type [Bengtson et al. 2011; Rondon et al. 2008; Vazou et al. 2013], which effectively extends type specifications with (SMT decidable) logical assertions. Refinement types have been applied to various verification domains such as security [Bengtson et al. 2011] or the verification of data-structures [Kawaguchi et al. 2009; Xi and Pfenning 1998], and are being incorporated in full-fledged programming languages, e.g., ML [Freeman and Pfenning 1991] Haskell [Vazou et al. 2014], F* [Swamy et al. 2011], JavaScript [Vekris et al. 2016].

With the aim of supporting common meta-programming idioms in the domain of web programming, [Chlipala 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 $\mathbf{if}\ \varphi\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$ and $\mathbf{if}\ T :: K\ \mathbf{as}\ t \Rightarrow e_1\ \mathbf{else}\ e_2$ 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 λ-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., typestate 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

Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. 1991. Dynamic Typing in a Statically Typed Language. *ACM Trans. Program. Lang. Syst.* 13, 2 (1991), 237–268. https://doi.org/10.1145/103135.103138

Thorsten Altenkirch and Conor McBride. 2002. Generic Programming within Dependently Typed Programming. In *Generic Programming, IFIP TC2/WG2.1 Working Conference on Generic Programming, July 11-12, 2002, Dagstuhl, Germany (IFIP Conference Proceedings)*, Jeremy Gibbons and Johan Jeuring (Eds.), Vol. 243. Kluwer, 1–20.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. 2011. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.* (2011).

Cristiano Calcagno, Eugenio Moggi, and Tim Sheard. 2003. Closed types for a safe imperative MetaML. *J. Funct. Program.* 13, 3 (2003), 545–571. https://doi.org/10.1017/S0956796802004598

Luca Cardelli. 1991. Typeful Programming. *IFIP State-of-the-Art Reports: Formal Description of Programming Concepts* (1991), 431–507.

Adam Chlipala. 2010. Ur: statically-typed metaprogramming with type-level record computation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 122–133. https://doi.org/10.1145/1806596.1806612

Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. 1986. *Implementing mathematics with the Nuprl proof development system.* Prentice Hall. http://dl.acm.org/citation.cfm?id=10510

CoqDevelopmentTeam. 2004. *The Coq proof assistant reference manual.* LogiCal Project. http://coq.inria.fr Version 8.0.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Manuel Fähndrich, Michael Carbin, and James R. Larus. 2006. Reflective program generation with patterns. In *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings*, Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen (Eds.). ACM, 275–284. https://doi.org/10.1145/1173706.1173748

Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, David S. Wise (Ed.). ACM, 268–277. https://doi.org/10.1145/113445.113468

Jean-Yves Girard. 1986. The system F of variable types, fifteen years later. *Theoretical Computer Science* 45 (1986), 159 – 192. https://doi.org/10.1016/0304-3975(86)90044-7

Robert Harper, Furio Honsell, and Gordon D. Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (1993), 143–184.

Robert Harper and Frank Pfenning. 2005. On equivalence and canonical forms in the LF type theory. *ACM Trans. Comput. Log.* 6, 1 (2005), 61–101.

Robert Harper and Benjamin C. Pierce. 1991. A Record Calculus Based on Symmetric Concatenation. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, David S. Wise (Ed.). ACM Press, 131–142. https://doi.org/10.1145/99583.99603

Martin Hofmann. 1997. *Extensional constructs in intensional type theory.* Springer.

Shan Shan Huang and Yannis Smaragdakis. 2008. Expressive and safe static reflection with MorphJ. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008.* 79–89.

Ming Kawaguchi, Patrick Maxim Rondon, and Ranjit Jhala. 2009. Type-based data structure verification. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 304–315. https://doi.org/10.1145/1542476.1542510

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology.

Tomas Petricek, Gustavo Guerra, and Don Syme. 2016. Types from data: making structured data first-class citizens in F#. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 477–490. https://doi.org/10.1145/2908080.2908115

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008.* 159–169.

John M. Rushby, Sam Owre, and Natarajan Shankar. 1998. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Trans. Software Eng.* 24, 9 (1998), 709–720. https://doi.org/10.1109/32.713327

Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More Sound Static Handling of Java Reflection. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings.* 485–503.

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 266–278. https://doi.org/10.1145/2034773.2034811

Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings.* 209–228. https://doi.org/10.1007/978-3-642-37036-6_13

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 269–282. https://doi.org/10.1145/2628136.2628161

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 310–325. https://doi.org/10.1145/2908080.2908110

Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with explicit kind equality. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013.* 275–286. https://doi.org/10.1145/2500365.2500599

Hongwei Xi. 2007. Dependent ML An approach to practical programming with dependent types. *J. Funct. Program.* 17, 2 (2007), 215–286. https://doi.org/10.1017/S0956796806006216

Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, Jack W. Davidson, Keith D. Cooper, and A. Michael Berman (Eds.). ACM, 249–257. https://doi.org/10.1145/277650.277732

Appendix

Refinement Kinds
A Theory of Type-Safe Meta-Programming

Additional definitions and proofs of the main materials.

A FULL SYNTAX, JUDGMENTS AND RULES

We define the syntax of kinds K, K', refinements φ, φ', types T, S, R, and terms M, N below. We assume countably infinite sets of type variables $\mathcal{X}$, names $\mathcal{N}$ and term variables $\mathcal{V}$. 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'	$::=$	$\mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K'$	Refined and Dependent Kinds
	$\mathcal{K}$	$::=$	$\mathsf{Rec} \mid \mathsf{Col} \mid \mathsf{Fun} \mid \mathsf{Ref} \mid \mathsf{Nm}$	Base Kinds
		$\mid$	$\mathsf{Type} \mid \mathsf{Gen}_K$	
Types	T, S, R	$::=$	$t \mid \lambda t{::}K.T \mid T\,S$	Type-level Functions
		$\mid$	$\boldsymbol{\mu} F : (\Pi t{:}K.\,K').\lambda t{::}K.\,T$	Structural Recursion
		$\mid$	$\forall t{::}K.T \mid \mathbf{tmap}(T)\,S$	Polymorphism
		$\mid$	$L \mid \langle\rangle \mid \langle L : T\rangle @ S$	Record Type constructors
		$\mid$	$\mathbf{headLabel}(T) \mid \mathbf{headType}(T) \mid \mathbf{tail}(T)$	Record Type destructors
		$\mid$	$T^\star \mid \mathbf{colOf}(T)$	Collection Types
		$\mid$	$\mathbf{ref}\,T \mid \mathbf{refOf}(T)$	Reference Types
		$\mid$	$T \rightarrow S \mid \mathbf{dom}(T) \mid \mathbf{img}(T)$	Function Types
		$\mid$	$\mathbf{if}\ T :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow S\ \mathbf{else}\ U$	Kind Case
		$\mid$	$\mathbf{if}\ \varphi\ \mathbf{then}\ T\ \mathbf{else}\ S$	Property Test
		$\mid$	$\bot \mid \top$	Empty and Top Types
		$\mid$	$\mathsf{Bool} \mid \mathbf{1} \mid \ldots$	Basic Data Types
Refinements	φ, ψ	$::=$	$P(T_1, \ldots, T_n)$	Type Predicates
		$\mid$	$\varphi \supset \psi \mid \varphi \wedge \psi \mid \ldots$	Propositional Logic
		$\mid$	$T \equiv S :: K$	Equality

Terms	M, N	$::=$	$x \mid \lambda x{:}T.M \mid M\,N$	Functions
		$\mid$	$\Lambda t{::}K.M \mid M[T]$	Type Abstraction and Application
		$\mid$	$\langle\rangle \mid \langle \ell = M\rangle @ N \mid \mathbf{recTail}(M)$	
		$\mid$	$\mathbf{recHeadLabel}(M) \mid \mathbf{recHeadTerm}(M)$	Records
		$\mid$	$\diamond$	Unit Element
		$\mid$	$\mathbf{if}\ M\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2$	
		$\mid$	$\mathit{true} \mid \mathit{false}$	Booleans
		$\mid$	$\mathbf{if}\ \varphi\ \mathbf{then}\ M\ \mathbf{else}\ N$	Property Test
		$\mid$	$\mathbf{if}\ T :: K\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N$	Kind Case
		$\mid$	$\varepsilon \mid M :: N$	
		$\mid$	$\mathbf{colHead}(M) \mid \mathbf{colTail}(M)$	Collections
		$\mid$	$\mathbf{ref}\,M \mid\ !M \mid M := N \mid l$	References
		$\mid$	$\boldsymbol{\mu} F{:}T.M$	Recursion

A.1 Kinding and Typing

Our type theory is defined by the following judgments:

$\Gamma \vdash$	Γ is a well-formed context
$\Gamma \vdash K$	K is a well-formed kind under the assumptions in Γ
$\Gamma \vdash \varphi$	Refinement φ is well-formed under the assumptions in Γ
$\Gamma \vdash T :: K$	Type T is a (well-formed) type of kind K under the assumptions in Γ
$\Gamma \vdash_S M : T$	Term M has type T under the assumptions in Γ and store typing S
$\Gamma \models \varphi$	Refinement φ holds under the assumptions in Γ
$\Gamma \vdash \varphi \equiv \psi$	Refinements φ and ψ 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:

$$\begin{array}{lcl}
\textbf{headLabel} & :: & \Pi t{:}\{r{::}\mathsf{Rec} \mid \mathsf{nonEmpty}(r)\}.\mathsf{Nm} \\
\textbf{headType} & :: & \Pi t{:}\{r{::}\mathsf{Rec} \mid \mathsf{nonEmpty}(r)\}.\mathsf{Type} \\
\textbf{tail} & :: & \Pi t{:}\{r{::}\mathsf{Rec} \mid \mathsf{nonEmpty}(r)\}.\mathsf{Rec} \\
\textbf{refOf} & :: & \Pi t{:}\mathsf{Ref}.\mathsf{Type} \\
\textbf{colOf} & :: & \Pi t{:}\mathsf{Col}.\mathsf{Type} \\
\textbf{dom} & :: & \Pi t{:}\mathsf{Fun}.\mathsf{Type} \\
\textbf{img} & :: & \Pi t{:}\mathsf{Fun}.\mathsf{Type} \\
\textbf{tmap} & :: & \Pi t{:}\mathsf{Gen}_K.\Pi s{:}K.\mathsf{Type}
\end{array}$$

We write $elim_K(T)$ to range over elimination forms for a given (base) kind K applied to type T.

Context Well-formedness.

$$\frac{\Gamma \vdash K \quad \Gamma \vdash}{\Gamma, t : K \vdash} \qquad \frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash}{\Gamma, x : T \vdash} \qquad \frac{\Gamma \vdash \varphi \quad \Gamma \vdash}{\Gamma, \varphi \vdash} \qquad \frac{\Gamma; S \vdash \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma; S, l : T \vdash} \qquad \frac{}{\cdot \vdash}$$

$$\frac{}{\Gamma; \cdot \vdash}$$

Kind well-formedness.

$$\frac{\Gamma \vdash \quad K \in \{\mathsf{Rec}, \mathsf{Col}, \mathsf{Fun}, \mathsf{Ref}, \mathsf{Nm}, \mathsf{Type}\}}{\Gamma \vdash K} \qquad \frac{\Gamma \vdash K \quad \Gamma, t{:}K \vdash K'}{\Gamma \vdash \Pi t{:}K.K'}$$

$$\frac{\Gamma \vdash K}{\Gamma \vdash \mathsf{Gen}_K} \qquad \frac{\Gamma \vdash \mathcal{K} \quad \Gamma, t{:}\mathcal{K} \vdash \varphi}{\Gamma \vdash \{t{::}\mathcal{K} \mid \varphi\}}$$

Refinement Well-formedness. We presupose a signature Σ 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.

$$\frac{P : K_1, \ldots, K_n \in \Sigma \quad \forall i \in \{1, \ldots, n\}. \Gamma \vdash T_i :: K_i}{\Gamma \vdash P(T_1, \ldots, T_n)} \quad + \text{ Well-formedness of propositional logic formulas}$$

$$\frac{\Gamma \vdash T :: K \quad \Gamma \vdash S :: K}{\Gamma \vdash T \equiv S :: K}$$

Refinement Satisfiability.

Propositional Logic

$$\frac{\Gamma \models T \equiv S :: K \quad \Gamma, x : K \vdash \varphi \quad \Gamma \models \varphi\{T/x\}}{\Gamma \models \varphi\{S/x\}} \ (\textsc{EqElim})$$

Kinding.

$$\frac{t{:}K \in \Gamma \quad \Gamma \vdash}{\Gamma \vdash t :: K} \qquad \frac{\Gamma \vdash T :: K \quad \Gamma \vdash K \leq K'}{\Gamma \vdash T :: K'} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \top :: \mathsf{Type}}$$

$$\frac{\Gamma \vdash T :: \Pi t{:}K.K' \quad \Gamma \vdash S :: K}{\Gamma \vdash T\, S :: K'\{S/t\}} \qquad \frac{\Gamma \vdash K \quad \Gamma, t{:}K \vdash T :: K'}{\Gamma \vdash \lambda t{::}K.T :: \Pi t{:}K.K'}$$

$$\frac{\Gamma \vdash K \quad \Gamma, t{:}K \vdash T :: \mathcal{K}}{\Gamma \vdash \forall t{::}K.T :: \mathsf{Gen}_K} \qquad \frac{\Gamma \vdash \ell \in \mathcal{N}}{\Gamma \vdash \ell :: \mathsf{Nm}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Bool} :: \mathsf{Type}}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \langle\rangle :: \mathsf{Rec}} \qquad \frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t{::}\mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \vdash \langle L : T\rangle @ S :: \mathsf{Rec}}$$

$$\frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \mathcal{K}'}{\Gamma \vdash T \to S :: \mathsf{Fun}} \qquad \frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T^\star :: \mathsf{Col}} \qquad \frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash \mathbf{ref}\, T :: \mathsf{Ref}}$$

$$\frac{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid elim_K(t) \equiv T' :: K'\} \quad \Gamma \vdash T'\{T/t\} :: K'\{T/t\}}{\Gamma \vdash elim_{\mathcal{K}}(T) :: K'\{T/t\}}$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma, \varphi \vdash T :: K \quad \Gamma, \neg\varphi \vdash S :: K}{\Gamma \vdash \mathbf{if}\, \varphi\, \mathbf{then}\, T\, \mathbf{else}\, S :: K} \qquad \frac{\Gamma \vdash \mathcal{K} \quad \Gamma \vdash T :: \mathcal{K}'' \quad \Gamma, t{:}\mathcal{K} \vdash S :: K' \quad \Gamma \vdash U :: K'}{\Gamma \vdash \mathbf{if}\, T :: \mathcal{K}\, \mathbf{as}\, t \Rightarrow S\, \mathbf{else}\, U :: K'}$$

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

$$\frac{\Gamma \models \bot \quad \Gamma \vdash K}{\Gamma \vdash \bot :: K} \qquad \frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{:}\mathcal{K} \mid \varphi\}}$$

Sub-kinding.

$$\frac{\Gamma \vdash K \equiv K'}{\Gamma \vdash K \leq K'} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathcal{K} \leq \mathsf{Type}}$$

$$\frac{\Gamma \vdash \mathcal{K} \quad \Gamma, t{:}\mathcal{K} \vdash \varphi}{\Gamma \vdash \{t{::}\mathcal{K} \mid \varphi\} \leq \mathcal{K}} \qquad \frac{\Gamma \vdash \mathcal{K} \leq \mathcal{K}' \quad \Gamma, t{:}\mathcal{K}' \models \varphi \equiv \varphi'}{\Gamma \vdash \{t{::}\mathcal{K} \mid \varphi\} \leq \{t{::}\mathcal{K}' \mid \varphi'\}}$$

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.

$$\textsc{(var)}\ \frac{(x{:}T) \in \Gamma \quad \Gamma; S \vdash \quad \Gamma \vdash}{\Gamma \vdash_S x : T} \qquad (1\mathrm{I})\ \frac{\Gamma \vdash}{\Gamma \vdash \diamond : \mathbf{1}} \qquad (\to\mathrm{I})\ \frac{\Gamma \vdash_S T :: \mathsf{Type} \quad \Gamma, x{:}T \vdash_S M : U}{\Gamma \vdash_S \lambda x{:}T.M : T \to U}$$

$$(\to\mathrm{E})\ \frac{\Gamma \vdash T_1 :: \{t{::}\mathsf{Fun} \mid \mathbf{dom}(t) \equiv T_2 :: \mathcal{K} \wedge \mathbf{img}(t) = U :: \mathcal{K}'\} \quad \Gamma \vdash_S M : T_1 \quad \Gamma \vdash_S N : T_2}{\Gamma \vdash_S M\,N : U\{T_1/t\}} \qquad (\forall\mathrm{I})\ \frac{\Gamma \vdash K \quad \Gamma, t{:}K \vdash_S M : T}{\Gamma \vdash_S \Lambda t{::}K.M : \forall t{::}K.T}$$

$$(\forall\mathrm{E})\ \frac{\Gamma \vdash T' :: \{f{::}\mathsf{Gen}_K \mid \mathbf{tmap}(f)\,T \equiv U :: \mathcal{K}\} \quad \Gamma \vdash_S M : T' \quad \Gamma \vdash T :: K \quad \Gamma \vdash U :: \mathcal{K}}{\Gamma \vdash_S M[T] : U} \qquad (\langle\rangle I_1)\ \frac{\Gamma \vdash \quad \Gamma; S \vdash}{\Gamma \vdash_S \langle\rangle : \langle\rangle}$$

$$(\langle\rangle I_2)\ \frac{\Gamma \vdash_S L :: \mathsf{Nm} \quad \Gamma \vdash S :: \{t{::}\mathsf{Rec} \mid L \notin \mathbf{lab}(t)\} \quad \Gamma \vdash_S M : T \quad \Gamma \vdash_S N : U}{\Gamma \vdash_S \langle L = M\rangle@N : \langle L : T\rangle@U}$$

$$\textsc{(recLabel)}\ \frac{\Gamma \vdash_S M : U \quad \Gamma \vdash U :: \{t{::}\mathsf{Rec} \mid \mathbf{headLabel}(t) \equiv L :: \mathsf{Nm}\}}{\Gamma \vdash_S \mathbf{recHeadLabel}(M) : L\{U/t\}}$$

$$\textsc{(recTerm)}\ \frac{\Gamma \vdash_S M : U \quad \Gamma \vdash U :: \{t{::}\mathsf{Rec} \mid \mathbf{headType}(t) \equiv T :: \mathcal{K}\}}{\Gamma \vdash_S \mathbf{recHeadTerm}(M) : T\{U/t\}} \qquad \textsc{(recTail)}\ \frac{\Gamma \vdash_S M : U \quad \Gamma \vdash U :: \{t{::}\mathsf{Rec} \mid \mathbf{tail}(t) \equiv T :: \mathcal{K}\}}{\Gamma \vdash_S \mathbf{tail}(M) : T\{U/t\}}$$

$$\textsc{(true)}\ \frac{\Gamma \vdash \quad \Gamma; S \vdash}{\Gamma \vdash_S \mathit{true} : \mathsf{Bool}} \qquad \textsc{(false)}\ \frac{\Gamma \vdash \quad \Gamma; S \vdash}{\Gamma \vdash_S \mathit{false} : \mathsf{Bool}} \qquad \textsc{(bool-ite)}\ \frac{\Gamma \vdash_S M : \mathsf{Bool} \quad \Gamma \vdash_S N_1 : T \quad \Gamma \vdash_S N_2 : T}{\Gamma \vdash_S \mathbf{if}\ M\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2 : T}$$

$$\textsc{(emp)}\ \frac{\Gamma \vdash T :: \mathsf{Type} \quad \Gamma; S \vdash}{\Gamma \vdash_S \varepsilon : T^{\star}} \qquad \textsc{(cons)}\ \frac{\Gamma \vdash U :: \{t{::}\mathsf{Col} \mid \mathbf{colOf}(t) \equiv T :: \mathcal{K}\} \quad \Gamma \vdash_S M : T\{U/t\} \quad \Gamma_S \vdash N : U}{\Gamma \vdash_S M :: N : U} \qquad \textsc{(head)}\ \frac{\Gamma \vdash T_c :: \{t{::}\mathsf{Col} \mid \mathbf{colOf}(t) \equiv T :: \mathcal{K}\} \quad \Gamma \vdash M : T_c}{\Gamma \vdash \mathbf{colHead}(M) : T}$$

$$\textsc{(tail)}\ \frac{\Gamma \vdash M : T_c \quad \Gamma \vdash T_c :: \{t{::}\mathsf{Col} \mid \mathbf{colOf}(t) \equiv T :: \mathcal{K}\}}{\Gamma \vdash \mathbf{colTail}(M) : T\{T_c/t\}} \qquad \textsc{(loc)}\ \frac{\Gamma \vdash \quad \Gamma; S \vdash \quad S(l) = T}{\Gamma \vdash_S l : \mathbf{ref}\ T}$$

$$\textsc{(ref)}\ \frac{\Gamma \vdash_S M : T}{\Gamma \vdash_S \mathbf{ref}\ M : \mathbf{ref}\ T} \qquad \textsc{(deref)}\ \frac{\Gamma \vdash U :: \{t{::}\mathsf{Ref} \mid \mathbf{refOf}(t) \equiv T :: \mathcal{K}\} \quad \Gamma \vdash_S M : U}{\Gamma \vdash_S\ !M : T\{U/t\}} \qquad \textsc{(assign)}\ \frac{\Gamma \vdash U :: \{t{::}\mathsf{Ref} \mid \mathbf{refOf}(t) \equiv T :: \mathcal{K}\} \quad \Gamma \vdash_S M : U \quad \Gamma \vdash_S N : T}{\Gamma \vdash_S M := N : \mathbf{1}}$$

$$\textsc{(prop-ite)}\ \frac{\Gamma \vdash \varphi \quad \Gamma, \varphi \vdash_S M : T_1 \quad \Gamma, \neg\varphi \vdash_S N : T_2}{\Gamma \vdash_S \mathbf{if}\ \varphi\ \mathbf{then}\ M\ \mathbf{else}\ N : \mathbf{if}\ \varphi\ \mathbf{then}\ T_1\ \mathbf{else}\ T_2} \qquad \textsc{(kindcase)}\ \frac{\Gamma \vdash T :: \mathcal{K}' \quad \Gamma \vdash \mathcal{K} \quad \Gamma, t{:}\mathcal{K} \vdash_S M : U \quad \Gamma \vdash_S N : U}{\Gamma \vdash_S \mathbf{if}\ T :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N : U}$$

$$\textsc{(conv)}\ \frac{\Gamma \vdash_S M : U \quad \Gamma \models U \equiv T :: \mathcal{K}}{\Gamma \vdash_S M : T} \qquad \textsc{(fix)}\ \frac{\Gamma, F : T \vdash_S M : T \quad \mathsf{structural}(F, M)}{\Gamma \vdash_S \boldsymbol{\mu} F{:}T.M : T}$$

Kind and Refinement Equality.

Reflexivity, Transitivity, Symmetry + Congruence+

$$\frac{\Gamma \vdash \mathcal{K} \equiv \mathcal{K}' \quad \Gamma, t{:}\mathcal{K} \vdash \varphi \equiv \psi}{\Gamma \vdash \{t{:}\mathcal{K} \mid \varphi\} \equiv \{t{:}\mathcal{K}' \mid \psi\}} \qquad \frac{\Gamma \models \varphi \supset \psi \quad \Gamma \models \psi \supset \varphi \quad \Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \equiv \psi}$$

$$\frac{P : (K_1, \ldots, K_n) \in \Sigma \quad \forall i \in \{1, \ldots, n\}.\Gamma \models T_i \equiv S_i :: K_i}{\Gamma \models P(T_1, \ldots, T_n) \equiv P(S_1, \ldots, S_n)}$$

Type equality.

Reflexivity, Transitivity, Symmetry+

$$\frac{\Gamma \models T_1 \equiv S_1 :: \Pi t{:}K_1.K_2 \quad \Gamma \models T_2 \equiv S_2 :: K_1}{\Gamma \models T_1\, T_2 \equiv S_1\, S_2 :: K_2\{T_2/t\}}$$

$$\frac{\Gamma \models K_1 \equiv K_1' \quad \Gamma, t{:}K_1 \models T_1 \equiv T_2 :: K_2}{\Gamma \models \lambda t{::}K_1.T_1 \equiv \lambda t{::}K_1'.T_2 :: \Pi t{:}K_1.K_2} \qquad \frac{\Gamma, t{:}K \vdash T :: K' \quad \Gamma \vdash S :: K}{\Gamma \models (\lambda t{::}K.T)\, S \equiv T\{S/t\} :: K'\{S/t\}}$$

$$\frac{\Gamma \models K_1 \equiv K_2 \quad \Gamma, t{:}K_1 \models T \equiv S :: \mathcal{K}}{\Gamma \models \forall t{::}K_1.T \equiv \forall t{:}K_2.S :: \mathsf{Gen}_{K_1}} \qquad \frac{\Gamma \models T_1 \equiv S_1 :: \mathsf{Gen}_K \quad \Gamma \models T_2 \equiv S_2 :: K}{\Gamma \models \mathbf{tmap}(T_1)T_2 \equiv \mathbf{tmap}(S_1)S_2 :: \mathsf{Type}}$$

$$\frac{\Gamma, t{:}K \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: K}{\Gamma \models \mathbf{tmap}(\forall t{::}K.T)\, S \equiv T\{S/t\} :: \mathsf{Type}} \qquad \frac{\Gamma \models \bot \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \models \bot \equiv T :: \mathcal{K}}$$

$$\frac{\Gamma \models L \equiv L' :: \mathsf{Nm} \quad \Gamma \models T \equiv T' :: \mathcal{K} \quad \Gamma \models S \equiv S' :: \{t{::}\mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \models \langle L : T \rangle @S \equiv \langle L' : T' \rangle @S' :: \mathsf{Rec}}$$

$$\frac{\Gamma \models T \equiv S :: \{r{::}\mathsf{Rec} \mid \mathsf{nonEmpty}(r)\}}{\Gamma \models \mathbf{headLabel}(T) \equiv \mathbf{headLabel}(S) :: \mathsf{Nm}} \qquad \frac{\Gamma \models T \equiv S :: \{r{::}\mathsf{Rec} \mid \mathsf{nonEmpty}(r)\}}{\Gamma \models \mathbf{headType}(T) \equiv \mathbf{headType}(S) :: \mathsf{Type}}$$

$$\frac{\Gamma \models T \equiv S :: \{r{::}\mathsf{Rec} \mid \mathsf{nonEmpty}(r)\}}{\Gamma \models \mathbf{tail}(T) \equiv \mathbf{tail}(S) :: \mathsf{Rec}}$$

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t{::}\mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \models \mathbf{headLabel}(\langle L : T\rangle @ S) \equiv L :: \mathsf{Nm}}$$

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t{::}\mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \models \mathbf{headType}(\langle L : T\rangle @ S) \equiv T :: \mathsf{Type}}$$

$$\frac{\Gamma \vdash L :: \mathsf{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t{::}\mathsf{Rec} \mid L \notin \mathbf{lab}(t)\}}{\Gamma \models \mathbf{tail}(\langle L : T\rangle @ S) \equiv S :: \mathsf{Rec}}$$

$$\frac{\Gamma \models T \equiv S :: \mathcal{K}}{\Gamma \models T^\star \equiv S^\star :: \mathsf{Col}} \qquad \frac{\Gamma \models T \equiv S :: \mathsf{Col}}{\Gamma \models \mathbf{colOf}(T) \equiv \mathbf{colOf}(S) :: \mathsf{Type}}$$

$$\frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \models \mathbf{colOf}(T^\star) \equiv T :: \mathsf{Type}} \qquad \frac{\Gamma \models T \equiv S :: \{t{::}\mathcal{K} \mid \mathit{elim}_{\mathcal{K}}(T) \equiv T' :: K'\} \quad \Gamma \vdash T'\{T/t\} :: K'\{T/t\}}{\Gamma \models \mathit{elim}_{\mathcal{K}}(T) \equiv T'\{T/t\} :: K'\{T/t\}}$$

$$\frac{\Gamma \models T \equiv S :: \mathcal{K}}{\Gamma \models \mathbf{ref}\, T \equiv \mathbf{ref}\, S :: \mathsf{Ref}} \qquad \frac{\Gamma \models T \equiv S :: \mathsf{Ref}}{\Gamma \models \mathbf{refOf}(T) \equiv \mathbf{refOf}(S) :: \mathsf{Type}}$$

$$\frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \models \mathbf{refOf}(\mathbf{ref}\, T) \equiv T :: \mathsf{Type}} \qquad \frac{\Gamma \models T \equiv S :: \mathcal{K} \quad \Gamma \models T' \equiv S' :: \mathcal{K}}{\Gamma \models T \to T' \equiv S \to S' :: \mathsf{Fun}}$$

$$\frac{\Gamma \models T \equiv S :: \mathsf{Fun}}{\Gamma \models \mathbf{dom}(T) \equiv \mathbf{dom}(S) :: \mathsf{Type}} \qquad \frac{\Gamma \models T \equiv S :: \mathsf{Fun}}{\Gamma \models \mathbf{img}(T) \equiv \mathbf{img}(S) :: \mathsf{Type}}$$

$$\frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \mathcal{K}'}{\Gamma \models \mathbf{dom}(T \to S) \equiv T :: \mathsf{Type}} \qquad \frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \mathcal{K}'}{\Gamma \models \mathbf{img}(T \to S) \equiv S :: \mathsf{Type}}$$

$$\frac{\Gamma \models T \equiv T' :: \mathcal{K}_0 \quad \Gamma \models \mathcal{K} \equiv \mathcal{K}' \quad \Gamma, t{:}\mathcal{K} \models S \equiv S' :: K'' \quad \Gamma \models U \equiv U' :: K''}{\Gamma \models \mathbf{if}\, T :: \mathcal{K}\, \mathbf{as}\, t \Rightarrow S\, \mathbf{else}\, U \equiv \mathbf{if}\, T' :: \mathcal{K}'\, \mathbf{as}\, t \Rightarrow S'\, \mathbf{else}\, U' :: K''}$$

$$\frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma, t{:}\mathcal{K} \vdash S :: K' \quad \Gamma \vdash U :: K'}{\Gamma \models \mathbf{if}\, T :: \mathcal{K}\, \mathbf{as}\, t \Rightarrow S\, \mathbf{else}\, U \equiv S\{T/t\} :: K'}$$

$$\frac{\Gamma \vdash T :: \mathcal{K}_0 \quad \Gamma \vdash \mathcal{K}_0 \not\equiv \mathcal{K} \quad \Gamma, t{:}\mathcal{K} \vdash S :: K' \quad \Gamma \vdash U :: K'}{\Gamma \models \mathbf{if}\, T :: \mathcal{K}\, \mathbf{as}\, t \Rightarrow S\, \mathbf{else}\, U \equiv U :: K'}$$

$$\frac{\Gamma \models \varphi \equiv \psi \quad \Gamma, \varphi \models T_1 \equiv S_1 :: K \quad \Gamma, \neg\varphi \models T_2 \equiv S_2 :: K}{\Gamma \models \mathbf{if}\ \varphi\ \mathbf{then}\ T_1\ \mathbf{else}\ T_2 \equiv \mathbf{if}\ \psi\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 :: K}$$

$$\frac{\Gamma \models \varphi \quad \Gamma, \varphi \vdash T_1 :: K \quad \Gamma, \neg\varphi \vdash T_2 :: K}{\Gamma \models \mathbf{if}\ \varphi\ \mathbf{then}\ T_1\ \mathbf{else}\ T_2 \equiv T_1 :: K} \qquad \frac{\Gamma \models \neg\varphi \quad \Gamma, \varphi \vdash T_1 :: K \quad \Gamma, \neg\varphi \vdash T_2 :: K}{\Gamma \models \mathbf{if}\ \varphi\ \mathbf{then}\ T_1\ \mathbf{else}\ T_2 \equiv T_2 :: K}$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma, \varphi \vdash T :: K \quad \Gamma, \neg\varphi \vdash T :: K}{\Gamma \models \mathbf{if}\ \varphi\ \mathbf{then}\ T\ \mathbf{else}\ T \equiv T :: K} \qquad \frac{\Gamma \models T \equiv S :: K \quad \Gamma \vdash K \leq K'}{\Gamma \models T \equiv S :: K'}$$

$$\frac{\begin{array}{c}\mathsf{structural}(T, F, t) \quad \mathsf{structural}(S, F, t)\\ \Gamma \models K_1 \equiv K_1' \quad \Gamma \models K_2 \equiv K_2' \quad \Gamma, F{:}\Pi t{:}K_1.K_2, t{:}K_1 \models T \equiv S :: K_2\end{array}}{\Gamma \models \boldsymbol{\mu} F : (\Pi t{:}K_1.\, K_2).\lambda t{::}K_1.\, T \equiv \boldsymbol{\mu} F : (\Pi t{:}K_1'.\, K_2').\lambda t{::}K_1'.\, S :: \Pi t{:}K_1.K_2}$$

$$\frac{\Gamma, t{:}K_1 \vdash K_2 \quad \Gamma, F{:}\Pi t{:}K_1.K_2, t{:}K_1 \vdash T :: K_2 \quad \Gamma \vdash S :: K_1 \quad \mathsf{structural}(T, F, t)}{\Gamma \models (\boldsymbol{\mu} F : (\Pi t{:}K_1.\, K_2).\lambda t{::}K_1.\, T)\ S \equiv T\{S/t\}\{(\boldsymbol{\mu} F : (\Pi t{:}K_1.\, K_2).\lambda t{::}K_1.\, T)/F\} :: K_2\{S/t\}}$$

B FULL OPERATIONAL SEMANTICS

The type reduction relation, $T \to T'$ 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 (EQ-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:

$$T_v, S_v \quad ::= \quad \lambda t{::}K.T \mid \forall t{::}K.T \mid \ell \mid \langle\rangle \mid \langle \ell : T_v \rangle @ S_v \mid T_v{}^{\star} \mid \mathbf{ref}\, T_v \mid T_v \to S_v \mid \bot \mid \mathsf{Bool} \mid \mathbf{1} \mid t$$

The type reduction rules are:

$$\frac{T \to T'}{T\, S \to T'\, S} \qquad \frac{S \to S'}{(\lambda t{::}K.T)\, S \to (\lambda t{::}K.T)\, S'} \qquad \frac{}{(\lambda t{::}K.T)\, S_v \to T\{S_v/t\}}$$

$$\frac{}{(\boldsymbol{\mu} F : (\Pi t{:}K.\, K').\lambda t{::}K.\, T)\, S_v \to T\{S_v/t\}\{\boldsymbol{\mu} F : (\Pi t{:}K.\, K').\lambda t{::}K.\, T/F\}}$$

$$\frac{L \to L'}{\langle L : T\rangle @ S \to \langle L' : T\rangle @ S} \qquad \frac{T \to T'}{\langle \ell : T\rangle @ S \to \langle \ell : T'\rangle @ S} \qquad \frac{S \to S'}{\langle \ell : T_v\rangle @ S \to \langle \ell : T_v\rangle @ S'}$$

$$\frac{T \to T'}{\mathbf{headLabel}(T) \to \mathbf{headLabel}(T')} \qquad \frac{T \to T'}{\mathbf{headType}(T) \to \mathbf{headType}(T')} \qquad \frac{T \to T'}{\mathbf{tail}(T) \to \mathbf{tail}(T')}$$

$$\frac{}{\mathbf{headLabel}(\langle \ell : T_v\rangle @ S_v) \to \ell} \qquad \frac{}{\mathbf{headType}(\langle \ell : T_v\rangle @ S_v) \to T_v} \qquad \frac{}{\mathbf{tail}(\langle \ell : T_v\rangle @ S_v) \to S_v}$$

$$\frac{T \to T'}{T^{\star} \to T'^{\star}} \qquad \frac{T \to T'}{\mathbf{colOf}(T) \to \mathbf{colOf}(T')} \qquad \frac{}{\mathbf{colOf}(T_v{}^{\star}) \to T_v}$$

$$\frac{T \to T'}{\mathbf{ref}\, T \to \mathbf{ref}\, T'} \qquad \frac{T \to T'}{\mathbf{refOf}(T) \to \mathbf{refOf}(T')} \qquad \frac{}{\mathbf{refOf}(\mathbf{ref}\, T_v) \to T_v}$$

$$\frac{T \to T'}{(T \to S) \to (T' \to S)} \qquad \frac{S \to S'}{(T_v \to S) \to (T_v \to S')} \qquad \frac{T \to T'}{\mathbf{dom}(T) \to \mathbf{dom}(T')} \qquad \frac{T \to T'}{\mathbf{img}(T) \to \mathbf{img}(T')}$$

$$\frac{}{\mathbf{dom}(T_v \to S_v) \to T_v} \qquad \frac{}{\mathbf{img}(T_v \to S_v) \to S_v}$$

$$\frac{\Gamma \models \varphi}{\mathbf{if}\ \varphi\ \mathbf{then}\ T\ \mathbf{else}\ S \to T} \qquad \frac{\Gamma \models \neg\varphi}{\mathbf{if}\ \varphi\ \mathbf{then}\ T\ \mathbf{else}\ S \to S} \qquad \frac{T \to T'}{\mathbf{if}\ T :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow S\ \mathbf{else}\ U \to \mathbf{if}\ T' :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow S\ \mathbf{else}\ U}$$

$$\frac{\Gamma \vdash T_v :: \mathcal{K}}{\mathbf{if}\ T_v :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow S\ \mathbf{else}\ U \to S\{T_v/t\}} \qquad \frac{\Gamma \vdash T_v :: \mathcal{K}' \quad \Gamma \vdash \mathcal{K}' \not\equiv \mathcal{K}}{\mathbf{if}\ T_v :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow S\ \mathbf{else}\ U \to U}$$

The rules of our operational semantics are as follows:

R-RecConsLab
$$\frac{\langle H; L\rangle \longrightarrow \langle H'; L'\rangle}{\langle H; \langle L = M\rangle @N\rangle \longrightarrow \langle H'; \langle L' = M\rangle @N\rangle}$$

R-RecConsL
$$\frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; \langle \ell = M\rangle @N\rangle \longrightarrow \langle H'; \langle \ell = M'\rangle @N\rangle}$$

R-RecConsR
$$\frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; \langle \ell = v\rangle @M\rangle \longrightarrow \langle H'; \langle \ell = v\rangle @M'\rangle}$$

R-RecHdLab
$$\frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; \mathbf{recHeadLabel}(M)\rangle \longrightarrow \langle H'; \mathbf{recHeadLabel}(M')\rangle}$$

R-RecHdLabV
$$\frac{}{\langle H; \mathbf{recHeadLabel}(\langle \ell = v\rangle @v')\rangle \longrightarrow \langle H; \ell\rangle}$$

R-RecHdVal
$$\frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; \mathbf{recHeadTerm}(M)\rangle \longrightarrow \langle H'; \mathbf{recHeadTerm}(M')\rangle}$$

R-RecHdValV
$$\frac{}{\langle H; \mathbf{recHeadTerm}(\langle \ell = v\rangle @v')\rangle \longrightarrow \langle H; v\rangle}$$

R-RecTail
$$\frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; \mathbf{recTail}(M)\rangle \longrightarrow \langle H'; \mathbf{recTail}(M')\rangle}$$

R-RecTailV
$$\frac{}{\langle H; \mathbf{recTail}(\langle \ell = v\rangle @v')\rangle \longrightarrow \langle H; v'\rangle}$$

R-Ref
$$\frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; \mathbf{ref}\ M\rangle \longrightarrow \langle H'; \mathbf{ref}\ M'\rangle}$$

R-RefV
$$\frac{l \notin \mathsf{dom}(H)}{\langle H; \mathbf{ref}\ v\rangle \longrightarrow \langle H[l \mapsto v]; l\rangle}$$

R-Deref
$$\frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; !M\rangle \longrightarrow \langle H'; !M'\rangle}$$

R-DerefV
$$\frac{H(l) = v}{\langle H; !l\rangle \longrightarrow \langle H; v\rangle}$$

R-AssignL
$$\frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; M := N\rangle \longrightarrow \langle H'; M' := N\rangle}$$

R-AssignR
$$\frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; l := M\rangle \longrightarrow \langle H'; l := M'\rangle}$$

R-AssignV
$$\frac{}{\langle H; l := v\rangle \longrightarrow \langle H[l \mapsto v]; v\rangle}$$

R-PropT
$$\frac{\Gamma \models \varphi}{\langle H; \mathbf{if}\ \varphi\ \mathbf{then}\ M\ \mathbf{else}\ N\rangle \longrightarrow \langle H; M\rangle}$$

R-PropF
$$\frac{\Gamma \models \neg\varphi}{\langle H; \mathbf{if}\ \varphi\ \mathbf{then}\ M\ \mathbf{else}\ N\rangle \longrightarrow \langle H; N\rangle}$$

R-IfT
$$\frac{}{\langle H; \mathbf{if}\ \mathit{true}\ \mathbf{then}\ M\ \mathbf{else}\ N\rangle \longrightarrow \langle H; M\rangle}$$

$$\frac{}{\langle H;\,\mathbf{if}\ \mathit{false}\ \mathbf{then}\ M\ \mathbf{else}\ N\rangle \longrightarrow \langle H; N\rangle}\ \textsc{R-IfF} \qquad \frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H;\,\mathbf{if}\ M\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2\rangle \longrightarrow \langle H';\,\mathbf{if}\ M'\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2\rangle}\ \textsc{R-If}$$

$$\frac{T \to T'}{\langle H; (\Lambda t{::}K.\,M)[T]\rangle \longrightarrow \langle H; (\Lambda t{::}K.\,M)[T']\rangle}\ \textsc{R-TAppTRed}$$

$$\frac{}{\langle H; \boldsymbol{\mu} F{:}T.M\rangle \longrightarrow \langle H; M\{\boldsymbol{\mu} F{:}T.M/F\}\rangle}\ \textsc{R-Fix} \qquad \frac{}{\langle H; (\Lambda t{::}K.\,M)[T_v]\rangle \longrightarrow \langle H; M\{T_v/t\}\rangle}\ \textsc{R-TApp}$$

$$\frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; M[T]\rangle \longrightarrow \langle H'; M'[T]\rangle}\ \textsc{R-TAppL} \qquad \frac{}{\langle H; (\lambda x : T.M)\ v\rangle \longrightarrow \langle H; M\{v/x\}\rangle}\ \textsc{R-AppV}$$

$$\frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; M\ N\rangle \longrightarrow \langle H'; M'\ N\rangle}\ \textsc{R-AppL} \qquad \frac{\langle H; N\rangle \longrightarrow \langle H'; N'\rangle}{\langle H; (\lambda x : T.M)\ N\rangle \longrightarrow \langle H'; (\lambda x : T.M)\ N'\rangle}\ \textsc{R-AppR}$$

$$\frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; M :: N\rangle \longrightarrow \langle H'; M' :: N\rangle}\ \textsc{R-ColConsL} \qquad \frac{\langle H; N\rangle \longrightarrow \langle H'; N'\rangle}{\langle H; v :: N\rangle \longrightarrow \langle H'; v :: N'\rangle}\ \textsc{R-ColConsR}$$

$$\frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; \mathbf{colHead}(M)\rangle \longrightarrow \langle H'; \mathbf{colHead}(M')\rangle}\ \textsc{R-ColHd} \qquad \frac{}{\langle H; \mathbf{colHead}(v :: v')\rangle \longrightarrow \langle H; v\rangle}\ \textsc{R-ColHdV}$$

$$\frac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; \mathbf{colTail}(M)\rangle \longrightarrow \langle H'; \mathbf{colTail}(M')\rangle}\ \textsc{R-ColTl} \qquad \frac{}{\langle H; \mathbf{colTail}(v :: v')\rangle \longrightarrow \langle H; v'\rangle}\ \textsc{R-ColTlV}$$

$$\frac{T \to T'}{\langle H; \mathbf{if}\ T :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N\rangle \longrightarrow \langle H; \mathbf{if}\ T' :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N\rangle}\ \textsc{R-KindType}$$

$$\frac{\Gamma \vdash T :: \mathcal{K}}{\langle H; \mathbf{if}\ T :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N\rangle \longrightarrow \langle H; M\{T/t\}\rangle}\ \textsc{R-KindL} \qquad \frac{\Gamma \vdash T :: \mathcal{K}_0 \quad \Gamma \vdash \mathcal{K}_0 \not\equiv \mathcal{K}}{\langle H; \mathbf{if}\ T :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N\rangle \longrightarrow \langle H; N\rangle}\ \textsc{R-KindR}$$

C PROOFS

In the development below we pressupose the signature Σ has been checked for well-formedness.

Lemma 5.1 (Substitution).

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

Case: $\dfrac{\Gamma, t{:}K, \Gamma' \vdash \mathcal{K}' \quad \Gamma, t{:}K, \Gamma', s{:}\mathcal{K}' \vdash \varphi}{\Gamma, t{:}K, \Gamma' \vdash \{s : \mathcal{K}' \mid \varphi\}}$

$\Gamma, \Gamma'\{T/t\} \vdash \mathcal{K}'\{T/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\}, s{:}\mathcal{K}'\{T/t\} \vdash \varphi\{T/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash \{s : \mathcal{K}'\{T/t\} \mid \varphi\{T/t\}\}$ by rule

Case: $\dfrac{P : K_1, \ldots, K_n \in \Sigma \quad \forall i \in \{1, \ldots, n\}.\Gamma, t{:}K, \Gamma' \vdash T_i :: K_i}{\Gamma, t{:}K, \Gamma' \vdash P(T_1, \ldots, T_n)}$

$\forall i \in \{1, \ldots, n\}.\Gamma, \Gamma'\{T/t\} \vdash T_i\{T/t\} :: K_i\{T/t\}$ by i.h. i times
$\Gamma, \Gamma'\{T/t\} \vdash P(T_1\{T/t\}, \ldots, T_n\{T/t\})$ by rule

Case: $\dfrac{\Gamma, t{:}K, \Gamma' \vdash T_1 \equiv T_2 :: K'}{\Gamma, t{:}K, \Gamma' \models T_1 \equiv T_2 :: K'}$

$\Gamma, \Gamma'\{T/t\} \vdash T_1\{T/t\} \equiv T_2\{T/t\} :: K'\{T/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \models T_1\{T/t\} \equiv T_2\{T/t\} :: K'\{T/t\}$ by rule

Case: $\dfrac{\Gamma, t{:}K, \Gamma' \models T_1 \equiv T_2 :: K' \quad \Gamma, t{:}K, \Gamma', x : K' \vdash \varphi \quad \Gamma, t{:}K, \Gamma' \models \varphi\{T_1/x\}}{\Gamma, t{:}K, \Gamma' \models \varphi\{S_2/x\}}$

$\Gamma, \Gamma'\{T/t\} \models T_1\{T/t\} \equiv T_2\{T/t\} :: K'\{T/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\}, x : K'\{T/t\} \vdash \varphi\{T/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \models \varphi\{T_1/x\}\{T/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \models \varphi\{T/t\}\{T_1\{T/t\}/x\}$ by definition
$\Gamma, t{:}K, \Gamma' \models \varphi\{T/t\}\{T_2\{T/t\}/x\}$ by rule

Case: $\dfrac{\Gamma, t{:}K, \Gamma' \vdash K' \quad \Gamma, t{:}K, \Gamma', s{:}K' \vdash T' :: \mathcal{K}}{\Gamma, t{:}K, \Gamma' \vdash \forall s{:}K'.T' :: \mathsf{Gen}_{K'}}$

$\Gamma, \Gamma'\{T/t\} \vdash K'\{T/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\}, s{:}K'\{T/t\} \vdash T'\{T/t\} :: \mathcal{K}$ by i.h.
$\Gamma, t{:}K, \Gamma' \vdash \forall s{:}K'\{T/t\}.T'\{T/t\} :: \mathsf{Gen}_{K'\{T/t\}}$ by rule

Case: $\dfrac{\Gamma, t{:}K, \Gamma' \vdash L :: \mathsf{Nm} \quad \Gamma, t{:}K, \Gamma' \vdash T' :: \mathcal{K} \quad \Gamma, t{:}K, \Gamma' \vdash S' :: \{t : \mathsf{Rec} \mid L\#t\}}{\Gamma, t{:}K, \Gamma' \vdash \langle L : T' \rangle @ S' :: \mathsf{Rec}}$

$\Gamma, \Gamma'\{T/t\} \vdash L\{T/t\} :: \mathsf{Nm}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash T'\{T/t\} :: \mathcal{K}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash S'\{T/t\} :: \{t : \mathsf{Rec} \mid L\{T/t\}\#t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash \langle L\{T/t\} : T'\{T/t\} \rangle @ S'\{T/t\} :: \mathsf{Rec}$ by rule

Case: $\dfrac{\Gamma, t{:}K, \Gamma' \vdash T' :: \{t : K' \mid \mathit{elim}_{K'}(t) \equiv T'' :: K''\}}{\Gamma, t{:}K, \Gamma' \vdash \mathit{elim}_{K'}(T') :: K''}$

$\Gamma, \Gamma'\{T/t\} \vdash T'\{T/t\} :: \{t : K'\{T/t\} \mid \mathit{elim}_{K'\{T/t\}}(t) \equiv T''\{T/t\} :: K''\{T/t\}\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash \mathit{elim}_{K'\{T/t\}}(T'\{T/t\}) :: K''\{T/t\}$ by rule

Case: $$\frac{\Gamma, t{:}K, \Gamma' \vdash \varphi \quad \Gamma, t{:}K, \Gamma', \varphi \vdash T' :: K' \quad \Gamma, t{:}K, \Gamma', \neg\varphi \vdash S :: K'}{\Gamma, t{:}K, \Gamma' \vdash \textbf{if } \varphi \textbf{ then } T' \textbf{ else } S :: K'}$$

$\Gamma, \Gamma'\{T/t\} \vdash \varphi\{T/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\}, \varphi\{T/t\} \vdash T'\{T/t\} :: K'\{T/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\}, \neg\varphi\{T/t\} \vdash S\{T/t\} :: K'\{T/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash \textbf{if } \varphi\{T/t\} \textbf{ then } T'\{T/t\} \textbf{ else } S\{T/t\} :: K'\{T/t\}$ by rule

Case: $$\frac{\Gamma, t{:}K, \Gamma' \vdash S :: \{t : \mathsf{Rec} \mid \ell \# t\} \quad \Gamma, t{:}K, \Gamma' \vdash M : T' \quad \Gamma, t{:}K, \Gamma' \vdash N : S}{\Gamma, t{:}K, \Gamma' \vdash \langle \ell = M \rangle @ N : \boldsymbol{\langle} \ell : T' \boldsymbol{\rangle} @ S}$$

$\Gamma, \Gamma'\{T/t\} \vdash S :: \{t : \mathsf{Rec} \mid \ell \# t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash M\{T/t\} : T'\{T/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash N\{T/t\} : S\{T/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash \langle \ell = M\{T/t\} \rangle @ N\{T/t\} : \boldsymbol{\langle} \ell : T'\{T/t\} \boldsymbol{\rangle} @ S\{T/t\}$ by rule

Case: $$\frac{\Gamma, t{:}K, \Gamma' \vdash M : S \quad \Gamma, t{:}K, \Gamma' \vdash S :: \{s : \mathsf{Rec} \mid \textbf{headLabel}(s) \equiv L :: \mathsf{Nm}\}}{\Gamma, t{:}K, \Gamma' \vdash \textbf{recHeadLabel}(M) : L\{S/s\}}$$

$\Gamma, \Gamma'\{T/t\} \vdash M\{T/t\} : S\{T/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash S\{T/t\} :: \{s : \mathsf{Rec} \mid \textbf{headLabel}(s) \equiv L\{T/t\} :: \mathsf{Nm}\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash \textbf{recHeadLabel}(M\{T/t\}) : L\{T/t\}\{S\{T/t\}/s\}$ by rule

Case: $$\frac{\Gamma, t{:}K, \Gamma' \vdash M : S \quad \Gamma, t{:}K, \Gamma' \vdash S :: \{s : \mathsf{Rec} \mid \textbf{headType}(s) \equiv T' :: K'\}}{\Gamma, t{:}K, \Gamma' \vdash \textbf{recHeadTerm}(M) : T'\{S/s\}}$$

$\Gamma, \Gamma'\{T/t\} \vdash M\{T/t\} : S\{T/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash S\{T/t\} :: \{s : \mathsf{Rec} \mid \textbf{headType}(s) \equiv T'\{T/t\} :: K'\{T/t\}\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash \textbf{recHeadTerm}(M\{T/t\}) : T'\{T/t\}\{S\{T/t\}/s\}$ by rule

Case: $$\frac{\begin{array}{c}\Gamma, t'{:}K, \Gamma', t{:}K_1 \vdash K_2 \quad \Gamma, t'{:}K, \Gamma' \vdash S :: K_1 \\ \Gamma, t'{:}K, \Gamma', F{:}\Pi t{:}K_1.K_2, s{:}K_1 \vdash T' :: K_2 \quad \mathsf{structural}(T', F, t)\end{array}}{\Gamma, t'{:}K, \Gamma' \models (\boldsymbol{\mu} F : (\Pi t{:}K_1.\, K_2).\lambda t{::}K_1.\, T')\, S \equiv T'\{S/t\}\{(\boldsymbol{\mu} F : (\Pi t{:}K_1.\, K_2).\lambda t{::}K_1.\, T')/F\} :: K_2\{S/t\}}$$

$\Gamma, \Gamma'\{T/t'\}, t{:}K_1\{T/t'\} \vdash K_2\{T/t'\}$ by i.h.
$\Gamma, \Gamma'\{T/t'\} \vdash S\{T/t'\} :: K_1\{T/t'\}$ by i.h.
$\Gamma, \Gamma'\{T/t'\}, F{:}\Pi t{:}K_1\{T/t'\}.K_2\{T/t'\}, s{:}K_1\{T/t'\} \vdash T'\{T/t'\} :: K_2\{T/t'\}$ by i.h.
$\mathsf{structural}(T'\{T/t'\}, F, t)$ by ???
$\Gamma, \Gamma'\{T/t'\} \models (\boldsymbol{\mu} F : (\Pi t{:}K_1\{T/t'\}.\, K_2\{T/t'\}).\lambda t{::}K_1\{T/t'\}.\, T'\{T/t'\})\, S\{T/t'\} \equiv$
$T'\{T/t'\}\{S\{T/t'\}/t\}\{(\boldsymbol{\mu} F : (\Pi t{:}K_1\{T/t'\}.\, K_2\{T/t'\}).\lambda t{::}K_1\{T/t'\}.\, T'\{T/t'\})/F\}$
$:: K_2\{T/t'\}\{S\{T/t'\}/t\}$ by rule

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

Lemma 5.2 (Context Conversion).

(a) *Let* $\Gamma, x{:}T \vdash$ *and* $\Gamma \vdash T' :: K$. *If* $\Gamma, x{:}T \vdash \mathcal{J}$ *and* $\Gamma \models T \equiv T' :: K$ *then* $\Gamma, x{:}T' \vdash \mathcal{J}$.
(b) *Let* $\Gamma, t{:}K \vdash$ *and* $\Gamma \vdash K'$. *If* $\Gamma, t{:}K \vdash \mathcal{J}$ *and* $\Gamma \vdash K \equiv K'$ *then* $\Gamma, t{:}K' \vdash \mathcal{J}$.

Proof. Follows by weakening and substitution.
(a)

$\Gamma, x : T' \vdash x : T'$ by variable rule
$\Gamma \vdash T' \equiv T :: K$ by symmetry
$\Gamma, x{:}T' \vdash x : T$ by conversion

$\Gamma, x' : T \vdash \mathcal{J}\{x'/x\}$ alpha conversion, for fresh x'
$\Gamma, x : T', x' {:} T \vdash \mathcal{J}\{x'/x\}$ by weakening
$\Gamma, x' {:} T \vdash \mathcal{J}\{x'/x\}\{x/x'\}$ by substitution
$\Gamma, x {:} T' \vdash \mathcal{J}$ by definition

Statement **(b)** follows by the same reasoning. □

Lemma 5.5 (Functionality of Kinding and Refinements). *Assume* $\Gamma \models T \equiv S :: K$, $\Gamma \vdash T :: K$ *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* $\Gamma, \Gamma'\{T/t\} \vdash K\{T/t\} \equiv K\{S/t\}$.
(c) *If* $\Gamma, t{:}K, \Gamma' \models \varphi$ *then* $\Gamma, \Gamma'\{T/t\} \models \varphi\{T/t\} \equiv \varphi\{S/t\}$

Proof. By induction on the given kinding/kind well-formedness and entailment judgments. Functionality follows by substitution and the congruence rules of definitional equality.

Case: $\dfrac{\Gamma, t : K, \Gamma' \vdash \mathcal{K}' \quad \Gamma, t : K, \Gamma', t'{:}\mathcal{K}' \vdash \varphi}{\Gamma, t : K, \Gamma' \vdash \{t' : \mathcal{K}' \mid \varphi\}}$

$\Gamma, \Gamma'\{T/t\} \vdash \mathcal{K}'\{T/t\} \equiv \mathcal{K}'\{S/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\}, t'{:}\mathcal{K}'\{T/t\} \vdash \varphi\{T/t\} \equiv \varphi\{S/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash \{t' : \mathcal{K}'\{T/t\} \mid \varphi\{T/t\}\} \equiv \{t' : \mathcal{K}'\{S/t\} \mid \varphi\{S/t\}\}$ by kind ref. equality

Case: $\dfrac{\Gamma, t : K, \Gamma' \models T' \equiv S' :: K' \quad \Gamma, t : K, \Gamma', x : K' \vdash \varphi \quad \Gamma, t : K, \Gamma' \models \varphi\{T'/x\}}{\Gamma, t : K, \Gamma' \models \varphi\{S'/x\}}$

$\Gamma \models T \equiv S :: K, \Gamma \vdash T :: K$ and $\Gamma \vdash S :: K$ by assumption
$\Gamma, \Gamma'\{T/t\} \models \phi\{S'/x\}\{T/t\}$ by substitution
$\Gamma, \Gamma'\{S/t\} \models \phi\{S'/x\}\{S/t\}$ by substitution
$\Gamma, \Gamma'\{T/t\} \models \phi\{S'/x\}\{S/t\}$ by ctxt. conversion
$\Gamma, \Gamma'\{T/t\} \models \phi\{S'/x\}\{T/t\} \supset \phi\{S'/x\}\{S/t\}$ by weakening and $\supset$I
$\Gamma, \Gamma'\{T/t\} \models \phi\{S'/x\}\{S/t\} \supset \phi\{S'/x\}\{T/t\}$ by weakening and $\supset$I
$\Gamma, \Gamma'\{T/t\} \models \varphi\{S'/x\}\{T/t\} \equiv \varphi\{S'/x\}\{S/t\}$ by definition of refinement equivalence

Case: $\dfrac{\Gamma, t : K, \Gamma' \vdash K \quad \Gamma, t : K, \Gamma, s{:}K' \vdash T' :: \mathcal{K}}{\Gamma, t : K, \Gamma \vdash \forall s{:}K'.T' :: \mathsf{Gen}_K}$

$\Gamma, \Gamma'\{T/t\} \vdash K'\{T/t\} \equiv K'\{S/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\}, t'{:}K'\{T/t\} \vdash T'\{T/t\} \equiv T'\{S/t\} :: \mathcal{K}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash \forall s : K'\{T/t\}.T'\{T/t\} \equiv \forall s : K'\{S/t\}.T'\{S/t\} :: \mathsf{Gen}_{K'\{T/t\}}$ by $\forall$ Eq.

Case: $\dfrac{\Gamma, t : K, \Gamma' \vdash L :: \mathsf{Nm} \quad \Gamma, t : K, \Gamma' \vdash T' :: \mathcal{K} \quad \Gamma, t : K, \Gamma' \vdash S' :: \{t : \mathsf{Rec} \mid L\#t\}}{\Gamma, t : K, \Gamma' \vdash \langle L : T' \rangle @ S' :: \mathsf{Rec}}$

$\Gamma, \Gamma'\{T/t\} \vdash L\{T/t\} \equiv L\{S/t\} :: \mathsf{Nm}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash T'\{T/t\} \equiv T'\{S/t\} :: \mathcal{K}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash S'\{T/t\} \equiv S'\{S/t\} :: \{t : \mathsf{Rec} \mid L\{T/t\}\#t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \vdash \langle L\{T/t\} : T'\{T/t\} \rangle @ S'\{T/t\} \equiv \langle L\{S/t\} : T'\{S/t\} \rangle @ S'\{S/t\} :: \mathsf{Rec}$ by Rec Eq.

Case: $\dfrac{\Gamma, t : K, \Gamma' \vdash T' :: \{s : \mathcal{K}' \mid \mathit{elim}_{\mathcal{K}'}(s) \equiv T'' :: \mathcal{K}''\}}{\Gamma, t : K, \Gamma' \vdash \mathit{elim}_{\mathcal{K}'}(T') :: \mathcal{K}''}$

$\Gamma, \Gamma'\{T/t\} \models T'\{T/t\} \equiv T'\{S/t\} :: \{s : \mathcal{K}'\{T/t\} \mid \mathit{elim}_{\mathcal{K}'}(s) \equiv T''\{T/t\} :: K''\{T/t\}\}$ by i.h.
$\Gamma, \Gamma'\{T/t\} \models \mathit{elim}_{\mathcal{K}'}(T'\{T/t\}) \equiv T''\{T/t\} :: K''\{T/t\}$ by elim_K eq. rule

$\Gamma, \Gamma'\{T/t\} \models \mathit{elim}_{\mathcal{K}'}(T'\{S/t\}) \equiv T''\{T/t\} :: K''\{T/t\}$ by symmetry and elim_K eq. rule
$\Gamma, \Gamma'\{T/t\} \models \mathit{elim}_{\mathcal{K}'}(T'\{T/t\}) \equiv \mathit{elim}_{\mathcal{K}'}(T'\{S/t\}) :: K''\{T/t\}$ by sym. and transitivity

Case: $\dfrac{\Gamma, t{:}K, \Gamma' \vdash \varphi \quad \Gamma, t{:}K, \Gamma', \varphi \vdash T' :: K' \quad \Gamma, t{:}K, \Gamma', \neg\varphi \vdash S' :: K'}{\Gamma, t{:}K, \Gamma' \vdash \textbf{if } \varphi \textbf{ then } T' \textbf{ else } S' :: K'}$

$\Gamma, \Gamma'\{T/t\}, \varphi\{T/t\} \vdash T'\{T/t\} \equiv T'\{S/t\} :: K'\{T/t\}$ by i.h.
$\Gamma, \Gamma'\{T/t\}, \neg\varphi\{T/t\} \vdash S'\{T/t\} \equiv S'\{S/t\} :: K'\{T/t\}$ by i.h.
$\Gamma, t : K, \Gamma', \varphi \models \varphi$ 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
$\Gamma, \Gamma'\{T/t\} \models \varphi\{S/t\} \supset \varphi\{T/t\}$ by $\supset$I
$\Gamma, \Gamma'\{S/t\}, \varphi\{S/t\} \models \varphi\{S/t\}$ by substitution
$\Gamma, \Gamma'\{T/t\}, \varphi\{T/t\} \models \varphi\{S/t\}$ by ctxt. conversion
$\Gamma, \Gamma'\{T/t\} \models \varphi\{T/t\} \supset \varphi\{S/t\}$ by $\supset$I
$\Gamma, \Gamma'\{T/t\} \vdash \varphi\{T/t\} \equiv \varphi\{S/t\}$ by definition
$\Gamma, \Gamma'\{T/t\} \models \textbf{if } \varphi\{T/t\} \textbf{ then } T'\{T/t\} \textbf{ else } S'\{T/t\} \equiv$
$\qquad \textbf{if } \varphi\{S/t\} \textbf{ then } T'\{S/t\} \textbf{ else } S'\{S/t\} :: K'\{T/t\}$ by rule

Case: $\dfrac{\Gamma, t{:}K, \Gamma' \models \varphi\{T'/s\} \quad \Gamma, t{:}K, \Gamma' \vdash T' :: \mathcal{K}'}{\Gamma, t{:}K, \Gamma' \vdash T' :: \{s{:}\mathcal{K}' \mid \varphi\}}$

$\Gamma, \Gamma'\{T/t\} \models \varphi\{T'/s\}\{T/t\} \equiv \varphi\{T'/s\}\{S/t\}$ 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: $\dfrac{\Gamma \vdash \mathcal{K} \equiv \mathcal{K}' \quad \Gamma, t{:}\mathcal{K} \vdash \varphi \equiv \psi}{\Gamma \vdash \{t{:}\mathcal{K} \mid \varphi\} \equiv \{t{:}\mathcal{K}' \mid \psi\}}$

$\Gamma \vdash \mathcal{K}$ and $\Gamma \vdash \mathcal{K}'$ by i.h.
$\Gamma, t : \mathcal{K} \vdash \varphi$ and $\Gamma, t : \mathcal{K} \vdash \psi$ 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: $\dfrac{\Gamma \models T_1 \equiv S_1 :: \mathrm{Gen}_K \quad \Gamma \models T_2 \equiv S_2 :: K}{\Gamma \models \mathbf{tmap}(T_1)T_2 \equiv \mathbf{tmap}(S_1)S_2 :: \mathrm{Type}}$

$\Gamma \vdash T_1 :: \mathrm{Gen}_K$ and $\Gamma \vdash S_1 :: \mathrm{Gen}_K$ by i.h.
$\Gamma \vdash T_2 :: K$ and $\Gamma \vdash S_2 :: K$ by i.h.
$\Gamma \vdash \mathbf{tmap}(T_1)\, T_2 :: \mathrm{Type}$ by kinding
$\Gamma \vdash \mathbf{tmap}(S_1)\, S_2 :: \mathrm{Type}$ by kinding

Case: $\dfrac{\Gamma, t{:}K \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: K}{\Gamma \models \mathbf{tmap}(\forall t{:}K.T)\, S \equiv T\{S/t\} :: \mathrm{Type}}$

$\Gamma, t{:}K \vdash T :: \mathcal{K}$ by inversion
$\Gamma \vdash S :: K$ by inversion
$\Gamma \vdash \forall t{:}K.T :: \mathsf{Gen}_K$ by kinding
$\Gamma \vdash \mathbf{tmap}(\forall t{:}K.T)\, S :: \mathsf{Type}$ by kinding
$\Gamma \vdash T\{S/t\} :: \mathcal{K}\{S/t\}$ by substitution
$\Gamma \vdash T\{S/t\} :: \mathsf{Type}$ by subkinding

Case: $$\frac{\Gamma \models T \equiv S :: \{t{:}\mathcal{K} \mid elim_{\mathcal{K}}(t) \equiv T' :: K'\} \quad [\Gamma \vdash T'\{T/t\} :: K'\{T/t\}]}{\Gamma \models elim_{\mathcal{K}}(T) \equiv T'\{T/t\} :: K'\{T/t\}}$$

$\Gamma \vdash T :: \{t : \mathcal{K} \mid elim_{\mathcal{K}}(t) \equiv T' :: K'\}$ and $\Gamma \vdash S :: \{t : \mathcal{K} \mid elim_{\mathcal{K}}(t) \equiv T' :: K'\}$ by i.h.
$\Gamma \vdash elim_{\mathcal{K}}(T) :: K'\{T/t\}$ by kinding
$\Gamma \vdash T'\{T/t\} :: K'\{T/t\}$ by assumption

Case: $$\frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \models \mathbf{colOf}(T^\star) \equiv T :: \mathsf{Type}}$$

$\Gamma \vdash T :: \mathcal{K}$ by inversion
$\Gamma \vdash T :: \mathsf{Type}$ by subkinding
$\Gamma \vdash T^\star :: \mathsf{Col}$ by kinding
$\Gamma \vdash \mathbf{colOf}(T^\star) :: \mathsf{Type}$ by kinding

Remaining cases follow by a similar reasoning.

□

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

Proof. Immediate from equality validity since $T \to S$ implies $T \equiv S$. □

Lemma 5.6 (Functionality of Equality). *Assume $\Gamma \models T_0 \equiv S_0 :: K$:*
(a) *If $\Gamma, t{:}K \models T \equiv S :: K'$ then $\Gamma \models T\{T_0/t\} \equiv S\{S_0/t\} :: K'\{T_0/t\}$.*
(b) *If $\Gamma, t{:}K \vdash K_1 \equiv K_2$ then $\Gamma \vdash K_1\{T_0/t\} \equiv K_2\{S_0/t\}$.*
(c) *If $\Gamma, t{:}K \vdash \varphi \equiv \psi$ then $\Gamma \vdash \varphi\{T_0/t\} \equiv \psi\{S_0/t\}$.*

Proof. **(a)**

$\Gamma, t{:}K \models T \equiv S :: K'$ assumption
$\Gamma \vdash T_0 \equiv S_0 :: K$ assumption
$\Gamma \vdash T_0 :: K$ and $\Gamma \vdash S_0 :: K$ by eq. validity
$\Gamma, t{:}K \vdash T :: K'$ and $\Gamma, t{:}K \vdash S :: K'$ by eq. validity
$\Gamma \vdash T\{T_0/t\} \equiv S\{T_0/t\} :: K'\{T_0/t\}$ by substitution
$\Gamma \vdash S\{T_0/t\} \equiv S\{S_0/t\} :: K'\{T_0/t\}$ by functionality
$\Gamma \vdash T\{T_0/t\} \equiv S\{S_0/t\} :: K'\{T_0/t\}$ by transitivity

(b)

$\Gamma \vdash T_0 \equiv S_0 :: K$ assumption
$\Gamma, t : K \vdash K_1 \equiv K_2$ assumption
$\Gamma \vdash T_0 :: K$ and $\Gamma \vdash S_0 :: K$ by eq. validity
$\Gamma, t : K \vdash K_1$ and $\Gamma, t : K \vdash K_2$ by eq. validity
$\Gamma \vdash K_1\{T_0/t\} \equiv K_2\{T_0/t\}$ by substitution
$\Gamma \vdash K_2\{T_0/t\} \equiv K_2\{S_0/t\}$ by functionality
$\Gamma \vdash K_1\{T_0/t\} \equiv K_2\{S_0/t\}$ by transitivity

(c)

$\Gamma \vdash T_0 \equiv S_0 :: K$	assumption
$\Gamma, t : K \vdash \varphi \equiv \psi$	assumption
$\Gamma \vdash T_0 :: K$ and $\Gamma \vdash S_0 :: K$	by eq. validity
$\Gamma, t : K \vdash \varphi$ 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

□

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* $\Gamma \vdash T :: \mathsf{Type}$.

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* $\Gamma \models T' \equiv T_1 \to T_2 :: \mathsf{Fun}$, $\Gamma \models T \equiv T_1 :: \mathsf{Type}$ *and* $\Gamma, x{:}T_1 \vdash M : T_2$.
(d) *If* $\Gamma \vdash \langle L : T\rangle @ S :: K$ *then* $\Gamma \vdash L :: \mathsf{Nm}$, $\Gamma \vdash T :: \mathsf{Type}$, $\Gamma \vdash S :: \{t{::}\mathsf{Rec} \mid L \notin t\}$ *and* $\Gamma \vdash K \equiv \mathsf{Rec}$.
(e) *If* $\Gamma \vdash \langle L = M\rangle @ N : T$ *then there is* L', T_1, T_2 *such that* $\Gamma \models L \equiv L' :: \mathsf{Nm}$, $\Gamma \vdash \langle L' : T_1\rangle @ T_2 :: \mathsf{Rec}$, $\Gamma \models T \equiv \langle L' : T_1\rangle @ T_2 :: \mathsf{Rec}$, $\Gamma \vdash M : T_1$ *and* $\Gamma \vdash N : T_2$.
(f) *If* $\Gamma \vdash T :: \{t{::}K \mid \varphi\}$ *then* $\Gamma \models \varphi\{T/t\}$, $\Gamma \vdash T :: K$ *and* $\Gamma, t{:}K \vdash \varphi$.
(g) *If* $\Gamma \vdash \mathit{elim}_{\mathcal{K}}(T) :: K$ *then* $\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \mathit{elim}_{\mathcal{K}}(t) \equiv T' :: K'\}$ *and* $\Gamma \vdash T'\{T/t\} :: K'\{T/t\}$ *and* $\Gamma \vdash K \equiv K'\{T/t\}$.
(h) *If* $\Gamma \vdash \mathbf{if}\,\varphi\,\mathbf{then}\,M\,\mathbf{else}\,N : T$ *then* $\Gamma \models T \equiv \mathbf{if}\,\varphi\,\mathbf{then}\,T_1\,\mathbf{else}\,T_2 :: K$ *with* $\Gamma, \varphi \vdash M : T_1$ *and* $\Gamma, \neg\varphi \vdash N : T_2$.
(i) *If* $\Gamma \vdash \mathbf{if}\,\varphi\,\mathbf{then}\,T\,\mathbf{else}\,S :: K$ *then* $\Gamma \vdash \varphi$, $\Gamma, \varphi \vdash T :: K$ *and* $\Gamma, \neg\varphi \vdash S :: K$.
(j) *If* $\Gamma \vdash T \to S :: K$ *then* $\Gamma \vdash K \equiv \mathsf{Fun}$, $\Gamma \vdash T :: \mathcal{K}$ *and* $\Gamma \vdash S :: \mathcal{K}'$, *for some* $\mathcal{K}, \mathcal{K}'$.
(k) *If* $\Gamma \vdash M :: N :: T$ *then* $\Gamma \models T \equiv S :: \{t{::}\mathsf{Col} \mid \mathbf{colOf}(t) \equiv T' :: \mathcal{K}\}$, $\Gamma \vdash N : S$ *and* $\Gamma \vdash M : T'\{S/t\}$, *for some* $T', \mathcal{K}, S, T'$.
(l) *If* $\Gamma \vdash T'^{\star} :: K$ *then* $\Gamma \vdash K \equiv \mathsf{Col}$ *and* $\Gamma \vdash T' :: \mathcal{K}$, *for some* $\mathcal{K}$.
(m) *If* $\Gamma \vdash \mathbf{if}\,T' :: K\,\mathbf{as}\,t \Rightarrow M\,\mathbf{else}\,N : T$ *then* $\Gamma \vdash T' :: \mathcal{K}$, $\Gamma \vdash K$, $\Gamma, t : K \vdash M : S$ *and* $\Gamma \vdash N : S$, *with* $\Gamma \vdash T \equiv S :: \mathcal{K}'$ *for some* $\mathcal{K}, \mathcal{K}', S$.
(n) *If* $\Gamma \vdash \mathbf{if}\,T' :: K\,\mathbf{as}\,t \Rightarrow S\,\mathbf{else}\,S' :: K'$ *then* $\Gamma \vdash T' :: \mathcal{K}$, $\Gamma \vdash K$, $\Gamma, t{:}K \vdash S :: K''$, $\Gamma\ \mathit{vdashS}' :: K''$ *and* $\Gamma \vdash K' \equiv K''$, *for some* $\mathcal{K}, K''$.
(o) *If* $\Gamma \vdash \boldsymbol{\mu} F{:}T.M : T$ *then* $\Gamma, F : T \vdash M : T$ *and* $\mathsf{structural}(F, M)$.
(p) *If* $\Gamma \vdash \boldsymbol{\mu} F : (\Pi t{:}K_1.\,K_2).\lambda t{::}K_1.\,T' :: K$ *then* $\Gamma, F{:}\Pi t{:}K_1.K_2, t{:}K_1 \vdash T' :: K_2$, $\mathsf{structural}(T', F, t)$ *and* $\Gamma \vdash K \equiv \Pi t{:}K_1.K_2$.

(q) *If* $\Gamma \vdash \textbf{recHeadLabel}(M) : T$ *then* $\Gamma \models T \equiv L\{S/t\} :: \mathsf{Nm}$, $\Gamma \vdash M : S$ *and* $\Gamma \vdash S :: \{t{:}\mathsf{Rec} \mid \textbf{headLabel}(t) \equiv L :: \mathsf{Nm}\}$
(r) *If* $\Gamma \vdash \textbf{recHeadTerm}(M) : T$ *then* $\Gamma \models T \equiv T'\{S/t\} :: \mathcal{K}\{S/t\}$, $\Gamma \vdash M : S$ *and* $\Gamma \vdash S :: \{t{:}\mathsf{Rec} \mid \textbf{headType}(t) \equiv T' :: \mathcal{K}\}$
(s) *If* $\Gamma \vdash \textbf{tail}(M) : T$ *then* $\Gamma \models T \equiv T'\{S/t\} :: \mathcal{K}\{S/t\}$, $\Gamma \vdash M : S$ *and* $\Gamma \vdash S :: \{t{:}\mathsf{Rec} \mid \textbf{tail}(t) \equiv T' :: \mathcal{K}\}$.
(t) *If* $\Gamma \vdash \textbf{colHead}(M) : T$ *then* $\Gamma \vdash M : T^\star$
(u) *If* $\Gamma \vdash \textbf{colTail}(M) : T$ *then* $\Gamma \models T \equiv T'\{T_c/t\} :: \mathcal{K}$, $\Gamma \vdash M : T_c$ *and* $\Gamma \vdash T_c :: \{t{::}\mathsf{Col} \mid \textbf{colOf}(t) \equiv T' :: \mathcal{K}\}$
(v) *If* $\Gamma \vdash \textbf{ref}\, M : T$ *then* $\Gamma \models T \equiv \textbf{ref}\, T'$ *and* $\Gamma \vdash M : T'$
(w) *If* $\Gamma \vdash !M : T$ *then* $\Gamma \models T \equiv T'\{S/t\} :: \mathcal{K}$, $\Gamma \vdash M : S$, $\Gamma \vdash N : T'$ *and* $\Gamma \vdash S :: \{t{::}\mathsf{Ref} \mid \textbf{refOf}(t) \equiv T' :: \mathcal{K}\}$
(x) *If* $\Gamma \vdash M := N : T$ *then* $\Gamma \models T \equiv T'\{S/t\} :: \mathcal{K}$, $\Gamma \vdash M : S$, $\Gamma \vdash N : T$, $\Gamma \vdash S :: \{t{::}\mathsf{Ref} \mid \textbf{refOf}(t) \equiv T' :: \mathcal{K}\}$
(y) *If* $\Gamma \vdash M\,N : T$ *then* $\Gamma \vdash M : T_1$, $\Gamma \vdash N : T_2$, $\Gamma \vdash T_1 :: kref\,t{::}\mathsf{Fun}\textbf{dom}(t) \equiv T_2 :: \mathcal{K} \wedge \textbf{img}(t) = U :: \mathcal{K}'$ *and* $\Gamma \vdash T \equiv U\{T_1/t\} :: \mathcal{K}'\{T_1/t\}$
(z) *If* $\Gamma \vdash M[T] : S$ *then* $\Gamma \vdash M : T'$, $\Gamma \vdash T :: K$, $\Gamma \vdash U :: \mathcal{K}$, $\Gamma \vdash T' :: \{f{::}\mathsf{Gen}_K \mid \textbf{tmap}(f)\,T \equiv U :: \mathcal{K}\}$ *and* $\Gamma \vdash S \equiv U :: \mathcal{K}$.

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

Case: $\dfrac{\Gamma \vdash \lambda t : K.T :: K'' \quad \Gamma \vdash K'' \leq K'}{\Gamma \vdash \lambda t : K.T :: K'}$

$\Gamma \vdash K'' \equiv \Pi t{:}K_1'.K_2'$, $\Gamma \vdash K \equiv K_1'$ and $\Gamma, t : K_1' \vdash T :: K_2'$ by i.h.
$\Gamma \vdash K'' \leq \Pi t : K_1.K_2$, for some K_1, K_2 with $\Gamma \vdash K_1' \leq K_1$ and $\Gamma, t : K_1' \vdash K_2' \leq K_2$ by inversion
$\Gamma \vdash K_1' \equiv K_1$ and $\Gamma, t : K_1' \vdash K_2' \equiv K_2$ by inversion
$\Gamma, t : K_1 \vdash T :: K_2'$ by ctxt. conversion
$\Gamma, t : K_1 \vdash T :: K_2$ by conversion
$\Gamma \vdash K \equiv K_1$ by transitivity

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

□

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* $\Gamma \models T \equiv \lambda t : K_1.T_2 :: K'$ *then* $\Gamma \models T \equiv \lambda t : K_0.T_2' :: \Pi t : K_0.K''$ *with* $\Gamma \vdash K_0 \equiv K_1$ *and* $\Gamma, t : K_0 \models T_2 \equiv T_2' :: K''$, *for some* K''.
(2) *If* $\Gamma \models T \equiv T_0\, S_0 :: K$ *then* $\Gamma \models T \equiv T_1\, S_1 :: K$ *with* $\Gamma \models T_1 \equiv T_0 :: \Pi t : K_1.K_0, \Gamma S_1 \equiv S_0 :: K_1$ *and* $K = K_0\{S_1/t\}$.
(3) *If* $\Gamma \models T \equiv \langle L : T\rangle @S :: K$ *then* $\Gamma \models T \equiv \langle L' : T'\rangle @S' :: K$ *with* $\Gamma \models L \equiv L' :: \mathsf{Nm}$, $\Gamma \models T' \equiv T :: \mathcal{K}$, $\Gamma \models S' \equiv S :: \{t : \mathsf{Rec} \mid L \notin t\}$ *and* $K = \mathsf{Rec}$.
(4) *If* $\Gamma \vdash K \equiv \{t : \mathcal{K} \mid \varphi\}$ *then* $\Gamma \vdash K \equiv \{t : \mathcal{K}' \mid \psi\}$ *with* $\Gamma \vdash \mathcal{K} \equiv \mathcal{K}'$ $\Gamma \vdash \varphi \equiv \psi$
(5) *If* $\Gamma \models T \equiv elim_{\mathcal{K}}(S) :: K$ *then* $\Gamma \models T \equiv elim_{\mathcal{K}'}(S') :: K$ *with* $\Gamma \models S \equiv S' :: \{t{:}\mathcal{K} \mid elim_{\mathcal{K}}(t) \equiv T' :: K'\}$, $\Gamma \vdash \mathcal{K} \equiv \mathcal{K}'$, $\Gamma \vdash T' :: K'\{S/t\}$ *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

$\Gamma \models T \equiv S' :: K$ and $\Gamma \models S' \equiv elim_{\mathcal{K}}(S) :: K$ — assumption
$\Gamma \models S' \equiv elim_{\mathcal{K}'}(S'') :: K$ with $\Gamma \models S \equiv S'' :: \{t{:}\mathcal{K} \mid elim_{\mathcal{K}}(t) \equiv T' :: K'\}$,
$\Gamma \vdash \mathcal{K}' \equiv \mathcal{K}, \Gamma \vdash T' :: K'\{S/t\}$ and $K = K'\{S/t\}$ — by i.h.
$\Gamma \models T \equiv elim_{\mathcal{K}'}(S'') :: K$ — by transitivity

Case: $$\frac{\Gamma, t{:}K_0 \vdash T_1 :: K' \quad \Gamma \vdash T_2 :: K_0}{\Gamma \models (\lambda t{:}K_0.T_1)\, T_2 \equiv T_1\{T_2/t\} :: K'\{T_2/t\}}$$

$\Gamma, t{:}K_0 \vdash T_1 :: K', \Gamma \vdash T_2 :: K_0$ and $elim_{\mathcal{K}}(S) = T_1\{T_2/t\}$ and $K = K'\{T_2/t\}$ — assumption
Subcase 1: $T_1 = t, T_2 = elim_{\mathcal{K}}(S)$
$K_0 = K' = K$ — assumption
$\Gamma \vdash elim_{\mathcal{K}}(S) :: K$ — assumption
$\Gamma \vdash S :: \{t{:}\mathcal{K} \mid elim_{\mathcal{K}}(t) \equiv T' :: K'\}$ and $\Gamma \vdash T'\{S/t\} :: K'\{S/t\}$ with $K = K'\{S/t\}$ — by inversion
$\Gamma \models S \equiv S :: \{t{:}\mathcal{K} \mid elim_{\mathcal{K}}(t) \equiv T' :: K'\}$ — by reflexivity
$\Gamma \vdash elim_{\mathcal{K}}(S) \equiv elim_{\mathcal{K}}(S) :: K$ — by reflexivity
Subcase 2: $T_1 = elim_{\mathcal{K}'}(S')$ such that $elim_{\mathcal{K}'\{T_2/t\}}(S'\{T_2/t\}) = elim_{\mathcal{K}}(S)$
$\Gamma, t : K_0 \vdash elim_{\mathcal{K}'}(S') :: K'$ — assumption
$\Gamma \vdash elim_{\mathcal{K}'\{T_2/t\}}(S'\{T_2/t\}) :: K'\{T_2/t\}$ — by substitution
$\Gamma \vdash S'\{T_2/t\} :: \{t{:}\mathcal{K} \mid elim_{\mathcal{K}}(t) \equiv T' :: K'\}$ and $\Gamma \vdash T'\{S/t\} :: K'\{S/t\}$ with $K = K'\{S/t\}$
— by inversion
$\Gamma \vdash S'\{T_2/t\} \equiv S'\{T_2/t\} :: \{t{:}\mathcal{K} \mid elim_{\mathcal{K}}(t) \equiv T' :: K'\}$ — by reflexivity
$\Gamma \vdash elim_{\mathcal{K}'\{T_2/t\}}(S'\{T_2/t\}) \equiv elim_{\mathcal{K}'\{T_2/t\}}(S'\{T_2/t\}) :: K'\{T_2/t\}$ — by reflexivity

□

Lemma C.5 (Subkinding Inversion).
(1) *If* $\Gamma \vdash \mathcal{K} \leq \mathcal{K}'$ *then* $\Gamma \vdash \mathcal{K} \equiv \mathcal{K}'$ *or* $\Gamma \vdash \mathcal{K}' \equiv \mathsf{Type}$.
(2) *If* $\Gamma \vdash K \leq \{t{:}\mathcal{K}' \mid \varphi\}$ *then* $\Gamma \vdash K \equiv \{t{:}\mathcal{K} \mid \psi\}$ *with* $\Gamma \vdash \mathcal{K} \leq \mathcal{K}'$ *and* $\Gamma \models \psi \supset \varphi$.
(3) *If* $\Gamma \vdash \{t{:}\mathcal{K}' \mid \varphi\} \leq \mathcal{K}$ *then* $\Gamma \vdash \mathcal{K} \leq \mathcal{K}$ *and* $\Gamma, t{:}\mathcal{K}' \vdash \varphi$.

Proof. By induction on the given derivation, using equality inversion. □

Lemma C.6. *If* $\Gamma \models T \equiv S :: K, \Gamma \vdash T :: K'$ *and* $\Gamma \vdash S :: K'$ *and* $\Gamma \vdash K' \leq K$ *then* $\Gamma \vdash T \equiv S :: K'$.

Proof. By induction on the given equality derivation. □

Theorem 5.8 (Unicity of Types and Kinds).
(1) *If* $\Gamma \vdash M : T$ *and* $\Gamma \vdash M : S$ *then* $\Gamma \vdash T \equiv S :: K$ *and* $\Gamma \vdash K \leq \mathsf{Type}$.
(2) *If* $\Gamma \vdash T :: K$ *and* $\Gamma \vdash T :: K'$ *then* $\Gamma \vdash K \leq K'$ *or* $\Gamma \vdash K' \leq K$.

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

Case: M is $\langle \ell = M' \rangle @ N'$

$\Gamma \vdash \langle \ell = M' \rangle @ N' : T$ and $\Gamma \vdash \langle \ell = M' \rangle @ N' : S$ — assumption
$\Gamma \vdash M' : T_1, \Gamma \vdash N' : T_2, \Gamma \vdash \ell \equiv L' :: \mathsf{Nm}, \Gamma \vdash \langle L' = T_1 \rangle @ T_2 :: \mathsf{Rec}$
and $\Gamma \models T \equiv \langle L' = T_1 \rangle @ T_2 :: \mathsf{Rec}$ — inversion
$\Gamma \vdash M' : S_1, \Gamma \vdash N' : S_2, \Gamma \vdash \ell \equiv L'' :: \mathsf{Nm}, \Gamma \vdash \langle L'' = S_1 \rangle @ S_2 :: \mathsf{Rec}$
and $\Gamma \models S \equiv \langle L'' = S_1 \rangle @ S_2 :: \mathsf{Rec}$ — inversion
$\Gamma \models T_1 \equiv S_1 :: K_1$ and $\Gamma \vdash K_1 \leq \mathsf{Type}$ — by i.h.
$\Gamma \models T_1 \equiv S_1 :: \mathsf{Type}$ — by conversion

$\Gamma \models T_2 \equiv S_2 :: K_2$ and $\Gamma \vdash K_2 \leq \mathsf{Type}$ by i.h.
$\Gamma \vdash T_1 :: \mathsf{Rec}$ and $\Gamma \vdash T_2 :: \mathsf{Rec}$ by inversion and conversion
$\Gamma \models T_2 \equiv S_2 :: \mathsf{Rec}$ by Lemma C.6

Case: T is $\langle L = S_1 \rangle @ S_2$

$\Gamma \vdash \langle L = S_1 \rangle @ S_2 :: K$ and $\Gamma \vdash \langle L = S_1 \rangle @ S_2 :: K'$ assumption
$\Gamma \vdash L :: \mathsf{Nm}, \Gamma \vdash S_1 :: \mathsf{Type}, \Gamma \vdash S_2 :: \{t{:}\mathsf{Rec} \mid K \notin t\}$ and $\Gamma \vdash K \equiv \mathsf{Rec}$ by inversion
$\Gamma \vdash L :: \mathsf{Nm}, \Gamma \vdash S_1 :: \mathsf{Type}, \Gamma \vdash S_2 :: \{t{:}\mathsf{Rec} \mid K \notin t\}$ and $\Gamma \vdash K' \equiv \mathsf{Rec}$ by inversion
$\Gamma \vdash \mathsf{Rec} \leq \mathsf{Rec}$ by reflexivity

Case: M is $\mathbf{if}\ \varphi\ \mathbf{then}\ M'\ \mathbf{else}\ N'$

$\Gamma \vdash \mathbf{if}\ \varphi\ \mathbf{then}\ M'\ \mathbf{else}\ N' : T$ and $\Gamma \vdash \mathbf{if}\ \varphi\ \mathbf{then}\ M'\ \mathbf{else}\ N' : S$ assumption
$\Gamma, \varphi \vdash M' : T_1, \Gamma, \neg\varphi \vdash N' : T_2$ and $\Gamma \models T \equiv \mathbf{if}\ \varphi\ \mathbf{then}\ T_1\ \mathbf{else}\ T_2$ by inversion
$\Gamma, \varphi \vdash M' : S_1, \Gamma, \neg\varphi \vdash N' : S_2$ and $\Gamma \models S \equiv \mathbf{if}\ \varphi\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2$ by inversion
$\Gamma, \varphi \models T_1 \equiv S_1 :: K_1$ with $\Gamma \vdash K_1 \leq \mathsf{Type}$ by i.h.
$\Gamma, \neg\varphi \models T_2 \equiv S_2 :: K_2$ with $\Gamma \vdash K_2 \leq \mathsf{Type}$ by i.h.
$\Gamma \models \mathbf{if}\ \varphi\ \mathbf{then}\ T_1\ \mathbf{else}\ T_2 \equiv \mathbf{if}\ \varphi\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 :: \mathsf{Type}$ by rule

Case: M is $\mathbf{if}\ T' :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow M'\ \mathbf{else}\ N'$

$\Gamma \vdash \mathbf{if}\ T' :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow M'\ \mathbf{else}\ N' : T$ and $\Gamma \vdash \mathbf{if}\ T' :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow M'\ \mathbf{else}\ N' : S$ assumption
$\Gamma \vdash T' :: \mathcal{K}', \Gamma \vdash \mathcal{K}, \Gamma, t : \mathcal{K} \vdash M' : T$ and $\Gamma \vdash N' : T$ by inversion
$\Gamma \vdash T' :: \mathcal{K}', \Gamma \vdash \mathcal{K}, \Gamma, t : \mathcal{K} \vdash M' : S$ and $\Gamma \vdash N' : S$ by inversion
$\Gamma \models T \equiv S :: K$ with $\Gamma \vdash K \leq \mathsf{Type}$ by i.h.

Case: T is $\mathbf{if}\ T' :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow S_1\ \mathbf{else}\ S_2$

$\Gamma \vdash \mathbf{if}\ T' :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow S_1\ \mathbf{else}\ S_2 :: K$ and $\Gamma \vdash \mathbf{if}\ T' :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow S_1\ \mathbf{else}\ S_2 :: K'$ assumption
$\Gamma \vdash T' :: \mathcal{K}', \Gamma \vdash \mathcal{K}, \Gamma, t : \mathcal{K} \vdash S_1 :: K$ and $\Gamma \vdash S_2 :: K$ by inversion
$\Gamma \vdash T' :: \mathcal{K}', \Gamma \vdash \mathcal{K}, \Gamma, t : \mathcal{K} \vdash S_1 :: K'$ and $\Gamma \vdash S_2 :: K'$ by inversion
$\Gamma \vdash K \leq K'$ or $\Gamma \vdash K' \leq K$ by i.h.

Case: M is $\boldsymbol{\mu} F{:}T.M'$

$\Gamma \vdash \boldsymbol{\mu} F{:}T.M' : T$ and $\Gamma \vdash \boldsymbol{\mu} F{:}T.M' : S$ assumption
$\Gamma \models T \equiv T' :: \mathcal{K}$ and $\Gamma, F : T \vdash M' : T'$ by inversion
$\Gamma \models S \equiv S' :: \mathcal{K}'$ and $\Gamma, F : T \vdash M' : S'$ by inversion
$\Gamma, F : T \models T' \equiv S' :: K$ with $\Gamma \vdash K \leq \mathsf{Type}$ by i.h.
$\Gamma, F : T \models T \equiv T' :: \mathcal{K}$ and $\Gamma, F : T \models S \equiv S' :: \mathcal{K}'$ by weakening
$\Gamma, F : T \models T \equiv S :: \mathsf{Type}$ by transitivity and conversion
$\Gamma \models T \equiv S :: \mathsf{Type}$ by strengthening

Case: T is $\boldsymbol{\mu} F : (\Pi t{:}K.\, K').\lambda t{::}K.\, T'$

$\Gamma \vdash \boldsymbol{\mu} F : (\Pi t{:}K_1.\, K_2).\lambda t{::}K_1.\, T' :: K$ and $\Gamma \vdash \boldsymbol{\mu} F : (\Pi t{:}K_1.\, K_2).\lambda t{::}K_1.\, T' :: K'$ assumption
$\Gamma, F : \Pi t : K_1.K_2, t : K_1 \vdash T' :: K_2$, $\mathsf{structural}(T', F, t)$ and $\Gamma \vdash K \equiv \Pi t : K_1.K_2$ by inversion
$\Gamma, F : \Pi t : K_1.K_2, t : K_1 \vdash T' :: K_2$, $\mathsf{structural}(T', F, t)$ and $\Gamma \vdash K' \equiv \Pi t : K_1.K_2$ by inversion
$\Gamma \vdash K \leq K'$ by transitivity

Case: M is $\mathbf{recHeadTerm}(M')$

$\Gamma \vdash \mathbf{recHeadTerm}(M') : T$ and $\Gamma \vdash \mathbf{recHeadTerm}(M') : S$ assumption
$\Gamma \vdash M : T', \Gamma \vdash T' :: \{t{:}\mathsf{Rec} \mid \mathbf{headType}(t) \equiv T'' :: \mathcal{K}\}$ and $\Gamma \models T \equiv T''\{T'/t\} :: \mathcal{K}\{T'/t\}$
by inversion
$\Gamma \vdash M : S', \Gamma \vdash S' :: \{t{:}\mathsf{Rec} \mid \mathbf{headType}(t) \equiv S'' :: \mathcal{K}\}$ and $\Gamma \models S \equiv S''\{S'/t\} :: \mathcal{K}\{S'/t\}$
by inversion

$\Gamma \models T' \equiv S' :: K$ with $K \leq \mathsf{Type}$ by i.h.
$\Gamma \vdash T' :: \mathsf{Rec}$ and $\Gamma \models \mathbf{headType}(T') \equiv T''\{T'/t\} :: \mathcal{K}\{T'/t\}$ by inversion
$\Gamma \vdash S' :: \mathsf{Rec}$ and $\Gamma \models \mathbf{headType}(S') \equiv S''\{S'/t\} :: \mathcal{K}\{S'/t\}$ by inversion
$\Gamma \vdash T' :: \{t{:}\mathsf{Rec} \mid \mathsf{nonEmpty}(t)\}$ by conversion
$\Gamma \vdash S' :: \{t{:}\mathsf{Rec} \mid \mathsf{nonEmpty}(t)\}$ by conversion
$\Gamma \models T' \equiv S' :: \{t{:}\mathsf{Rec} \mid \mathsf{nonEmpty}(t)\}$ by Lemma C.6
$\Gamma \models \mathbf{headType}(T') \equiv \mathbf{headType}(S') :: \mathsf{Type}$ by equality rule
$\Gamma \models T \equiv T''\{T'/t\} :: \mathsf{Type}$ by conversion
$\Gamma \models S \equiv S''\{S'/t\} :: \mathsf{Type}$ by conversion
$\Gamma \models T''\{T'/t\} \equiv S''\{S'/t\} :: \mathsf{Type}$ by transitivity

Case: T is $\mathbf{headType}(T')$

$\Gamma \vdash \mathbf{headType}(T') :: K_1$ and $\Gamma \vdash \mathbf{headType}(T') :: K_2$ assumption
$\Gamma \vdash T' :: \{t{:}\mathcal{K} \mid \mathbf{headType}(t) \equiv T'' :: K'\}, \Gamma \vdash T''\{T'/t\} :: K'\{T'/t\}$ and $\Gamma \vdash K_1 \equiv K'\{T'/t\}$
by inversion
$\Gamma \vdash T' :: \{t{:}\mathcal{K}' \mid \mathbf{headType}(t) \equiv T''' :: K''\}, \Gamma \vdash T'''\{T'/t\} :: K''\{T'/t\}$ and $\Gamma \vdash K_2 \equiv K''\{T'/t\}$
by inversion
$\Gamma \vdash \{t{:}\mathcal{K} \mid \mathbf{headType}(t) \equiv T'' :: K'\} \leq \{t{:}\mathcal{K}' \mid \mathbf{headType}(t) \equiv T''' :: K''\}$
or $\Gamma \vdash \{t{:}\mathcal{K} \mid \mathbf{headType}(t) \equiv T'' :: K'\} \geq \{t{:}\mathcal{K}' \mid \mathbf{headType}(t) \equiv T''' :: K''\}$ by i.h.
Subcase 1: $\Gamma \vdash \{t{:}\mathcal{K} \mid \mathbf{headType}(t) \equiv T'' :: K'\} \leq \{t{:}\mathcal{K}' \mid \mathbf{headType}(t) \equiv T''' :: K''\}$
$\Gamma \vdash \mathcal{K} \leq \mathcal{K}'$ and $\Gamma, t{:}\mathcal{K}' \models \mathbf{headType}(t) \equiv T'' :: K' \equiv \mathbf{headType}(t) \equiv T''' :: K''$ by inversion
$\Gamma, t{:}\mathcal{K} \vdash K' \equiv K''$ by entailment
$\Gamma \vdash K'\{T'/t\} \equiv K''\{T'/t\}$ by substitution
$\Gamma \vdash K_1 \leq K_2$
Subcase 2 is symmetric.

Case: T is $\mathbf{tmap}(T_1)\, T_2$

$\Gamma \vdash \mathbf{tmap}(T_1)\, T_2 :: K$ and $\Gamma \vdash \mathbf{tmap}(T_1)\, T_2 :: K'$ assumption
$\Gamma \vdash T_1 :: \mathsf{Gen}_{\mathcal{K}}, \Gamma \vdash T_2 :: \mathcal{K}$ and $\Gamma \vdash K \equiv \mathsf{Type}$ by inversion
$\Gamma \vdash T_1 :: \mathsf{Gen}_{\mathcal{K}'}, \Gamma \vdash T_2 :: \mathcal{K}'$ and $\Gamma \vdash K' \equiv \mathsf{Type}$ by inversion
$\Gamma \vdash K \leq K'$ since $\Gamma \vdash \mathsf{Type} \leq \mathsf{Type}$

□

Theorem 5.9 (Type Preservation). *Let $\Gamma \vdash_S M : T$ and $\Gamma \vdash_s H$. If $\langle H; M\rangle \longrightarrow \langle H'; M'\rangle$ then there exists S' such that $S \subseteq S'$, $\Gamma \vdash_{S'} H'$ and $\Gamma \vdash_{S'} M' : T$.*

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

Case: $$\frac{T_0 \to T_0'}{\langle H; (\Lambda t{::}K.\, M)[T_0]\rangle \longrightarrow \langle H; (\Lambda t{::}K.\, M)[T_0']\rangle}$$

$\Gamma \vdash T \equiv U :: \mathcal{K}$ where $\Gamma \vdash \Lambda t{::}K.\, M : T_1, \Gamma \vdash T_0 :: K, \Gamma \vdash U :: \mathcal{K}$,
$\Gamma \vdash T_1 :: \{f{::}\mathsf{Gen}_K \mid \mathbf{tmap}(f)\, T_0 \equiv U :: \mathcal{K}\}$ by inversion
$\Gamma \vdash T_1 \equiv \forall t{::}K.S :: \mathsf{Gen}_K$ and $\Gamma, t{::}K \vdash S :: \mathcal{K}$ by inversion
$\Gamma \vdash S\{T_0/t\} :: \mathcal{K}$ by substitution
$\Gamma \vdash T_0 \equiv T_0' :: K$ by definition
$\Gamma \vdash S\{T_0/t\} \equiv S\{T_0'/t\} :: \mathcal{K}$ by functionality
$\Gamma \vdash \mathbf{tmap}(\forall t{::}K.S)\, T_0 \equiv \mathbf{tmap}(\forall t{::}K.S)\, T_0' :: \mathcal{K}$ by equality
$\Gamma \vdash U \equiv S\{T_0/t\} :: \mathcal{K}$ by transitivity
$\Gamma \vdash U \equiv S\{T_0'/t\} :: \mathcal{K}$ by transitivity

$\Gamma \vdash (\Lambda t{::}K.\,M)[T_0'] : S\{T_0'/t\}$ by typing
$\Gamma \vdash (\Lambda t{::}K.\,M)[T_0'] : U$ by conversion

Case: $\dfrac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; \langle \ell = M\rangle @N\rangle \longrightarrow \langle H'; \langle \ell = M'\rangle @N\rangle}$

$\Gamma \vdash_S T \equiv \langle\!\langle L : T'\rangle\!\rangle @T'', \Gamma \vdash_S \ell \equiv L :: \mathsf{Nm}, \Gamma \vdash_S M : T'$ and $\Gamma \vdash_S N : T''$ by inversion
$\exists S'$ such that $S \subseteq S', \Gamma \vdash_{S'} H'$ and $\Gamma \vdash_{S'} M' : T'$ by i.h.
$\Gamma \vdash_{S'} \langle \ell = M'\rangle @N : \langle\!\langle L : T'\rangle\!\rangle @T''$ by RecCons rule

Case: $\dfrac{\langle H; M\rangle \longrightarrow \langle H'; M'\rangle}{\langle H; \mathbf{recHeadTerm}(M)\rangle \longrightarrow \langle H'; \mathbf{recHeadTerm}(M')\rangle}$

$\Gamma \vdash_S T \equiv T'\{S'/t\} :: \mathcal{K}\{S'/t\}, \Gamma \vdash_S M : S'$ and
$\Gamma \vdash S' :: \{t{:}\mathsf{Rec} \mid \mathbf{headType}(t) \equiv T' :: \mathcal{K}\}$ by inversion
$\exists S_0$ such that $S \subseteq S_0, \Gamma \vdash_{S_0} H'$ and $\Gamma \vdash_{S_0} M' : S'$ by i.h.
$\Gamma \vdash_{S_0} \mathbf{recHeadTerm}(M') : T'\{S'/t\}$ by typing rule

Case: $\langle H; \mathbf{recHeadTerm}(\langle \ell = v\rangle @v')\rangle \longrightarrow \langle H; v\rangle$

$\Gamma \vdash_S \mathbf{recHeadTerm}(\langle \ell = v\rangle @v') : T'$ and $\Gamma \vdash_s v : T'$ by inversion

Case: $\dfrac{\overline{\Gamma} \models \varphi}{\langle H; \mathbf{if}\ \varphi\ \mathbf{then}\ M\ \mathbf{else}\ N\rangle \longrightarrow \langle H; M\rangle}$

$\Gamma \models T \equiv \mathbf{if}\ \varphi\ \mathbf{then}\ T_1\ \mathbf{else}\ T_2 :: K$ with $\Gamma, \varphi \vdash_S M : T_1$ and $\Gamma, \neg\varphi \vdash_S N : T_2$ by inversion
$\Gamma \models \mathbf{if}\ \varphi\ \mathbf{then}\ T_1\ \mathbf{else}\ T_2 \equiv T_1 :: K$ by eq. rule
$\Gamma \models T \equiv T_1 :: K$ by transitivity
$\Gamma \vdash_S M : T_1$ by cut

Case: $\dfrac{}{\langle H; \boldsymbol{\mu} F{:}T.M\rangle \longrightarrow \langle H; M\{\boldsymbol{\mu} F{:}T.M/F\}\rangle}$

$\Gamma, F : T \vdash M : T$ and structural(F, M) by inversion $\Gamma \vdash M\{\boldsymbol{\mu} F{:}T.M/F\} : T$ by substitution

Case: $\dfrac{\Gamma \vdash T :: K}{\langle H; \mathbf{if}\ T' :: K\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N\rangle \longrightarrow \langle H; M\{T'/t\}\rangle}$

$\Gamma \vdash T' :: K', \Gamma \vdash K, \Gamma, t{:}K \vdash M : T''$ and $\Gamma \vdash N : T''$ by inversion
$\Gamma \vdash T :: K$ assumption
$\Gamma \vdash M\{T'/t\} : T''$ by substitution

□

Lemma 5.10 (Type Progress). *If $\Gamma \vdash T :: K$ then either T is a type value or $T \to T'$, for some T'.*

Proof. Straightforward induction on kinding. □

Theorem 5.11 (Progress). *Let $\cdot \vdash_S M : T$ and $\cdot \vdash_S H$. Then either M is a value or there exists S' and M' such that $\langle H; M\rangle \longrightarrow \langle H'; M'\rangle$.*

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. □