Refinement Kinds

Type-Safe Programming with Practical Type-Level Computation

LUÍS CAIRES, NOVA-LINCS, FCT-NOVA, Universidade Nova de Lisboa, Portugal
BERNARDO TONINHO, NOVA-LINCS, FCT-NOVA, Universidade Nova de Lisboa, 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-directed 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 checker, type checker and interpreter for our language.

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

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

ACM Reference Format:
Luís Caires and Bernardo Toninho. 2019. Refinement Kinds: Type-Safe Programming with Practical Type-Level Computation. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 131 (October 2019), 30 pages. https://doi.org/10.1145/3360557

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 making programming more accessible to non-experts, and, more generally, increasing the level of abstraction expressible in

Authors' addresses: Luís Caires, Departamento de Informática, NOVA-LINCS, FCT-NOVA, Universidade Nova de Lisboa, Portugal, lcaires@fct.unl.pt; Bernardo Toninho, Departamento de Informática, NOVA-LINCS, FCT-NOVA, Universidade Nova de Lisboa, Portugal, btoninho@fct.unl.pt.

2475-1421/2019/10-ART131
https://doi.org/10.1145/3360557

languages and tools for program construction. Meta-programming is better supported by so-called dynamic languages and related frameworks, such as Ruby and Ruby on Rails, JavaScript and Node.js, but is 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 extant 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 $\langle\rangle$ and the "cons" cell $\langle L : T\rangle @ R$, which constructs the record type obtained by adding a field declaration $\langle L : T\rangle$ to the record type R.

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

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

The addFieldType *type-level* function takes a label l, a type t and any record type r that does not contain label l, and returns the expected extended record type of kind Rec. Notice that the *kind* of all record types that do not contain label l is represented by the refinement kind $\{s{::}\mathsf{Rec} \mid l \notin \mathbf{lab}(s)\}$.

A refinement kind in our system is noted $\{t{::}\mathcal{K} \mid \varphi(t)\}$, where $\mathcal{K}$ is a bas kind, and the logical formula $\varphi(t)$ expresses a constraint on the type t that inhabits $\mathcal{K}$. As in refinement type systems [Bengtson et al. 2011; Swamy et al. 2011; Vazou et al. 2014], 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

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

where $\Gamma \models \varphi$ denotes entailment in the refinement logic. Basic formulas of our refinement logic include propositional logic, equality, and some useful predicates and functions on types, including the primitive type constructors and destructors, such as $\mathbf{lab}(R)$ (record label set), $L \in S$ (label set membership), $S\#S'$ (label set apartness), $R@S$ (concatenation), $\mathbf{dom}(F)$ (function domain selector). Interestingly, given the presence of equality in refinements, it is always possible to define for any type T of kind $\mathcal{K}$ a precise singleton kind of the form $\{t :: \mathcal{K} \mid t = T\}$. As another simple example, consider the kind Auto of automorphisms, defined as $\{t :: \mathsf{Fun} \mid \mathbf{dom}(t) = \mathbf{img}(t)\}$.

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

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

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

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

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

An instance of this judgement yields:

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

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

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

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

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

Contributions. We summarise the main contributions of this work:

- 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).
- We technically develop our refinement kind system (Section 3), targeting a polymorphic λ-calculus (Section 4) with records, references, collections and supporting type-level computation over types of all kinds, thus capturing the essence of an ML-like language.
- We establish the key meta-theoretical result (Section 5) of type safety through type unicity, type preservation and progress (Theorems 5.5, 5.6 and 5.8, respectively).
- We report on our implementation of a prototype kind and type-checker for our theory (Section 6), which validates our examples and the overall feasibility of our approach.
- We give a detailed overview of related work (Section 7), and offer some concluding remarks and discussion of future work (Section 8).

A companion technical report [Caires and Toninho 2019] lists omitted definitions of the type theory, its semantics and proof outlines.

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:

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

Given a non-empty record type S, function genConstr returns a constructor *function* for a mutable record whose fields are specified by S. We use a pragmatic notation to express recursive definitions (coinciding 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 $\lambda x{:}\mathsf{String}.\lambda y{:}\mathsf{Int}.\langle \mathit{age} = \mathbf{ref}\ y; \mathit{name} = \mathbf{ref}\ x\rangle$.

Notice that function genConstr accepts any non-empty record type S, and proceeds by recursion on the structure 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 $\mathbf{lab}(V)\#\mathbf{lab}(S)$, we can automatically deduce that $\mathbf{headLabel}(S) \notin \mathbf{lab}(V)$, needed to kind check $\langle\mathbf{headLabel}(S) : \mathbf{ref}\ \mathbf{headType}(S)\rangle@V$; and $\mathbf{lab}(\langle\mathbf{headLabel}(S) : \mathbf{ref}\ \mathbf{headType}(S)\rangle@V)\#\mathbf{lab}(\mathbf{tail}(S))$, required to kind and type check the recursive call. In our language, genConstr can be typed as follows:

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

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

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

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

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

From Record Types to XML Tables. As a second example, we develop a generic function MkTable that generates 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 first introduce 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.

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

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

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

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

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

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

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

Hence $M.\mathbf{headLabel}(M).tag$: String. By a similar reasoning, we conclude $r.\mathbf{recHeadLabel}(M)$: $\mathbf{headType}(R)$. In Section 3 we show how refinements and equalities derived therein are integrated into typing and kinding. Moreover, in Section 6 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 MkMut that generates a getter/setter wrapper for any mutable record (i.e. a record where all its fields are of reference type). We first define the auxiliary type-level MutableRec function, that returns the mutable record type obtained from a record type R in terms of Map:

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

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

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

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

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

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

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

where (MkMut Person r) : SetGetRec Person.

3 A TYPE THEORY WITH KIND REFINEMENTS

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

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

Record Types. 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 $\langle\rangle$, and the constructor $\langle L : T\rangle@R$, which given a record type R that does not contain the label L, generates a record type that is an extension of R with the label L associated with type T. Record types are associated with three destructors: $\mathbf{headLabel}(T)$, which projects the label of the head of the record T (when seen as a list); $\mathbf{headType}(T)$ which produces the type at the head of the record T; and $\mathbf{tail}(T)$ which produces the tail of the record T (i.e. drops its first label and type pair). As we will see (Example 3.1), since our type-level λ-calculus allows for structural recursion, we can *define* a suitable record projection type construct in terms of these lower-level primitives.

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

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

Kinds	K, K'	::=	$\mathcal{K} \mid \{t{::}\mathcal{K} \mid \varphi\} \mid \Pi t{:}K.K'$	Refinement Kinds
	$\mathcal{K}$	::=	$\mathsf{Rec} \mid \mathsf{Col} \mid \mathsf{Fun} \mid \mathsf{Ref} \mid \mathsf{Nm}$	Basic Kinds
		|	$\mathsf{Type} \mid \mathsf{Gen}_K$	
Types	T, S, R	::=	$t \mid \lambda t{::}K.T \mid T\ S$	Type-level Functions
		|	$\boldsymbol{\mu} F : (\Pi t{:}K.\ K').\lambda t{::}K.\ T$	Structural Recursion
		|	$\forall t{::}K.T$	Polymorphism
		|	$L \mid \langle\rangle \mid \langle L : T\rangle @ S$	Record Type constructors
		|	$\mathbf{headLabel}(T) \mid \mathbf{headType}(T)$	Record Type destructors
		|	$\mathbf{tail}(T)$	
		|	$T^\star \mid \mathbf{colOf}(T)$	Collection Types
		|	$\mathbf{ref}\ T \mid \mathbf{refOf}(T)$	Reference Types
		|	$T \to S \mid \mathbf{dom}(T) \mid \mathbf{img}(T)$	Function Types
		|	$\mathbf{if}\ T :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow S\ \mathbf{else}\ U$	Kind Case
		|	$\mathbf{if}\ \varphi\ \mathbf{then}\ T\ \mathbf{else}\ S$	Property Test
		|	$\mathsf{Bool} \mid \mathbf{1} \mid \ldots$	Basic Data Types
Extended Types	$\mathcal{T}, \mathcal{S}$	::=	$T \mid \mathbf{lab}(T) \mid \mathcal{T} +\!+ \mathcal{S}$	
Refinements	φ, ψ	::=	$\varphi \supset \psi \mid \varphi \wedge \psi \mid \ldots$	Propositional Logic
		|	$\mathsf{empty}(\mathcal{T})$	Empty Record Test
		|	$\mathcal{T} = \mathcal{S}$	Equality
		|	$\mathcal{T} \in \mathcal{S}$	Label Set Membership
		|	$\mathcal{T} \# \mathcal{S}$	Label Set Apartness

Fig. 1. Syntax of Kinds, Types and Refinements

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

Kind Test. Just as many programming languages have a type case construct [Abadi et al. 1991] that allows for the runtime testing of the type of a given expression, our λ-calculus of types has a *kind case* construct, $\mathbf{if}\ T :: \mathcal{K}\ \mathbf{as}\ t \Rightarrow S\ \mathbf{else}\ U$, which checks the kind of type T against kind $\mathcal{K}$, computing to type S if the kinds match and to U otherwise. 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 λ-calculus for the manipulation of such structures and so includes functions from types to types, $\lambda t{::}K.T$, and their respective application, written $T\ S$. We also include a type-level structural recursion operator $\boldsymbol{\mu} F : (\Pi t{:}K.\ K').\lambda t{::}K.\ T$, which allows us to define recursive type functions from kind K to K'. While written as a fixpoint operator, we syntactically enforce that recursive calls must always take structurally smaller arguments to ensure well-foundedness.

Type-level functions are *dependently kinded*, with kind $\Pi t{:}K.K'$ (i.e. the kind of the image type in a type λ-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 $\{t{::}\mathcal{K} \mid \varphi\}$, where $\mathcal{K}$ is a *basic* kind, and φ is a logical formula (with t bound in φ), characterises types T of kind $\mathcal{K}$ such that the property φ holds of T (i.e. $\varphi\{T/t\}$ is true). The language of properties φ can refer to the syntax of types, extended with a refinement-level notion of label set of a (record) type, $\mathbf{lab}(T)$, and a notion of label set concatenation, $\mathcal{T} +\!\!+ \mathcal{S}$, where $\mathcal{T}$ is such an *extended* type. Refinements φ, ψ consist of propositional logic formulae, (logical) equality, $\mathcal{T} = \mathcal{S}$, an empty record predicate $\mathrm{empty}(\mathcal{T})$, and basic label set predicates and such as label inclusion ($\mathcal{T} \in \mathcal{S}$) and set apartness ($\mathcal{T} \# \mathcal{S}$). 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 $\{t{::}\mathsf{Fun} \mid \mathbf{img}(t) = \mathsf{Bool}\}$, 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:

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

The function dropField above takes label l and a record type with a field labelled by l and removes the corresponding field and type pair from the record type (recall that $\mathbf{lab}(r)$ denotes the refinement-level set of labels of r). Such a function combines structural recursion (where $\mathbf{tail}(t)$ is correctly deemed as structurally smaller than t) with our type-level refinement test, $\mathbf{if}\ \varphi\ \mathbf{then}\ T\ \mathbf{else}\ S$. We note that the well-kindedness of such a function relies on the ability to derive that, when the record label $\mathbf{headLabel}(t)$ is not l, since we know that l must be in t, $\mathbf{tail}(t)$ 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 Γ. Contexts contain assumptions of the form $t{:}K$, $x{:}T$ and $\varphi - t$ stands for a type of kind K, x stands for a term of type T and refinement φ is assumed to hold, respectively. Kinding relies on a context well-formedness judgment, written $\Gamma \vdash$, a kind well-formedness judgment $\Gamma \vdash K$, subkinding judgment $\Gamma \vdash K \leq K'$ and the refinement well-formedness and entailment judgments, $\Gamma \vdash \varphi$ and $\Gamma \models \varphi$. Context well-formedness simply checks that all types, kinds and refinements in Γ are well-formed. Kind well-formedness is defined in the standard way, relying on refinement well-formedness (see [Caires and Toninho 2019]), 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 $\mathcal{K}$ and a *valid* property φ of T, we are justified in stating that T is of kind $\{t{::}\mathcal{K} \mid \varphi\}$.

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: \mathcal{K}}{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid \varphi\}}\ (\textsc{kref}) \qquad \frac{\Gamma \vdash \varphi \quad \mathsf{Valid}(\llbracket\Gamma\rrbracket \Rightarrow \llbracket\varphi\rrbracket)}{\Gamma \models \varphi}\ (\textsc{entails})$$

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 basic kind) and if the representation of the context Γ and the refinement φ as an implicational formula is SMT-valid. The context and refinement representation is discussed in Section 6.

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 β-equivalence reasoning for records allows us to derive $\langle \ell : \mathsf{Bool} \to \mathsf{Bool}\rangle@\langle\rangle :: \{t{::}\mathsf{Rec} \mid \mathbf{headType}(t) = \mathsf{Bool} \to \mathsf{Bool}\}$. In general, we provide an equality elimination rule for refinements (R-EQELIM), internalizing such equalities in definitional equality of our theory:

$$\frac{\Gamma \vdash T :: \{t{::}\mathcal{K} \mid t = S\} \quad \Gamma \vdash S :: \mathcal{K}}{\Gamma \models T \equiv S :: \mathcal{K}}\ (\textsc{r-eqelim})$$

These principles become particularly interesting when reasoning from refinements that appear in type variables. For instance, the type $\forall t{::}\{f{:}\mathsf{Fun} \mid \mathbf{dom}(f) = \mathsf{Bool} \wedge \mathbf{img}(f) = \mathsf{Bool}\}.t \to \mathsf{Bool}$ can be used to type the term $\Lambda t{::}\{f{:}\mathsf{Fun} \mid \mathbf{dom}(f) = \mathsf{Bool} \wedge \mathbf{img}(f) = \mathsf{Bool}\}.\lambda f{:}t.(f\ \mathit{true})$, where Λ is the binder for polymorphic functions, as usual. Crucially, typing (and kinding) exploits not only the fact that we know that the type variable t stands for a function type, but also that the domain and image are the type Bool, which then warrants the application of f to a boolean in order to produce a boolean, despite the basic kinding information only specifying that f is 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 $\mathbf{if}\ \varphi\ \mathbf{then}\ T\ \mathbf{else}\ S$. Provided that the property φ is well-formed, if T is of kind K assuming φ and S of kind K assuming $\neg\varphi$, then the conditional test is well-kinded, as specified by the rule (K-ITE). The equality principles for the property test rely on validity of the specified property (with a degenerate case where both branches are equal types). We note that $\Gamma, \varphi \vdash T :: K$ can effectively be represented as $\Gamma, x : \{\_ \mid \varphi\} \vdash T :: K$ where x is fresh. This representation encodes φ in the context through a "dummy" refinement that simply asserts the property.

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

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

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):

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

$$\frac{\Gamma \vdash T :: \Pi t{:}K_1.K_3 \quad \Gamma, t{:}K_1 \vdash T\,t :: K_2 \quad x \notin fv(T)}{\Gamma \vdash T :: \Pi t{:}K_1.K_2}\ (\textsc{k-ext}) \qquad \frac{\Gamma \vdash S :: \Pi t{:}K_1.K_3 \quad \Gamma \vdash T :: \Pi t{:}K_1.K_4 \quad \Gamma, t{:}K_1 \vdash S\,t \equiv T\,t :: K_2}{\Gamma \vdash S \equiv T :: \Pi t{:}K_1.K_2}\ (\textsc{eq-funext})$$

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

Rules (K-EXT) and (EQ-FUNEXT) allow for basic extensionality principles on type-level functions. The former states that an η-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-FUNEXT) is the analogous rule for type equality. We note that such rules, although they allow us to equate types such as $\lambda t{::}\{s{:}\mathsf{Type} \mid t = \mathsf{Bool} \to \mathsf{Bool}\}.t$ and $\lambda t{::}\{s{:}\mathsf{Type} \mid t = \mathsf{Bool} \to \mathsf{Bool}\}.\mathsf{Bool} \to \mathsf{Bool}$, 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:

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

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

The predicate $\mathsf{structural}(T, F, t)$ enforces that calls of F in T must take arguments that are structurally smaller than t (i.e. the arguments must be syntactically equal to t applied to a destructor). More precisely, the predicate $\mathsf{structural}(T, F, t)$ holds iff all occurrences of F in T are applied to terms smaller than t, where the notion of size is given by $elim(t) < t$, where $elim(t)$ stands for an appropriate destructor applied to t (e.g., if t is of kind Fun then $\mathbf{dom}(t) < t$). 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 Gen_K:

$$\frac{\Gamma \vdash K \quad \Gamma, t{:}K \vdash T :: \mathcal{K}}{\Gamma \vdash \forall t{::}K.T :: \mathsf{Gen}_K}\ (\textsc{k-}\forall)$$

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:

$$\frac{\Gamma \vdash T :: \mathcal{K} \quad \Gamma \vdash S :: \mathcal{K}'}{\Gamma \vdash T \to S :: \mathsf{Fun}}\ (\textsc{k-fun}) \qquad \frac{\Gamma \vdash T :: \mathsf{Fun}}{\Gamma \vdash \mathbf{dom}(T) :: \mathsf{Type}}\ (\textsc{k-dom}) \qquad \frac{\Gamma \vdash T :: \mathsf{Fun}}{\Gamma \vdash \mathbf{img}(T) :: \mathsf{Type}}\ (\textsc{k-codom})$$

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

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

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

$$\frac{\Gamma \vdash T :: \{t{::}\mathsf{Rec} \mid \neg\mathsf{empty}(t)\}}{\Gamma \vdash \mathbf{headType}(T) :: \mathsf{Type}}\ (\textsc{k-hdt}) \qquad \frac{\Gamma \vdash T :: \{t{::}\mathsf{Rec} \mid \neg\mathsf{empty}(t)\}}{\Gamma \vdash \mathbf{headLabel}(T) :: \mathsf{Nm}}\ (\textsc{k-hdl}) \qquad \frac{\Gamma \vdash T :: \{t{::}\mathsf{Rec} \mid \neg\mathsf{empty}(t)\}}{\Gamma \vdash \mathbf{tail}(T) :: \mathsf{Rec}}\ (\textsc{k-tail})$$

The rule for non-empty records 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-kinded.

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

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

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.

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

Conversion and Subkinding. As we have informally described earlier, our theory of kinds is predicated on the idea that we can distinguish between the different specialized types at the kind level. For instance, the kind of record types 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:

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

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

Rule (SUB-REFKIND) specifies that a refined kind is always a subkind of its unrefined variant. Rule (SUB-REF) allows for subkinding between refined kinds, by requiring that the basic kind respects subkinding and that the 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 (K-KCASE)), accounting for the case where the kind of type T matches the specified kind $\mathcal{K}'$ with

type S and with type U otherwise.

$$\frac{\Gamma \vdash \mathcal{K} \quad \Gamma \vdash T :: \mathcal{K}'' \quad \Gamma, t{:}\mathcal{K} \vdash S :: K' \quad \Gamma \vdash U :: K'}{\Gamma \vdash \textbf{if}\ T :: \mathcal{K}\ \textbf{as}\ t \Rightarrow S\ \textbf{else}\ U :: K'}\ (\textsc{k-kcase}) \qquad \frac{\Gamma \models \bot \quad \Gamma \vdash K}{\Gamma \vdash \bot :: K}\ (\textsc{k-bot})$$

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

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

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

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 $L :: \mathsf{Nm}$ produces a recursive type-function that essentially iterates over a type record of kind $\{r{::}\mathsf{Rec} \mid \ell \in \textbf{lab}(r)\}$:

$$\lambda L{::}\mathsf{Nm}.\boldsymbol{\mu} F : (\Pi t{:}\{r{::}\mathsf{Rec} \mid L \in \textbf{lab}(r)\}.\,\mathsf{Type}).\lambda t{::}\{r{::}\mathsf{Rec} \mid L \in \textbf{lab}(r)\}.$$
$$\textbf{if}\ \textbf{headLabel}(t) = L\ \textbf{then}\ \textbf{headType}(t)\ \textbf{else}\ F(\textbf{tail}(t)) :: \Pi L : \mathsf{Nm}.\Pi t : \{r{::}\mathsf{Rec} \mid L \in \textbf{lab}(r)\}.\mathsf{Type}$$

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 match and recurring otherwise. It is instructive to consider the kinding for the property test construct (let Γ_0 be $L{:}\mathsf{Nm}, F{:}\Pi t{:}\{r{::}\mathsf{Rec} \mid L \in \textbf{lab}(r)\}.\mathsf{Type}, t{:}\{r{:}\mathsf{Rec} \mid L \in \textbf{lab}(r)\}$):

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

where $\mathcal{D}$ is a derivation of $\Gamma_0, \textbf{headLabel}(t) = L \vdash \textbf{headType}(t) :: \mathsf{Type}$ and $\mathcal{E}$ is a derivation of $\Gamma_0, \neg(\textbf{headLabel}(t) = L) \vdash F(\textbf{tail}(t)) :: \mathsf{Type}$. To show that $\textbf{headLabel}(t) = L$ is well-formed we must be able to derive $t :: \{r{::}\mathsf{Rec} \mid \neg\mathsf{empty}(r)\}$ from $t :: \{r{::}\mathsf{Rec} \mid L \in \textbf{lab}(r)\}$, which is achieved via subkinding, by appealing to entailment in our underlying theory (see Section 6). Similarly, the derivation $\mathcal{E}$ requires the ability to conclude that $\textbf{tail}(t) :: \{r{::}\mathsf{Rec} \mid L \in \textbf{lab}(r)\}$, using the information that $t :: \{r{::}\mathsf{Rec} \mid L \in \textbf{lab}(r)\}$ and $\neg(\textbf{headLabel}(t) = L)$, which is also a valid entailment.

Example 3.2 (Generic Pairing of Objects). The following example consists of an object (implemented as a record of methods) combinator Pairer which takes two object types X and Y and for every method that X and Y have in common, Pairer X Y contains a method with the same name and domain types, but where the return type is a pair of the two original return types. This practical example is inspired by an example found in Huang and Smaragdakis [2008], which uses pattern-based nested reflection in the context of Java.

We first define a Pair type constructor as a type-level function that takes two types X and Y as argument and produces a two-element record, where the label fst denotes the first element of the pair (of type X) and the label snd denotes the second element of the pair (of type Y):

$$\begin{aligned}\mathsf{Pair} &:: \Pi X{::}\mathsf{Type}.\Pi Y{::}\mathsf{Type}.\{r :: \mathsf{Rec} \mid \varphi\}\\ \mathsf{Pair} &\triangleq \lambda X{::}\mathsf{Type}.\lambda Y{::}\mathsf{Type}.\langle \mathsf{fst} : X\rangle@\langle \mathsf{snd} : Y\rangle@\langle\rangle\end{aligned}$$

$\varphi \triangleq \mathbf{headLabel}(r) = \mathsf{fst} \wedge \mathbf{headType}(r) = X \wedge \mathbf{headLabel}(\mathbf{tail}(r)) = \mathsf{snd} \wedge \mathbf{headType}(\mathbf{tail}(r)) = Y$

For the sake of conciseness, we make use of a predicate isObj on record types which holds if a record is a record of functions (i.e. object methods). For simplicity we assume that methods are functions of exactly one argument.

We now define a Pairer type-level function which takes two object types X and Y to produce a new object type which contains the same methods of X and Y, but where the methods that X and Y have in common (i.e. methods with the same name and same argument types) have as result type the pairing of the two original return types.

$$
\begin{array}{ll}
\mathsf{Pairer} & :: \Pi X :: \{r :: \mathsf{Rec} \mid \mathsf{isObj}(r)\}.\Pi Y :: \{r :: \mathsf{Rec} \mid \mathsf{isObj}(r)\}.\{r :: \mathsf{Rec} \mid \mathsf{isObj}(r)\} \\
\mathsf{Pairer} & \triangleq \lambda X :: \{r :: \mathsf{Rec} \mid \mathsf{isObj}(r)\}.\lambda Y :: \{r :: \mathsf{Rec} \mid \mathsf{isObj}(r)\}. \\
& \quad \mathbf{if}\ \neg\mathsf{empty}(X) \wedge \neg\mathsf{empty}(Y)\ \mathbf{then} \\
& \quad\quad (\mathbf{if}\ \mathbf{headLabel}(X) \in \mathbf{lab}(Y) \wedge \mathbf{dom}(Y.\mathbf{headLabel}(X)) = \mathbf{dom}(\mathbf{headType}(X))\ \mathbf{then} \\
& \quad\quad \langle \mathbf{headLabel}(X) : \mathbf{dom}(\mathbf{headType}(X)) \to \\
& \quad\quad\quad\quad \mathsf{Pair}\,(\mathbf{img}(\mathbf{headType}(X)))\,(\mathbf{img}(Y.\mathbf{headLabel}(X)))\rangle @ \mathsf{Helper} \\
& \quad\quad \mathbf{else}\ \langle \mathbf{headLabel}(X) : \mathbf{headType}(X)\rangle @ \mathsf{Helper}) \\
& \quad \mathbf{else} \\
& \quad\quad \mathbf{if}\ \neg\mathsf{empty}(X)\ \mathbf{then}\ X\ \mathbf{else}\ Y \\
\mathsf{Helper} & = \mathbf{if}\ (\mathbf{headLabel}(Y) \in \mathbf{lab}(X) \wedge \mathbf{dom}(X.\mathbf{headLabel}(Y)) = \mathbf{dom}(\mathbf{headType}(Y)) \wedge \\
& \quad \mathbf{headLabel}(X) \neq \mathbf{headLabel}(Y))\ \mathbf{then} \\
& \quad\quad \langle \mathbf{headLabel}(Y) : \mathbf{dom}(\mathbf{headType}(Y)) \to \\
& \quad\quad\quad \mathsf{Pair}\,(\mathbf{img}(X.\mathbf{headLabel}(Y)))\,(\mathbf{img}(\mathbf{headType}(Y)))\rangle @ \\
& \quad\quad\quad \mathsf{dropField}(\mathbf{headLabel}(Y), \mathsf{Pairer}\,(\mathbf{tail}(X))\,(\mathsf{dropField}(\mathbf{headLabel}(X), \mathbf{tail}(Y)))) \\
& \quad \mathbf{else}\ \mathsf{Pairer}\,(\mathbf{tail}(X))\,(\mathsf{dropField}(\mathbf{headLabel}(X), Y))
\end{array}
$$

The Pairer function above proceeds recursively over the records X and Y. When Y is empty, the function returns X since there is nothing left to pair, and similarly for when X is empty. When neither X or Y are empty, we test whether the head label of X is in the label set of Y with a matching domain type, if not, then there is no pairing to be done with the method at the head of X and the resulting record copies the method signature from X. If the conditional holds, then we produce a function type with the appropriate domain and where the image is the pairing of the two image types. In both cases (to ease with the formating) the tail of the record is defined by a Helper definition.

The Helper definition tests whether the head label of Y is in X with matching domain types, but is not the first label of X (which is handled in the previous test). If the condition holds, then we must include the head method of Y with the appropriately paired image type. The recursive call to Pairer makes use of the dropField type-level function, which removes a record entry, to ensure that the head label of X is removed from the tail of Y and that the head label of Y is removed from the result of the recursive call. When the condition does not hold we simply recurse on the tail of X and on Y with the method labelled by $\mathbf{headLabel}(X)$ removed.

4 A PROGRAMMING LANGUAGE WITH KIND REFINEMENTS

Having covered the key details of kinding and type equality, we introduce the syntax and typing for our programming language *per se*, capturing the essence of an ML-style 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 ε and the concatenation of an element M with a collection N, $M :: N$,

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

Fig. 2. Syntax of Terms

with the usual case analysis $\mathbf{case}\ M\ \mathbf{of}\ (\varepsilon \Rightarrow N_1 \mid x{::}xs \Rightarrow N_2)$ 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 xs with its tail. We allow for recursive terms via a fixpoint construct $\boldsymbol{\mu} F{:}T.M$, noting that since there are no type dependencies, non-termination in the term language does not affect the overall soundness of the development. We also mirror the type-level property test and kind case constructs in the term language as $\mathbf{if}\ \varphi\ \mathbf{then}\ M\ \mathbf{else}\ N$ and $\mathbf{if}\ T :: K\ \mathbf{as}\ t \Rightarrow M\ \mathbf{else}\ N$, respectively. As we have initially stated, our language has general higher-order references, represented with the constructs $\mathbf{ref}\ M$, $!M$ and $M := N$, which create a reference to M, dereference a reference M and assign N to the reference M, respectively. As usual in languages with a store, we use l to stand for the runtime values of memory locations.

The typing rules for the language are given in Figure 3. The typing judgment is written as $\Gamma \vdash_S M : T$, where S is a location typing environment. We write $\Gamma; S \vdash$ to state that S is a valid mapping from locations to well-kinded types, according to the typing context Γ. Notably, despite the fairly advanced type-level features, the typing rules are virtually unchanged when compared to those of a language in the ML family.

In fact, the advanced kinding and type equality features manifest themselves in typing via the (CONV) conversion rule, (KINDCASE) and the ($\langle\rangle I_2$) 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:

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

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 $t = s \longrightarrow \mathsf{Bool}$ 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.

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

$$\frac{\Gamma \vdash_S M : T \to S \quad \Gamma \vdash_S N : T}{\Gamma \vdash_S M\,N : S}\ (\to\mathrm{E}) \qquad \frac{\Gamma \vdash K \quad \Gamma, t{:}K \vdash_S M : T}{\Gamma \vdash_S \Lambda t{::}K.M : \forall t{::}K.T}\ (\forall\mathrm{I}) \qquad \frac{\Gamma \vdash_S M : \forall t :: K.S \quad \Gamma \vdash T :: K}{\Gamma \vdash_S M[T] : S\{T/t\}}\ (\forall\mathrm{E})$$

$$\frac{\Gamma \vdash \quad \Gamma; S \vdash}{\Gamma \vdash_S \langle\rangle : \langle\rangle}\ (\langle\rangle I_1) \qquad \frac{\Gamma \vdash_S L :: \mathsf{Nm} \quad \Gamma \vdash_S M : T_1 \quad \Gamma \vdash T_2 :: \{t{::}\mathsf{Rec} \mid L \notin \mathbf{lab}(t)\} \quad \Gamma \vdash_S N : T_2}{\Gamma \vdash_S \langle L = M\rangle @ N : \langle L : T_1\rangle @ T_2}\ (\langle\rangle I_2)$$

$$\frac{\Gamma \vdash_S M : \langle L : T\rangle @ U}{\Gamma \vdash_S \mathbf{recHeadLabel}(M) : L}\ (\textsc{reclabel}) \qquad \frac{\Gamma \vdash_S M : \langle L : T\rangle @ U}{\Gamma \vdash_S \mathbf{recHeadTerm}(M) : T}\ (\textsc{recterm}) \qquad \frac{\Gamma \vdash_S M : \langle L : T\rangle @ U}{\Gamma \vdash_S \mathbf{tail}(M) : U}\ (\textsc{rectail})$$

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

$$\frac{\Gamma \vdash T :: \mathsf{Type} \quad \Gamma; S \vdash}{\Gamma \vdash_S \varepsilon : T^\star}\ (\textsc{emp}) \qquad \frac{\Gamma \vdash_S M : T \quad \Gamma \vdash_S N : T^\star}{\Gamma \vdash_S M :: N : T^\star}\ (\textsc{cons}) \qquad \frac{\Gamma \vdash_S M : T^\star \quad \Gamma \vdash N_1 : S \quad \Gamma, x{:}T, xs{:}T^\star \vdash N_2 : S}{\Gamma \vdash_S \mathbf{case}\ M\ \mathbf{of}\ (\varepsilon \Rightarrow N_1 \mid x{::}xs \Rightarrow N_2) : S}\ (\textsc{case})$$

$$\frac{\Gamma \vdash \quad \Gamma; S \vdash \quad S(l) = T}{\Gamma \vdash_S l : \mathbf{ref}\ T}\ (\textsc{loc}) \qquad \frac{\Gamma \vdash_S M : T}{\Gamma \vdash_S \mathbf{ref}\ M : \mathbf{ref}\ T}\ (\textsc{ref}) \qquad \frac{\Gamma \vdash_S M : \mathbf{ref}\ T}{\Gamma \vdash_S\ !M : T}\ (\textsc{deref}) \qquad \frac{\Gamma \vdash_S M : \mathbf{ref}\ T \quad \Gamma \vdash_S N : T}{\Gamma \vdash_S M := N : \mathbf{1}}\ (\textsc{assign})$$

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

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

Fig. 3. Typing Rules

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

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

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

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

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

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

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

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

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

with $\mathcal{D}$ a derivation of

$$\Gamma_0 \vdash \mathbf{if}\ (\mathbf{headLabel}(T) = L)\ \mathbf{then}\ \mathbf{recHeadTerm}(x)\ \mathbf{else}\ F[\mathbf{tail}(T)](\mathbf{tail}(x)) : T_0$$

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

The **else** branch requires showing that $\Gamma_0, \neg\mathbf{headLabel}(T) = L \vdash F[\mathbf{tail}(T)](\mathbf{tail}(x)) : \mathbf{tail}(T).L$, which is derivable from $F : \forall t :: \mathcal{R}.t \to (t.L)$ and $x{:}T$ as follows: $\mathbf{tail}(T) :: \mathcal{R}$ follows from $\neg\mathbf{headLabel}(T) = L$ and $T :: \mathcal{R}$ (see Section 6), thus $F[\mathbf{tail}(T)] : \mathbf{tail}(T) \to \mathbf{tail}(T).L$. Since $\mathbf{tail}(x) : \mathbf{tail}(T)$ from $x : T$ and $x : \langle \mathbf{headLabel}(T) : \mathbf{headType}(T)\rangle @ \mathbf{tail}(T)$ 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:

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

Example 4.2 (Generic Object Pairing). We now produce the term-level implementation of Example 3.2, which takes two objects x and y of types X and Y and produces a new object of type $\mathsf{Pairer}\ X\ Y$. We first define a constructor for pairs, PCons:

$$\begin{aligned}\mathsf{PCons} &: \forall X :: \mathsf{Type}.\forall Y :: \mathsf{Type}.X \to Y \to \mathsf{Pair}\ X\ Y\\ \mathsf{PCons} &\triangleq \Lambda X :: \mathsf{Type}.\Lambda Y :: \mathsf{Type}.\lambda x{:}X.\lambda y{:}Y.\langle \mathsf{fst} = x, \mathsf{snd} = y\rangle\end{aligned}$$

We now define the pair-object constructor, which makes use of Pairer in its typing and of term-level record projection and record field removal in its definition:

$$
\begin{array}{lll}
\mathsf{ObjPair} & : \forall X :: \{r :: \mathsf{Rec} \mid \mathsf{isObj}(r)\}.\forall Y :: \{r :: \mathsf{Rec} \mid \mathsf{isObj}(r)\}.X \to Y \to \mathsf{Pairer}\, X\, Y \\
\mathsf{ObjPair} & \triangleq \Lambda X :: \{r :: \mathsf{Rec} \mid \mathsf{isObj}(r)\}.\Lambda Y :: \{r :: \mathsf{Rec} \mid \mathsf{isObj}(r)\}.\lambda x{:}X.\lambda y{:}Y. \\
& \quad \textbf{if } \neg\mathsf{empty}(X) \wedge \neg\mathsf{empty}(Y) \textbf{ then} \\
& \quad\quad (\textbf{if } \mathbf{headLabel}(X) \in \mathbf{lab}(Y) \wedge \mathbf{dom}(Y.\mathbf{headLabel}(X)) = \mathbf{dom}(\mathbf{headType}(X)) \textbf{ then} \\
& \quad\quad \langle \mathbf{recHeadLabel}(x) = \lambda z{:}\mathbf{dom}(\mathbf{headType}(X)). \\
& \quad\quad\quad\quad \mathsf{PCons}\,(\mathbf{recHeadTerm}(x)\, z)\,(y.\mathbf{recHeadLabel}(x)\, z)\rangle @ \mathsf{PHelper} \\
& \quad\quad \textbf{else } \langle \mathbf{recHeadLabel}(x) = \mathbf{recHeadTerm}(x)\rangle @ \mathsf{PHelper}) \\
& \quad \textbf{else} \\
& \quad\quad \textbf{if } \neg\mathsf{empty}(X) \textbf{ then } x \textbf{ else } y \\
\\
\mathsf{PHelper} & = \textbf{if } (\mathbf{headLabel}(Y) \in \mathbf{lab}(X) \wedge \mathbf{dom}(X.\mathbf{headLabel}(Y)) = \mathbf{dom}(\mathbf{headType}(Y)) \wedge \\
& \quad \mathbf{headLabel}(X) \neq \mathbf{headLabel}(Y)) \textbf{ then} \\
& \quad\quad \langle \mathbf{recHeadLabel}(y) = \lambda z{:}\mathbf{dom}(\mathbf{headType}(Y)). \\
& \quad\quad\quad \mathsf{PCons}\,(x.\mathbf{recHeadLabel}(y)\, z)\,(\mathbf{recHeadTerm}(y)\, z)\rangle @ \\
& \quad \mathsf{dropField}(\mathbf{recHeadLabel}(y), \mathsf{ObjPair}\,(\mathbf{tail}(x))\,(\mathsf{dropField}(\mathbf{recHeadLabel}(x), \mathbf{tail}(y)))) \\
& \quad \textbf{else } \mathsf{ObjPair}\,(\mathbf{tail}(x))\,(\mathsf{dropField}(\mathbf{recHeadLabel}(x), y))
\end{array}
$$

The structure of the code follows that of the Pairer definition. The key point is the new method construction, where we define a function that takes a value z in the domain of the head type of one of the records and pairs up the result of applying the corresponding methods of x and y to z.

5 OPERATIONAL SEMANTICS AND METATHEORY

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

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

Moreover, since types in our language are themselves structured objects with computational significance, we make use of a type reduction relation, written $T \to T'$, defined as a call-by-value reduction semantics on types when seen as a λ-calculus. It is convenient to define a notion of *type value*, denoted by T_v, S_v and given by the following grammar:

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

We note that it follows from the literature on F_ω and related systems that type reduction is strongly normalizing [Giménez 1998; Norell 2007; Pierce 2002; Stone and Harper 2000]. The values of the *term* language are defined by the grammar:

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

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

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

congruence rules such as:

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

where the record field labelled by ℓ is evaluated (and the resulting modifications in store H to H' are propagated accordingly). The reduction rules enforce a call-by-value, left-to-right evaluation order and are listed in Figure 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 [Caires and Toninho 2019] for the complete set of rules.

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

5.1 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 F_ω [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 5.1 (Assumed Properties of Entailment).

Substitution: *If* $\Gamma \vdash T :: K$ *and* $\Gamma, t{:}K, \Gamma' \models \varphi$ *then* $\Gamma, \Gamma'\{T/k\} \models \varphi\{T/t\}$;
Weakening: *If* $\Gamma \models \varphi$ *then* $\Gamma' \models \varphi$ *where* $\Gamma \subseteq \Gamma'$;
Functionality: *If* $\Gamma \models T \equiv S :: K$ *and* $\Gamma, t : K, \Gamma' \vdash \varphi$ *then* $\Gamma \models \varphi\{T/t\} \Leftrightarrow \varphi\{S/t\}$.
Soundness: *If* $\mathsf{Valid}(\llbracket\Gamma\rrbracket \Rightarrow \llbracket\varphi\rrbracket)$, *then* $\llbracket\Gamma\rrbracket \Rightarrow \llbracket\varphi\rrbracket$ *is valid; If* $\mathsf{Valid}(\llbracket\Gamma\rrbracket \Rightarrow \llbracket\varphi\rrbracket)$ *answers negatively, then it is not the case that* $\neg(\llbracket\Gamma\rrbracket \Rightarrow \llbracket\varphi\rrbracket)$ *is valid.*

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

Lemma 5.1 (Substitution).

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

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

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

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

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

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

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

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

$$\text{(R-ColCaseEmp)}\ \frac{}{\langle H; \mathbf{case}\ \varepsilon\ \mathbf{of}\ (\varepsilon \Rightarrow N_1 \mid x{::}xs \Rightarrow N_2) \rangle \longrightarrow \langle H; N_1 \rangle}$$

$$\text{(R-ColCaseCons)}\ \frac{}{\langle H; \mathbf{case}\ v :: vs\ \mathbf{of}\ (\varepsilon \Rightarrow N_1 \mid x{::}xs \Rightarrow N_2) \rangle \longrightarrow \langle H; N_2\{v/x, vs/xs\} \rangle}$$

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

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

Fig. 4. Operational Semantics (Excerpt)

Lemma 5.2 (Context Conversion).

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

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

Theorem 5.3 (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* $\Gamma \vdash K, \Gamma \vdash T :: K$ *and* $\Gamma \vdash T' :: K$.
(c) *If* $\Gamma \models \psi \Leftrightarrow \varphi$ *and* $\Gamma \vdash$ *then* $\Gamma \vdash \psi$ *and* $\Gamma \vdash \varphi$

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

This setup then allows us to show so-called functionality properties of kinding and equality (see [Caires and Toninho 2019]), 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 5.5) and type preservation (Theorem 5.6). Moreover, kinding crucially ensures that all types of refinement kind are such that the corresponding refinement is SMT-valid.

Theorem 5.5 (Unicity of Types and Kinds).
(1) *If* $\Gamma \vdash M : T$ *and* $\Gamma \vdash M : S$ *then* $\Gamma \models T \equiv S :: K$ *and* $\Gamma \vdash K \leq \mathsf{Type}$.
(2) *If* $\Gamma \vdash T :: K$ *and* $\Gamma \vdash T :: K'$ *then* $\Gamma \vdash K \leq K'$ *or* $\Gamma \vdash K' \leq K$.

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

Theorem 5.6 (Type Preservation). *Let* $\Gamma \vdash_S M : T$ *and* $\Gamma \vdash_s H$. *If* $\langle H; M\rangle \longrightarrow \langle H'; M'\rangle$ *then there exists* S' *such that* $S \subseteq S'$, $\Gamma \vdash_{S'} H'$ *and* $\Gamma \vdash_{S'} M' : T$.

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

Lemma 5.7 (Type Progress). *If* $\cdot \vdash T :: K$ *then either* T *is a type value or* $T \longrightarrow T'$, *for some* T'.

Theorem 5.8 (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* $\langle H; M\rangle \longrightarrow \langle H'; M'\rangle$.

6 ALGORITHMIC TYPE CHECKING AND IMPLEMENTATION

This section provides a general description of our practical design choices and OCaml 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. 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):

$$\frac{\Gamma \vdash \varphi \quad \mathsf{Valid}([\![\Gamma]\!] \Rightarrow [\![\varphi]\!])}{\Gamma \models \varphi}\ (\textsc{entails})$$

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 basic 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). In this representation, the "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 a function symbol appTyp to the theory that is used to encode type-level application, effectively implementing defunctionalization [Reynolds 1972]. For instance, if f is a variable of some higher-kinded type, such that some equality on f is present in refinement formulae, e.g. $\{x :: \mathcal{K} \mid f(x) = t\}$, an equation corresponding to $\mathsf{appTyp}(f, x) = [\![t]\!]$ will be added to the SMT proof context.

Refinements are encoded as logical formulae that make use of the theory of finite sets in order to represent reasoning about record label set membership and apartness. We add two auxiliary functional symbols to the theory: $\mathsf{isRec} : \mathsf{Types} \to \mathsf{Bool}$ and $\mathsf{lab} : \mathsf{Types} \to \mathsf{Set\ of\ String}$, 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 representation of the empty record or a cons-cell, such that the 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, $\mathsf{apart} : (\mathsf{Set\ of\ String}, \mathsf{Set\ of\ String}) \to \mathsf{Bool}$) 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.

$$\begin{array}{lcl}
[\![\emptyset]\!] & \triangleq & \mathsf{True} \\
[\![\Gamma, t : \{x :: \mathcal{K} \mid \phi(x)\}]\!] & \triangleq & [\![\Gamma]\!] \wedge t : [\![\mathcal{K}]\!] \wedge [\![\phi(t)]\!] \\
[\![\Gamma, t : \Pi s : K.K']\!] & \triangleq & [\![\Gamma]\!] \wedge t : \mathsf{Types} \\
[\![\mathcal{K}]\!] & \triangleq & \mathsf{Types} \\
[\![\{x :: \mathcal{K} \mid \phi(x)\}]\!] & \triangleq & [\![\mathcal{K}]\!]
\end{array}$$

To simplify the presentation, we overload the $[\![-]\!]$ notation on contexts, types and kinds. All basic kinds are translated to the representation type Types. At the level of contexts, type variables

of basic 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 represent type variables t of higher-kind we encode them as variables of representation type Types, such that occurrences of t in refinements are defunctionalized using the technique described above.

Outline of the Algorithm. The main function of our checker takes a term (i.e. an expression or a type), an expected type or kind (and other auxiliary parameters explained below) and either raises an exception if the term is ill-typed or ill-kinded, or returns the type or kind of the term. The returned value is a lower bound of the expected type or kind. The OCaml type signature of the function is:

```
val typecheck: termenv -> termenv -> term -> term -> term -> bool -> ihenv -> term
```

The parameters of (OCaml) type `termenv` respectively hold the typing and evaluation contexts. The evaluation context holds bindings between variables and corresponding value bindings, necessary to normalize (open) types during type and kind-checking. The parameters of OCaml type `term` are respectively the term to be typed, its expected type, and the expected kind of such type. The `typecheck` function is also used to (kind) check types, in which case the three terms hold the type to be kinded, its expected kind, and the (conventional) well-kinded classifier `KindOK` constant. The parameter of type `bool` is used to approximate whether the typing environment is known to be inconsistent (used to implement the kinding and typing rules for $\bot$), and the parameter of type `ihenv` holds the induction environment for recursive types.

The algorithm crucially relies on auxiliary functions to compute normal forms of types using an evaluation strategy that is confluent and normalizing, and relies on the SMT solver to decide conditional predicates and equality tests. In particular, unfolding of recursive types is only performed when the recursion argument is equal (up to refinements) to a constructor (see [CoqDevelopmentTeam 2004; Giménez 1998]), so that the condition imposed on the rules for recursive types (decreasing size on the argument on recursive calls) suffices to ensure termination.

We highlight our adopted solution for interleaving type-level computation with type checking of value terms. When considering a value term for type checking, the algorithm first considers the structure of the (weak-head normalized) expected type. It then focuses on the structure of the type if its head term takes the form of a conditional, an application, or a recursive type, and applies the appropriate typing rule before recursing on the structure of the value term. Recursive types are handled using the induction environment mentioned above, allowing typing of recursive calls to be discharged using the appropriate kind declared in the recursive definition, as specified in the corresponding kinding rule. We illustrate a run of our type checker (and interpreter) on the concrete syntax for Examples 3.1 and 4.1, implementing record field selection in types and values.

```
# type
let Proj:: Pi L:: Lab.Pi t:: { r::Rec | L inl labSet(r) }.Type =
  fun L::Lab ->
    letrec G :: Pi t :: { r :: Rec | L inl labSet(r) }.Type =
      fun t :: { r :: Rec | L inl labSet(r) } ->
        if (headlb(t) == L) then head(t) else (G (tail(t)))
    in G end
in letrec proj : All L::Lab. All t:: { r::Rec | L inl labSet(r) }.(t -> (Proj L t)) =
       fun  L::Lab -> fun t :: { r::Rec | L inl labSet(r) } -> fun r:t ->
         if (headlb(t) == L) then head(r) else (proj L (tail(t)) tail(r))
   in
       ( proj `a [|`b : int, `a:bool|] [`b=5, `a=false] )
   end
end;;
 - : bool = false
```

The type `(Proj L t)` defines the projection of the type associated with label `L` in (record type) `t`, and `(proj L t r)` defines the projection of value associated with label with label `L` in record `r` (of type `t`). Notice the declared kind of `Proj` and the declared type of `proj`.

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 (e.g., those that manipulate records) and any instance of subkinding or kind equality in the presence of refinements is discharged via the encoding into CVC4. Kind-checking (of types) only requires type-level computation to take place while handling refinements predicates in kinds: those are normalized prior to encoding.

Type Equality. 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 basic kinds against variables of refined kind are offloaded to the SMT solver, implementing extensionality for types of basic kind (e.g., deriving that $t \equiv \mathsf{Bool} \to \mathsf{Bool}$ if $t :: \{f{:}\mathsf{Fun} \mid \mathbf{dom}(f) = \mathsf{Bool} \wedge \mathbf{img}(f) = \mathsf{Bool}\}$). The type checking algorithm itself (which makes use of the type equality algorithm) is mostly straightforward, since the typing rules of our language are essentially those of an ML-like language.

In terms of our overall approach to type and kind-checking, it follows closely known type-checking algorithms for related systems and so is clearly sound. Completeness holds relative to the underlying SMT theories, as is generally the case in related works on SMT-based refinement [Vazou et al. 2013, 2014]. Our prototype implementation consists of around 5000 lines of OCaml code (not counting the generated lexer and parser code) which includes parsuning, kind-checking, type-checking and an interpreter for our system (using the operational semantics of Section 5). The implementation validates the examples found in the paper. The interaction with the SMT solver to discharge refinements produces some overheads, but that is mostly due to the unoptimized nature of our proof-of-concept implementation.

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 adaptation 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-Star [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 supporting type-level record computation with similar aims as ours, avoiding type dependency. In our case, we generalize 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 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 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 ℓ is mapped to $R_1.\ell \longrightarrow R_2.\ell$. 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 a suitable λ-calculus.

Weirich et al. [2013] study 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 is designed to manipulate explicit type and kind coercions as part of the core language itself, which have a non-trivial structure (as required by the various type features 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 work of Altenkirch and McBride takes a dual approach to ours: While we take the stance of not having a dependently typed language, their work starts from a dependent type theory with universes and so-called large eliminations and shows how one can encode generic programming (i.e., the ability to define functions by recursion on the structure of datatypes) by defining an appropriate universe of types and a coding function. Thus, their general framework is naturally more expressive than ours, but lacks the general purpose practical programming setting of ours.

The work of Calcagno et al. focuses on multi-staging in the presence of effects. Staged computation is a form of meta-programming where program fragments can be safely quoted and executed in different phases. This form of metaprogramming is fundamentally different from that formulated in our work, being targeted towards efficiency and optimizations based on *safe* partial evaluation.

The concept of a statically checked type-case construct was introduced by Abadi et al. [1991]; however, our refinement kind checking of dynamic type conditionals on types and kinds $\mathbf{if}\ \varphi\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$ and $\mathbf{if}\ T :: K\ \mathbf{as}\ t \Rightarrow e_1\ \mathbf{else}\ e_2$ greatly extends the precision of type and kind checking, and 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 real-world general purpose languages such as Java and C# (or significant fragments of those languages). By using the standard record-based encoding of objects (as done in the examples of Sections 1 and 2), several of the meta-programming patterns found in their works are representable using our framework of refinement kinds (e.g., generating constructors, changing field types, generating accessor and modifier methods). However, since those works target object-oriented languages, they support OO-specific features that are out of the scope of our work (e.g. inheritance, method visibility), which does not deal with object orientation concepts but rather with a minimal ML-style language in order to illustrate the core ideas and their general expressiveness.

We further highlight the recent work of Kazerounian et al. [2019], which addresses arbitrary type-level computation in Ruby libraries and injects appropriate *run-time checks* to ensure that library methods abide by their computed type. Their work allows for arbitrary Ruby functions to be called during type-level computation and is thus more expressive than our more strictly stratified framework. Their typing discipline also exploits singleton-like types, that can be used in the context of database column and table names, to assign precise type signatures to database query methods (i.e., type-level computations can interact with a database to find the schema of a table that is then used as part of a type). While we can define types of the form $F(T_1) \to T_2$, where the domain type is the result of a computation F on T_1, we are restricted to more limited type-level reasoning, whereas their work is closer to general dependent types. For instance, we can define:

$$\mathsf{ColRecPreds} \triangleq \lambda s :: \mathsf{Type}.(\mathbf{if}\ s :: \mathsf{Col}\ \mathbf{as}\ t \Rightarrow (\mathbf{if}\ \mathbf{colOf}(t) :: \mathsf{Rec}\ \mathbf{as}\ t \Rightarrow t\ \mathbf{else}\ \mathbf{1})\ \mathbf{else}\ \mathbf{1}) \to \mathsf{Bool}$$

where ColRecPreds is a type-level function that given a type s, provided s is a collection type of records of some record type, produces the type of predicates on that record type (i.e. a function from that record type to Bool) and otherwise returns the trivial predicate type (i.e. $\mathbf{1} \to \mathsf{Bool}$, where $\mathbf{1}$ is the unit type). We can use ColRecPreds to type a program akin to a generic record existence test in a table (i.e. a collection of records):

$$\begin{array}{l}
\mathsf{exists} : \forall C :: \mathsf{Col}.C \to \mathsf{ColRecPreds}(C) \\
\mathsf{exists} \triangleq \Lambda C :: \mathsf{Col}.\mathbf{if}\ \mathbf{colOf}(C) :: \mathsf{Rec}\ \mathbf{as}\ t \Rightarrow \boldsymbol{\mu} F{:}C \to \mathsf{ColRecPreds}(C). \\
\qquad\qquad \lambda c{:}C.\lambda x{:}t.\mathbf{case}\ c\ \mathbf{of}\ (\varepsilon \Rightarrow \mathit{false} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid r{::}rs \Rightarrow \mathbf{if}\ r = x\ \mathbf{then}\ \mathit{true}\ \mathbf{else}\ F(rs)(x)) \\
\qquad\qquad \mathbf{else}\ \lambda c{:}C.\lambda x{:}\mathbf{1}.\mathit{false}
\end{array}$$

The example above contrasts with [Kazerounian et al. 2019], where a related example is formulated such that the equivalent of our ColRecPreds type-level function actually queries a database for the appropriate table schema, whereas ColRecPreds can only inspect the "shape" of its type arguments to obtain the table schema (i.e. the types of records contained in the collection).

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# [Petricek 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 λ-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 6, apart from the peculiarities specific to the refinement logic, our implementation is not significantly more involved than standard algorithms for type-checking system F_ω 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.

Following the approach of Kazerounian et al. [2019], it would be interesting to study a version of our theory of refinement kinds that is applied to a real-world dynamically typed language by inserting run-time checks to ensure methods follow their specified 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.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and suggestions. This work is supported by NOVA LINCS (Ref. UID/CEC/04516/2019), Fundação para a Ciência e Tecnologia project CLAY (PTDC/EEICTP/4293/2014), and the NOVA LINCS & OutSystems FLEX-AGILE project.

REFERENCES

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

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

Kshitij Bansal, Clark Barrett, Andrew Reynolds, and Cesare Tinelli. 2018. Reasoning with Finite Sets and Cardinality Constraints in SMT. *Logical Methods in Computer Science* 14, 4 (2018). https://doi.org/10.23638/LMCS-14(4:12)2018

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

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

Luís Caires and Bernardo Toninho. 2019. Refinement Kinds: Type-safe Programming with Practical Type-level Computation (Extended Version). *CoRR* abs/1908.00441 (2019). http://arxiv.org/abs/1908.00441

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

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

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

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

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

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

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

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

Benedict R. Gaster and Mark P. Jones. 1996. *A Polymorphic Type System for Extensible Records and Variants.* Technical Report NOTTCS-TR-96-3. Functional Programming Research Group, Department of Computer Science, University of Nottingham.

Eduardo Giménez. 1998. Structural Recursive Definitions in Type Theory. In *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*. 397–408. https://doi.org/10.1007/BFb0055070

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

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. 1996. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (1996), 109–138. https://doi.org/10.1145/227699.227700

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

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

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

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

Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S. Foster, and David Van Horn. 2019. Type-level computations for Ruby libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. 966–979. https://doi.org/10.1145/3314221.3314630

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly typed heterogeneous collections. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*. 96–107. https://doi.org/10.1145/1017472.1017488

Daan Leijen and Erik Meijer. 1999. Domain specific embedded compilers. In *Proceedings of the Second Conference on Domain-Specific Languages (DSL '99), Austin, Texas, USA, October 3-5, 1999*. 109–122. https://doi.org/10.1145/331960.331977

J. Garrett Morris and James McKinna. 2019. Abstracting extensible data types: or, rows by any other name. *PACMPL* 3, POPL (2019), 12:1–12:28. https://dl.acm.org/citation.cfm?id=3290325

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

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

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

Benjamin C. Pierce. 2004. *Advanced Topics in Types and Programming Languages*. The MIT Press.

Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstic, Morgan Deters, and Clark Barrett. 2013. Quantifier Instantiation Techniques for Finite Model Finding in SMT. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*. 377–391. https://doi.org/10.1007/978-3-642-38574-2_26

John C. Reynolds. 1972. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2 (ACM '72)*. ACM, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

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

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

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

Christopher A. Stone and Robert Harper. 2000. Deciding Type Equivalence with Singleton Kinds. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*. 214–227. https://doi.org/10.1145/325694.325724

Christopher A. Stone and Robert Harper. 2006. Extensional equivalence and singleton types. *ACM Trans. Comput. Log.* 7, 4 (2006), 676–722. https://doi.org/10.1145/1183278.1183281

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

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

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

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

Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. 60–76. https://doi.org/10.1145/75277.75283

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

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

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