Refinement Kinds
Type-safe Programming with Practical Type-level Computation

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 develop in the context of an explicitly polymorphic ML-like language with type-level computation. Just 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 meta-programming, which are often found in modern software development, but not typically expressible in a type-safe manner in general purpose languages. We validate our approach both formally and pragmatically by establishing the standard meta-theoretical results of type safety and via a prototype implementation of a kind, type-checker and interpreter for our language.

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

1 INTRODUCTION

Current software development practices increasingly rely on many forms of automation, often based on tools that generate code from various types of specifications, leveraging the various reflection and meta-programming facilities that modern programming languages provide. A simple example would be a function that given any record type would produce a factory of mutable instances of the given record type. As a more involved and useful example consider a code generator that given as input an XML database schema, produces all the code needed to create and manipulate a database instance of such schema with some appropriate database connector.

Automated code generation, domain specific languages, and meta-programming are increasingly becoming productivity drivers for the software industry, while also bringing programming more accessible to non-experts, and, more generally, increasing the level of abstraction expressible in languages and tools for program construction. Meta-programming are better 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 other facilities, allowing both code and 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 challenge is then the key motivation for our paper, which 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 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 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 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. Moreover, by leveraging the stratification between types and kinds, our design shows that arguably advanced type-level features can be integrated into a general purpose language without the need to fundamentally alter the language’s type system and its associated rules.

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 give rise to a notion of subkinding by entailment in the refinement logic. For example, we introduce a least upper bound kind for each kind, 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. Moreover, the deduction of relevant structural properties of such tree representations of types is amenable to rather efficient implementation, unlike properties on the typical value domains (e.g., integers, arrays) manipulated by mainstream languages, and easier to automate using off-the-shelf SMT solvers (e.g. [Barrett et al. 2011; de Moura and Bjørner 2008]). Remarkably, even if types in our system can essentially be 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 extent type theories in a fully automated way.

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

The record type destructors are functions headLabel(R), headType(R) and 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 are 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 types Type via subkinding, which can be further specialized via kind refinement.

For example, we may define the record type Person ≜ ⟨name : String⟩@⟨age : Int⟩@(), which we conveniently abbreviate by ⟨name : String; age : Int⟩. We then have that headLabel(Person) =
The addField function is denoted by \( \text{addField} \) and it 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 \( r \). Notice that the \( \text{kind} \) of all record types that do not contain label \( l \) is represented by the refinement kind \( \{s:\text{Rec} \mid l \notin \text{lab}(s)\} \).

A refinement kind in our system is noted \( \{t:\mathcal{K} \mid \varphi(t)\} \), where \( \mathcal{K} \) is a basic kind, and the logical formula \( \varphi(t) \) expresses a constraint on the type \( t \) that inhabits \( \mathcal{K} \). As expected in refinement type systems \cite{Bengtson2011,Swamy2011,Vazou2014}, our underlying logic of refinements includes 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 by

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

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

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

A use of the type-level function addField above is, for instance, the definition of the following \textit{term-level} polymorphic record extension function

\[
\text{addField} : \forall l::\text{Nm}.\forall t::\text{Type}.\forall r::\{s::\text{Rec} \mid l \notin \text{lab}(s)\}.t \rightarrow r \rightarrow \text{addFieldType} l t r
\]

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:

\[
l::\text{Nm}, t::\text{Type}, r::\{s::\text{Rec} \mid l \notin \text{lab}(s)\}, x:t, y:r \vdash \text{addField} l t r x y : \text{addFieldType} l t r
\]

An instance of this judgement yields:

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

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

\[
(\text{addField} \text{name} \text{String} \langle \text{age} : \text{Int} \rangle \langle \text{age} = 20 \rangle) \rightarrow^* \langle \text{name} = \text{“jack”}; \text{age} = 20 \rangle
\]

\[
(\text{addFieldType} \text{name} \text{String} \langle \text{age} : \text{Int} \rangle) \equiv \langle \text{name} : \text{String}; \text{age} : \text{Int} \rangle
\]
Using the available refinement principles, our system can also derive the following more precise
kinding for the type addFieldType l t r:

\[ l::Nm, t::Type, r::s::Rec | l \notin \text{lab}(s) ] \mapsto \text{addFieldType } l t r :: s::Rec | s = \langle l : t \rangle @r \]

**Contributions.** We summarise the main contributions of this work: First, we illustrate 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 an ML-like polymorphic λ-calculus (Section 4) with
records, references and collections, supporting type-level computation over types of all kinds. Third,
we report on our implementation of a prototype kind and type-checker for our theory (Section 5),
which validates the examples of our paper and the overall feasibility of our approach. We plan to
submit this prototype as an artifact of this paper. Finally, we establish the key meta-theoretical result
(Section 6) of type safety through type unicity, type preservation and progress (Theorems 6.5, 6.6
and 6.8, respectively).

We conclude with an overview of related work (Section 7), and offer some concluding remarks
and discussion of future work (Section 8). Appendices A, B and C in the anonymised supplement
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 challenging (from a static typing perspective) meta-programming idioms.

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

\[
\text{genConstr} \triangleq \Delta S::\{r::\text{Rec} | \neg \text{empty}(r)\}, \Delta V::\{v::\text{Rec} | \text{lab}(v)\#\text{lab}(S)\}, \lambda v:V.
\]

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

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

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

Given a non-empty record type \( S \), function genConstr returns a constructor function for a mutable
record whose fields are specified by \( S \). We use a pragmatic notation to express recursive definitions
(coincident with that of our implementation), 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 \( \langle \rangle \langle \rangle \) evaluates
to a value equivalent to \( \lambda x::\text{String}, \lambda y::\text{Int}. \langle \text{age } = \text{ref } y; \text{name } = \text{ref } x \rangle \).

Notice that function genConstr accepts any non-empty record type \( S \), and proceeds by recur-
sion on the structure of \( S \), 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 \( \langle \rangle \) (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 \( \text{lab}(V)\#\text{lab}(S) \), we can automatically deduce
that \( \text{headLabel}(S) \notin \text{lab}(V) \), needed to kind check \( \langle \text{headLabel}(S) : \text{ref headType}(S) \rangle @V; \) and
As a second example, we develop a generic function From Record Types to XML Tables.

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

\[
\text{Gtype} :: \forall S:(r::\text{Rec} \mid \neg \text{empty}(r)). \forall V:(v::\text{Rec} | \text{lab}(v)\#\text{lab}(S)). V \rightarrow (\text{Gtype} S V)
\]

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

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

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], but where refinement kinds allow for extreme simplicity. We start by introducing an auxiliary type-level Map function, that computes 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 \(R\).

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

\[
\text{Map} \triangleq \lambda G:(\Pi X :: \text{Type}. \text{Type}). \lambda R::\text{Rec}.
\]

\[
\begin{cases}  
\text{if } \neg \text{empty}(R) \text{ then } (\text{headLabel}(R) : G \text{headType}(R))@(\text{Map } G \text{tail}(R)) \text{ else } \langle \rangle 
\end{cases}
\]

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

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

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

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

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

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

\[
\text{MkTable} \triangleq \lambda R::\text{Rec}. \lambda M::\text{MkTableType } R. \lambda r::R. \begin{cases}  
\text{if } \neg \text{empty}(R) \text{ then }  
\langle \text{tr} > < \text{th} > + M.\text{recHeadLabel}(M).\text{tag} + " < / \text{th} > " + M.\text{recHeadLabel}(M).\text{toStr } r.\text{recHeadLabel}(M) + " < / \text{td} > </ \text{tr} > "  
\end{cases}
\]

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 \(M.\text{recHeadLabel}(M).\text{tag}\). Notice that, by declaration, \(R::\text{Rec}\) and \(r::R\). However, the expression under consideration is to be typed under the assumption that \(\neg \text{empty}(R)\), which is added to the current set of refinement...
We thus derive \( \text{headLabel}(TT) \equiv \text{headLabel}(R) \). Then

\[
\text{headType}(\text{MkTableType } R) \equiv \text{XForm } \text{headType}(R) \equiv \langle \text{tag} : \text{String}; \text{toStr} : \text{headType}(R) \rightarrow \text{String} \rangle
\]

Hence \( M.\text{headLabel}(M) \cdot \text{tag} : \text{String} \). By a similar reasoning, we conclude \( r.\text{recHeadLabel}(M) : \text{headType}(R) \). In Section 3 we show how refinements and equalities derived therein are integrated into typing and kinding. Moreover, in Section 5 we detail how refinements can be represented and discharged via SMT solvers in order to make fully precise the reasoning sketched above.

Generating Getters and Setters. As a final introductory example, we develop a generic function \( \text{MkMut} \) 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 \( \text{MutableRec} \) function, that returns the mutable record type obtained from a record type \( R \) in terms of Map:

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

We then define the auxiliary type-level \( \text{SetGet} \) function, that returns the record type that exposes the getter/setter interface generated from record type \( R \):

\[
\begin{align*}
\text{SetGetRec} & :: \Pi R :: \text{Rec}. \{ r :: \text{Rec} \mid \text{lab}(r) = \text{set++lab}(R) \cup \text{get++lab}(R) \} \\
\text{SetGetRec} & \equiv \lambda R :: \text{Rec}. \\
& \quad \text{if } \neg \text{empty}(R) \text{ then} \\
& \quad \quad \langle \text{set++headLabel}(R) : 1 \rightarrow \text{headType}(R) \rangle @ \\
& \quad \quad \langle \text{set++headLabel}(R) : \text{headType}(R) \rightarrow 1 \rangle @ \\
& \quad \quad \text{SetGetRec} \text{ tail}(R) \\
& \quad \text{else } () 
\end{align*}
\]

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 \( \text{SetGet} \) 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 \( \text{MkMut} \) function:

\[
\begin{align*}
\text{MkMut} & :: \forall R :: \text{Rec}. \text{MutableRec } R \rightarrow \text{SetGetRec } R \\
\text{MkMut} & \equiv \Delta R :: \text{Rec}. \\
& \quad \lambda r :: \text{MutableRec } R. \\
& \quad \text{if } \neg \text{empty}(R) \text{ then} \\
& \quad \quad \langle \text{get++headLabel}(R) = \lambda x : 1.!(r.\text{recHeadLabel}(R)) \rangle @ \\
& \quad \quad \langle \text{set++headLabel}(R) = \lambda x : \text{headType}(R). r.\text{recHeadLabel}(R) := x \rangle @ \\
& \quad \quad \text{MkMut} \text{ tail}(R) \text{ recTail}(r) \\
& \quad \text{else } () 
\end{align*}
\]

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

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

where \( \langle \text{MkMut } \text{Person } r \rangle :: \text{SetGetRec } \text{Person} \).
We write 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, types and the type-level operations enabled by this fine-grained view of kinds, addressing kind refinements.

Our notion of record type, as introduced in Section 2, is essentially a type-level list of pairs of labels and types which maintains the invariant that all labels in a record must be distinct.
We thus have the type of empty records (), and the constructor \( \langle L : T \rangle @ R \), 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 produces 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 \( \lambda \)-calculus allows for structural recursion, we can define a suitable record projection type construct in terms of these lower-level primitives.

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

Polymorphic function types are represented by \( \forall t::K.T \) (with \( t \) bound in \( T \), as usual). Note that the kind annotation for the type variable \( t \) allows us to express not only general parametric polymorphic functions (by specifying the kind as Type) but also a form of sub-kind polymorphism, since we can restrict the kind of \( t \) to a specific kind such as \( \text{Ref} \) or \( \text{Fun} \), or to a refined kind. For instance, we can specify the type \( \forall t::\text{Fun}.t \to \text{dom}(t) \to \text{img}(t) \) of functions that, given a function type \( t \), a function of such a type and a value in its domain produce a value in its image (i.e. the type of function application).

**Collections and References.** The type of collections of elements of type \( T \) is written as \( T^* \), with the associated type destructor colOf\((T)\), which projects out the type of the collection elements.

Similarly, reference types \( \text{ref} T \) are bundled with a destructor refOf\((T)\) which determines the type of the referenced elements.

**Kind Test.** Just as many programming languages have a type case construct [Abadi et al. 1991] that allows for the runtime testing of the type of a given expression, our \( \lambda \)-calculus of types has a kind case construct, if \( T :: K \), as \( t \Rightarrow S \) else \( U \), which checks the kind of type \( T \) against kind \( K \), computing to type \( S \) if the kinds match and to \( U \) otherwise. Coupled with a term-level analogue, this enables ad-hoc polymorphism, allowing us to express non-parametric polymorphic functions.

### 3.1 Type-level Functions and Refinements

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

Type-level functions are **dependently kinded**, with kind \( \Pi t::K.K' \) (i.e. the kind of the image type in a type \( \lambda \)-abstraction can refer to its type 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 the kinding system to specify and enforce logical constraints on the structure of types.

A kind refinement, written \( \{t::K | \varphi \} \), where \( K \) is a basic kind, and \( \varphi \) is a logical formula (with \( t \) bound in \( \varphi \)), characterises types \( T \) of kind \( K \) such that the property \( \varphi \) holds of \( T \) (i.e. \( \varphi \{T/t\} \) is true). The language of properties \( \varphi \) can refer to the syntax of types, extended with a refinement-level notion of label set of a (record) type, \( \text{lab}(T) \), and a notion of label set concatenation, \( T ++ S \), where \( T \) is such an extended type. Refinements \( \varphi, \psi \) consist of propositional logic formulae, (logical) equality, \( T = S \), an empty record predicate empty\((T)\), and basic label set predicates and such as...
label inclusion ($T \in S$) and set apartness ($T \not\# S$). The intended target logic is a typed first-order logic with uninterpreted functions, finite sets, inductive datatypes and equality [Barrett et al. 2011]. While such theories are in general undecidable, the state-of-the-art in SMT solving [Bansal et al. 2018; Reynolds et al. 2013] procedures can be applied to effectively cover the automated reasoning needed in our work.

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

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

if $\text{headLabel}(t) = l$ then $\text{tail}(t)$ else $(\text{headLabel}(t) : \text{headType}(t))\@ (F(\text{tail}(t)))$

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 lab($r$) denotes the refinement-level set of labels of $r$). Such a function combines structural recursion (where tail($t$) is correctly deemed as structurally smaller than $t$) with our type-level refinement test, if $\varphi$ then $T$ else $S$. We note that the well-kindness of such a function relies on the ability to derive that, when the record label headLabel($t$) is not $l$, since we know that $l$ must be in $t$, tail($t$) is a record type containing $l$. This kind of reasoning is easily decided using SMT-based techniques [Barrett et al. 2011].

### 3.2 Kinding and Type Equality

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

The kinding judgment is written $\Gamma \vdash T :: K$, denoting that type $T$ has kind $K$ under the assumptions in the context $\Gamma$. Contexts contain assumptions of the form $t:K$, $x:T$ and $\varphi \rightarrow t$ stands for a type of kind $K$, $x$ stands for a term of type $T$ and refinement $\varphi$ is assumed to hold, respectively. Kinding relies on a context well-formedness judgment, written $\Gamma \vdash$, a kind well-formedness judgment $\Gamma \vdash K$, subkinding judgment $\Gamma \vdash K \leq K'$ and the refinement well-formedness and entailment judgments, $\Gamma \vdash \varphi$ and $\Gamma \models \varphi$. Context well-formedness simply checks that all types, kinds and refinements in $\Gamma$ are well-formed. Kind well-formedness is defined in the standard way, relying on refinement well-formedness (see Appendix A.1), which requires that formulae and types in refinements be well-formed. Subkinding codifies the informal reasoning from the start 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 logically equivalent refinements. We define equality between $K$ and $K'$ by requiring $K \leq K'$ and $K' \leq K$.

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 and Type Properties.** A kind refinement is introduced by the rule (kref) below. Given a type $T$ of kind $K$ and a valid property $\varphi$ of $T$, we are justified in stating that $T$ is of kind $\{t::K \mid \varphi\}$.

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

Rule (entails) specifies that a refinement formula is satisfiable if it is well-formed (i.e., a syntactically well-formed boolean expression which may include equalities on terms of base kind) and if the
representation of the context \( \Gamma \) and the refinement \( \varphi \) as an implicational formula is SMT-valid. The context and refinement representation is discussed in Section 5.

Crucially, since we rely on an underlying logic with inductive types (which includes constructor and destructor equality reasoning), refinements can specify the shape of the refined types. For instance, the expected \( \beta \)-equivalence reasoning for records allows us to derive \( \langle \ell : \text{Bool} \to \text{Bool} \rangle @ \{ t : \text{Rec} | \text{headType}(t) = \text{Bool} \to \text{Bool} \} \). In general, we provide an equality elimination rule for refinements (eq-eqelim), allowing for such logical equalities to be internalized in definitional equality of our theory:

\[
\frac{\Gamma \vdash T :: \{ t : \mathcal{K} | t = S \} \quad \Gamma \vdash S :: \mathcal{K}}{\Gamma \vdash T \equiv S :: \mathcal{K}} \quad \text{(r-eqelim)}
\]

These principles become particularly interesting when reasoning from refinements that appear in type variables. For instance, the type \( \forall t : \{ f : \text{Fun} | \text{dom}(f) = \text{Bool} \land \text{img}(f) = \text{Bool} \}. t \to \text{Bool} \) can be used to type the term \( \Lambda t : \{ f : \text{Fun} | \text{dom}(f) = \text{Bool} \land \text{img}(f) = \text{Bool} \}. \lambda f : t. (f \text{ true}) \), where \( \Lambda \) is the binder for polymorphic functions, as usual. Crucially, typing (and kinding) exploits not only the fact that we know that the type variable \( t \) stands for a function type, but also that the domain and image are the type \( \text{Bool} \), 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 of function kind. This style of reasoning, which is not readily available even in powerful type theories such as that of Coq [CoqDevelopmentTeam 2004], is akin to that of a setting with singleton kinds [Stone and Harper 2006].

As we have shown in Section 2, properties can also be tested in types through a conditional construct if \( \varphi \) then \( T \) else \( S \). Provided that the property \( \varphi \) is well-formed, if \( T \) is of kind \( K \) assuming \( \varphi \) and \( S \) of kind \( K \) assuming \( \neg \varphi \), then the conditional test is well-kindred, as specified by the rule (k-ite). The equality principles for the property test rely on validity of the specified property, as expected (with a degenerate case where both branches are equal types). We note that \( \Gamma, \varphi \vdash T :: K \) can effectively be represented as \( \Gamma, x : \{ _\varphi | \varphi \} \vdash T :: K \) where \( x \) is fresh. This representation encodes \( \varphi \) in the context through a "dummy" refinement that simply asserts the property.

\[
\begin{align*}
\frac{\Gamma \vdash \varphi \quad \Gamma, \varphi, T :: K \quad \Gamma, \neg \varphi, S :: K}{\Gamma \vdash \text{if} \varphi \text{ then } T \text{ else } S :: K} & \quad \text{(k-ite)} \\
\frac{\Gamma \vdash \text{if} \varphi \text{ then } T \text{ else } S :: K}{\Gamma \vdash \varphi \quad \Gamma, \varphi, T_1 :: K \quad \Gamma, \neg \varphi, T_2 :: K}{\Gamma \vdash \text{if} \varphi \text{ then } T_1 \text{ else } T_2 :: K} & \quad \text{(eq-iteT)} \\
\frac{\Gamma \vdash \text{if} \varphi \text{ then } T_1 \text{ else } T_2 :: K}{\Gamma \vdash \varphi \quad \Gamma, \varphi, T :: K \quad \Gamma, \neg \varphi, T :: K}{\Gamma \vdash \text{if} \varphi \text{ then } T \text{ else } T :: K} & \quad \text{(eq-iteE)}
\end{align*}
\]

Type Functions and Function Types. The rules that govern kinding and equality of type-level functions consist of the standard rules plus the extensionality principles of [Stone and Harper 2006] (to streamline the presentation, we omit the congruence rules for equality):

\[
\begin{align*}
\frac{\Gamma \vdash K \quad \Gamma, t : K, T :: K'}{\Gamma \vdash \lambda t : K, T :: \Pi t : K.K'} & \quad \text{(k-fun)} \\
\frac{\Gamma \vdash T :: \Pi t : K.K' \quad \Gamma \vdash S :: K}{\Gamma \vdash T \mapsto S :: K'} & \quad \text{(k-app)} \\
\frac{\Gamma \vdash T :: \Pi t : K_1.K_3 \quad \Gamma, t : K_1 :: K_2 \quad x \neq f v(T) }{\Gamma \vdash T :: \Pi t : K_1.K_2} & \quad \text{(k-ext)} \\
\frac{\Gamma \vdash S :: \Pi t : K_1.K_3 \quad \Gamma \vdash T :: \Pi t : K_1.K_4}{\Gamma \vdash S \equiv T :: \Pi t : K_2} & \quad \text{(eq-funext)} \\
\frac{\Gamma \vdash T :: K' \quad \Gamma \vdash S :: K}{\Gamma \vdash (\lambda t : K, T) S :: T/S :: K' / (S/t)} & \quad \text{(eq-funapp)}
\end{align*}
\]
Rules \((k\text{-ext})\) and \((eq\text{-funext})\) allow for basic extensionality principles on type-level functions. The former states that an \(\eta\)-like typing rule, where a type \(T\) that is a type-level function from \(K_1\) to \(K_3\) can be seen as a type-level function from \(K_1\) to \(K_2\) if \(T\) applied to a fresh variable of type \(K_1\) can derive a type of kind \(K_2\). Rule \((eq\text{-funext})\) is the analogue rule for type equality. We note that such rules, although they allow us to equate types such as \(\lambda t::s:\text{Type} \mid t = \text{Bool} \rightarrow \text{Bool}\).\(t\) and \(\lambda t::s:\text{Type} \mid t = \text{Bool} \rightarrow \text{Bool}\).\(t\) in general, they do not disturb the decidability of kinding or equality [Stone and Harper 2006].

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

\[
\frac{\Gamma, F:\Pi t::K. K', t:K \vdash T :: K' \quad \text{structural}(T, F, t)}{\Gamma \vdash \mu F : (\Pi t::K. K').\lambda t::K. T :: \Pi t::K.K'} \quad (k\text{-fix})
\]

\[
\frac{\Gamma; t::K \vdash K_2 \quad \Gamma, F:\Pi t::K_1.K_2, t::K_1 \vdash T :: K_2 \quad \Gamma \vdash \mu S :: \Gamma_1 \quad \text{structural}(T, F, t)}{\Gamma \models (\mu F : (\Pi t::K_1, K_2).\lambda t::K_1, K_2). T \equiv T/S/t) \{((\mu F : (\Pi t::K_1, K_2).\lambda t::K_1, K_2). T)/\}/ K_2\{S/t\}}
\]

The predicate \(\text{structural}(T, F, t)\) 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 \(\text{structural}(T, F, t)\) holds if all occurrences of \(F\) in \(T\) are applied to terms smaller than \(t\), where the notion of size is given by \(\text{elim}(t) < t\), where \(\text{elim}(t)\) stands for an appropriate destructor applied to \(t\) (e.g., if \(t\) is of kind \(\text{Fun}\) then \(\text{dom}(t) < t\)). The equality rule allows for the appropriate unfolding of the recursion to take place. Naturally, the implementation of this rule follows the standard lazy unfolding approach to recursive definitions.

Polymorphic function types are assigned kind \(\text{Gen}_K\), as expected:

\[
\frac{\Gamma \vdash K \quad \Gamma, t::K \vdash T :: K' \quad \Gamma \vdash \forall t::K. T :: \text{Gen}_K}{\Gamma \vdash \forall t::K.T :: \text{Gen}_K} \quad (k\text{-}\forall)
\]

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

\[
\frac{\Gamma \vdash T :: K \quad \Gamma \vdash S :: K' \quad \Gamma \vdash \text{Fun}}{\Gamma \vdash T \rightarrow S :: \text{Fun}} \quad (k\text{-fun})
\]

\[
\frac{\Gamma \vdash T :: \text{Fun} \quad \Gamma \vdash \text{dom}(T) :: \text{Type}}{\Gamma \vdash \text{img}(T) :: \text{Type}} \quad (k\text{-codom})
\]

\[
\frac{\Gamma \vdash \text{dom}(T \rightarrow S) :: T \equiv T :: \text{Type}}{\Gamma \vdash \text{img}(T \rightarrow S) :: S :: \text{Type}} \quad (eq\text{-img})
\]

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

\[
\frac{\Gamma \vdash \{\}}{\Gamma \vdash \{:: \text{Rec}} \quad (k\text{-recnil})
\]

\[
\frac{\Gamma \vdash L :: \text{Nm}}{\Gamma \vdash \{:: \text{Rec}} \quad (k\text{-reccons})
\]

\[
\frac{\Gamma \vdash \{:: L :: \text{Type}}{\Gamma \vdash \langle L :: T \rangle @ S :: \text{Rec}} \quad (k\text{-label})
\]

\[
\frac{\Gamma \vdash \text{headType}(T) :: \text{Type}}{\Gamma \vdash \text{headLabel}(T) :: \text{Nm}} \quad (k\text{-tail})
\]

The rule for non-empty records crucially requires that the tail \(S\) of the record type must not contain the field label \(L\). The rules for the various destructors require that the record be non-empty, projecting out the appropriate data. The equality principles for the three destructors are fairly straightforward, projecting out the appropriate record type component, provided the record is well-kindred.
\[(\text{EQ-HEADLABEL})\]
\[
\Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{ t : \text{Rec} \mid L \notin \text{lab}(t) \}
\]
\[
\Gamma \models \text{headLabel}(\langle L : T \rangle_{S}) \equiv L :: \text{Nm}
\]
\[(\text{EQ-HEADTYPE})\]
\[
\Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{ t : \text{Rec} \mid L \notin \text{lab}(t) \}
\]
\[
\Gamma \models \text{headType}(\langle L : T \rangle_{S}) \equiv T :: \text{Type}
\]
\[(\text{EQ-TAIL})\]
\[
\Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{ t : \text{Rec} \mid L \notin \text{lab}(t) \}
\]
\[
\Gamma \models \text{tail}(\langle L : T \rangle_{S}) \equiv S :: \text{Rec}
\]

Collections and Reference Types. At the level of kinding, there is virtually no difference between a collection 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.

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 specialized types at the kind level. For instance, the kind of record types \(\text{Rec}\) is a specialisation of Type, the kind of all types, and similarly for the other type-level base constructs of the theory. We formalise this via a subkinding relation, which also internalises kind equality, and the corresponding subsumption rule:

\[
(\text{K-COL})\]
\[
\Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash T :: \text{Col}
\]
\[
\Gamma \vdash \text{refOf}(T) :: \mathcal{K}
\]
\[
\Gamma \models \text{colOf}(T) \equiv T :: \text{Type}
\]
\[(\text{K-REF})\]
\[
\Gamma \vdash T :: \mathcal{K}
\]
\[
\Gamma \vdash \text{refOf}(\text{ref} T) \equiv T :: \text{Type}
\]

Rule (\text{SUB-REFKIND}) specifies that a refined kind is always a subkind of its unrefined variant. Rule (\text{SUB-REF}) allows for subkinding between refined kinds, by requiring that the basic kind respects subkinding and that the refinement of the more precise kind implies that of the more general one.

Kind Case and Bottom. The kind case type-level mechanism is kinded in a natural way (rule (\text{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.

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

Our treatment of \(\bot\) allows for \(\bot\) to be of any (well-formed) kind, provided one can conclude \(\bot\) is valid. The associated equality principles implement the kind case by testing the specified kind against the derivable kind of type \(T\). When \(\bot\) is provable from \(\Gamma\) then we can derive any equality via rule (\text{EQ-BOT}).

\[
(\text{EQ-KCASE})\]
\[
\Gamma \vdash T :: \mathcal{K} \quad \Gamma, t : \mathcal{K} + S :: \mathcal{K}' \quad \Gamma \vdash U :: \mathcal{K}'
\]
\[
\Gamma \models \text{if } T :: \mathcal{K} \text{ as } t \Rightarrow S \text{ else } U :: \mathcal{T}(T/t) :: \mathcal{K}'
\]
\[
\Gamma \models \bot \quad \Gamma \vdash T :: \mathcal{K}
\]
\[
\Gamma \models \bot \equiv T :: \mathcal{K}
\]
\[
\Gamma \vdash T :: \mathcal{K}_0 \quad \Gamma \vdash \mathcal{K}_0 \neq \mathcal{K} \quad \Gamma, t : \mathcal{K} + S :: \mathcal{K}' \quad \Gamma \vdash U :: \mathcal{K}'
\]
\[
\Gamma \models \text{if } T :: \mathcal{K} \text{ as } t \Rightarrow S \text{ else } U :: \mathcal{K}'
\]
Example 3.1 (Representing Record Field Selection in types and values). With the development presented up to this point we can 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 $L :: Nm$ produces a recursive type-function that essentially iterates over a type record of kind $\{r::Rec | \ell \in lab(r)\}$:

$$
\lambda L::Nm. \mu F : (\Pi t : \{r::Rec | L \in lab(r)\}. \text{Type}). \lambda t : \{r::Rec | L \in lab(r)\}. \text{Type}
$$

if $headLabel(t) = L$ then $headType(t)$ else $F(tail(t)) :: \Pi L : Nm. \Pi t : \{r::Rec | L \in lab(r)\}. \text{Type}$

The function iteratively tests the label at the head of the record against $L$, producing the type at the head of the record on a matches and recurring otherwise. It is instructive to consider the kinding for the property test construct (let $\Gamma_0$ be $L::Nm, F::\Pi t : \{r::Rec | L \in lab(r)\}. \text{Type}, t : \{r::Rec | L \in lab(r)\}$):

$$
\Gamma_0 \vdash \text{headLabel}(t) = L \quad \mathcal{D} \quad \mathcal{E}
$$

where $\mathcal{D}$ is a derivation of $\Gamma_0, \text{headLabel}(t) = L \vdash \text{headType}(t) :: \text{Type}$ and $\mathcal{E}$ is a derivation of $\Gamma_0, \neg(\text{headLabel}(t) = L) \vdash F(tail(t)) :: \text{Type}$. To show that $\text{headLabel}(t) = L$ is well-formed we must be able to derive $t : \{r::Rec | \neg\text{empty}(r)\}$ from $t : \{r::Rec | L \in lab(r)\}$, which is achieved via subkinding, by appealing to entailment in our underlying theory (see Section 5). Similarly, the derivation $\mathcal{E}$ requires the ability to conclude that $\text{tail}(t) : \{r::Rec | L \in lab(r)\}$, using the information that $t : \{r::Rec | L \in lab(r)\} \land \neg(\text{headLabel}(t) = L)$, which is also a valid entailment.

4 A PROGRAMMING LANGUAGE WITH KIND REFINEMENTS

Having covered the key details of kinding and type equality, we finally introduce the syntax and typing for our programming language per se, an ML-like functional language with a higher-order store. The syntax of which is given in Figure 2. Most constructs are standard.

We highlight the treatment of records, mirroring that of record types, as heterogeneous lists of (pairs of) field labels and terms equipped with the appropriate destructors. Collections are built from the empty collection $\varepsilon$ and the concatenation of an element $M$ with a collection $N$, $M :: N$, with the usual case analysis $\text{case } M \text{ of } (\varepsilon \Rightarrow N_1 | x::xs \Rightarrow N_2)$ that reduces to $N_1$ when $M$ evaluates to the empty collection and to $N_2$ otherwise, where $x$ is instantiated with the head of the collection.

and 

and 

and 

Fig. 3. Typing Rules
In fact, the advanced kinding and type equality features manifest themselves in typing via the (CONV) conversion rule, (KINDCASE) and the (\langle )L_2 \rangle record formation rule – this further reveals a potential strength of our approach, since it allows for a clean integration of powerful type-level reasoning and meta-programming without dramatically changing the surface-level language. For instance, the following term is well-typed:

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

Despite not knowing the exact form of the function type that is to be instantiated for \( t \), by refining its domain and image types we can derive that \( t = s \to \text{Bool} \) and give a type to applications of terms of type \( t \) correctly. Note that this is in contrast with what happens in dependent type theories 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.

We also highlight the typing of the property test term construct,

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

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

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

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

Finally, as expected, the type conversion rule (CONV) allows us to coerce between equal types, 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 \( \{ r::\text{Rec} \mid L \in \text{lab}(r) \} \), we define the record selection construct \( M.L \) as (for conciseness, let \( \mathcal{R} = \{ r::\text{Rec} \mid L \in \text{lab}(r) \} \)):

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

if headLabel\( (t) = L \) then recHeadTerm\( (x) \) else \( F[t[\text{tail}(t)][\text{tail}(x)]])[L][T] \) \( M \)

such that \( M.L : T.L \). The typing requires crucial use of type conversion to allow for the unfolding of the recursive type function to take place (let \( \Gamma_0 \) be \( L :: \text{Nm}, F::\forall t :: \mathcal{R}. t \Rightarrow (t.L), x:T \)):

\[ \text{(CONV)} \]

\[ \Downarrow \] \[ \Gamma_0 \vdash (\text{if headLabel}(T) = L \text{ then headType}(T) \text{ else tail}(T)).L) \Rightarrow T.L :: \text{Type} \]

with \( \Downarrow \) a derivation of \( \Gamma_0 \vdash (\text{if headLabel}(T) = L \text{ then recHeadTerm}(x) \text{ else } F[t[\text{tail}(T)][\text{tail}(x)]])[L][T_0] \)
where \( T_0 \) is if \( \text{headLabel}(T) = L \) then \( \text{headType}(T) \) else \( \text{tail}(T) . L \), requiring a similar appeal to logical entailment to that of Example 3.1. Specifically, in the then branch we must show that \( \Gamma_0 . \text{headLabel}(T) = L \vdash \text{recHeadTerm}(x) : \text{headType}(T) \), which is derivable from \( x : T \) and \( x : \langle \text{headLabel}(T) : \text{headType}(T) \rangle @ \text{tail}(T) \) – the latter following from type conversion due to the refinement \( L \in T \) allowing us to establish \( \neg \text{empty}(T) \) – via typing rule (recterm).

The else branch requires showing that \( \Gamma_0 . \neg \text{headLabel}(T) = L \vdash F[\text{tail}(T)](\text{tail}(x)) : \text{tail}(T).L \), which is derivable from \( F : \forall t :: R. t \rightarrow (t . L) \) and \( x : T \) as follows: \( \text{tail}(T) :: R \) follows from \( \neg \text{headLabel}(T) = L \) and \( T :: R \) (see Section 5), thus \( F[\text{tail}(T)] : \text{tail}(T) \rightarrow \text{tail}(T).L \). Since \( \text{tail}(x) : \text{tail}(T) \) from \( x : T \) and \( x : \langle \text{headLabel}(T) : \text{headType}(T) \rangle @ \text{tail}(T) \) via rule (rectail), we conclude using the application rule. Thus, combining the type and term-level record projection constructs we have that the following is admissible:

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

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

## 5 ALGORITHMIC TYPE CHECKING AND IMPLEMENTATION

This section provides a general description of our practical design choices and implementation of the type theory of the previous sections. While a detailed description of the formulation of our typing and kinding algorithm is not given for the sake of conciseness, we describe the representation and entailment of refinements and the implementation strategy for typing, kinding and equality.

From a conceptual point of view, 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 NuPRL [Constable et al. 1986] is of the former variety. Languages with refinement types such as Liquid Haskell [Vazou et al. 2014] and F-Star [Swamy et al. 2011] (or with limited 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 definitional equality through refinement predicates but disallowing the full power of extensional theories (i.e. allowing arbitrary properties to be used as refinements). The goal of such languages is to allow for non-trivial equalities on types while preserving decidability of type-checking, typically off-loading the non-trivial reasoning about entailment of refinement predicates to some external solver.

### Kind Refinements through SMT Solving

As expected, our approach follows in this tradition, and our system is implemented by offloading validity checks of refinement predicates to the SMT solver CVC4 [Barrett et al. 2011], embodied by the rule for refinement entailment (and for subkinding between two refinement kinds):

\[
\Gamma \vdash \varphi \quad \text{Valid}(\Gamma) \Rightarrow \{ \varphi \} (\text{ENTAILS})
\]

The solver includes first-order theories (with equality) on strings, finite sets and inductive types (with their associated constructors, destructors and congruence principles), and so allows us to represent our refinement language in a fairly direct manner. Crucially, since our theory maintains the distinction between types and terms, we need only represent the type-level constructs of our theory in the solver.

Types of base kind are encoded using an inductive type with a constructor and destructor for each type constructor and destructor in our language, respectively. Labels are represented by strings (i.e. finite sequences). This representation “type of all types” is named Types. Types of higher-kind are encoded as first-order terms, so they can be encoded adequately in the theory of the solver. To
do this in a general way, we add an uninterpreted function symbol to the theory that is used to
code type-level application, effectively implementing defunctionalization [Reynolds 1972].

Refinements are encoded as logical formulae that make use of the theory of finite sets in or-
dered to represent reasoning about record label set membership and apartness. We add two aux-
iliary functional symbols to the theory: isRec : Types → Bool and lab : Types → Set of String,
whose meaning is given through appropriate defining axioms. The isRec predicate codifies that
a given term (representing a type) is a well-formed record, specifying that it is either the repre-
sentation of the empty record or a cons-cell, such that label at the head of the record does not
occur in the label set of its tail. lab encodes the label set of a record representation, essentially
projecting out its labels accordingly. We can then define apartness of two label sets (formally,
apart : (Set of String, Set of String) → Bool) as the formula that holds iff the intersection of the
two sets is empty. Label concatenation and its lifting to label sets is defined in terms of string
concatenation. The empty record test and its negation is encoded via an equality test to the empty
record and the appropriate negation.

To map types to their representation in the SMT solver we make use of a representation function
[−] on contexts which collects variable names (which will be universally quantified in the resulting
formula) and assumed refinements from the context as a conjunction. Without loss of generality,
we assume that all basic kinds appear at the top level in the context as a refinement, all context
variables are distinct and all bound occurrences of variables are distinct.

\[
\begin{align*}
[\emptyset] & \triangleq \text{True} \\
[\Gamma, t : \{ x :: \mathcal{K} \mid \phi(x)\}] & \triangleq [\Gamma] \land t : [\mathcal{K}] \land [\phi(t)] \\
[\Gamma, t : \Pi s : \mathcal{K}. \mathcal{K}'] & \triangleq [\Gamma] \land t : [\mathcal{K}] \rightarrow [\mathcal{K}'] \\
[\mathcal{K}] & \triangleq \text{Types} \\
\{ x :: \mathcal{K} | \phi(x) \} & \triangleq \{ x :: \mathcal{K} | \phi(x) \}
\end{align*}
\]

To simplify the presentation, we overload the [−] notation on contexts, types and kinds. All base
kinds are translated to the representation type Type. At the level of contexts, type variables of base
kind are translated to a declaration of a variable of the appropriate target type and the refinement
is translated straightforwardly making use of the auxiliary predicates defined above. To faithfully
represent type variables of higher-kind (which must denote first-order functions in order to be
representable in the SMT logic) we encode them as uninterpreted functions of the appropriate
(encoded) type.

**Kinding Algorithm.** The implementation of kind checking follows a standard algorithm for type-
checking a λ-calculus with lists, pairs, subtyping and structurally recursive function definitions
[Pierce 2002]. Kinding rules that make use of refinements (i.e., those that manipulate records) and
any instance of subkinding or kind equality in the presence of refinements is discharged via the
encoding into CVC4. All types occurring in refinements are normalized prior to encoding.

**Type Equality and Typing.** As in most type theories, the crux of our implementation lies in
a suitable implementation of type equality. Since our notion of type equality has flavours of
extensionality (recall the examples of Section 3.2) and is essentially kind sensitive, we make use
of the now folklore equivalence checking algorithms that exploit weak-head normalization and
type information [Pierce 2004]. In our setting, we use weak-head normalization of types and
exploit *kinding* information [Stone and Harper 2000, 2006]. The algorithm alternates between
weak-head normalization and kind-oriented equality checking phases. In the former phase, weak-
head reduction of types that form a λ-calculus is used. In the latter phase, extensionality of type-
level functions is implemented essentially by the equivalent of rule (EQ-FUNEXT) read bottom
up and comparisons at base kinds against variables of refined kind are offloaded to the SMT
solver, implementing extensionality for types of base kind (e.g., deriving that \( t \equiv \text{Bool} \rightarrow \text{Bool} \) if
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, denoted by \( T_v, S_v \) and given by the following grammar:

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

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

\[
v, v' ::= \text{true} | \text{false} | \emptyset | \ell :: v | \lambda x:T_v . M | \Lambda t::K.M | v :: v' | \epsilon | l
\]

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

For convenience of notation we write \( \ell :: 1, \ldots, ℓ_n : T_n \) for \( \ell_1 :: 1 @ \cdots @ ℓ_n : T_n @ \emptyset \), and similarly \( ℓ_1 = M_1, \ldots, ℓ_n = M_n \) for \( ℓ_1 = M_1 @ \cdots @ ℓ_n = M_n @ \emptyset \).

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

\[
\frac{(H; M) \rightarrow (H'; M')}{(H; ℓ = M) @ N \rightarrow (H'; ℓ = M') @ N} \quad \text{(R-RecConsL)}
\]

where the record field labelled \( ℓ \) 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 4 (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)

Fig. 4. Operational Semantics (Excerpt)

and (R-PROP) are the only ones that appeal to the entailment relation for refinements, making use of the running environment $\Gamma$ which is threaded through the reduction rules straightforwardly. Similarly, rules (R-KINDL) and (R-KINDR) mimic the equality rules of the kind case construct, testing the kind of type $T$ against $\mathcal{K}$.  

6.1 Metatheory

We now develop the main metatheoretical results of type preservation, progress and uniqueness of kinding and typing. We begin by noting that types and their kinding system are essentially as complex as a type theory with singletons [Stone and Harper 2000, 2006]. Theories of singleton kinds essentially amount to $E_{\omega}$ [Girard 1986] with kind dependencies and a fairly powerful but decidable definitional equality. This is analogous to our development, but where singletons are replaced by kind refinements and the additional logical reasoning on said refinements, and the type language includes additional primitives to manipulate types as data. Notably, when we consider terms and their typing there is no significant added complexity since our typing rules are essentially those of an ML-style, quotiented by a more intricate notion of type equality.

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 entailment is defined by appealing to SMT-validity, we require some basic soundness assumptions on the entailment relation, which we list below.

**Postulate 6.1 (Assumed Properties of Entailment).**

- **Substitution:** If $\Gamma \vdash T :: K$ and $\Gamma, t:K, \Gamma' \models \varphi$ then $\Gamma, \Gamma'\{T/t\} \models \varphi\{T/t\};$
- **Weakening:** If $\Gamma \models \varphi$ then $\Gamma' \models \varphi$ where $\Gamma \subseteq \Gamma';$
- **Functionality:** If $\Gamma \models T \equiv S :: K$ and $\Gamma, t : K, \Gamma' \vdash \varphi$ then $\Gamma \models T\{t/\} \leftrightarrow \varphi\{S/t\}.$
- **Soundness:** If $\text{Valid}([\Gamma] \Rightarrow [\varphi])$, then $[\Gamma] \Rightarrow [\varphi]$ is valid; if $\text{Valid}([\Gamma] \Rightarrow [\varphi])$ answers negatively, then it is not the case that $\neg([\Gamma] \Rightarrow [\varphi])$ is valid.

The general structure of the development is as follows: we first establish basic structural properties of substitution (Lemma 6.1) and weakening, which we can then use to show that we can apply type and kind conversion inside contexts (Lemma 6.2), which then can be used to show a so-called validity property for equality (Theorem 6.3), stating that equality derivations only manipulate well-formed objects (from which kind preservation – Lemma 6.4 – follows).

**Lemma 6.1 (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 6.2 (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 $\Gamma \vdash K \leq K'$ then $\Gamma, t:K' \vdash \mathcal{J}.$

**Theorem 6.3 (Validity for Equality).**

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

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

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

**Lemma 6.4 (Kind Preservation).** If $\Gamma \vdash T :: K$ and $T \rightarrow T'$ then $\Gamma \vdash T' :: K.$

This setup then allows us to show so-called functionality properties of kinding and equality (see Appendix C), stating that substitution is consistent with our theory’s definitional equality and that definitional equality is compatible with substitution of definitionally equal terms.

With functionality and the previous properties we can then establish the so-called validity theorem for our theory, which is a general well-formedness property of the judgments of the language. Validity is crucial in establishing the various type and kind 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 6.5) and type preservation (Theorem 6.6). Moreover, kinding crucially ensures that all types of refinement kind are such that the corresponding refinement is SMT-valid.

**Theorem 6.5** (Unicity of Types and Kinds).

1. If $\Gamma \vdash M : T$ and $\Gamma \vdash M : S$ then $\Gamma \vdash T \equiv S :: K$ and $\Gamma \vdash K \leq \text{Type}$.
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 $I$ in $\text{dom}(H)$ we have that $\Gamma \vdash S I : \text{ref } T$ with $\cdot \vdash H(I) : T$. We write $S \subseteq S'$ to denote that $S'$ is an extension of $S$ (i.e. it preserves the location typings of $S$).

**Theorem 6.6** (Type Preservation). Let $\Gamma \vdash S M : T$ and $\Gamma \vdash S H$. If $\langle H; M \rangle \rightarrow \langle H'; M' \rangle$ 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 6.7** (Type Progress). If $\cdot \vdash T :: K$ then either $T$ is a type value or $T \rightarrow T'$, for some $T'$.

**Theorem 6.8** (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 $\langle H; M \rangle \rightarrow \langle H'; M' \rangle$.

## 7 RELATED WORK

To the best of our knowledge, ours is the first work to explore the concept of refinement kind and illustrate their expressiveness as a practical language feature that integrates statically typed meta-programming features such as type reflection, ad-hoc polymorphism, and type-level computation which allows us to specify structural properties of function, collection and record types.

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] develops a type system to support type-level record computations with similar aims as ours, 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. The idea of expressing constraints (e.g., disjointness) on record labels with predicates goes back to [Harper and Pierce 1991]. We note that our system admits convenient predicates and operators in the refinement logic that are applicable not just to record types, but also to other kinds of types such as function and collection types.

The work of [Kiselyov et al. 2004] implements a library of strongly-typed heterogeneous collections in Haskell via an encoding using the language extensions of multi-parameter type classes and functional dependencies. Their library includes heterogeneous lists and extensible records,
with a semantics that is akin to that of our record types. Since their development is made on top of Haskell and its type-class system, they explicitly encode all the necessary type manipulation (type-level) functions through the type-class system. To do this, they must also encode several auxiliary type-level data such as type-level natural numbers, type-level booleans, type-level occurrence and deletion predicates, to name but a few. To adequately manipulate these types, they also reify type equality and type unification as explicit type classes. This is in sharp contrast with our development, which leverages the expressiveness of refinement kinds to produce the same style of reasoning but with significantly less machinery. We also highlight the work of [Leijen and Meijer 1999], a domain specific embedded compiler for SQL in Haskell by using so-called phantom types, which follows a related approach.

[Morris and McKinna 2019] study a general framework of extensible data types by introducing a notion of row theory which gives a general account of record concatenation and projection. Their work is based on a generalization of row types using qualified types that can refer to some properties of row containment and combination. The ability to express these properties at the type-level is similar to our work, although we can leverage the more general concept of refinement kind to easily express programs and structural properties of records that are not definable in their work: the Map and SetGetRec record transformations from Section 2, the ability to state that a record does not contain a given label [Gaster and Jones 1996], or the general case of a combinator that takes two records $R_1$ and $R_2$ and produces a record where each label $\ell$ is mapped to $R_1.\ell \to R_2.\ell$.

Their work develops an encoding of row theories into System F satisfying coherence. It would be interesting to explore a similar encoding of our work into an appropriate $\lambda$-calculus such as $F_\omega$ with product types.

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 has a non-trivial structure (as required by the various type features and extensions of GHC), and so 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 language with an advanced typing system such as Haskell.

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 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 $\text{if } T :: K \text{ as } t \Rightarrow e_1 \text{ else } e_2$ greatly extends the precision of type and kind checking, and 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 how the fundamental concept of refinement kinds suggests itself as a general type-theoretic principle that accounts for 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# [Petrick et al. 2016]) which seems to be out of reach of 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 expressed by refinements.

8 CONCLUDING REMARKS

This work introduces the concept of refinement kinds and develops its associated type theory, in the context of higher-order polymorphic \( \lambda \)-calculus with imperative constructs, several kinds of datatypes, and type-level computation. The resulting programming language supports static typing of sophisticated features such as type-level reflection with ad-hoc and parametric polymorphism, which can be elegantly combined to implement non-trivial meta-programming idioms, as we have illustrated with several examples. Crucially, the typing system for our language is essentially that of an ML-like language but with a more intricate notion of type equality and kinding, which are defined independently from typing.

We have validated our theory by establishing the standard type safety results and by further developing a prototype implementation for our theory, making use of the SMT solver CVC4 [Barrett et al. 2011] to discharge the verification of refinements. Our implementation demonstrates the practicality and effectiveness of our approach, and validates all examples in the paper. Moreover, as discussed in Section 5, apart from the peculiarities specific to the refinement logic, our implementation is not significantly more involved than standard algorithms for type-checking system \( F_\omega \) or those for singleton kinds [Pierce 2002, 2004; Stone and Harper 2000].

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 investigate limited forms of dependent 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.

The relationship between our refinement kind system and the notion of type class [Wadler and Blott 1989], popularised by Haskell [Hall et al. 1996], also warrants further investigation. Type classes integrate ad-hoc polymorphism with parametric polymorphism by allowing for the specification of functional interfaces that quantified types must satisfy. In principle, type classes can be realized by appropriate type-level records of functions and may thus be representable in our general framework. Finally, to ease the burden on programmers, we plan to investigate how to integrate our algorithmic system with partial type inference mechanisms.

REFERENCES

Appendix

Refinement Kinds:
Type-safe Programming with Practical Type-level Computation

Additional definitions and proofs of the main materials.
A  FULL SYNTAX, JUDGMENTS AND RULES

We define the syntax of kinds $K, K'$, refinements $\phi, \phi'$, types $T, S, R$, and terms $M, N$ below. We assume countably infinite sets of type variables $\mathcal{X}$, 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

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

Refined and Dependent Kinds

$$\mathcal{K} ::= \text{Rec} \mid \text{Col} \mid \text{Fun} \mid \text{Ref} \mid \text{Nm}$$

Base Kinds

Types

$$T, S, R ::= t \mid \lambda t : K. T \mid T S$$

Type-level Functions

$$| \mu F : (\Pi t : K. K'). \lambda t : K. T$$

Structural Recursion

$$| \forall t : K. T$$

Polymorphism

$$\mid L \mid \langle L : T \rangle @ S$$

Record Type constructors

$$\mid \text{headLabel}(T) \mid \text{headType}(T) \mid \text{tail}(T)$$

Record Type destructors

$$\mid T^* \mid \text{colOf}(T)$$

Collection Types

$$\mid \text{ref} T \mid \text{refOf} (T)$$

Reference Types

$$\mid T \to S \mid \text{dom}(T) \mid \text{img}(T)$$

Function Types

$$\mid \text{if } T :: \mathcal{K} \text{ as } t \Rightarrow S \text{ else } U$$

Kind Case

$$\mid \text{if } \phi \text{ then } T \text{ else } S$$

Property Test

$$\mid \bot \mid \top$$

Empty and Top Types

$$\mid \text{Bool} \mid 1 \mid \ldots$$

Basic Data Types

Extended Types

$$\mathcal{T}, \mathcal{S} ::= T \mid \text{lab}(T) \mid \mathcal{T} + + \mathcal{S}$$

Refinements

$$\phi, \psi ::= \phi \supset \psi \mid \phi \land \psi \mid \ldots$$

Propositional Logic

$$\mid \text{empty}(T)$$

Empty Record Test

$$\mid \mathcal{T} = S$$

Equality

$$\mid \mathcal{T} \in S$$

Label Set Inclusion

$$\mid \mathcal{T} \# S$$

Label Set Apartness

Terms

$$M, N ::= x \mid \lambda x : T. M \mid M \ N$$

Functions

$$\mid \Delta t : K. M \mid M[T]$$

Type Abstraction and Application

$$\mid \langle \rangle \mid \langle \ell = M \rangle @ N \mid \text{recTail}(M)$$

Records

$$\mid \text{recHeadLabel}(M) \mid \text{recHeadTerm}(M)$$

Unit Element

$$\mid \text{if } M \text{ then } N_1 \text{ else } N_2$$

Booleans

$$\mid \text{true} \mid \text{false}$$

Property Test

$$\mid \text{if } \phi \text{ then } M \text{ else } N$$

Kind Case

$$\mid \epsilon \mid M :: N$$

References

$$\mid \text{case } M \text{ of } (\epsilon \Rightarrow N_1 \mid x :: xs \Rightarrow N_2)$$

Collections

$$\mid \text{ref } M \mid !M \mid M := N \mid I$$

Recursion

$$\mid \mu F : T. M$$

A.1 Kinding and Typing

Our type theory is defined by the following judgments:

\[ \begin{align*}
\Gamma \vdash & \quad \text{\(\Gamma\) is a well-formed context}\ \\
\Gamma \vdash K & \quad \text{\(K\) is a well-formed kind under the assumptions in \(\Gamma\)} \\
\Gamma \vdash \varphi & \quad \text{Refinement \(\varphi\) is well-formed under the assumptions in \(\Gamma\)} \\
\Gamma \vdash T :: K & \quad \text{Type \(T\) is a (well-formed) type of kind \(K\) under the assumptions in \(\Gamma\)} \\
\Gamma \vdash S \ M : T & \quad \text{Term \(M\) has type \(T\) under the assumptions in \(\Gamma\) and store typing \(S\)} \\
\Gamma \vdash K \equiv K' & \quad \text{Kinds \(K\) and \(K'\) are equal} \\
\Gamma \vdash K \leq K' & \quad \text{Kind \(K\) is a sub-kind of \(K'\)} \\
\Gamma \vdash T \equiv T' :: K & \quad \text{Types \(T\) and \(T'\) of kind \(K\) are equal}
\end{align*} \]

Context Well-formedness.

\[ \begin{align*}
\Gamma \vdash K & \quad \Gamma \vdash T :: \text{Type} & \quad \Gamma \vdash \varphi & \quad \Gamma \vdash \Gamma ; S \vdash T :: K & \quad \cdot \vdash \\
\Gamma , t : K \vdash & \quad \Gamma , x : T \vdash & \quad \Gamma , \varphi \vdash & \quad \Gamma ; S , I : T \vdash \\
\cdot \vdash & \quad \Gamma ; \cdot \vdash
\end{align*} \]

Kind well-formedness.

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

Refinement Well-formedness. Refinement well-formedness simply requires context well-formedness and that all logical predicates are well-sorted (i.e., logical expressions of type \(\text{Bool}\)). All types occurring in refinements must be well-kinded (we write \(p\) to stand for any logical predicate or uninterpreted function of the theory with the appropriate sort):

\[ \begin{align*}
\forall i \in \{1, \ldots, n\}. \Gamma \vdash T_i :: \mathcal{K} & \quad \Gamma \vdash p(T_1, \ldots, T_n)
\end{align*} \]

Refinement Satisfiability. A refinement is satisfiable if it is well-formed and if the representation of the context \(\Gamma\) and the refinement \(\varphi\) as an implicational formula is SMT-valid.

\[ \begin{align*}
\Gamma \vdash \varphi & \quad \text{Valid(}[\Gamma] \Rightarrow [\varphi]) \\
\Gamma \models \varphi
\end{align*} \]
Kind Equality and Sub-kinding.

\[
\begin{align*}
\Gamma & \vdash t : K & \Gamma & \vdash T : K' & \Gamma & \vdash K \leq K' & \Gamma & \vdash \text{Bool} : \text{Type} & \Gamma & \vdash 1 : \text{Type} & \Gamma & \vdash \ell \in \mathcal{N} \\
\Gamma & \vdash T : \Pi t : K. K' & \Gamma & \vdash S : K & \Gamma & \vdash K, t : K \vdash T : K' & \Gamma, t : K_1 \vdash T, t : K_2 & x \notin f \psi(T) & \Gamma & \vdash T : \Pi t : K_1. K_2 \\
\Gamma & \vdash T : K' \{S/t\} & \Gamma & \vdash \lambda t : K. T : \Pi t : K. K' & \Gamma & \vdash T : \Pi t : K_1. K_2 \\
\Gamma & \vdash K, t : K \vdash T : \text{Type} & \Gamma & \vdash \forall t : K. T : \text{Gen}_K \\
\Gamma & \vdash \emptyset : \text{Rec} & \Gamma & \vdash \langle L : T \rangle @ \text{S} : \text{Rec} \\
\Gamma & \vdash T : \{t : \text{Rec} \mid \lnot \text{empty}(t)\} & \Gamma & \vdash T : \{t : \text{Rec} \mid \lnot \text{empty}(t)\} & \Gamma & \vdash T : \{t : \text{Rec} \mid \lnot \text{empty}(t)\} \\
\Gamma & \vdash \text{headType}(T) : \text{Type} & \Gamma & \vdash \text{headLabel}(T) : \text{Nm} & \Gamma & \vdash \text{tail}(T) : \text{Rec} \\
\Gamma & \vdash T : \text{Type} & \Gamma & \vdash S : \text{Type} & \Gamma & \vdash T : \text{Fun} & \Gamma & \vdash T : \text{Fun} \\
\Gamma & \vdash T \rightarrow S : \text{Fun} & \Gamma & \vdash \text{dom}(T) : \text{Type} & \Gamma & \vdash \text{img}(T) : \text{Type} & \Gamma & \vdash \text{dom}(T) : \text{Type} \\
\Gamma & \vdash T : \text{Type} & \Gamma & \vdash T : \text{Col} & \Gamma & \vdash \text{colOf}(T) : \text{Type} & \Gamma & \vdash T : \text{Col} \\
\Gamma & \vdash T : \text{Col} & \Gamma & \vdash \text{colOf}(T) : \text{Type} & \Gamma & \vdash T : \text{Col} & \Gamma & \vdash \text{refOf}(T) : \text{Type} \\
\Gamma & \vdash \text{ref \ T} : \text{Ref} & \Gamma & \vdash \text{refOf}(T) : \text{Type} & \Gamma & \vdash T : \text{Ref} \\
\Gamma & \vdash \varphi & \Gamma, \varphi, t : K & \Gamma, \lnot \varphi, S : K & \Gamma & \vdash \mathcal{K} & \Gamma & \vdash T : \mathcal{K}'' & \Gamma, t : \mathcal{K} \vdash S : K' & \Gamma & \vdash U : K' \\
\Gamma & \vdash \text{if } \varphi \text{ then } T \text{ else } S : K & \Gamma & \vdash \mathcal{K} & \Gamma & \vdash \mathcal{K}' & \Gamma \vdash \mathcal{T / t} & \Gamma & \vdash T : \mathcal{K} \\
\Gamma, F : \Pi t : K. K', t : K + T : K' & \text{structural}(T, F, t) & \Gamma \models \varphi(T/t) & \Gamma & \vdash T : \mathcal{K} \\
\Gamma & \vdash \mu F : (\Pi t : K. K') \lambda t : K. T : \Pi t : K. K' & \Gamma \models \varphi(T/t) & \Gamma \vdash T : \{t : \mathcal{K} \mid \varphi\} \\
\end{align*}
\]
Type Equality.

Reflexivity, Transitivity, Symmetry+

\[ \Gamma \vdash T :: \{ t :: \mathcal{K} | t = S \} \quad \Gamma \vdash S :: \mathcal{K} \]

\[ \Gamma \vdash T \equiv S :: \mathcal{K} \]

\[ \Gamma \vdash T_1 \equiv S_1 :: \Pi t : K_1.K_2 \quad \Gamma \vdash T_2 \equiv S_2 :: K_1 \]

\[ \Gamma \vdash T_1.T_2 \equiv S_1.S_2 :: K_2\{T_2/t\} \]

\[ \Gamma \vdash S :: \Pi t : K_1.K_2 \quad \Gamma, t : K_1 \vdash t \equiv T t :: K_2 \]

\[ \Gamma \vdash T :: \Pi t : K_1.K_2 \]

\[ \Gamma \vdash K_1 \equiv K'_1 \quad \Gamma, t : K_1 \vdash T \equiv T :: \mathcal{K} \]

\[ \Gamma \vdash \lambda t :: K_1.T \equiv \lambda t :: K'_1.T :: \Pi t : K_1.K_2 \]

\[ \Gamma \vdash \lambda : K.T \equiv \lambda : K'.T :: \mathcal{K}'(S/t) \]

\[ \Gamma \vdash K_1 \equiv K_2 \quad \Gamma, t : K_1 \vdash T :: \mathcal{K} \]

\[ \Gamma \vdash \forall t :: K_1.\tau \equiv \forall t :: K_2.\tau :: \text{Gen}_{K_1} \]

\[ \Gamma \vdash L \equiv L' :: \text{Nm} \quad \Gamma \vdash T \equiv T' :: \mathcal{K} \]

\[ \Gamma \vdash S \equiv S' :: \{ t :: \text{Rec} | L \not\equiv \text{lab}(t) \} \]

\[ \Gamma \vdash \langle L : T \rangle@S \equiv \langle L' : T' \rangle@S' :: \text{Rec} \]

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

\[ \Gamma \vdash \text{headLabel}(T) \equiv \text{headLabel}(S) :: \text{Nm} \]

\[ \Gamma \vdash \text{headType}(T) \equiv \text{headType}(S) :: \text{Type} \]

\[ \Gamma \vdash \langle L : T \rangle@S \equiv L :: \text{Nm} \]

\[ \Gamma \vdash \text{headLabel}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash \text{headType}(\langle L : T \rangle@S) \equiv T :: \text{Type} \]

\[ \Gamma \vdash \text{headType}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \]

\[ \Gamma \vdash S :: \{ t :: \text{Rec} | L \not\equiv \text{lab}(t) \} \]

\[ \Gamma \vdash \text{tail}(T) \equiv \text{tail}(S) :: \text{Rec} \]

\[ \Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \]

\[ \Gamma \vdash S :: \{ t :: \text{Rec} | L \not\equiv \text{lab}(t) \} \]

\[ \Gamma \vdash \text{headType}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash \text{headLabel}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash \text{headType}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \]

\[ \Gamma \vdash S :: \{ t :: \text{Rec} | L \not\equiv \text{lab}(t) \} \]

\[ \Gamma \vdash \text{tail}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \]

\[ \Gamma \vdash S :: \{ t :: \text{Rec} | L \not\equiv \text{lab}(t) \} \]

\[ \Gamma \vdash \text{headType}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash \text{headLabel}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash \text{headType}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \]

\[ \Gamma \vdash S :: \{ t :: \text{Rec} | L \not\equiv \text{lab}(t) \} \]

\[ \Gamma \vdash \text{tail}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \]

\[ \Gamma \vdash S :: \{ t :: \text{Rec} | L \not\equiv \text{lab}(t) \} \]

\[ \Gamma \vdash \text{headType}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash \text{headLabel}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash \text{headType}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \]

\[ \Gamma \vdash S :: \{ t :: \text{Rec} | L \not\equiv \text{lab}(t) \} \]

\[ \Gamma \vdash \text{tail}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \]

\[ \Gamma \vdash S :: \{ t :: \text{Rec} | L \not\equiv \text{lab}(t) \} \]

\[ \Gamma \vdash \text{headType}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash \text{headLabel}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash \text{headType}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \]

\[ \Gamma \vdash S :: \{ t :: \text{Rec} | L \not\equiv \text{lab}(t) \} \]

\[ \Gamma \vdash \text{tail}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash L :: \text{Nm} \quad \Gamma \vdash T :: \mathcal{K} \]

\[ \Gamma \vdash S :: \{ t :: \text{Rec} | L \not\equiv \text{lab}(t) \} \]

\[ \Gamma \vdash \text{headType}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash \text{headLabel}(\langle L : T \rangle@S) \equiv \text{Rec} \]

\[ \Gamma \vdash \text{headType}(\langle L : T \rangle@S) \equiv \text{Rec} \]
\[
\Gamma \vdash T \equiv T' :: \mathcal{K}_0 \quad \Gamma \vdash \mathcal{K} \equiv \mathcal{K}' \quad \Gamma, t: \mathcal{K} \vdash S \equiv S' :: K'' \quad \Gamma \vdash U \equiv U' :: K''
\]

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

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

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

\[
\Gamma \vdash \varphi \Leftrightarrow \psi \quad \Gamma, \varphi \vdash T_1 \equiv S_1 :: K \quad \Gamma, \neg \varphi \vdash T_2 \equiv S_2 :: K
\]

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

\[
\Gamma \vdash \varphi \quad \Gamma, \varphi \vdash T_1 :: K \quad \Gamma, \neg \varphi \vdash T_2 :: K
\]

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

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

\[
\Gamma \vdash \text{structural}(T, F, t)
\]

\[
\Gamma \vdash K_1 \equiv K_1' \quad \Gamma \vdash K_2 \equiv K_2' \quad \Gamma, F: \Pi t: \mathcal{K}_1, \mathcal{K}_2, t: \mathcal{K}_1 \vdash T \equiv S :: K_2
\]

\[
\Gamma \vdash \mu F : (\Pi t: \mathcal{K}_1, \mathcal{K}_2). \lambda t: \mathcal{K}_1. T \equiv \mu F : (\Pi t: \mathcal{K}_1', \mathcal{K}_2'). \lambda t: \mathcal{K}_1'. \mu S :: \Pi t: \mathcal{K}_1, \mathcal{K}_2
\]

\[
\Gamma, t: \mathcal{K}_1 : t_2 \quad \Gamma, F: \Pi t: \mathcal{K}_1, \mathcal{K}_2, t: \mathcal{K}_1 \vdash T :: K_2 \quad \Gamma \vdash S :: K_1 \quad \text{structural}(T, F, t)
\]

\[
\Gamma \vdash (\mu F : (\Pi t: \mathcal{K}_1, \mathcal{K}_2). \lambda t: \mathcal{K}_1. T) S \equiv T \{S/t\} \{(\mu F : (\Pi t: \mathcal{K}_1, \mathcal{K}_2). \lambda t: \mathcal{K}_1. T)/F\} :: K_2 \{S/t\}
\]

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

\[
\begin{align*}
\text{(VAR)} & \quad (x:T) \in \Gamma \\
\text{(1I)} & \quad \Gamma \vdash T \coloneqq \text{Type} \\
\text{(-I)} & \quad \Gamma \vdash x : T \\
\text{(-E)} & \quad \Gamma \vdash M : T \rightarrow S \\
\text{(VI)} & \quad \Gamma \vdash \top \\
\text{(VE)} & \quad \Gamma \vdash T : K \\
\text{(true)} & \quad \Gamma \vdash \text{true} : \text{Bool} \\
\text{(false)} & \quad \Gamma \vdash \text{false} : \text{Bool} \\
\text{(cons)} & \quad \Gamma \vdash \text{if } \varphi \text{ then } M \text{ else } N : T \\
\text{(_.emp)} & \quad \Gamma \vdash T \coloneqq \text{Type} \\
\text{(_.loc)} & \quad \Gamma \vdash l : \text{ref } T \\
\text{(_.prop-ite))} & \quad \Gamma \vdash \text{if } \varphi \text{ then } M \text{ else } N : T \\
\text{(prop-ite)} & \quad \Gamma \vdash \varphi, \varphi \vdash M : T_1 \\
\text{(_.ref)} & \quad \Gamma \vdash \text{ref } M : \text{ref } T \\
\text{(_.ref)} & \quad \Gamma \vdash \text{ref } M : \text{ref } T \\
\text{(_.case)} & \quad \Gamma \vdash \text{case } M \text{ of } (\varepsilon \Rightarrow N_1 | x:x:T \Rightarrow N_2) : S \\
\text{(_.derefer)} & \quad \Gamma \vdash \text{ref } M : \text{ref } T \\
\text{(_.assign)} & \quad \Gamma \vdash !M : T \\
\text{(_.kindcase)} & \quad \Gamma \vdash T : \mathcal{K}' \\
\text{(_.fix)} & \quad \Gamma \vdash \mu F : T \text{.M} \\
\end{align*}
\]

B. FULL OPERATIONAL SEMANTICS

The type reduction relation, $T \rightarrow T'$, is defined as a call-by-value reduction semantics on types $T$, obtained by orienting the computational rules of type equality from left to right (thus excluding rule (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:

\[
T_v, S_v \coloneqq \lambda t : K . T | \forall t : K . T | \ell | \emptyset | \langle \ell : T_v \rangle \! @ S_v | T_v^* | \text{ref } T_v | T_v \rightarrow S_v | \bot | \text{Bool} | 1 | t
\]
The type reduction rules are:

\[
\begin{align*}
T \to T' & \quad S \to S' \\
TS \to T'S & \quad (\lambda t::K.T)S \to (\lambda t::K.T)S' \\
\to & \quad (\lambda t::K.T)S_v \to T[S_v/t] \\
\to & \quad \lambda t::K.T \to \lambda t::K.T' \\
\forall t::K.T & \to \forall t::K.T' \\
L \to L' & \quad \langle L : T @ S \rangle \to \langle L' : T @ S \rangle \\
T \to T' & \quad \langle \ell : T @ S \rangle \to \langle \ell : T' @ S \rangle \\
S \to S' & \quad \langle \ell : T_v @ S \rangle \to \langle \ell : T_v @ S' \rangle \\
\to & \quad \text{headLabel}(T) \to \text{headLabel}(T') \\
\to & \quad \text{headType}(T) \to \text{headType}(T') \\
\to & \quad \text{tail}(T) \to \text{tail}(T') \\
\text{headLabel}(\langle \ell : T_v @ S_v \rangle) & \to \ell \\
\text{headType}(\langle \ell : T_v @ S_v \rangle) & \to T_v \\
\text{tail}(\langle \ell : T_v @ S_v \rangle) & \to S_v \\
\end{align*}
\]
The rules of our operational semantics are as follows:

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

\[ \langle H; \text{false} \rangle M N \rightarrow \langle H; N \rangle \]

\[ \langle H; \text{if} M \text{ then } N_1 \text{ else } N_2 \rangle \rightarrow \langle H'; \text{if } M' \text{ then } N_1 \text{ else } N_2 \rangle \]

R-TAppTRed

\[ T \rightarrow T' \]

\[ \langle H; (\Lambda t \text{::K}. M)[T] \rangle \rightarrow \langle H; (\Lambda t \text{::K}. M)[T'] \rangle \]

R-Fix

\[ \langle H; T_u . M \rangle \rightarrow \langle H; M(T_u / M) \rangle \]

R-AppL

\[ \langle H; M \rangle \rightarrow \langle H'; M' \rangle \]

\[ \langle H; M[T] \rangle \rightarrow \langle H'; M'[T] \rangle \]

R-AppR

\[ \langle H; \Lambda x . T . M \rangle \rightarrow \langle H; M/V \rangle \]

R-TApp

\[ \langle H; \text{if } then \rangle T \rightarrow \langle H; \text{if } then \rangle T' \]

R-AppV

\[ \langle H; (\Lambda x . T . M) V \rangle \rightarrow \langle H; M/V \rangle \]

R-AppLT

\[ \langle H; \Lambda x . T . M \rangle \rightarrow \langle H'; \Lambda x . T'. M \rangle \]

R-AppR

\[ \langle H; N \rangle \rightarrow \langle H'; N' \rangle \]

R-ColC eyel L

\[ \langle H; M :: N \rangle \rightarrow \langle H'; M' :: N \rangle \]

R-ColConsR

\[ \langle H; v :: N \rangle \rightarrow \langle H'; v :: N' \rangle \]

R-ColTLv

\[ \langle H; \text{case } M \text{ of } (\varepsilon \Rightarrow N_1 | x :: x \Rightarrow N_2) \rangle \rightarrow \langle H'; \text{case } M' \text{ of } (\varepsilon \Rightarrow N_1 | x :: x \Rightarrow N_2) \rangle \]

(R-ColCaseEmp)

\[ \langle H; \text{case } \varepsilon \text{ of } (\varepsilon \Rightarrow N_1 | x :: x \Rightarrow N_2) \rangle \rightarrow \langle H; N_1 \rangle \]

(R-ColCaseCons)

\[ \langle H; \text{case } v :: v s \text{ of } (\varepsilon \Rightarrow N_1 | x :: x \Rightarrow N_2) \rangle \rightarrow \langle H; N_2 \{ u / x, v s / x s \} \rangle \]
R-KindL \[ \Gamma \vdash T :: \mathcal{K} \]

R-KindR \[ \Gamma \vdash T :: \mathcal{K}_0 \quad \Gamma \vdash \mathcal{K}_0 \neq \mathcal{K} \]

\[ \langle H; \text{if } T :: \mathcal{K} \text{ as } t \Rightarrow M \text{ else } N \rangle \rightarrow \langle H; M(T/t) \rangle \]

\[ \langle H; \text{if } T :: \mathcal{K} \text{ as } t \Rightarrow M \text{ else } N \rangle \rightarrow \langle H; N \rangle \]

C PROOFS

**Lemma 6.1 (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) 

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

\[ \text{(KREF)} \]

**Case:** 

\[ \Gamma, t : K, \Gamma' \vdash T :: \{t : \mathcal{K} | \varphi\} \]

\[ \Gamma, \Gamma'\{T/t\} \vdash \varphi\{s/s\}\{T/t\} \]

by i.h.

\[ \Gamma, \Gamma'\{T/t\} = \mathcal{K}\{T/t\} \]

by i.h.

\[ \Gamma, \Gamma'\{T/t\} \vdash \varphi\{T/t\}\{T/t\} \]

by properties of substitution

\[ \Gamma, \Gamma'\{T/t\} + \mathcal{S}\{T/t\} :: \{s : \mathcal{K}\{T/t\} | \varphi\{T/t\}\} \]

by rule

\[ \Gamma, t : K, \Gamma' \vdash \varphi \quad \text{Valid} \left( \Gamma, t : K, \Gamma' \parallel \varphi\{s/s\} \right) \]

\[ \text{(ENTAILS)} \]

**Case:** 

\[ \Gamma, t : K, \Gamma' \parallel \varphi(T/t) \]

\[ \text{by i.h.} \]

\[ \text{Valid}(\varphi(T/t)) \]

by logical substitution / congruence

\[ \Gamma, \Gamma'\{T/t\} \vdash \varphi(T/t) \]

by rule

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

\[ \text{(R-EQELIM)} \]

**Case:** 

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

\[ \Gamma, \Gamma'\{T/t\} + \mathcal{S}\{T/t\} :: \{s : \mathcal{K}\{T/t\} | s = S\{T/t\}\} \]

by i.h.

\[ \Gamma, \Gamma'\{T/t\} + \mathcal{S}\{T/t\} :: \mathcal{K}\{T/t\} \]

by i.h.

\[ \Gamma, \Gamma'\{T/t\} + \mathcal{S}\{T/t\} :: \mathcal{K}\{T/t\} \]

by rule

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

\[ \text{(R-EQELIM)} \]

**Case:** 

\[ \Gamma, t : K, \Gamma' \vdash \forall s : \mathcal{K} \cdot T' :: \text{Gen}_{\mathcal{K}'} \]

\[ \Gamma, \Gamma'\{T/t\} \vdash \mathcal{K}\{T/t\} \]

by i.h.

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

by i.h.

\[ \Gamma, t : K, \Gamma' \vdash \forall s : \mathcal{K}\{T/t\}. T' \{T/t\} :: \text{Gen}_{\mathcal{K}'\{T/t\}} \]

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

Lemma 6.2 (Context Conversion).

(a) Let $\Gamma, x: T \vdash \Delta \vdash \Gamma$ and $\Gamma \vdash T \equiv K$ then $\Gamma, x: T' \equiv \Delta$. 
(b) Let $\Gamma, t: K \vdash \Delta$ and $\Gamma \vdash K \equiv K'$. If $\Gamma, t: K' \vdash \Delta$ then $\Gamma, t: K' \vdash \Delta$. 

Proof. Follows by weakening and substitution.

(a)

Γ, x : T′ ⊢ x : T

Γ ⊢ T′ ≡ T :: K

Γ, x : T′ ⊢ x : T

Γ, x′ : T ⊢ J{x′/x}

Γ, x : T′, x′:T ⊢ J{x′/x}

Γ, x : T′ ⊢ J by definition

Lemma C.1 (Functionality of Kinding and Refinements).

Assume Γ ⊨ T = S :: K, Γ ⊢ T :: K and Γ ⊢ S :: K:

(a) If Γ, t:K, Γ′ ⊢ T′ :: K′ then Γ, Γ′′(T/t) = T′{S/t} :: K′{T/t}

(b) If Γ, t:K, Γ′ ⊢ K′ then Γ, Γ′′(T/t) ⊢ K{S/t}.

(c) If Γ, t:K, Γ′ ⊢ φ then Γ, Γ′′(T/t) ⊢ φ{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:

Γ, t : K, Γ′ ⊢ K′
Γ, t : K, Γ′ ⊢ t′:K′ + φ

Γ, t : K, Γ′ + {t′ : K′ | φ}

Γ, Γ′(T/t) ⊢ K′{S/t} by i.h.

Γ, Γ′(T/t), t′:K′{T/t} + φ{T/t} ≡ φ{S/t} by i.h.

Γ, Γ′(T/t) ⊢ {t′ : K′{S/t} | φ{S/t}} by kind ref. equality

Γ, t : K, Γ′ + K

Case:

Γ, t : K, Γ + ∀s:K′, T′ :: GenK

Γ, Γ′(T/t) ⊢ K′{S/t} by i.h.

Γ, Γ′(T/t), t′:K′{T/t} + T′{S/t} :: K by i.h.

Γ, Γ′(T/t) ⊢ ∀s : K′{S/t}.T′{S/t} :: GenK′{T/t} by ∀ Eq.

Γ, t : K, Γ′ + L :: Nm

Γ, t : K, Γ + T′ :: K

Γ, t : K, Γ′ + S′ :: {t : Rec | L(t) ≠ t}

Case:

Γ, t : K, Γ′ + Lm{L : T′}@S′ :: Rec

Γ, Γ′(T/t) ⊢ L{T/t} = L{S/t} :: Nm by i.h.

Γ, Γ′(T/t) ⊢ T′{T/t} ≡ T′{S/t} :: K by i.h.

Γ, Γ′(T/t) ⊢ S′{T/t} ≡ S′{S/t} :: {t : Rec | L(T) ≠ t} by i.h.

Γ, Γ′(T/t) ⊢ L{T/t} : T′{T/t}@S′{T/t} = L{S/t} : T′{S/t}@S′{S/t} :: Rec by Rec Eq.

Γ, t : K, Γ′ ⊢ T′ :: Fun

Case:

Γ, t : K, Γ′ + dom(T′) :: Type

Γ, Γ′(T/t) ⊢ T′{T/t} = T′{S/t} :: Fun by i.h.

Γ, Γ′(T/t) ⊢ dom(T{T/t}) = dom(T′{S/t}) :: Type by congruence rule

(k-recons)

Γ, t : K, Γ′ ⊢ L :: Nm

Γ, t : K, Γ′ + U :: K

Γ, t : K, Γ′ + W :: {s : Rec | L ≤ lab(s)}

Case:

Γ t : K, Γ′ + Lm{L : U}@W :: Rec
Γ, Γ′{T/t} ⊨ L{T/t} ≡ L{S/t} :: Nm
by i.h.
Γ, Γ′{T/t} ⊨ U{T/t} ≡ U{S/t} :: K{T/t}
by i.h.
Γ, Γ′{T/t} ⊨ W{T/t} ≡ W{S/t} :: {s : Rec | L{T/t} ∉ lab(s)}
by i.h.
Γ, Γ′{T/t} ⊨ ⟨L{T/t} : U{T/t}⟩@W{T/t} ≡ ⟨L{S/t} : U{S/t}⟩@W{S/t} :: Rec
by congruence rule

Case: Γ, t : K, Γ′, ϕ ⊨ K ∨ Γ, t : K, Γ′, ϕ + T′ ⊨ K′ ∨ Γ, t : K, Γ′, ϕ + S′ ⊨ K′

Γ, Γ′{T/t}, ϕ{T/t} + T′{T/t} ≡ T′{S/t} :: K′{T/t}
by i.h.
Γ, Γ′{T/t}, ϕ{T/t} + S′{T/t} ≡ K′{T/t}
by i.h.

Γ, t : K, Γ′, ϕ ≡ ϕ
by substitution
Γ, Γ′{T/t}, ϕ{T/t} ≡ ϕ{T/t}
by substitution
Γ, Γ′{T/t}, ϕ{S/t} ≡ ϕ{T/t}
by context conversion
Γ, Γ′{S/t}, ϕ{S/t} ≡ ϕ{S/t}
by substitution
Γ, Γ′{T/t}, ϕ{T/t} ≡ ϕ{S/t}
by context conversion
Γ, Γ′{T/t} ≡ ϕ{T/t} ⊃ ϕ{S/t}
by ⊃I
Γ, Γ′{T/t} + ϕ{T/t} + S′{T/t} ≡ K′{T/t}
by definition

If ϕ{T/t} then T′{T/t} else S′{T/t} ⊨ K′{T/t}

Case: Γ, t : K, Γ′ + T′ :: K′

Γ, t : K, Γ′ + T′ :: {s : K′ | ϕ}
by i.h.
Γ, Γ′{T/t} ≡ ϕ{T′/s}{T/t} ≡ ϕ{T′/s}{S/t}
by context conversion
Γ, Γ′{T/t} ≡ T′{T/t} ≡ T′{S/t} :: K′{T/t}
by i.h.
Γ, Γ′{T/t} ≡ T′{T/t} ≡ T′{S/t} :: {s : K′{T/t} | ϕ{T/t}}
by i.h.

Γ, Γ + K ⊨ K and Γ + K′
by i.h.
Γ, t : K + ϕ
by inversion
Γ, t : K + ψ
by inversion
Γ, t : K′ + ϕ
by context conversion
Γ + {t : K | ϕ}
by refinement kind w.f.
Γ + {t : K′ | ϕ}
by refinement kind w.f.
Γ ⊨ T :: {s : K | s = S} + S :: K
by refinement kind w.f.

Case: Γ, t : K + ψ ⇒ ψ

Γ + T :: {s : K | s = S} + S :: K
(R-EQELIM)
by i.h.
Γ, Γ + S :: K
by inversion
Γ ⊨ T :: {s : K | s = S}
by inversion
Γ ⊨ T :: K
by subsumption

Theorem 6.3 (Validity for Equality).

(a) If Γ ⊨ K ≤K′ and Γ ⊨ then Γ ⊨ K and Γ + K′.
(b) If Γ ⊨ T ≡ T′ :: K and Γ ⊨ then Γ ⊨ K, Γ ⊨ T :: K and Γ ⊨ T′ :: K.
(c) If Γ ⊨ ψ ⇔ ϕ and Γ ⊨ then Γ ⊨ ψ and Γ ⊨ ϕ

Proof. By induction on the given derivation.

Case: Γ + K ⊨ K and Γ + K′

Γ + {t : K | ϕ} ≤ {t : K′ | ϕ}
by i.h.
Γ, t : K + ϕ
by inversion
Γ, t : K + ψ
by inversion
Γ, t : K′ + ψ
by context conversion
Γ + {t : K | ϕ}
by refinement kind w.f.
Γ + {t : K′ | ϕ}
by refinement kind w.f.
Γ ⊨ T :: {s : K | s = S} + S :: K
by refinement kind w.f.

Case: Γ ⊨ T ≡ S :: K
(R-EQELIM)
by i.h.
Γ ⊨ S :: K
by inversion
Γ ⊨ T :: {s : K | s = S}
by inversion
Γ ⊨ T :: K
by subsumption

\( \Gamma \vdash \mathcal{K} \) by kind w.f.

\[ \frac{\Gamma \vdash L :: Nm \quad \Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \{t::Rec \mid L \not\in \text{lab}(t)\}}{\Gamma \vdash \text{headType}(\langle L : T \rangle@S) \equiv T :: \text{Type}} \] Case: \( \Gamma \vdash T :: \mathcal{K} \) by inversion
\( \Gamma \vdash T :: \text{Type} \) by subsumption
\( \Gamma \vdash \langle L : T \rangle@S :: \text{Rec} \) by kinding
\( \Gamma \vdash \langle L : T \rangle@S :: \{t::Rec \mid \neg \text{empty}(t)\} \) by subsumption
\( \Gamma \vdash \text{headType}(\langle L : T \rangle@S) :: \text{Type} \) by kinding
\( \Gamma \vdash \text{Type} \) by kind w.f.

\[ \frac{\Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash \text{colOf}(T^*) :: T :: \text{Type}} \] Case: \( \Gamma \models \text{colOf}(T^*) \equiv T :: \text{Type} \)
\( \Gamma \vdash T :: \mathcal{K} \) by inversion
\( \Gamma \vdash T :: \text{Type} \) by subkinding
\( \Gamma \vdash T^* :: \text{Col} \) by kinding
\( \Gamma \vdash \text{colOf}(T^*) :: \text{Type} \) by kinding
\( \Gamma \vdash \text{Type} \) by kind w.f.

Remaining cases follow by a similar reasoning.

\( \square \)

**Lemma 6.4 (Kind Preservation).** If \( \Gamma \vdash T :: K \) and \( T \rightarrow T' \) then \( \Gamma \vdash T' :: K \).

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

**Lemma C.2 (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 \( \Gamma, t:K + \varphi \equiv \psi \) then \( \Gamma \models \varphi\{T_0/t\} \equiv \psi\{S_0/t\} \).

**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
THEOREM C.3 (Validity).

(a) If $\Gamma \vdash K$ then $\Gamma \vdash$

(b) If $\Gamma \vdash T : K$ then $\Gamma \vdash K$

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

PROOF. Straightforward induction on the given derivation.

LEMMA C.4 (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.5 (Injectivity via Subtyping). 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.6 (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 $\Gamma, t : K_1 \vdash K_2$.

(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 $\Gamma \models T' \equiv T_1 \rightarrow T_2 :: \text{Fun}, \Gamma \models T \equiv T_1 :: \text{Type}$ and $\Gamma, x : T_1 \vdash M : T_2$.

(d) If $\Gamma \vdash (L : T)@S :: K$ then $\Gamma \vdash L :: \text{Nm}, \Gamma \vdash T :: \text{Type}, \Gamma \vdash S :: \{t :: \text{Rec} \mid L \notin t\}$ and $\Gamma \vdash K \equiv \text{Rec}$.

(e) If $\Gamma \vdash (L = M)@N : T$ then there is $L', T_1, T_2$ such that $\Gamma \models L \equiv L' :: \text{Nm}, \Gamma \models \langle L' : T_1 \rangle@T_2 :: \text{Rec}, \Gamma \models T \equiv \langle L' : T_1 \rangle@T_2 :: \text{Rec}, \Gamma \vdash M : T_1$ and $\Gamma \vdash N : T_2$.

(f) If $\Gamma \vdash t :: \{t : K \mid \varphi \}$ then $\Gamma \vdash \varphi(T/t), \Gamma \vdash T :: K$ and $\Gamma, t : T_1$.

(g) If $\Gamma \vdash t :: K_1$ and $\varphi \vdash M : T_1$ and $\neg \varphi \vdash N : T_2$.

(h) If $\Gamma \vdash t :: S$ then $\Gamma \vdash \varphi, \Gamma, \varphi \vdash T :: K$ and $\neg \varphi \vdash S :: K$.

(i) \[ \Gamma \vdash t : S \rightarrow T \vdash \varphi, \Gamma, \varphi \vdash T :: K \text{ and } \neg \varphi \vdash S :: K. \]

(j) If $\Gamma \vdash T : S :: K$ then $\Gamma \vdash K : S$ and $\Gamma \vdash S : \mathcal{K}$.

(k) If $\Gamma \vdash M : N :: T$ then $\Gamma \vdash T \equiv S^* :: \text{Col}, \Gamma \vdash N : S^*$ and $\Gamma \vdash M : S$, for some $S$.

(l) If $\Gamma \vdash T' :: K$ then $\Gamma \vdash K :: \text{Col}$ and $\Gamma \vdash T' :: \mathcal{K}$ for some $\mathcal{K}$.

(m) If $\Gamma \vdash t :: K$ as $t \Rightarrow M$ else $N : T$ then $\Gamma \vdash T' :: \mathcal{K}, \Gamma \vdash K, \Gamma, t : K :: M : S$ and $\Gamma \vdash N : S$.

(n) If $\Gamma \vdash t :: K$ as $t$ then $\text{else} S' :: K'$ then $\Gamma \vdash T' :: \mathcal{K}, \Gamma \vdash K, \Gamma, t : K :: S' :: K''$, $\Gamma \vdash S' :: K''$ and $\Gamma \vdash K' \equiv K''$, for some $\mathcal{K}, \mathcal{K}'$.

(o) If $\Gamma \vdash \mu F. T : M$ then $\Gamma, F :: T : M : T$ and $\text{structural}(F, M)$.

(p) If $\Gamma \vdash \mu F :: (\Pi t : K_1, K_2). \lambda t : K_1. T' :: K$ then $\Gamma, F : \Pi t : K_1, K_2, t : K_1 \vdash T' :: K_2$, structural($T', F, t$) and $\Gamma \vdash K \equiv \Pi t : K_1, K_2$.

(q) If $\Gamma \vdash \text{recHeadLabel}(M) : T$ then $\Gamma \vdash T :: L :: \text{Nm}, \Gamma \vdash M : \langle L :: S \rangle@U$.

(r) If $\Gamma \vdash \text{recHeadTerm}(M) : T$ then $\Gamma \vdash T :: \text{Type and } \Gamma \vdash M : \langle L :: S \rangle@U$.

(s) If $\Gamma \vdash \text{tail}(M) : T$ then $\Gamma \vdash T :: \text{Rec and } \Gamma \vdash M : \langle L :: S \rangle@U$.

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

1:41
Case: $\Gamma \vdash \lambda t : K.T :: K'' \quad \Gamma \vdash K'' \leq K'$. 

$\Gamma \vdash \lambda t : K.T :: K'$. 

$\Gamma \vdash K'' \equiv \Pi t : K', K'' \equiv \Gamma \vdash K_1, K_2$, for some $K_1, K_2$ with $\Gamma \vdash \lambda t : K_1, K_2$ and $\Gamma, t : K_1 \vdash \lambda t : K_2$, by i.h. 

$\Gamma \vdash K'' \equiv \Pi t : K_1, K_2$, for some $K_1, K_2$ with $\Gamma \vdash K_1 \leq K_1$ and $\Gamma, t : K_1 \vdash K_2$, by inversion 

$\Gamma, t : K_1 \vdash T :: K_1'$, by inversion 

$\Gamma, t : K_1 \vdash T :: K_2$, by conversion 

$\Gamma \vdash K \equiv K_1$, by transitivity 

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

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

(a)
LEMMA C.8 (Subtyping Inversion).

1. If $\Gamma \vdash \mathcal{K} \leq \mathcal{K}'$ then $\Gamma \vdash \mathcal{K} \equiv \mathcal{K}'$ or $\Gamma \vdash \mathcal{K}' \equiv \text{Type}$.
2. If $\Gamma \vdash K \leq \{t:K' \mid \varphi\}$ then $\Gamma \vdash K \equiv \{t:K \mid \psi\}$ with $\Gamma \vdash \mathcal{K} \leq \mathcal{K}'$ and $\Gamma \vdash \varphi \Rightarrow \psi$.
3. If $\Gamma \vdash \{t:K' \mid \varphi\} \leq \mathcal{K}$ then $\Gamma \vdash \mathcal{K} \leq \mathcal{K}'$ and $\Gamma \vdash K \vdash \varphi$.

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

LEMMA C.9. If $\Gamma \models T \equiv S :: K$ and $\Gamma \vdash T :: K'$ and $\Gamma \vdash K' \leq K$ then $\Gamma \vdash T \equiv S :: K'$.

Proof. By induction on the given equality derivation. □

THEOREM 6.5 (Unicity of Types and Kinds).

1. If $\Gamma \vdash M \vdash T$ and $\Gamma \vdash M : S$ then $\Gamma \vdash T \equiv S :: K$ and $\Gamma \vdash K \leq \text{Type}$.
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 $(\ell = M')@N'$

- $\Gamma \vdash (\ell = M')@N' : T$ and $\Gamma \vdash (\ell = M')@N' : S$ (assumption)
- $\Gamma \vdash M' : T_1$, $\Gamma \vdash N' : T_2$, $\Gamma \vdash \ell \equiv L' :: \text{Nm}$, $\Gamma \vdash (L' = T_1)@T_2 :: \text{Rec}$ (by definition)
- $\Gamma \vdash M' : S_1$, $\Gamma \vdash N' : S_2$, $\Gamma \vdash \ell \equiv L'' :: \text{Nm}$, $\Gamma \vdash (L'' = S_1)@S_2 :: \text{Rec}$ (by inversion)
- $\Gamma \vdash S \equiv (L'' = S_1)@S_2 :: \text{Rec}$ (by inversion and conversion)
- $\Gamma \models T_1 \equiv S_1 :: K_1$ and $\Gamma \vdash K_1 \leq \text{Type}$ (by i.h.)
- $\Gamma \models T_1 \equiv S_1 :: \text{Type}$ (by conversion)
- $\Gamma \models T_2 \equiv S_2 :: K_2$ and $\Gamma \vdash K_2 \leq \text{Type}$ (by i.h.)
- $\Gamma \vdash T_3 :: \text{Rec}$ and $\Gamma \vdash T_2 :: \text{Rec}$ (by inversion and conversion)
- $\Gamma \models T_2 \equiv S_2 :: \text{Rec}$ (by Lemma C.9)

Case: $T$ is $(L = S_1)@S_2$

- $\Gamma \vdash (L = S_1)@S_2 :: K$ and $\Gamma \vdash (L = S_1)@S_2 :: K'$ (assumption)
- $\Gamma \vdash L :: \text{Nm}$, $\Gamma \vdash S_1 :: \text{Type}$, $\Gamma \vdash S_2 :: \{t: \text{Rec} \mid K \not= t\}$ and $\Gamma \vdash K \equiv \text{Rec}$ (by inversion)
- $\Gamma \vdash L :: \text{Nm}$, $\Gamma \vdash S_1 :: \text{Type}$, $\Gamma \vdash S_2 :: \{t: \text{Rec} \mid K \not= t\}$ and $\Gamma \vdash K' \equiv \text{Rec}$ (by inversion)
- $\Gamma \vdash \text{Rec} \leq \text{Rec}$ (by reflexivity)

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

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

Case: $M$ is if $T' :: \mathcal{K}$ as $t$ then $M'$ else $N'$
Theorem 6.6 (Type Preservation). Let $\Gamma \vdash S M : T$ and $\Gamma \vdash S H$. If $(H; M) \rightarrow (H'; M')$ 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: \( T_0 \rightarrow T'_0 \)

\[ \langle H; (\Lambda t::K. M)[T_0] \rangle \rightarrow \langle H; (\Lambda t::K. M)[T'_0] \rangle \]

\( \Gamma \vdash T \equiv U :: \mathcal{K} \) where \( \Gamma \vdash \Lambda t::K. M : T_1 \), \( \Gamma \vdash T_0 :: K \), \( \Gamma \vdash U :: \mathcal{K} \),

\( \Gamma \vdash T_1 \equiv \forall t::K.S :: \text{Gen}_{K} \) and \( \Gamma \vdash t::K + S :: \mathcal{K} \)

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

\( \Gamma \vdash U \equiv S\{T_0/t\} :: \mathcal{K} \)

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: \( \langle H; M \rangle \rightarrow \langle H'; M' \rangle \)

\( \langle H; \ell = M @ N \rangle \rightarrow \langle H'; \ell = M' @ N \rangle \)

\( \Gamma \vdash S \ T \equiv L :: T'' \) \( \Gamma \vdash S \ell \equiv L :: \text{Nm} \), \( \Gamma \vdash S M : T' \) 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.

\( \Gamma \vdash S' \langle \ell = M' @ N \rangle : L : T'' \)

by RecCons rule

Case: \( \langle H; \text{recHeadTerm}(M) \rangle \rightarrow \langle H'; \text{recHeadTerm}(M') \rangle \)

\( \Gamma \vdash S \ T \equiv T' \{S'/t\} :: \mathcal{K}\{S'/t\} \), \( \Gamma \vdash S M : S' \) and

\( \Gamma \vdash S' :: \{t: \text{Rec} \mid \text{headType}(t) \equiv T' :: \mathcal{K} \} \)

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.

\( \Gamma \vdash S_0 \text{recHeadTerm}(M') : T' \{S'/t\} \)

by i.h.

Case: \( \langle H; \text{recHeadTerm}(\langle \ell = v \rangle @ v') \rangle \rightarrow \langle H; v \rangle \)

\( \Gamma \vdash \text{recHeadTerm}(\langle \ell = v \rangle @ v') : T' \) and \( \Gamma \vdash v : T' \)

by typing rule

Case: \( \Gamma \vdash \varphi \)

by i.h.

Case: \( \langle H; \text{if } \varphi \text{ then } M \text{ else } N \rangle \rightarrow \langle H; M \rangle \)

\( \Gamma \models \text{if } \varphi \text{ then } T_1 \text{ else } T_2 :: K \) with \( \Gamma \models \varphi M : T_1 \) and \( \Gamma \models \neg \varphi N : T_2 \)

by inversion

\( \Gamma \models \text{if } \varphi \text{ then } T_1 \text{ else } T_2 :: K \)

by eq. rule

\( \Gamma \models T :: K \)

by transitivity

\( \Gamma \vdash S M : T_1 \)

by cut

Case: \( \langle H; \mu F : T. M \rangle \rightarrow \langle H; M\{\mu F : T. M/F \} \rangle \)

\( \Gamma, F : T + M : T \) and structural \( (F, M) \)

by inversion

\( \Gamma \vdash M\{\mu F : T.M/F \} : T \)

by substitution

Case: \( \langle H; \text{if } T' :: K \text{ as } t \Rightarrow M \text{ else } N \rangle \rightarrow \langle H; M\{T'/t\} \rangle \)

\( \Gamma \vdash T' :: K', \Gamma \vdash K, \Gamma, t : K + M : T'' \) and \( \Gamma \vdash N : T'' \)

by inversion

\( \Gamma \vdash T :: K \)

assumption

\( \Gamma \vdash M\{T'/t\} : T'' \)

by substitution
Lemma 6.7 (Type Progress). If $\cdot \vdash T :: K$ then either $T$ is a type value or $T \rightarrow T'$, for some $T'$.

Proof. Straightforward induction on kinding, relying on the decidability of logical entailment.

Theorem 6.8 (Progress). Let $\cdot \vdash S : M$ and $\cdot \vdash S : H$. Then either $M$ is a value or there exists $S'$ and $M'$ such that $\langle H; M \rangle \rightarrow \langle H'; M' \rangle$.

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