The Essence of Generalized Algebraic Data Types

The Essence of Generalized Algebraic Data Types

FILIP SIECZKOWSKI, Heriot-Watt University, UK
SERGEI STEPANENKO, Aarhus University, Denmark
JONATHAN STERLING, University of Cambridge, UK
LARS BIRKEDAL, Aarhus University, Denmark

This paper considers direct encodings of generalized algebraic data types (GADTs) in a minimal suitable lambda-calculus. To this end, we develop an extension of System F_ω with recursive types and internalized type equalities with injective constant type constructors. We show how GADTs and associated pattern-matching constructs can be directly expressed in the calculus, thus showing that it may be treated as a highly idealized modern functional programming language. We prove that the internalized type equalities in conjunction with injectivity rules increase the expressive power of the calculus by establishing a non-macro-expressibility result in F_ω, and prove the system type-sound via a syntactic argument. Finally, we build two relational models of our calculus: a simple, unary model that illustrates a novel, two-stage interpretation technique, necessary to account for the equational constraints; and a more sophisticated, binary model that relaxes the construction to allow, for the first time, formal reasoning about data-abstraction in a calculus equipped with GADTs.

CCS Concepts: • **Theory of computation → Type theory**; *Program reasoning*; • **Software and its engineering → Data types and structures**.

Additional Key Words and Phrases: Generalized Algebraic Data Types, Logical Relations

ACM Reference Format:
Filip Sieczkowski, Sergei Stepanenko, Jonathan Sterling, and Lars Birkedal. 2024. The Essence of Generalized Algebraic Data Types. *Proc. ACM Program. Lang.* 8, POPL, Article 24 (January 2024), 29 pages. https://doi.org/10.1145/3632866

1 INTRODUCTION

Since their introduction twenty years ago, *generalized algebraic data types* [Cheney and Hinze 2003; Sheard and Pasalic 2008; Xi et al. 2003], commonly referred to as GADTs, have become a staple of modern functional programming languages. First introduced as an extension of the Glasgow Haskell Compiler, they were since adopted by OCaml, Scala and many other programming languages, due to their ability to establish more precise specifications for algebraic type constructors. These implementations, and the attendant difficulties, have also resulted in a rich literature that tackles the problems of type inference in languages with ML-style polymorphism and GADTs [Jones et al. 2006; Pottier and Régis-Gianas 2006]. However, one feature that remains understudied is the semantics of GADTs and the nature and expressivity of these types. Since the seminal paper of Xi et al. [2003], GADT calculi have usually been equipped with primitive notions of algebraic types and their constructors. While this is desirable for studying the problems of inference, it tends to obscure the essence of the structure of the types, and where their expressive power stems from.

Authors' addresses: Filip Sieczkowski, Heriot-Watt University, Edinburgh, UK, f.sieczkowski@hw.ac.uk; Sergei Stepanenko, Aarhus University, Aarhus, Denmark, sergei.stepanenko@cs.au.dk; Jonathan Sterling, University of Cambridge, Department of Computer Science and Technology, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK, js2878@cl.cam.ac.uk; Lars Birkedal, Aarhus University, Aarhus, Denmark, birkedal@cs.au.dk.

ACM 2475-1421/2024/1-ART24
https://doi.org/10.1145/3632866

In this paper, we intend to shed more light on these aspects of GADTs by introducing what we believe to be the first calculus that does not rely on the notion of a data type constructor to express GADTs. We then proceed by studying soundness and expressive power of the calculus, as well as provide some interesting and difficult open problems.

Beyond algebraic data types. Algebraic data types are one of the fundamental features of modern functional programming languages, arguably as important to programming practice as higher-order functions themselves. One of the canonical examples is that of a binary tree with nodes labeled by elements of some type, which can be defined as follows, using Haskell syntax:[1]

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

These types are often modeled in λ-calculi with a combination of simple types — disjoint sums for distinct constructors, products for their arguments — and iso-recursive types [Pierce 2002, Chapter 21]. For instance, the type of trees labeled with elements of type α would correspond to the following type:

$$\mathsf{Tree}\ \alpha \triangleq \mu\beta.\ \mathsf{unit} + (\beta \times \alpha \times \beta)$$

Above, the variable β bound by μ stands for the recursive occurrences of the Tree type constructor.

There are two important points to note about the type of trees. First, it really is a *type constructor*, rather than a type: we only get a type after the definition of Tree is applied to an argument, the type of labels. Second, the definition is *uniform* in the argument: all the recursive arguments are indexed with the same type of elements as the complete tree. Over the last quarter of a century, numerous generalizations have been proposed to extend the expressive power of algebraic data types and allow definitions to more precisely describe the shape of the data.

The most natural way to relax the constraints on the shapes of algebraic data types is to remove the uniformity condition; types obtained in this way are often dubbed *nested* algebraic data types. A simple example due to Bird and Meertens [1998] is the type of perfectly balanced trees, defined below:

```
data PTree a = PLeaf | PNode a (PTree (a * a))
```

This definition models a perfectly balanced tree as a list of levels; at each depth, we require twice as many elements as at the previous one, by virtue of the fact that the recursive call to the type constructor is taken at type `a * a`. Since the type is no longer uniform, simple recursive types no longer suffice to encode it in a λ-calculus based system. We may nonetheless encode it in a version of polymorphic λ-calculus with recursive type constructors, which forms the basis of most modern functional programming languages. Under this extension, type-level recursion can be used to define not only ground types but also proper type constructors. This allows us to encode the type of perfectly-balanced trees as follows:

$$\mathsf{PTree} \triangleq \mu\beta :: \mathsf{T} \Rightarrow \mathsf{T}.\ \lambda\alpha :: \mathsf{T}.\ \mathsf{unit} + (\alpha \times (\beta\ (\alpha \times \alpha))).$$

Above we define PTree as a recursively-defined type constructor by specifying that the recursive variable β shall map types to types; then we introduce the type α of labels using a type-level lambda-abstraction. After that, the encoding follows the pattern described above.

While relaxing the uniformity requirement for algebraic data types is likely the most natural generalization, other possibilities abound. Readers familiar with inductive type families, as introduced in the Calculus of Inductive Constructions [Paulin-Mohring 1993] or found in the implementations of Coq or Agda, may notice that — even with the relaxed condition — the arguments still satisfy

[1] In this paper, we use Haskell syntax for presenting examples. However, our language differs from Haskell, as we use a different evaluation strategy. In this paper, all examples use call-by-value evaluation strategy, *i.e.* before performing any reductions, arguments should be reduced to values.

the conditions required of *parameters*, rather than the more general *indices*; the difference is that parameters may be constrained non-uniformly in recursive calls, whereas indices may additionally be non-uniform in the return types of the data constructors.[2] This additional non-uniformity is precisely what ***generalized algebraic data types*** (GADTs) allow for, albeit restricted to the non-dependently typed setting. This may be demonstrated by the following intrinsically well-typed representation of lambda terms:

```
data Tm a where
  Lift :: a -> Tm a
  Lam  :: (a -> Tm b) -> Tm (a -> b)
  App  :: Tm (a -> b) -> Tm a -> Tm b
```

There are several things to observe about the representation above. First of all, we re-use the meta-level types as the indices that indicate the type of the given lambda term. Secondly, although the `Lift` constructor is uniformly inhabited at any type for which we have an element, `Lam` is only ever inhabited at indices that are, syntactically, of the arrow type form, and the indices of the arguments and result of the `App` constructor express nontrivial constraints. This has far-reaching consequences for the behavior of pattern-matching: for instance, if we have an object of type `Tm (a * b)`, we can safely disregard the `Lam` constructor. This ability to enforce constraints on the *return type* of the constructor enables specifications of data types to express many invariants, lending them power that before was only found in dependent type theories. It is worth noting that this property of pattern matching in languages with GADTs expresses the *discriminability* of types whose head constructors are different; discriminability laws of this kind are *not* present in standard dependent type theories, which means that the naïve encoding of GADTs as inductive families in (*e.g.*) Coq does not allow the same programs to be written.[3]

With this additional expressive power comes new problems. Since classic algebraic data types can be directly translated to rather conservative extensions of System F, they are very well studied — not just in terms of type safety, but also stronger properties, such as relational parametricity. In fact, it is the parametricity results that allow us to talk about "free theorems" about simple algebraic data types, such as lists or trees. Since nested, or non-uniform data types can be similarly encoded in well-behaved extensions of System F, many of these results also extend to them. However, this line of results comes to a halt when GADTs are concerned: while many calculi have been developed, the main lines of work have focused on the difficult questions of typechecking in the presence of GADTs [Dunfield and Krishnaswami 2019] or the equally difficult matter of functorial initial algebra semantics for GADTs [Johann and Ghani 2008], and very few go beyond type safety results. This paper is an attempt to cross that boundary.

However, in order to investigate existence of relational models of GADTs, we need a calculus that is expressive enough to directly encode GADTs and the associated programming patterns, while at the same time remaining as simple as possible. To this end, in Section 2 we develop System $\mathrm{F}^{=i}_{\omega\mu}$, which extends System F_ω with recursive types and *internalized equalities*. System $\mathrm{F}^{=i}_{\omega\mu}$ also contains certain discriminability and type-constructor injectivity rules, which are crucial for GADT-style reasoning. In contrast to most of the GADT calculi, our approach allows us to leave out the notions of constructors of algebraic data types and their associated type constructors. We believe that this allows us to better capture the *essence* of how GADTs behave and remove language constructs

[2] In the context of dependent type theory, non-uniformity together with propositional equality suffice to encode indices. However, in the absence of equality types or constraints they behave parametrically.

[3] To account for the needed discriminability in the environment of inductive families in systems like Coq and Agda, one must value the index of `Tm` in a custom inductively defined universe rather than in the ambient universe of existing types.

that are useful in practice, but superfluous in terms of the semantic study of the calculus and its properties.

After having defined the calculus, we proceed, in Section 3, to study its relationship to its well-studied restriction, System F_ω. We establish that even in the absence of recursive types and with certain restrictions on injectivity of type constructors, our calculus is not expressible in F_ω, due to the presence of the typing rules that we use to model GADTs. Thus, any potential translation to the better-studied system must be a full-program compilation. This result cuts against the folklore understanding of GADTs as "existentials and type equalities": we show that the encodings depend crucially on how one can reason about type equality.

Since System $F^{=i}_{\omega\mu}$ is, to the best of our knowledge, not macro-expressible in known systems, in Section 4 we establish its type soundness through a non-trivial syntactic argument, involving a normalization process to cater for equality reasoning, and proceed to investigate the existence of relational models of the calculus in Section 5. Since the proof of non-expressibility strongly suggests that the usual approach of interpreting types as predicates or relations on values would not allow us to enforce the necessary injectivity rules, we develop a novel unary model construction that splits the relational interpretation process in two phases. First, we use a normalization-by-evaluation (NbE) interpretation to account for type-level computation (including injectivity of types); then, the closed, ground subset of the NbE interpretations of types is realized as predicates on values. We observe that while this construction is sufficient to construct a model, it does not allow to reason about syntactically ill-typed programs or data abstraction. To remedy this, we propose a simple extension of the technique, and use it to build a binary relational model that allows us to transport data abstraction results into the realm of GADTs.

With the exception of the expressibility result in Section 3, all the constructions presented in this paper are formalized in Coq. We believe that some of those developments, in particular the NbE implementation which bakes functoriality into the construction, thus alleviating some of the post-hoc reasoning necessary, are of independent interest. We discuss the techniques used in the formalization in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8.

Contributions and Outline. In summary, we make the following contributions:

- We present (in Section 2) a novel calculus $F^{=i}_{\omega\mu}$ extending System F_ω with (higher-kinded) iso-recursive types and internalized type-equalities satisfying enough injectivity and discriminability laws to model GADTs.
- We prove (in Section 3) that System $F^{=i}_{\omega\mu}$ is not macro-expressible in System F_ω, justifying our claim that GADTs require additional expressive power.
- We prove (in Section 4) type soundness of System $F^{=i}_{\omega\mu}$, with a methodologically novel use of normalization-by-evaluation within a syntactic type soundness proof.
- We present (in Section 5) two semantic models of System $F^{=i}_{\omega\mu}$, introducing a novel two-stage construction technique that allows us to enforce injectivity of type constructors in a relational model, and reason about syntactically ill-typed programs and data abstraction.
- We have formalized our constructions in the Coq Proof Assistant (Section 6).

2 POLYMORPHIC λ-CALCULUS WITH INTERNALIZED TYPE EQUALITIES

In this section we present our core calculus, dubbed System $F^{=i}_{\omega\mu}$, as an extension of System F_ω – the λ-calculus with impredicative polymorphism and type constructors, which is the theoretical foundation of modern functional programming languages. The core feature of our calculus is the *reification* of F_ω type equality, which is a judgment for which there is no *internal* syntax within the F_ω language.

$$
\begin{array}{lll}
\text{kinds} & \kappa & ::= \mathsf{T} \mid \kappa \Rightarrow \kappa \\
\text{constructors} & c & ::= \forall_\kappa \mid \exists_\kappa \mid \mu_\kappa \mid \rightarrow \mid \times \mid + \mid \mathsf{unit} \mid \mathsf{void} \\
\text{constraints} & \chi & ::= \sigma \equiv_\kappa \tau \\
\text{types} & \sigma, \tau & ::= c \mid \alpha \mid \lambda \alpha :: \kappa.\ \tau \mid \sigma\ \tau \mid \chi \rightarrow \tau \mid \chi \times \tau
\end{array}
$$

Fig. 1. The syntax of kinds, types and constraints.

$$
\forall_\kappa, \exists_\kappa :: (\kappa \Rightarrow \mathsf{T}) \Rightarrow \mathsf{T} \qquad \mu_\kappa :: (\kappa \Rightarrow \kappa) \Rightarrow \kappa \ (\mu) \qquad \rightarrow, \times, + :: \mathsf{T} \Rightarrow \mathsf{T} \Rightarrow \mathsf{T} \qquad \mathsf{unit}, \mathsf{void} :: \mathsf{T}
$$

$$
\frac{c :: \kappa}{\Delta \vdash c :: \kappa} \qquad \frac{}{\Delta \vdash \alpha :: \Delta(\alpha)} \qquad \frac{\Delta, \alpha :: \kappa \vdash \tau :: \kappa'}{\Delta \vdash \lambda \alpha :: \kappa.\ \tau :: \kappa \Rightarrow \kappa'} \qquad \frac{\Delta \vdash \sigma :: \kappa_1 \Rightarrow \kappa_2 \qquad \Delta \vdash \tau :: \kappa_1}{\Delta \vdash \sigma\ \tau :: \kappa_2}
$$

$$
\frac{\Delta \vdash \chi\ \mathsf{constr} \qquad \Delta \vdash \tau :: \mathsf{T}}{\Delta \vdash \chi \rightarrow \tau :: \mathsf{T}} \qquad \frac{\Delta \vdash \chi\ \mathsf{constr} \qquad \Delta \vdash \tau :: \mathsf{T}}{\Delta \vdash \chi \times \tau :: \mathsf{T}} \qquad \frac{(\Delta \vdash \tau_i :: \kappa)_{i \in \{1,2\}}}{\Delta \vdash \tau_1 \equiv_\kappa \tau_2\ \mathsf{constr}}
$$

Fig. 2. Well-kinded constructors and types, and well-formed equality constraints. The recursive type constructor (marked with a red (μ)) is not considered in a restricted sub-system, $\mathsf{F}^{=i}_{\omega}$.

Syntax of kinds and types. The structure of kinds, types and equality constraints is presented in Figure 1. The calculus has the standard kind structure: T stands for the kind of proper types, while $\kappa_1 \Rightarrow \kappa_2$ denotes the kind of *type constructors* that produce a type of kind κ_2 given a type of kind κ_1. The types are formed from variables, type abstraction and application, type constants — which include products, disjoint sums, arrows, as well as universal and existential quantifiers, and recursive type constructor — and two types that interact with *type equality constraints*. The first of these is an arrow type predicated on a constraint: its intended meaning is to qualify suspended computations that can only be run if the constraint is valid; the second is an analogue of a product type that provides a value together with a witness of validity of a constraints. The constraints, in turn, are kinded equalities between types. Figure 2 provides the kinding rules for types and well-formedness rules for constraints; the two judgments are mutually inductively defined.

By convention, we write binary type constants (*i.e.* arrows, products and sums) infix, and we write $\forall \alpha :: \kappa.\ \tau$ to mean $\forall_\kappa\ (\lambda \alpha :: \kappa.\ \tau)$ — and likewise for existential and recursive types. Note that the arrow and product syntax is somewhat ambiguous; however, it is always clear from context whether the type is a standard arrow/product, or the constrained variant.

This syntax allows us to express most of the standard types encountered in functional programming: for instance, we can express a uniform algebraic data type $\mathsf{List} \triangleq \lambda \alpha :: \mathsf{T}.\ \mu \beta :: \mathsf{T}.\ \mathsf{unit} + \alpha \times \beta$, with the unit type standing in for the nil constructor, and the pair for the cons. However, the fact that we can form recursive data types at higher kinds allows us to express nested data types, and the constrained types allow us to express GADTs. For example, consider the following type:

```
data Foo :: * -> * where
  C1 :: Bool -> Foo Bool
  C2 :: Foo a
```

We may encode the above by defining $\mathsf{Foo} \triangleq \lambda \alpha :: \mathsf{T}.\ (\mathsf{Bool} \times (\alpha \equiv_{\mathsf{T}} \mathsf{Bool})) + \mathsf{unit}$. We discuss expressivity and examples in more detail in Section 2.1.

$$\frac{c_1 \neq c_2 \qquad (\Delta \vdash c_i\ \overline{\tau}_i :: \kappa)_{i\in\{1,2\}}}{\Delta \Vdash c_1\ \overline{\tau}_1\ \#_\kappa\ c_2\ \overline{\tau}_2} \qquad \frac{\Delta \vdash c\ \overline{\tau} :: \mathsf{T} \qquad \Delta \vdash \chi \to \sigma :: \mathsf{T}}{\Delta \Vdash c\ \overline{\tau}\ \#_{\mathsf{T}}\ \chi \to \sigma}$$

$$\frac{\Delta \vdash c\ \overline{\tau} :: \mathsf{T} \qquad \Delta \vdash \chi \times \sigma :: \mathsf{T}}{\Delta \Vdash c\ \overline{\tau}\ \#_{\mathsf{T}}\ \chi \times \sigma} \qquad \frac{\Delta \vdash \chi_1 \to \tau_1 :: \mathsf{T} \qquad \Delta \vdash \chi_2 \times \tau_2 :: \mathsf{T}}{\Delta \Vdash \chi_1 \to \tau_1\ \#_{\mathsf{T}}\ \chi_2 \times \tau_2} \qquad \frac{\Delta \Vdash \tau_2\ \#_\kappa\ \tau_1}{\Delta \Vdash \tau_1\ \#_\kappa\ \tau_2}$$

$$\frac{\chi \in \Phi}{\Delta \mid \Phi \Vdash \chi} \qquad \frac{\Delta, \alpha :: \kappa_a \vdash \sigma :: \kappa_r \qquad \Delta \vdash \tau :: \kappa_a}{\Delta \mid \Phi \Vdash (\lambda\alpha :: \kappa_a.\ \sigma)\ \tau \equiv_{\kappa_r} \sigma[\tau/\alpha]} \qquad \frac{\Delta \vdash \tau :: \kappa_a \Rightarrow \kappa_r \qquad \alpha \notin \Delta}{\Delta \mid \Phi \Vdash \lambda\alpha :: \kappa_a.\ \tau\ \alpha \equiv_{\kappa_a \Rightarrow \kappa_r} \tau}$$

$$\frac{\Delta \mid \Phi \Vdash \chi \to \sigma \equiv_{\mathsf{T}} \xi \to \tau}{\Delta \mid \Phi \Vdash \sigma \equiv_{\mathsf{T}} \tau} \qquad \frac{\Delta \mid \Phi \Vdash \chi \times \sigma \equiv_{\mathsf{T}} \xi \times \tau}{\Delta \mid \Phi \Vdash \sigma \equiv_{\mathsf{T}} \tau} \qquad \frac{c :: (\kappa_i \Rightarrow)_i\ \kappa \qquad \Delta \mid \Phi \Vdash c\ (\sigma_i)_i \equiv_\kappa c\ (\tau_i)_i}{\left(\Delta \mid \Phi \Vdash \sigma_i \equiv_{\kappa_i} \tau_i\right)_i}$$

$$\frac{\Delta \mid \Phi \Vdash \chi \times \sigma \equiv_{\mathsf{T}} \xi \times \tau \qquad \Delta \mid \Phi \Vdash \chi}{\Delta \mid \Phi \Vdash \xi} \qquad \frac{\Delta \mid \Phi \Vdash \chi \to \sigma \equiv_{\mathsf{T}} \xi \to \tau \qquad \Delta \mid \Phi \Vdash \chi}{\Delta \mid \Phi \Vdash \xi}$$

Fig. 3. Discriminability of types and provability of equality constraints; for brevity, we omit the rules that make constructor equality a congruent equivalence relation.

Discriminability and provability of constraints. With the syntax of kinds and types defined, we can turn to the notion of provability of constraints, which is defined is Figure 3. The judgment $\Delta \mid \Phi \Vdash \chi$ denotes that the constraint χ — well-formed in the type-variable context Δ — is provable from the assumptions in Φ, which are also constraints well-formed in Δ. In addition to a rule that allows for the use of assumptions, the rules follow the standard type-equivalence pattern from System F_ω: the type equivalence is a congruence with respect to the type formers, and it is closed under β and η rules. However, in addition to these we also have *injectivity* rules for type constants (including the new variants of arrow and product), marked in purple in the definition. These ensure that if two types formed by the application of the same constructor to the same number of (well-kinded) arguments are equal, then so are, pairwise, their arguments. The consequences of including these rules will be discussed in detail in the following section; the rationale for their inclusion is that this is precisely the reasoning that GADTs require, particularly when performing pattern-matching. For now, let us observe that it is the injectivity rules that allow us to discern, from an assumption $\mathsf{List}\ \alpha \equiv_{\mathsf{T}} \mathsf{List}\ \beta$, that $\alpha \equiv_{\mathsf{T}} \beta$ holds. Note, however, that any essential use of an injectivity rule requires the existence of an assumed equality of two compound types in the context, from which we can derive equalities of components. That can serve as an intuitive explanation for why they are not necessary in System F_ω.

In addition to the provability judgment, we also define a simpler notion of *discriminability* of a constraint. This judgment holds when two types can never be made equal — in principle, whenever the two types are created by distinct type constants. Note that, for the purpose of simplicity, we only consider the top-most constructor, and that the constrained arrow and product types are discriminable from each other, as well as from the types constructed by appropriate application of type constants. The corresponding rules are marked in blue in Fig. 3.

Syntax of expressions and values. With the notions of types, constraints and provability defined, we now turn to the term-level of our calculus. The syntax is defined in Figure 4; for simplicity of presentation we use fine-grain call-by-value semantics [Levy et al. 2003], which significantly

$$
\begin{array}{lrcl}
\text{values} & v & ::= & x \mid \langle\rangle \mid \lambda x.\, e \mid \langle v_1, v_2\rangle \mid \mathsf{inj}_1\, v \mid \mathsf{inj}_2\, v \mid \Lambda.\, e \mid \mathsf{pack}\, v \mid \mathsf{roll}\, v \mid \lambda\bullet.\, e \mid \langle \bullet, v\rangle \\
\text{expressions} & e & ::= & v \mid \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \mid v_1\, v_2 \mid \mathsf{proj}_1\, v \mid \mathsf{proj}_2\, v \mid \mathsf{case}\, v\, [x.\, e_1 \mid y.\, e_2] \mid \\
 & & & \mathsf{abort}\, v \mid v * \mid \mathsf{let}\, (*, x) = v\, \mathsf{in}\, e \mid \mathsf{unroll}\, v \mid \mathsf{abort}\, \bullet \mid v\, \bullet \mid \mathsf{let}\, (\bullet, x) = v\, \mathsf{in}\, e \\
\text{eval. contexts} & E & ::= & \square \mid \mathsf{let}\, x = E\, \mathsf{in}\, e
\end{array}
$$

Fig. 4. The syntax of expressions and values.

reduces the metatheoretical overhead. In larger examples we typically forego these restrictions, assuming a left-to-right, call-by-value reading of the terms.

As a consequence of being an extension of System F_ω, most of the syntax of our calculus is standard, although note that we use $\mathsf{let}\, (*, x) = v\, \mathsf{in}\, e$ as an unpacking operation, "pattern-matching" on the packed value. Aside from that choice, we have two new values and three new expressions. The values are, respectively, the introduction forms for the constrained arrow and product types: a computation suspended pending a proof of constraint, and a pair consisting of a value and such a proof. However, since we do not introduce proof terms for our notion of provability, we use $\bullet$ in places where we might imagine a proof or an assumption. This is in direct correspondence to the treatment of universal and existential types, where witnesses are also elided.

In the same spirit, the "proof application", $v\ \bullet$, and "proof unpacking", $\mathsf{let}\, (\bullet, x) = v\, \mathsf{in}\, e$, serve as elimination forms for the two constructs. This leaves us with the final expression, $\mathsf{abort}\, \bullet$, whose behavior matches the eliminator for an empty type — *i.e.* the program should be considered erroneous whenever the expression is in an evaluation position. Its intended role will become clearer when we consider the type system, presented in Figure 5.

The typing judgment assigns a type (of kind T) to an expression or a value in three contexts: Δ, which assigns kinds to type variables, Φ, which lists the constraints we assume to hold, and Γ, which assigns types (of kind T in Δ) to term-level variables. Most rules follow System F_ω, with the additional context passed around; we discuss the other rules below. First, the form of the F_ω rule that allows one to replace the type of a term with any type equivalent to it is slightly different in our calculus, as we use the provability relation for the appropriate constraint, rather than the external type equivalence judgment. This means that the proof may depend on equalities, some of which can be introduced by the derivation — a fact that is crucial to encode pattern matching over GADTs. Second, the rules for values and expressions associated with the constrained types match the intuition of introduction and elimination rule. When type-checking the body of $\lambda\bullet.\, e$ we may assume that any well-formed constraint χ holds, and assign the type $\chi \longrightarrow \tau$ to the expression — but in order to use a value of this type, through $v\ \bullet$, we need to show that the required constraint does indeed hold. The rules for constrained pairs are analogous. Finally, the rule for abort $\bullet$ requires that a discriminable equality can be proved using our assumptions: in other words, that the assumptions are inconsistent. This rule is crucial, since it allows us to directly encode pattern matching for GADTs in a calculus where all case expressions are, by necessity, exhaustive: in a way, it corresponds to OCaml's "dot pattern", which forces the typechecker to try to ensure that a given case in a pattern-match is impossible due to the equalities that would have to hold. The final extension with respect to the standard System F_ω is the presence of the recursive types; the corresponding rules are marked with a red μ. Due to their higher-kinded nature, the recursive types need to be appropriately applied in order for the roll expression to typecheck; the same arguments are passed to the recursive type's unfolding. As usual, the typing rule for the unroll expression is simply an inverse of the one for roll.

In order to see these rules in action, consider the type constructor Foo, defined as $\mathsf{Foo} \triangleq \lambda\alpha :: \mathsf{T}.\, (\mathsf{Bool} \times (\alpha \equiv_{\mathsf{T}} \mathsf{Bool})) + \mathsf{unit}$, and the program: $e \triangleq \mathsf{case}\, x\, [y.\, \mathsf{let}\, (\bullet, y) = y\, \mathsf{in}\, \mathsf{abort}\, \bullet \mid \_.\, \langle\rangle]$.

$$\frac{x : \tau \in \Gamma}{\Delta \mid \Phi \mid \Gamma \vdash x : \tau} \qquad \frac{}{\Delta \mid \Phi \mid \Gamma \vdash \langle\rangle : \mathsf{unit}} \qquad \frac{\Delta \vdash \sigma :: \mathsf{T} \qquad \Delta \mid \Phi \mid \Gamma, x : \sigma \vdash e : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \lambda x.\, e : \sigma \to \tau}$$

$$\frac{(\Delta \mid \Phi \mid \Gamma \vdash v_i : \tau_i)_{i \in \{1,2\}}}{\Delta \mid \Phi \mid \Gamma \vdash \langle v_1, v_2 \rangle : \tau_1 \times \tau_2} \qquad \frac{\Delta \vdash \tau_2 :: \mathsf{T} \qquad \Delta \mid \Phi \mid \Gamma \vdash v : \tau_1}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{inj}_1\, v : \tau_1 + \tau_2}$$

$$\frac{\Delta \vdash \tau_1 :: \mathsf{T} \qquad \Delta \mid \Phi \mid \Gamma \vdash v : \tau_2}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{inj}_2\, v : \tau_1 + \tau_2} \qquad \frac{\Delta, \alpha :: \kappa \mid \Phi \mid \Gamma \vdash e : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \Lambda.\, e : \forall \alpha :: \kappa.\, \tau}$$

$$\frac{\Delta \vdash \sigma :: \kappa \qquad \Delta \mid \Phi \mid \Gamma \vdash v : \tau[\sigma/\alpha]}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{pack}\, v : \exists \alpha :: \kappa.\, \tau} \qquad \frac{\kappa = (\kappa_i \Rightarrow)_i\, \mathsf{T} \qquad (\Delta \vdash \sigma_i :: \kappa_i)_i \qquad \Delta \mid \Phi \mid \Gamma \vdash v : \tau[\mu\alpha :: \kappa.\, \tau/\alpha]\ (\sigma_i)_i}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{roll}\, v : (\mu\alpha :: \kappa.\, \tau)\ (\sigma_i)_i}\ (\mu)$$

$$\frac{\Delta \vdash \chi\ \mathsf{constr} \qquad \Delta \mid \Phi, \chi \mid \Gamma \vdash e : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \lambda \bullet.\, e : \chi \to \tau} \qquad \frac{\Delta \mid \Phi \Vdash \chi \qquad \Delta \mid \Phi \mid \Gamma \vdash v : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \langle \bullet, v \rangle : \chi \times \tau}$$

$$\frac{\Delta \mid \Phi \Vdash \tau_1 \equiv_{\mathsf{T}} \tau_2 \qquad \Delta \mid \Phi \mid \Gamma \vdash v : \tau_1}{\Delta \mid \Phi \mid \Gamma \vdash v : \tau_2}$$

$$\frac{\Delta \mid \Phi \mid \Gamma \vdash e_1 : \sigma \qquad \Delta \mid \Phi \mid \Gamma, x : \sigma \vdash e_2 : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau} \qquad \frac{\Delta \mid \Phi \mid \Gamma \vdash v_1 : \sigma \to \tau \qquad \Delta \mid \Phi \mid \Gamma \vdash v_2 : \sigma}{\Delta \mid \Phi \mid \Gamma \vdash v_1\, v_2 : \tau}$$

$$\frac{\Delta \mid \Phi \mid \Gamma \vdash v : \tau_1 \times \tau_2}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{proj}_1\, v : \tau_1} \qquad \frac{\Delta \mid \Phi \mid \Gamma \vdash v : \tau_1 \times \tau_2}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{proj}_2\, v : \tau_2} \qquad \frac{\Delta \vdash \tau :: \mathsf{T} \qquad \Delta \mid \Phi \mid \Gamma \vdash v : \mathsf{void}}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{abort}\, v : \tau}$$

$$\frac{\Delta \mid \Phi \mid \Gamma \vdash v : \tau_1 + \tau_2 \qquad (\Delta \mid \Phi \mid \Gamma, x_i : \tau_i \vdash e_i : \tau)_{i \in \{1,2\}}}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{case}\, v\, [x_1.\, e_1 \mid x_2.\, e_2] : \tau} \qquad \frac{\Delta \mid \Phi \mid \Gamma \vdash v : \forall \alpha :: \kappa.\, \tau \qquad \Delta \vdash \sigma :: \kappa}{\Delta \mid \Phi \mid \Gamma \vdash v * : \tau[\sigma/\alpha]}$$

$$\frac{\Delta \mid \Phi \mid \Gamma \vdash v : \exists \alpha :: \kappa.\, \sigma \qquad \Delta \vdash \tau :: \kappa \qquad \Delta, \alpha :: \kappa \mid \Phi \mid \Gamma, x : \sigma \vdash e : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{let}\, (*, x) = v\, \mathsf{in}\, e : \tau} \qquad \frac{\kappa = (\kappa_i \Rightarrow)_i\, \mathsf{T} \qquad (\Delta \vdash \sigma_i :: \kappa_i)_i \qquad \Delta \mid \Phi \mid \Gamma \vdash v : (\mu\alpha :: \kappa.\, \tau)\ (\sigma_i)_i}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{unroll}\, v : \tau[\mu\alpha :: \kappa.\, \tau/\alpha]\ (\sigma_i)_i}\ (\mu)$$

$$\frac{\Delta \mid \Phi \Vdash \sigma_1 \equiv_\kappa \sigma_2 \qquad \Delta \Vdash \sigma_1 \#_\kappa \sigma_2 \qquad \Delta \vdash \tau :: \mathsf{T}}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{abort}\, \bullet : \tau} \qquad \frac{\Delta \mid \Phi \mid \Gamma \vdash v : \chi \to \tau \qquad \Delta \mid \Phi \Vdash \chi}{\Delta \mid \Phi \mid \Gamma \vdash v\, \bullet : \tau}$$

$$\frac{\Delta \mid \Phi \mid \Gamma \vdash v : \chi \times \sigma \qquad \Delta \mid \Phi, \chi \mid \Gamma, x : \sigma \vdash e : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{let}\, (\bullet, x) = v\, \mathsf{in}\, e : \tau} \qquad \frac{\Delta \mid \Phi \Vdash \tau_1 \equiv_{\mathsf{T}} \tau_2 \qquad \Delta \mid \Phi \mid \Gamma \vdash e : \tau_1}{\Delta \mid \Phi \mid \Gamma \vdash e : \tau_2}$$

Fig. 5. Type system of $\mathsf{F}^{=i}_{\omega\mu}$. The roll and unroll rules (marked with a red (μ)) are not considered in $\mathsf{F}^{=i}_{\omega}$. The rules are grouped by separating values and expressions, and, within these two groups, by connectives.

$$\text{let } x = v \text{ in } e \mapsto e[v/x]$$
$$(\lambda x.\ e)\ v \mapsto e[v/x]$$
$$\left(\text{proj}_i\ \langle v_1, v_2\rangle \mapsto v_i\right)_{i\in\{1,2\}}$$
$$\left(\text{case inj}_i\ v\ [x.\ e_1 \mid x.\ e_2] \mapsto e_i[v/x]\right)_{i\in\{1,2\}}$$

$$(\Lambda.\ e)\ * \mapsto e$$
$$\text{let } (*, x) = \text{pack}\, v \text{ in } e \mapsto e[v/x]$$
$$\text{unroll}\,(\text{roll}\, v) \mapsto v \ \ (\mu)$$
$$(\lambda \bullet.\ e)\ \bullet \mapsto e$$
$$\text{let } (\bullet, x) = \langle \bullet, v\rangle \text{ in } e \mapsto e[v/x]$$

$$\frac{e_1 \mapsto e_2}{E[e_1] \to E[e_2]}$$

Fig. 6. Operational semantics: contraction and reduction relations; rules marked with a red (μ) are not considered in $\mathsf{F}^{=i}_{\omega}$

Assuming x : Foo unit — and that Bool and unit are discriminable — we can check that e : unit. This is because in the first branch of the case expression we unpack the constrained pair, introducing a constraint unit $\equiv_{\mathsf{T}}$ Bool as an assumption — but since this is a discriminable equality, the abort instruction typechecks, thus expressing the fact that our program should never reach this point during its evaluation. Admittedly, this example is not particularly useful: we discuss more practical examples and their encodings in the following section.

First, though, we turn to the operational semantics of our programs, presented in Figure 6. This, again, is largely standard — and the rules for the new constructs simply force appropriate computations or pass the argument to the computation that requires it. As usual, we model failure by a stuck computation: thus, neither of the abort expressions has any contraction rule associated with it.

Nontermination in $F^{=i}_{\omega\mu}$. It is clear that well-typed programs in our calculus are not necessarily terminating, due to the inclusion of recursive types. However, it is interesting to observe that recursive types are not the *only* source of non-terminating behavior: the combination of impredicative quantification and injectivity of type-constructors with higher-kinded arguments is sufficient.[4]

To observe this behavior, consider the following type, well-kinded at T if $\alpha :: \mathsf{T}$, for any constructor $c :: (\mathsf{T} \Rightarrow \mathsf{T}) \Rightarrow \mathsf{T}$ (either the existential, universal or recursive type constructor):

$$\tau_c^{\text{loop}} \triangleq \exists \beta :: \mathsf{T} \Rightarrow \mathsf{T}.\ (c\ \beta \equiv_{\mathsf{T}} \alpha) \times (\beta\ \alpha \to \text{void})$$

We can now use the same constructor to close-off our type, taking $\tau_c^{\text{cl}} \triangleq c\ (\lambda \alpha :: \mathsf{T}.\ \tau_c^{\text{loop}})$, and use it to give a type to the following variant of the staple of non-terminating computations:

$$v^{\text{loop}} \triangleq \lambda x.\ \text{let } (*, (\bullet, y)) = x \text{ in } y\ (\text{pack}\ \langle \bullet, y\rangle)$$

Indeed, we can easily show that, in closed contexts,

$$\vdash v^{\text{loop}} : \tau_c^{\text{loop}}[\tau_c^{\text{cl}}/\alpha] \to \text{void},$$

taking τ_c^{loop} itself as the existential witness, and reflexivity as the required proof of type equivalence, and using injectivity of c on the assumption retrieved from the package to ensure that y is typed — in both instances — at $\tau_c^{\text{loop}}[\tau_c^{\text{cl}}/\alpha]$. Analogous packaging of v^{loop} will ensure that $\vdash v^{\text{loop}}\ (\text{pack}\ \langle \bullet, v^{\text{loop}}\rangle)$: void, and we get a closed expression at the empty type whose evaluation does not terminate (since it reduces to itself).

[4]We adapt a proof of falsehood in Idris via injective type constructors, available at https://github.com/idris-lang/Idris-dev/issues/3687, itself an adaptation of a similar proof in Lean, discussed at https://github.com/leanprover/lean/issues/654. While these systems are significantly different, the main problem — injectivity at higher kinds and impredicativity leading to inconsistency — remains applicable.

```
data Z where
data S a where

data GTree a n where
  EmptyG :: GTree a Z
  NodeG :: GTree a n -> a -> GTree a n -> GTree a (S n)

left :: GTree a (S n) -> GTree a n
left (NodeG l _ _) = l
```

Fig. 7. An example GADT, the type of perfectly-balanced trees.

A simpler sub-calculus. Since the main object of our study, System $\mathrm{F}^{=i}_{\omega\mu}$, exhibits non-terminating behavior without utilizing the recursive types, the injective, internalized equalities clearly increase the expressive power over standard System F_ω. However, to make this point even stronger, and to better highlight some of the difficulties in extending relational techniques to systems with GADTs, in Section 3 we study a sub-calculus of $\mathrm{F}^{=i}_{\omega\mu}$, called $\mathrm{F}^{=i}_{\omega}$ and establish its relationship to System F_ω. This removes the recursive types and all the rules associated with them, *i.e.* all the rules marked with (μ) and — in order to avoid the nonterminating construction discussed above — prohibits the use of the injectivity rule on quantifiers (which, in effect, restricts its use to sum, product and arrow types).

2.1 Expressing GADTs and Pattern Matching

We now discuss how the features of $\mathrm{F}^{=i}_{\omega\mu}$ allow for encoding of common patterns of GADTs. As is folklore, this will be achieved through a combination of existential quantification and equality constraints over types — however, in order to demonstrate how the system operates, we pay special attention to the equality reasoning required, particularly discriminability and injectivity.

As mentioned in the previous section, the distinguishing feature of (proper) GADTs is restriction of the index of certain constructors of a type constructor. Below, we present an alternative encoding of perfectly balanced binary trees, which is uniform in the type of its labels, but uses a separate index to guarantee constant depth on all paths. In the absence of other extensions of the type system, the classic Haskell approach is to define "type-level natural numbers", *i.e.* two distinguishable type constructors, one of kind T, and one of kind $\mathsf{T} \Rightarrow \mathsf{T}$, which we would externally identify with zero and successor. The implementation of these, and the balanced binary tree, are presented in Fig. 7.

There are a few things of note in the implementation. First, note that both `Z` and `S a` are empty types — however, since the ML lineage of languages treats data types nominally, they are still distinguishable, and `S` is considered injective. Second, the discriminability of these two types ensures that the two constructors of `GTree` have distinct indices: this, in turn, ensures that both arguments of `NodeG` have the same height, as measured by our pseudo-natural number type that represents the index. Finally, we can write a type-safe `left` function that recovers the left subtree of a *non-empty* tree. The non-emptiness of the argument is enforced by a constraint on its index: the `EmptyG` pattern is impossible, due to discriminability.

Our encoding of this example will not be entirely direct, due to the structural treatment of types in $\mathrm{F}^{=i}_{\omega\mu}$. Therefore, we take the type unit as encoding the depth 0, and the type constructor $\lambda\alpha :: \mathsf{T}.\, \mathsf{unit} + \alpha$ as the successor.[5] This ensures that the two "constructors" of type-level natural

[5]In this example, the precise types do not matter, since their role is *phantom* — they do not influence the computation other than through reasoning about their equality. As we shall see, this is not always the case.

$$
\begin{array}{l}
\mathsf{GTree} :: \mathsf{T} \Rightarrow \mathsf{T} \Rightarrow \mathsf{T} \\
\mathsf{GTree} \triangleq \\
\quad \lambda\alpha_t :: \mathsf{T}.\ \mu\varphi :: \mathsf{T} \Rightarrow \mathsf{T}.\ \lambda\alpha_n :: \mathsf{T}. \\
\quad ((\alpha_n \equiv_{\mathsf{T}} \mathsf{unit}) \times \mathsf{unit}) \\
\quad + \exists \alpha'_n :: \\
\qquad \mathsf{T}.\ (\alpha_n \equiv_{\mathsf{T}} (\alpha'_n + \mathsf{unit})) \times \varphi\ \alpha'_n \times \alpha_t \times \varphi\ \alpha'_n
\end{array}
$$

$$
\begin{array}{l}
\mathsf{left} : \forall \alpha_t, \alpha_n :: \\
\quad \mathsf{T}.\ \mathsf{GTree}\ \alpha_t\ (\alpha_n + \mathsf{unit}) \to \mathsf{GTree}\ \alpha_t\ \alpha_n \\
\mathsf{left} \triangleq \Lambda.\ \Lambda.\ \lambda x.\ \mathsf{case\ unroll}\, x \\
\quad \mid \mathsf{inj}_1\ (\bullet, \_).\ \mathsf{abort}\ \bullet \\
\quad \mid \mathsf{inj}_2\ (*, (\bullet, \langle l, \langle \_, \_ \rangle \rangle)).\ l
\end{array}
$$

Fig. 8. $\mathsf{F}^{=i}_{\omega\mu}$-encoding of `GTree` and `left`. We use syntactic sugar in the definition of left to improve readability.

```
data Tm v where
  Lift :: a -> Tm a
  Abs  :: (a -> Tm b) -> Tm (a -> b)
  App  :: Tm (a -> b) -> Tm a -> Tm b

eval :: Tm a -> a
eval (Lift x) = x
eval (Abs f) = \x -> eval (f x)
eval (App f x) = (eval f) (eval x)
```

Fig. 9. GADT and evaluation of well-formed lambda terms

numbers are discriminable, and that the "successor" is indeed injective. Now we can proceed with the encoding, presented in Fig. 8. There are a few things to note here. First, notice that we can take the type of labels, α_t, as an *external* parameter to the recursive type, as it remains uniform throughout the definition. On the other hand, the depth-encoding index, α_n, varies through the recursive calls and has to be bound *within* the recursive type. Finally, the body of the type is a disjoint sum, with both branches restricting the index α_n through an equality constraint. The left branch is only available at index unit, while the right branch is available at any index $\alpha'_n + \mathsf{unit}$ — with the new, smaller index α'_n bound existentially, and used at the recursive calls.

In the definition of the left function, we use some syntactic sugar to offset the verbosity of our fine-grain call-by-value notation. We introduce the two type variables, α_t and α_n, and an argument x of type $\mathsf{GTree}\ \alpha_t\ (\alpha_n + \mathsf{unit})$, and pattern-match on its unrolled form. In the left case, we obtain a unit value, and a constraint $\alpha_n + \mathsf{unit} \equiv_{\mathsf{T}} \mathsf{unit}$, which is clearly discriminable: thus, we can use the abort expression to obtain any well-formed type. In the right case, on the other hand, we introduce a fresh type variable α'_n, the existential witness of the depth of our subtrees, and a constraint $\alpha_n + \mathsf{unit} \equiv_{\mathsf{T}} \alpha'_n + \mathsf{unit}$. Note that the variable l, which denotes the left subtree, has type $\mathsf{GTree}\ \alpha_t\ \alpha'_n$, which must not escape the scope of the case: thus, it is crucial to cast the type through an equality to one that is well-formed *outside* the case expression. We can achieve this due to the injectivity rules, since the constraint we introduced implies that $\alpha_n \equiv_{\mathsf{T}} \alpha'_n$ holds. Thus, we have that $\mathsf{left} : \forall \alpha_t :: \mathsf{T}.\ \forall \alpha_n :: \mathsf{T}.\ \mathsf{GTree}\ \alpha_t\ (\alpha_n + \mathsf{unit}) \to \mathsf{GTree}\ \alpha_t\ \alpha_n$, which is the expected type.

In the previous example, the indices at which we used type equalities were entirely separated from the term-level content of the types: they were *phantom*. This afforded us a lot of leeway in terms of the encoding, as we only needed to ensure that appropriate type constructors were injective or discriminable. However, it is not always the case that indices do not carry any semantic meaning — GADTs also can be used to encode data types whose indices are used in nontrivial manner. In Fig. 9 we revisit the well-formed lambda terms example from the introduction: note that the presence of the `Lift` constructor allows us to treat any metalanguage value of type `a` as a constant term of that type. This binds the type structure of the indices to the type structure at the metalevel: we are no longer free to choose the encoding of the other indices arbitrarily, as we would lose the connection to the lifted values at the appropriate index. This can be observed in the

```
Tm :: T ⇒ T
Tm ≜
  μφ :: T ⇒ T. λα :: T.
  α + (∃β, γ :: T. (α ≡_T (β → γ)) × (β → φ γ))
    + (∃β :: T. φ (β → α) × φ β)
```

```
eval : ∀α :: T. Tm α → α
eval ≜
  fix λf. Λ. λx.
  case unroll x
  | inj_1 y. y
  | inj_2 y. case y
    | inj_1 (∗, (∗, (•, g))). λz. f ∗ (g z)
    | inj_2 (∗, ⟨g, x⟩). (f ∗ g) (f ∗ x)
```

Fig. 10. $\mathrm{F}^{=i}_{\omega\mu}$-encoding of `Tm` and `eval`. We use syntactic sugar in the definition of eval to improve readability.

well-typed evaluator in Fig. 9, which transforms lambda-terms indexed with type a into values *of that type*. Although this pattern requires a rather tight connection between the indices of recursive types and the types themselves, we can encode it in $\mathrm{F}^{=i}_{\omega\mu}$ using the same approach presented above. The encodings are presented in Fig. 10. Note that there is no equality constraint in the first branch of the sum type: this means the corresponding case in pattern-matching is never discriminable, and thus has to be considered in any program. This matches the intended behavior, as we consider a non-exhaustive pattern matching a type error. Since the definition of eval is recursive, it utilizes a fix combinator, which is definable through the use of recursive types, using the standard construction.

3 NON-EXPRESSIBILITY OF $\mathrm{F}^{=i}_{\omega}$ IN F_{ω}

We begin the study of our calculus by establishing its relationship with System F_{ω} — the polymorphic lambda calculus with higher kinds, of which $\mathrm{F}^{=i}_{\omega}$ is an extension. For brevity, we do not present the standard rules of System F_{ω} for kinding, constructor equivalence, and typing.

It is well-known that System F_{ω} is strong enough to express some notions of equality using universal quantification at higher kinds. For instance, Atkey [2012] uses a Church-style definition of equality as its own eliminator, as the following type constructor, although a Leibniz-style definition is also possible [Barendregt 1993, §5.4, Definition 5.4.17]:

$$\mathsf{eq}_{\kappa} \triangleq \lambda\alpha, \beta :: \kappa.\ \forall\rho :: \kappa \Rightarrow \kappa \Rightarrow \mathsf{T}.\ (\forall\gamma :: \kappa.\ \rho\ \gamma\ \gamma) \to \rho\ \alpha\ \beta$$

This leads to a pertinent question: does the reification of equalities and the complex type system proposed in the previous section add any expressive power, or is it an over-complicated reimagining of F_{ω}?

To answer this question we study the restricted system, $\mathrm{F}^{=i}_{\omega}$, equipped with reified equality, discriminability and the injectivity rules for type constructors with ground arguments, but without considering the recursive types or injectivity on higher-kinded arguments, which would take the system beyond the expressive power of F_{ω} by allowing nonterminating behaviors. We study the problem of expressibility of $\mathrm{F}^{=i}_{\omega}$ in F_{ω} as an extension of Felleisen's [1991] approach to typed languages, *i.e.* the question of existence of a *structural* translation of the internalized equalities and associated constructs, which preserves the structure of the features already present in the target language. Through an appeal to a model of F_{ω} defined below, we establish that no such translation that preserves typing may exist.

The propositional model of F_{ω}. The model we construct in this section treats the F_{ω} types as propositions, in the usual Curry-Howard fashion: the idea is to enforce the condition that the inhabited types are mapped to true propositions, while the empty types are mapped to false ones.

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
[\![\alpha]\!]_\eta & \triangleq & \eta(\alpha) & [\![\times]\!]\ \varphi_1\ \varphi_2 & \triangleq & \varphi_1 \wedge \varphi_2 & [\![\mathsf{unit}]\!] & \triangleq & \top \\
[\![\lambda\alpha :: \kappa.\ \tau]\!]_\eta & \triangleq & \lambda\varphi.[\![\tau]\!]_{\eta[\alpha\mapsto\varphi]} & [\![+]\!]\ \varphi_1\ \varphi_2 & \triangleq & \varphi_1 \vee \varphi_2 & [\![\mathsf{void}]\!] & \triangleq & \bot \\
[\![\sigma\ \tau]\!]_\eta & \triangleq & [\![\sigma]\!]_\eta([\![\tau]\!]_\eta) & [\![\to]\!]\ \varphi_1\ \varphi_2 & \triangleq & \varphi_1 \Rightarrow \varphi_2 & [\![\forall_\kappa]\!]\ \varphi & \triangleq & \forall\psi : [\![\kappa]\!].\ \varphi(\psi) \\
[\![c]\!]_\eta & \triangleq & [\![c]\!] & & & & [\![\exists_\kappa]\!]\ \varphi & \triangleq & \exists\psi : [\![\kappa]\!].\ \varphi(\psi)
\end{array}
$$

Fig. 11. Propositional interpretation of the types (left) and constructors (middle and right) of F_ω.

Our formalization uses the Coq type of propositions; however, a set-theoretic model of truth is equally valid.

We begin by defining the interpretation of kinds:

$$
\begin{aligned}
[\![\mathsf{T}]\!] &\triangleq \mathrm{Prop} \\
[\![\kappa_1 \Rightarrow \kappa_2]\!] &\triangleq [\![\kappa_1]\!] \to [\![\kappa_2]\!],
\end{aligned}
$$

with the implicit notion that equivalent propositions are equal, and that functions preserve equality and are themselves equal extensionally. Now we can define the interpretation of well-kinded types of System F_ω, which is presented in Fig. 11. We take the usual type of the interpretation function as $[\![\Delta \vdash \tau :: \kappa]\!] : (\prod_\alpha [\![\Delta(\alpha)]\!]) \to [\![\kappa]\!]$, and omit the contexts and kinds to avoid cluttering the definition. Note that the constructors are interpreted as appropriate logical connectives; in particular, the empty type is interpreted as falsehood.

It is immediate that the definition is well-formed. As a soundness result, we obtain the following theorem, where the interpretation of the term-variable context is given as by the conjunction of the interpretations of types.

Theorem 3.1. *For any well-typed term $\Delta \mid \Gamma \vdash e : \tau$ in F_ω and any $\eta : \prod_\alpha [\![\Delta(\alpha)]\!]$ we have $[\![\Gamma]\!]_\eta \Rightarrow [\![\tau]\!]_\eta$.*

Note that since the types in Γ and τ are of kind T, the theorem is well-formed. With this result, we are ready to tackle the problem of expressibility of $\mathrm{F}_\omega^{=i}$.

Non-existence of a translation. Armed with the model of F_ω, we can now prove the following non-expressibility theorem:

Theorem 3.2. *There cannot exist a family of translations $\lceil - \rceil : F_\omega^{=i} \to F_\omega$ for expressions, values and types such that the following conditions hold:*

- *$\lceil - \rceil$ preserve closedness of types and terms;*
- *$\lceil - \rceil$ is homomorphic on constructs of F_ω;*
- *if $\cdot \mid \cdot \mid \cdot \vdash e : \tau$ holds in $F_\omega^{=i}$, then $\cdot \mid \cdot \vdash \lceil e \rceil : \lceil \tau \rceil$ holds in F_ω.*

Note that, while we restrict ourselves to conditions on typing of closed programs with no equality assumptions, the requirement that the translation is homomorphic ensures that the typing also needs to be preserved in open contexts that can arise as application of the typing rules of F_ω. Certainly, the most important feature of a translation from $\mathrm{F}_\omega^{=i}$ to F_ω would be the elimination of constraints and equality assumptions: we make no assumptions as to how this would be achieved, as the impossibility of the translation follows already from its required behavior on types and type constructors.

Proof. Assume that a translation that satisfies the requirements exists, and consider a type constructor $\varphi \triangleq \lambda\alpha :: \mathsf{T}.\ (\alpha \equiv_\mathsf{T} \mathsf{unit} + \mathsf{void}) \times \mathsf{unit}$, well-kinded in $\mathrm{F}_\omega^{=i}$ at kind $\mathsf{T} \Rightarrow \mathsf{T}$. Clearly, there exists an expression e, such that $\cdot \mid \cdot \mid \cdot \vdash e : \varphi\ (\mathsf{unit} + \mathsf{unit}) \to \mathsf{void}$, since the assumed constraint

allows us (by using injectivity) to cast the unit value to type void. Thus, by the properties of translation, we get that $\cdot \mid \cdot \vdash \ulcorner e \urcorner : \ulcorner \varphi \urcorner\ (\mathsf{unit} + \mathsf{unit}) \to \mathsf{void}$, and consequently, by the properties of the propositional model, $(\llbracket \ulcorner \varphi \urcorner \rrbracket\ \top) \Rightarrow \bot$.

However, observe that the type $\varphi\ (\mathsf{unit} + \mathsf{void})$ is clearly inhabited (by the value $\langle \bullet, \langle\rangle \rangle$), and thus, by properties of the translation and the soundness of the model, $\llbracket \ulcorner \varphi \urcorner \rrbracket . \top$ holds, leading to a contradiction. □

Note that the proof depends only on injectivity of the disjoint sum type constructor and the ability to cast through type equalities. An analogous construction can be performed for discriminability, since $\mathsf{unit} + \mathsf{unit}$ and unit are also equal in our model of F_ω. The intuitive content of the proof is that the requirement that the translation behaves homomorphically on type constructors cannot be fulfilled: while in System F_ω, a type constructor that builds a non-empty type from a non-empty argument will do so for *any* non-empty argument, this is not necessarily the case in the presence of discriminable (or injective) equality constraints.

Through this theorem, we have established that $\mathrm{F}^{=i}_\omega$ is not directly expressible in F_ω. We believe that this highlights the expressive power of GADTs, which depend heavily on injectivity rules to express the precise pattern-matching constructs. Moreover, it highlights that while the adage that GADTs are existential types and type equalities may be true, one ought to be very precise about what can be proved about the equality of types within the system.

Finally, the propositional model of F_ω serves to highlight the difficulties that any traditional relational interpretations is bound to run into: while the propositional model identifies all inhabited types, the standard relational approach tends to, at least, conflate all the *empty* types. It is also difficult to imagine how one could obtain injectivity for type arguments that appear in a contravariant position in the interpretation, such as for the left-hand side of the arrow type.

4 TYPE SOUNDNESS OF $\mathrm{F}^{=i}_{\omega\mu}$

Having established that our calculus is a non-trivial extension of the known systems, we turn to the question of its type soundness. In this section we apply the classic syntactic methodology of progress and preservation [Wright and Felleisen 1994]. However, since the type-system of $\mathrm{F}^{=i}_{\omega\mu}$ includes a complex system of equality reasoning — in particular, a system that is able to express that assumptions are contradictory — the progress lemma becomes non-trivial. On the other hand, the same fact allows us to state in one concise lemma all the properties that are required to prove canonical forms (also called inversion) lemmas for particular type formers. This lemma states the *consistency* of our proof system.

Lemma 4.1 (Consistency). *A discriminable constraint is not provable in an empty context: in other words,* $\emptyset \mid \emptyset \Vdash \tau_1 \equiv_\kappa \tau_2$ *and* $\emptyset \Vdash \tau_1 \#_\kappa \tau_2$ *are contradictory.*

We prove this lemma via a *normalization-by-evaluation* argument [Berger and Schwichtenberg 1991; Danvy 1996], which we discuss in more detail in Section 5.2; for now, we discuss the methodological implications of the lemma. In the case of $\mathrm{F}^{=i}_{\omega\mu}$ its statement follows naturally from having the discriminability judgment as part of the type system used to discharge contradictory equality assumptions — a feature that is rarely, if ever, present: most systems with non-trivial equality of types consider them up to relatively simple syntactic rearrangements or some forms of subtyping. However, we believe that even in those cases it may be a useful methodological approach to define these relations in such a way that a variant of Lemma 4.1 can be stated. Moreover, if the system admits a natural characterization of normal forms and non-trivial computation on the level of types, we believe that normalization-by-evaluation is a natural path to follow in proving consistency. To the best of our knowledge, this is the first time such a consistency property is used

explicitly as a crucial ingredient in a Wright–Felleisen-style proof of type soundness via progress and preservation [Wright and Felleisen 1994].

Lemma 4.1 can be used to eliminate impossible cases in progress and preservation lemmas. Its first application is in the proofs of the canonical forms and inversion lemmas, which have to account for possible applications of equality casting rule: here, it plays the role of traditional inversion lemmas in type systems with subtyping principles. The lemma allows us to eliminate the impossible cases, which are trivially discriminable: thus, the notion of discriminability serves to simplify and modularize these proofs. We observe this in the following instance of a canonical form lemma for the arrow types.

Lemma 4.2 (Canonical form for arrows). *If v is a closed value of type τ and τ is provably equal to some arrow type in an empty context, then v is a lambda-abstraction with a well-typed body.*

$$(\emptyset \mid \emptyset \Vdash \tau \equiv_{\mathsf{T}} (\tau_1 \to \tau_2)) \wedge (\emptyset \mid \emptyset \mid \Gamma \vdash v : \tau) \implies (\exists x\, e.\, v = \lambda x.\, e \wedge \emptyset \mid \emptyset \mid \Gamma, x : \tau_1 \vdash e : \tau_2)$$

Proof. By induction on the typing derivation. Of the applicable proof rules only type-cast and lambda abstraction rules are not discriminable; the remainder can be dispatched via Lemma 4.1 (*e.g.* if $\tau = \tau_3 + \tau_4$). In the first of the remaining cases we proceed by induction, since the type equality is transitive. Finally, for the lambda abstraction we only need to cast the variable and expression through appropriate equalities, which follow from injectivity of the arrow constructor. □

Lemma 4.3 (Preservation). *Reductions preserves typing in the sense that for any well-typed $\emptyset \mid \emptyset \mid \emptyset \vdash e : \tau$ such that $e \longrightarrow e'$, we also have $\emptyset \mid \emptyset \mid \emptyset \vdash e' : \tau$.*

Proof. The proof is by induction on a given typing derivation.[6] All the cases (except for type conversion, which is trivial) follow by case analysis of the reduction, with the appropriate canonical form lemmas used where necessary. □

Lemma 4.4 (Progress). *Any well-typed expression $\emptyset \mid \emptyset \mid \emptyset \vdash e : \tau$ can either be reduced or is already a value.*

Proof. The proof proceeds by induction on the typing derivation. All the cases, except for the (trivial) type conversion and discrimination of impossible equalities, follow from canonical form lemmas. The latter case holds due to the contradiction rule for equality constraints. In this case, we assume that $e = \mathsf{abort}\ \bullet$, and both $\emptyset \mid \emptyset \Vdash \tau \equiv_{\mathsf{T}} \sigma$ and $\emptyset \Vdash \tau \#_{\mathsf{T}} \sigma$ hold. By Lemma 4.1, we evidently have a contradiction. □

Definition 4.5 (Safety). A closed expression e is called *safe* when any expression that e reduces to is irreducible if and only if it is a value.

Theorem 4.6 (Soundness). *Any well-typed expression $\emptyset \mid \emptyset \mid \emptyset \vdash e : \tau$ is safe in the sense of Definition 4.5.*

Proof. The proof follows from progress and preservation lemmas. □

5 RELATIONAL MODELS OF INJECTIVE TYPE EQUALITIES

In this section we discuss relational models of $\mathrm{F}^{=i}_{\omega\mu}$. We begin by discussing in detail the challenge of constructing models that validate injectivity rules (this challenge could already be observed in Section 3). In Section 5.2 we follow this by constructing a unary model, which utilizes a normalization-by-evaluation view of the types, and which we can use to semantically justify type soundness. In Section 5.3 we discuss the limitations of this construction, in particular for the case of binary relations (as needed for reasoning about representation independence), and, finally, we

[6] This allows us to omit inversion lemmas for elimination forms.

$$\frac{c :: \kappa}{\vdash c :: \mathrm{Neu}^{\Delta}_{\kappa}} \qquad \frac{x :: \kappa \in \Delta}{\vdash x :: \mathrm{Neu}^{\Delta}_{\kappa}} \qquad \frac{\vdash \nu :: \mathrm{Neu}^{\Delta}_{\kappa_a \Rightarrow \kappa_r} \qquad \vdash \mu :: \mathrm{Nf}^{\Delta}_{\kappa_a}}{\vdash \nu\ \mu :: \mathrm{Neu}^{\Delta}_{\kappa_r}}$$

$$\frac{\vdash \chi :: \mathrm{NC}^{\Delta} \qquad \vdash \nu :: \mathrm{Neu}^{\Delta}_{\mathsf{T}}}{\vdash \chi \to \nu :: \mathrm{Neu}^{\Delta}_{\mathsf{T}}} \qquad \frac{\vdash \chi :: \mathrm{NC}^{\Delta} \qquad \vdash \nu :: \mathrm{Neu}^{\Delta}_{\mathsf{T}}}{\vdash \chi \times \nu :: \mathrm{Neu}^{\Delta}_{\mathsf{T}}}$$

$$\frac{\vdash \nu :: \mathrm{Neu}^{\Delta}_{\mathsf{T}}}{\vdash \nu :: \mathrm{Nf}^{\Delta}_{\mathsf{T}}} \qquad \frac{\vdash \mu :: \mathrm{Nf}^{\Delta,\alpha::\kappa_a}_{\kappa_r}}{\vdash \lambda\alpha :: \kappa_a.\ \mu :: \mathrm{Nf}^{\Delta}_{\kappa_a \Rightarrow \kappa_r}} \qquad \frac{\left(\vdash \mu_i :: \mathrm{Nf}^{\Delta}_{\kappa}\right)_{i \in \{1,2\}}}{\vdash \mu_1 \equiv_{\kappa} \mu_2 :: \mathrm{NC}^{\Delta}}$$

Fig. 12. A judgmental presentation of neutral and normal types, and normal constraints.

extend the model construction to allow for semantic binary relations as needed for reasoning about representation independence. We use the model to obtain the first known proofs of representation independence in a setting with generalized algebraic data types.

5.1 The Challenge of Injective Relational Interpretations

To appreciate the challenge of constructing a relational interpretation where all type constructors would be injective, consider the propositional interpretation of System F_{ω} presented in Figure 11. While relational interpretations — whether unary or binary — contain additional information, the basic structure remains very similar. In particular, whenever a given type is uninhabited, its interpretation is empty: this leads to the resurgence of the problems that threaten validity of the injectivity and discriminability rules. Therefore, a *direct* interpretation of types as predicates on values is *not* viable.

A universe of codes. A natural attempt to square this circle is through introduction of some universe of *codes* of types, into which we could interpret types in a way that would validate injectivity and discriminability, and which we could decode (or interpret, or realize), in a separate step, as predicates on values. This, however, presents its own challenges, as the structure of the types is rather rich. Indeed, there would be two conflicting requirements: (1) In order to obtain a justification of injectivity and discriminability rules, such a universe (or at least a significant part of it) would need to be inductively defined. But at the same time, (2) the injectivity of the quantifiers and recursive type constructors, and the rich equational theory involving the higher-kinded types, requires a rather semantic view of the codes, particularly at higher types. Fulfilling those seemingly conflicting requirements without succumbing to paradoxes is challenging — and it is the subject of the remainder of this section.

5.2 A Unary Model via Normalization-by-Evaluation

A key insight of the model construction we now present is that the above mentioned requirements for our universe of codes match very closely those known from the area of normalization-by-evaluation: we have the *inductively* defined normal forms, and *semantic* interpretations, which are used to justify the sophisticated equational or computational theory. In our case, the neutral and normal forms of types and constraints are presented in Figure 12.

Definition 5.1. We will write $\mathcal{K}$ for the category of kinding contexts Δ with morphisms $\delta : \Delta \to \Delta'$ given by kind-preserving *renamings* sending variables $\alpha \in \Delta'$ to variables $\delta^*\alpha : \Delta'(\alpha)$ in context Δ.

$$\begin{array}{ll}
\mathrm{reify} : [\![\kappa]\!] \Rightarrow \mathrm{Nf}_\kappa & \mathrm{reflect} : \mathrm{Neu}_\kappa \Rightarrow [\![\kappa]\!] \\
\mathrm{reify}(\nu : [\![\mathsf{T}]\!]) \triangleq \nu & \mathrm{reflect}(\nu : \mathrm{Neu}_\mathsf{T}) \triangleq \nu \\
\mathrm{reify}(\varphi : [\![\kappa_a \Rightarrow \kappa_r]\!]) \triangleq & \mathrm{reflect}(\nu : \mathrm{Neu}_{\kappa_a \Rightarrow \kappa_r})(\mu : [\![\kappa_a]\!]) \triangleq \\
\quad \lambda\alpha :: \kappa_a.\, \mathrm{reify}(\varphi\ (\mathrm{reflect}(\alpha))) & \quad \mathrm{reflect}(\nu\ (\mathrm{reify}(\mu)))
\end{array}$$

Fig. 13. Reification and reflection functions, defined in the internal language of $\mathrm{Pr}(\mathcal{K})$.

We will write $\mathbf{y}_\mathcal{K} : \mathcal{K} \hookrightarrow \mathrm{Pr}(\mathcal{K})$ for the ***Yoneda embedding*** of $\mathcal{K}$ into its category of presheaves $\mathrm{Pr}(\mathcal{K}) = \mathbf{Set}^{\mathcal{K}^{\mathrm{op}}}$ sending $\Delta \in \mathcal{K}$ to the representable functor $\hom_\mathcal{K}(-, \Delta)$.

For a given kind κ, the collections of normal constructors, well-formed neutral constructors, and normal constraints form *presheaves* Nf_κ, Neu_κ, and NC on the category $\mathcal{K}$, as described inductively in Fig. 12. For example, $\mathrm{Nf}_\kappa^\Delta$ is the set of normal forms of constructors of kind κ in context Δ. The functorial action is given by syntactic renaming of variables.

Definition 5.2 (Intepretation of kinds). We now define an NbE-inspired interpretation of kinds into presheaves on the category $\mathcal{K}$ of kinding contexts and renamings as follows:

$$[\![\mathsf{T}]\!] \triangleq \mathrm{Neu}_\mathsf{T}$$
$$[\![\kappa_a \Rightarrow \kappa_r]\!] \triangleq [\![\kappa_a]\!] \Rightarrow [\![\kappa_r]\!]$$

In the second clause above, the right-hand $\Rightarrow$ denotes the exponential presheaf [Mac Lane and Moerdijk 1992, §I.6, Proposition 1]. The interpretation of kinds κ is extended to kind contexts Δ pointwise, using the cartesian product of presheaves:

$$[\![\Delta]\!] \triangleq \prod_{\alpha :: \kappa \in \Delta} [\![\kappa]\!]$$

We use the interpretation of kinds defined above as the type of the interpretation function for types; the interpretation of constraints will target constraints in normal form.

The normalization procedure can now proceed in the usual way, by defining the injection of neutral forms and reification, and reflection of well-formed types into the interpretation of their kinds (and normalization of constraints).

We begin by defining, by mutual induction on the structure of kinds, the *reification* of interpretations of kinds as normal forms and the *reflection* of neutrals as interpretation of kinds (the main purpose of the latter is to perform a semantic analogue of eta-expansion). In order to enforce the well-formedness, these are defined as exponential presheaves; we present the implementation in Figure 13, making use of the cartesian closed structure of presheaves via the internal language of $\mathrm{Pr}(\mathcal{K})$. Note how the reification ensures that all functional types will be in eta-long form, and how the variable α needs to be reflected to be passed as an argument to φ, since its kind may be functional (and thus the variable may need to be eta-expanded as well).

Definition 5.3 (The identity environment). For each context Δ, we may define an *identity environment* $\mathrm{id}_\Delta : [\![\Delta]\!]^\Delta$ using the reflection operation, setting $\mathrm{id}_\Delta(\alpha :: \kappa \in \Delta) \triangleq \mathrm{reflect}(\alpha)$.

With reification and reflection in place, we can define the interpretation of types. As with the other two functions, well-formedness with respect to renaming is ensured by taking the denotation in the exponential presheaf:

$$[\![\Delta \vdash \tau :: \kappa]\!] : [\![\Delta]\!] \Rightarrow [\![\kappa]\!]$$

We present the implementation of this interpretation function in Figure 14, again using the internal language of $\mathrm{Pr}(\mathcal{K})$.

$$
\begin{aligned}
[\![\Delta \vdash \alpha :: \kappa]\!]_\eta &\triangleq \eta(\alpha) \\
[\![\Delta \vdash \lambda\alpha :: \kappa_a.\ \tau :: \kappa_a \Rightarrow \kappa_r]\!]_\eta &\triangleq \mu \mapsto [\![\Delta, \alpha :: \kappa_a \vdash \tau :: \kappa_r]\!]_{\eta[\alpha \mapsto \mu]} \\
[\![\Delta \vdash \sigma\ \tau :: \kappa_r]\!]_\eta &\triangleq [\![\Delta \vdash \sigma :: \kappa_a \Rightarrow \kappa_r]\!]_\eta\, [\![\Delta \vdash \tau :: \kappa_a]\!]_\eta \\
[\![\Delta \vdash c :: \kappa]\!]_\eta &\triangleq \mathrm{reflect}(c) \\
[\![\Delta \vdash \chi \to \tau :: \mathsf{T}]\!]_\eta &\triangleq [\![\Delta \vdash \chi\ \mathrm{constr}]\!]_\eta \to [\![\Delta \vdash \tau :: \mathsf{T}]\!]_\eta \\
[\![\Delta \vdash \chi \times \tau :: \mathsf{T}]\!]_\eta &\triangleq [\![\Delta \vdash \chi\ \mathrm{constr}]\!]_\eta \times [\![\Delta \vdash \tau :: \mathsf{T}]\!]_\eta \\
[\![\Delta \vdash \tau_1 \equiv_\kappa \tau_2\ \mathrm{constr}]\!]_\eta &\triangleq \mathrm{reify}([\![\Delta \vdash \tau_1 :: \kappa]\!]_\eta) \equiv_\kappa \mathrm{reify}([\![\Delta \vdash \tau_2 :: \kappa]\!]_\eta)
\end{aligned}
$$

Fig. 14. Interpretation of types and constraints, specified in the internal language of $\mathrm{Pr}(\mathcal{K})$.

$$
\begin{aligned}
\eta \mid \nu_1 \approx_{\mathsf{T}} \nu_2 &\triangleq [\![\nu_1]\!]_\eta = \nu_2 \\
\eta \mid \varphi_1 \approx_{\kappa_a \Rightarrow \kappa_r} \varphi_2 &\triangleq \forall \Delta_1', \Delta_2', (\delta_1 : \hom_{\mathcal{K}}(\Delta_1', \Delta_1), \delta_2 : \hom_{\mathcal{K}}(\Delta_2', \Delta_2)), (\eta' : [\![\Delta_1']\!]^{\Delta_2'}), \mu_1, \mu_2. \\
&\quad \big(\delta_2^* \eta = \lambda x.\ \eta'(\delta_1(x))\big) \to \big(\eta' \mid \mu_1 \approx_{\kappa_a} \mu_2\big) \to \big(\eta' \mid \varphi_1(\delta_1, \mu_1) \approx_{\kappa_r} \varphi_2(\delta_2, \mu_2)\big)
\end{aligned}
$$

Fig. 15. A logical relation connecting interpretations of types via an environment. The relation $\eta \mid \mu_1 \approx_\kappa \mu_2$ ranges over $\eta : [\![\Delta_1]\!]^{\Delta_2}$, $\mu_1 : [\![\kappa]\!]^{\Delta_1}$, and $\mu_2 : [\![\kappa]\!]^{\Delta_2}$.

While this definition bakes in the requirement that the interpretation of our types is well-behaved in terms of functoriality, we need more: due to the presence of injectivity, if the interpretation of our constraints is to verify the reasoning rules, reification needs to be injective, at least on the image of interpretation of types.

Lemma 5.4. *For any types τ_1, τ_2 of kind κ, well-formed in Δ and any* good *(defined in the following paragraph) environment $\eta : [\![\Delta]\!]^{\Delta'}$, if* $\mathrm{reify}([\![\tau_1]\!]_\eta) = \mathrm{reify}([\![\tau_2]\!]_\eta)$*, then* $[\![\tau_1]\!]_\eta = [\![\tau_2]\!]_\eta$.

This lemma does not depend on conventional soundness or completeness lemmas for NbE, nor is it useful in their proofs in the absence of injectivity. However, in the presence of injectivity of constructors, it is crucial to the proof of completeness of NbE, which needs to justify the reasoning rules of Fig. 3 — in particular, the injectivity rule. There, we can assume that $[\![c\ (\sigma_i)_i]\!]_\eta = [\![c\ (\tau_i)_i]\!]_\eta$, and need to prove that $\big([\![\sigma_i]\!]_\eta = [\![\tau_i]\!]_\eta\big)_i$. The types in our assumption are applications of a constructor to arguments, and thus neutral, but this only allows us to establish that $\big(\mathrm{reify}([\![\sigma_i]\!]_\eta) = \mathrm{reify}([\![\tau_i]\!]_\eta)\big)_i$: thus, to establish completeness, we need Lemma 5.4.

To prove this lemma, we introduce a novel logical relation $(\eta \mid \mu_1 \approx_\kappa \mu_2)$, as depicted in Fig. 15. Note that, in contrast to the usual relation that appears in the proof of soundness and relates syntactic view of types to the semantic view, this relation connects two semantic types via a mediating environment, which serves to reconcile the free variables in the two types.[7] The relation naturally extends to a pair of environments (with matching domain) related through a third, mediating environment. We say that a semantic type $\mu \in [\![\kappa]\!]^\Delta$ (respectively, environment) is *good* when it is related to itself via the identity environment:

$$\mathrm{good}(\mu) \triangleq (\mathrm{id}_\Delta \mid \mu \approx_\kappa \mu)$$

With this definition, we can establish the (limited) injectivity of reification that we need via a series of technical lemmas. The most important include the following pair of results.

[7] To the best of our knowledge, this approach has not been previously employed in the NbE literature.

Lemma 5.5. *If $\eta \mid \mu_1 \approx \mu_2$, then $[\![\mathrm{reify}(\mu_1)]\!]_\eta = \mu_2$.*

Lemma 5.6. *If $\eta \mid \eta_1 \approx \eta_2$, then $\eta \mid [\![\tau]\!]_{\eta_1} \approx [\![\tau]\!]_{\eta_2}$.*

The first of these establishes that we can obtain the related semantic type via reinterpretation of the reification; the second ensures that interpreting a type with related environments yields related results.

With Lemma 5.4, we can finally establish correctness of equational reasoning and discriminability.

Lemma 5.7. *If $\Delta \mid \Phi \Vdash \psi$ holds, and $\eta : [\![\Delta]\!]^{\Delta'}$ is a good environment and $[\![\varphi]\!]_\eta$ is true for any $\varphi \in \Phi$, then $[\![\psi]\!]_\eta$ is also true. Moreover, if $\Delta \Vdash \tau_1 \#_\kappa \tau_2$ holds, then $[\![\tau_1 \equiv_\kappa \tau_2]\!]_\eta$ is false.*

In the above, we say that (a normal form of) a constraint is true if the two types (in normal forms) are identical. As a corollary, we obtain the consistency lemma of Section 4.

Realizing NbE interpretations as predicates on values. While the normalization-by-evaluation argument has given us *some* interpretation of types that can be used to justify the equality and discriminability rules of the calculus, it is far from obvious, *a priori*, that we can use it to give a relational interpretation on values. To get an intuitive view of what remains to be done, note that we have largely dealt with the problem of open types, and the main problem that remains is that of open terms. Note also that, crucially, it is enough to consider evaluation in closed contexts, both for terms and types, and that we only need to give a relational interpretation to types of kind T. Therefore, we define a function $\mathcal{R} : \mathrm{Neu}^{\cdot}_{\mathsf{T}} \to \mathrm{Val} \to \mathit{iProp}$, using a mixture of guarded and structural recursion. Here *iProp* is a universe of step-indexed propositions, which comes equipped with a later modality and supports definition of recursive predicates when the recursion is guarded by the later modality (concretely, *iProp* can be understood as downwards-closed sets of natural numbers or as the universe of propositions in an appropriate logic framework, such as LSLR [Dreyer et al. 2011] or Iris [Jung et al. 2018]). The definition is presented in Figure 16. Note that, in contrast to the usual presentation of impredicative quantification, we *need* to use guarded recursion for universal and existential quantifiers to ensure that the interpretation is well-defined, as the normal form of the body of the quantified type needs to be *reinterpreted* to obtain a normal form of the resulting type. Thus, this approach does not conflict with the non-terminating example of Section 2: the fact that the model validates the injectivity rules for universal and existential quantifiers precludes an interpretation of the quantifiers that doesn't require us to count a computation step.

With the interpretation of the closed types defined (in two stages), we can now turn to the definition of the logical relation, which is presented in Figure 17. We pay attention to the fact that the environments for the type-variable context Δ are "good", *i.e.* that all the interpretations are self-related, which ensures that this property is inherited by any interpretation of a type with that environment. The environment needs to validate all the equality assumptions in Φ, and the closing environment is the standard extension of the relation on values. With this definition, it is easy to prove the fundamental theorem of logical relations:

Theorem 5.8 (Fundamental). *Any well-typed value, $\Delta \mid \Phi \mid \Gamma \vdash v : \tau$, is in the logical interpretation of its type,* i.e. $\Delta \mid \Phi \mid \Gamma \models v : \tau$. *Any well-typed expression, $\Delta \mid \Phi \mid \Gamma \vdash e : \tau$, is in the logical interpretation of its type for computations,* i.e. $\Delta \mid \Phi \mid \Gamma \models e : \tau$.

Proof. By induction on the structure of the derivation, using appropriate compatibility lemmas. For the type conversion rules, note that the assumptions of Lemma 5.7 are satisfied. Thus, the normal forms of the two types are equal, and so their realizers coincide. □

Limitations of the model based on the NbE interpretation of types. While the model presented in this section justifies semantically the type system of System $\mathrm{F}^{=i}_{\omega\mu}$, and the realizability interpretation

$$
\begin{aligned}
\mathcal{E}(P)(e) &\triangleq (\exists v.\ P(v) \wedge e = v) \vee (\exists e'.\ e \to e' \wedge \triangleright\, \mathcal{E}(P)(e')) \\
\mathcal{R}(\mathsf{unit})(v) &\triangleq v = \langle\rangle \\
\mathcal{R}(\mathsf{void})(v) &\triangleq \bot \\
\mathcal{R}(\nu_1 \times \nu_2)(v) &\triangleq \exists v_1, v_2.\ v = \langle v_1, v_2\rangle \wedge \bigwedge_{i\in\{1,2\}} \mathcal{R}(\nu_i)(v_i) \\
\mathcal{R}(\nu_1 + \nu_2)(v) &\triangleq \bigvee_{i\in\{1,2\}} \exists v'.\ v = \mathsf{inj}_i\, v' \wedge \mathcal{R}(\nu_i)(v') \\
\mathcal{R}(\nu_1 \to \nu_2)(v) &\triangleq \forall u.\ \mathcal{R}(\nu_1)(u) \to \mathcal{E}(\mathcal{R}(\nu_2))(v\ u) \\
\mathcal{R}(\forall\alpha :: \kappa.\ \tau)(v) &\triangleq \exists e.\ v = \Lambda.\ e \wedge \forall \mu \in [\![\kappa]\!](\cdot).\ \mathrm{good}(\mu) \to \triangleright\, \mathcal{E}(\mathcal{R}([\![\tau]\!]_{[\alpha\mapsto\mu]}))(e) \\
\mathcal{R}(\exists\alpha :: \kappa.\ \tau)(v) &\triangleq \exists v'.\ v = \mathsf{pack}\, v' \wedge \exists \mu \in [\![\kappa]\!](\cdot).\ \mathrm{good}(\mu) \wedge \triangleright\, \mathcal{R}([\![\tau]\!]_{[\alpha\mapsto\mu]})(v') \\
\mathcal{R}((\mu\alpha :: \kappa.\ \tau)\ \overline{\sigma})(v) &\triangleq \exists v'.\ v = \mathsf{roll}\, v' \wedge \triangleright\, \mathcal{R}([\![(\tau[\mu\alpha :: \kappa.\ \tau/\alpha])\ \overline{\sigma}]\!].)(v') \\
\mathcal{R}(\chi \to \nu)(v) &\triangleq \chi\,\mathsf{true} \to \mathcal{E}(\mathcal{R}(\nu))(v\ \bullet) \\
\mathcal{R}(\chi \times \nu)(v) &\triangleq \exists v'.v = \langle \bullet, v'\rangle \wedge \chi\,\mathsf{true} \wedge \mathcal{R}(\nu)(v')
\end{aligned}
$$

Fig. 16. The interpretation of closed, neutral ground types as predicates and an evaluation closure; both notions are defined via guarded recursion.

$$
\begin{aligned}
[\![\Phi]\!]_\eta\,\mathsf{true} &\triangleq \forall \varphi \in \Phi.\ [\![\varphi]\!]_\eta\,\mathsf{true} \\
[\![\Gamma]\!]_\eta &\triangleq \left\{\gamma \in \mathrm{dom}(\Gamma) \to \mathrm{Val} \;\middle|\; \forall x \in \mathrm{dom}(\Gamma).\ \mathcal{R}([\![\Gamma(x)]\!]_\eta)(\gamma(x))\right\} \\
\Delta \mid \Phi \mid \Gamma \models v : \tau &\triangleq \forall \eta \in [\![\Delta]\!](\cdot).\ \mathrm{good}(\eta) \to [\![\Phi]\!]_\eta\,\mathsf{true} \to \forall \gamma \in [\![\Gamma]\!]_\eta \to \mathcal{R}([\![\tau]\!]_\eta)(v[\gamma]) \\
\Delta \mid \Phi \mid \Gamma \models e : \tau &\triangleq \forall \eta \in [\![\Delta]\!](\cdot).\ \mathrm{good}(\eta) \to [\![\Phi]\!]_\eta\,\mathsf{true} \to \forall \gamma \in [\![\Gamma]\!]_\eta \to \mathcal{E}(\mathcal{R}([\![\tau]\!]_\eta))(e[\gamma])
\end{aligned}
$$

Fig. 17. The definition of the logical relation.

can be naturally extended to the binary case, with closed, neutral ground types interpreted as step-indexed relations rather than predicates, the interpretation has an important shortcoming. The limiting factor stems from the fact that all our semantic types can be reified as (a subset of) *well formed, syntactic types*. Thus, the universally or existentially quantified variables can only be instantiated with such types. (Concretely, for universal types, *e.g.*, the relation τ in Figure 16 is in $[\![\kappa]\!](\cdot)$.) In the binary case, this clearly prevents us from reasoning about representation independence for abstract data types, as we are obliged to *choose* the representation once and for all — and even in the unary case, it prevents us from proving safety of a (syntactically) ill-typed implementation of an interface by semantic means. In effect, by *allowing* the sophisticated equational reasoning with types whose syntactic structure (or part of it) must be known, we have severely circumscribed the model. So the question is: Can we relax these constraints for types whose syntactic structure we need not know and then obtain a more powerful semantic model supporting reasoning about semantic typing (in the unary case) and representation independence (in the binary case)? The answer is yes, as we now explain.

$$\frac{\varphi : \mathrm{Val}^2 \to \mathit{iProp}}{\varphi : \mathcal{U}_{\mathsf{T}}^{\natural\Delta}} \qquad \frac{c :: \kappa}{c : \mathcal{U}_{\kappa}^{\natural\Delta}} \qquad \frac{x :: \kappa \in \Delta}{x : \mathcal{U}_{\kappa}^{\natural\Delta}} \qquad \frac{\nu : \mathcal{U}_{\kappa_a \Rightarrow \kappa_r}^{\natural\Delta} \qquad \mu : \mathcal{U}_{\kappa_a}^{\Delta}}{\nu\ \mu : \mathcal{U}_{\kappa_r}^{\natural\Delta}}$$

$$\frac{\chi : \mathcal{U}_{\mathsf{C}}^{\Delta} \qquad \nu : \mathcal{U}_{\mathsf{T}}^{\natural\Delta}}{\chi \to \nu : \mathcal{U}_{\mathsf{T}}^{\natural\Delta}} \qquad \frac{\chi : \mathcal{U}_{\mathsf{C}}^{\Delta} \qquad \nu : \mathcal{U}_{\mathsf{T}}^{\natural\Delta}}{\chi \times \nu : \mathcal{U}_{\mathsf{T}}^{\natural\Delta}}$$

$$\frac{\nu : \mathcal{U}_{\mathsf{T}}^{\natural\Delta}}{\nu : \mathcal{U}_{\mathsf{T}}^{\Delta}} \qquad \frac{\mu : \mathcal{U}_{\kappa_r}^{\Delta, \alpha :: \kappa_a}}{\lambda \alpha :: \kappa_a.\ \mu : \mathcal{U}_{\kappa_a \Rightarrow \kappa_r}^{\Delta}} \qquad \frac{\left(\mu_i : \mathcal{U}_{\kappa}^{\Delta}\right)_{i \in \{1,2\}}}{\vdash \mu_1 \equiv_{\kappa} \mu_2 : \mathcal{U}_{\mathsf{C}}^{\Delta}}$$

Fig. 18. Open universes, generalizing the neutral and normal forms of types and normal constraints. As with normal forms, we define three universes by mutual induction: the *neutral* universe $\mathcal{U}_{\kappa}^{\natural\Delta}$, the *normal* universe $\mathcal{U}_{\kappa}^{\Delta}$ and the universe of *normal constraints* $\mathcal{U}_{\mathsf{C}}^{\Delta}$; as before, the functorial action is given by syntactic renaming.

5.3 A Relaxed Model for Representation Independence

We now show that the NbE model *can* be relaxed — albeit by relinquishing the canonical property that the inductively defined "normal forms" are a subset of well-formed types. Thus, rather than defining the neutral and normal forms of types, we define *open universes* (still split into neutral and normal) at each kind. These universes, presented in Figure 18, are largely as before, since we still need to enforce injectivity of our type constructors, except that the neutral universe at ground kind also contains the entire space of uniform relations on closed values! These are considered distinct from all the other, more syntactic, members of the universe and are considered equal when equivalent as (uniform) relations. Note that our universes remain functorial with respect to the type variable contexts, as the relations are inert, unchanging under the context morphism action.

We can now replay the same construction as presented in the previous section, with one key difference: where the reinterpretation of semantic types used to piggyback on the (implicit) coercion of neutral and normal forms to well-formed types, we cannot do so here. This, however, is not a problem: we replay the definition of Figure 14 twice: once for types (targeting our open universes), and once for elements of the universe themselves. An analogue of the logical relation in Figure 15 then lets us establish that Lemma 5.7 holds in this interpretation as well.

With the generalized NbE-style model, we can now give its realization (for closed, ground, neutral universe) as relations on values. The definition is presented in Figure 19; the construction proceeds in an entirely analogous fashion, with the main difference being the interpretation of the relation φ — which is simply realized as itself. The other minor difference is that the components of the recursive type have to be interpreted on their own and applied as the elements of the semantic interpretation, rather than coerced into a single type; this does not lead to any significant differences.

The remainder of the interpretation is standard: we interpret open types by quantifying over an environment of closing environments. The difference with respect to the simpler model is that the interpretations for open type variables (as well as the ones in the realizability interpretation of universal and existential quantifiers) now allow a closed relation on values to be picked. This allows picking safe-but-syntactically-ill-typed implementations (in the unary case), as well as implementations that relate different implementations (in the binary case), thus allowing us to recover some data abstraction capacities. Note that for this to be viable, the context cannot depend on the structure of the picked interpretation (through constraints): this is the cost of working in

$$
\begin{aligned}
\mathcal{E}(P)(e_1, e_2) &\triangleq (\exists v_1, v_2.\ e_1 = v_1 \wedge e_2 \twoheadrightarrow^* v_2 \wedge P(v_1, v_2)) \vee \\
&\quad (\exists e_1', e_2'.\ e_1 \to e_1' \wedge e_2 \twoheadrightarrow^* e_2' \wedge \triangleright\, \mathcal{E}(P)(e_1', e_2')) \\
\mathcal{R}(\varphi)(u, v) &\triangleq \varphi(u, v) \\
\mathcal{R}(\mathsf{unit})(u, v) &\triangleq u = v = \langle\rangle \\
\mathcal{R}(\mathsf{void})(u, v) &\triangleq \bot \\
\mathcal{R}(\nu_1 \times \nu_2)(u, v) &\triangleq \exists u_1, u_2, v_1, v_2.\ u = \langle u_1, u_2\rangle \wedge v = \langle v_1, v_2\rangle \wedge \bigwedge_{i \in \{1,2\}} \mathcal{R}(\nu_i)(u_i, v_i) \\
\mathcal{R}(\nu_1 + \nu_2)(u, v) &\triangleq \bigvee_{i \in \{1,2\}} \exists u', v'.\ u = \mathsf{inj}_i\, u' \wedge v = \mathsf{inj}_i\, v' \wedge \mathcal{R}(\nu_i)(u', v') \\
\mathcal{R}(\nu_1 \to \nu_2)(u, v) &\triangleq \forall u', v'.\ \mathcal{R}(\nu_1)(u', v') \to \mathcal{E}(\mathcal{R}(\nu_2))(u\ u', v\ v') \\
\mathcal{R}(\forall \alpha :: \kappa.\ \tau)(u, v) &\triangleq \exists e, e'.\ u = \Lambda.\ e \wedge v = \Lambda.\ e' \wedge \\
&\qquad \forall \mu \in [\![\kappa]\!](\cdot).\ \mathrm{good}(\mu) \to \triangleright\, \mathcal{E}(\mathcal{R}([\![\tau]\!]_{[\alpha \mapsto \mu]}))(e, e') \\
\mathcal{R}(\exists \alpha :: \kappa.\ \tau)(u, v) &\triangleq \exists u', v'.\ u = \mathsf{pack}\, u' \wedge v = \mathsf{pack}\, v' \wedge \\
&\qquad \exists \mu \in [\![\kappa]\!](\cdot).\ \mathrm{good}(\mu) \wedge \triangleright\, \mathcal{R}([\![\tau]\!]_{[\alpha \mapsto \mu]})(u', v') \\
\mathcal{R}(\mu_\kappa\ \varphi\ \overline{\sigma})(u, v) &\triangleq \exists u', v'.\ u = \mathsf{roll}\, u' \wedge v = \mathsf{roll}\, v' \wedge \triangleright\, \mathcal{R}([\![\varphi]\!].\ [\![\mu_\kappa\ \varphi]\!].\ \overline{[\![\sigma]\!].})(u', v') \\
\mathcal{R}(\chi \to \nu)(u, v) &\triangleq \chi\, \mathsf{true} \to \mathcal{E}(\mathcal{R}(\nu))(u\ \bullet, v\ \bullet) \\
\mathcal{R}(\chi \times \nu)(u, v) &\triangleq \exists u', v'. u = \langle \bullet, u'\rangle \wedge v = \langle \bullet, v'\rangle \wedge \chi\, \mathsf{true} \wedge \mathcal{R}(\nu)(u', v')
\end{aligned}
$$

Fig. 19. The realization of $\mathcal{U}_{\mathsf{T}}^{\natural \cdot}$ as relations on values.

an environment that can reason about the types and depend on non-trivial information about their structure.

Representation independence for non-empty lists. To illustrate that our model supports reasoning about representation independence, we provide a small synthetic example, in which we show contextual equivalence of two implementations of an existentially typed module for non-empty lists of natural numbers, with operations for obtaining the head of a list, constructing a list with a single element, and for inserting an element into a list:

$$\mathsf{nelist} \triangleq \exists \alpha :: \mathsf{T}.\ (\alpha \to \mathbb{N}) \times (\mathbb{N} \to \alpha) \times (\mathbb{N} \to \alpha \to \alpha)$$

The two implementations we consider are lists of natural numbers and vectors of natural numbers. The vectors are implemented as a GADT; they are indexed by their size, which we represent as type-level natural numbers. The implementation details can be found in Figure 20. It is worth noting that the GADT enables the definition of a total head operation for vectors.

We use our logical relation to show that two implementations $I_1 = \mathsf{pack}\,\langle \mathsf{head}, \langle \mathsf{inj}, \mathsf{cons}\rangle\rangle$ and $I_2 = \mathsf{pack}\,\langle \mathsf{vhead}, \langle \mathsf{vinj}, \mathsf{vcons}\rangle\rangle$ are contextually equivalent[8] at the type nelist. The key step in the proof is to choose an appropriate relation for the existentially quantified type variable α in the nelist type. We use the following relation, which specifies that both lists must be non-empty, and their respective head elements must be related as natural numbers:

$$\{(v_1, v_2) \mid v_1 = \mathsf{roll}\,\mathsf{inj}_2\,\langle u_1, u_2\rangle \wedge v_2 = \mathsf{pack}\,\mathsf{roll}\,\mathsf{inj}_2\,\langle u_3, u_4\rangle \wedge \mathcal{R}[\![\mathbb{N}]\!](u_1, u_3)\} \subseteq \mathrm{Val} \times \mathrm{Val}.$$

[8]For brevity, we have not included the standard argument showing that logical relatedness implies contextual approximation; it is included in the Coq formalization. To show contextual equivalence, we show logical relatedness in both directions.

```
natlist :: T
natlist ≜ μα :: T. unit + (ℕ × α)

cons : ℕ → natlist → natlist
cons x xs ≜ roll inj₂ ⟨x, xs⟩

inj : ℕ → natlist
inj x ≜ cons x (roll inj₁ ⟨⟩)

head : natlist → ℕ
head xs ≜
  case unroll xs
  | inj₁ _. diverge
  | inj₂ ⟨y, _⟩. y

natvec :: T ⇒ T
natvec ≜
  μφ :: T ⇒ T. λα :: T.
  ((α ≡_T void) × unit)
  + (ℕ × ∃β :: T. (α ≡_T (β + unit)) × (φ β))

nenatvec :: T
nenatvec ≜ ∃α :: T. natvec (α + unit)

vcons : ℕ → nenatvec → nenatvec
vcons x xs ≜
  let (∗, ys) = xs in
  pack roll inj₂ ⟨x, pack ⟨•, ys⟩⟩

vinj : ℕ → nenatvec
vinj x ≜ vcons x roll inj₁ ⟨•, ⟨⟩⟩

vhead : nenatvec → ℕ
vhead xs ≜
  let (∗, ys) = xs in
  case unroll ys
  | inj₁ (•, w). abort •
  | inj₂ ⟨y, _⟩. y
```

Fig. 20. $\mathsf{F}^{=i}_{\omega\mu}$-encoding of lists and non-empty type-indexed vectors. We use syntactic sugar throughout.

Parametricity. Our model is also strong enough to encompass a number of the classic consequences of parametricity. To demonstrate that, we present a simple free theorem: we show that any value v of type $\forall\alpha :: \mathsf{T}.\ \alpha \to \alpha$ approximates the identity function $\Lambda.\ \lambda x.\ x$.

Proof. It suffices to show that for any $\eta \in [\![\Delta]\!](\cdot)$ such that good(η) and $[\![\Phi]\!]_\eta$ hold, and any substitutions $(\gamma, \gamma') \in [\![\Gamma]\!]_\eta$,

$$\mathcal{R}([\![\forall\alpha :: \mathsf{T}.\ \alpha \to \alpha]\!])_\eta(v[\gamma], (\Lambda.\ \lambda x.\ x)[\gamma']).$$

Given the assumption that $\Delta \mid \Phi \mid \Gamma \vdash v : \forall\alpha :: \mathsf{T}.\ \alpha \to \alpha$, by the fundamental lemma we get $\mathcal{R}([\![\forall\alpha :: \mathsf{T}.\ \alpha \to \alpha]\!])_\eta(v[\gamma], v[\gamma'])$. By the definition of the value interpretation for universal types, $v[\gamma] = \Lambda.\ e$ and $v[\gamma'] = \Lambda.\ e'$. Moreover, $\forall\mu.\triangleright\mathcal{E}(\mathcal{R}([\![\alpha \to \alpha]\!]_{\eta,\alpha\mapsto\mu}))(e, e')$. We instantiate this interpretation with an empty relation. From that we can conclude that either e diverges, in which case the statement trivially holds, or e terminates at a value f. If e terminates at f, then, it suffices to prove the following statement:

$$\forall u\ v\ \mu.\triangleright(\mathcal{R}([\![\alpha]\!]_{\eta,\alpha\mapsto\mu})(u, v) \to \mathcal{E}(\mathcal{R}([\![\alpha]\!]_{\eta,\alpha\mapsto\mu}))(f\ u, v)).$$

We instantiate the interpretation from the fundamental lemma with a singleton relation $\{(u, v)\} \subseteq \mathrm{Val} \times \mathrm{Val}$. From that and determinism of operational semantics, we can conclude that either $t \triangleq f\ u$ terminates or diverges. If it terminates, the result is equal to u by our choice of the relation, in which case the statement trivially holds. If t diverges, then the statement vacuously holds. □

Note that the proof follows the standard pattern, thus providing evidence that the known results can be transferred to a calculus with GADTs. While the example we show here does not explicitly use GADTs, the approach scales: the usual theorems, such as map-map fusion or the free theorems of Wadler [1989], can be proved not just for lists, but also types that do use equality constraints,

such as vectors or perfect binary trees indexed with type-encoded depth. Note, however, that these properties reflect the payload type of the data structure (which behaves parametrically, even though the structure itself uses equality constraints), rather than the index encoding length or depth, which is much more constrained.

Power and limitations of the relational model. As the examples above demonstrate, our model is powerful enough to transport parametricity and representation independence results from simpler systems, without internalized, injective equalities. It is natural to ask whether *all* such results can be preserved in our model, and whether GADTs introduce *new* opportunities for exploiting parametricity.

For the first question, we are aware of certain limitations of our current model. These stem from the fact that we only allow inert relations as semantic elements of our open universes. Thus, we have no way of treating higher-kinded parameters semantically. Thus, for instance, we cannot use an ill-typed but well-behaved implementation of lists as a type constructor of kind $\mathsf{T} \Rightarrow \mathsf{T}$ (over the type of elements). In contrast, concrete instantiation of such a list constructed at known element type do fit into the setup as presented. Whether this limitation can be lifted is one of the questions left for future work.

As for the interaction of GADTs with parametricity, it is clear that type constructors with equality-constrained indices cannot be parametric in those indices: this much is enforced by the model. However, this does not mean that GADTs as a feature are orthogonal to representation independence, free theorems, etc.: the fact that they *can* be used as implementations of abstract data types, used as instances of free theorems, etc., with correct behavior enforced by virtue of the type system is a consequence of the model we have presented in this section.

6 FORMALIZATION

With the exception of the final part of the non-expressibility theorem in Section 3, all the constructions presented in this paper have been formalized in the Coq proof assistant. In this section we discuss some aspects of the formalization that we believe are of wider interest, either from a technical or a methodological standpoint.

6.1 Internal Languages *vs.* Explicit Presheaves

In the informal presentation of our results, we have made liberal use of the *internal language* of presheaf categories where possible. For instance, many important constructions involved in our NbE–style model lie in the category $\mathrm{Pr}(\mathcal{K})$ of presheaves on the category $\mathcal{K}$ of kind contexts and kind-preserving renamings. The most direct and intuitive presentation of these constructions, then, uses the internal language of $\mathcal{K}$ as a (locally) cartesian closed category. The use of the internal language is well-established as a way to avoid the proliferation of administrative parameters (quantifying over future worlds, renamings, *etc.*) that routinely obscure what are in essence simple constructions based on simple ideas.

The downside of our use of the internal language is that different parts of our development must take place in different categories (languages), corresponding to the boundary between the $\mathrm{Pr}(\mathcal{K})$ and **Set**. Whereas informal mathematical practice is highly optimized for working rigorously with such abstraction boundaries, the same does not currently hold of proof assistants like Coq which do not yet support mixing different type theories in the same development. For this reason, our Coq formalization explicitly unfolds the internal constructions to external ones; this process introduces many additional verification conditions (*e.g.* naturality laws, *etc.*) that are automatic in the internal presentation, but which we had to check explicitly in our Coq development.

6.2 Representation of Terms and Binders

The problem of representing abstract syntax with binding in theorem provers spawned a rich literature, whose review is beyond the scope of this paper. Here, we briefly note that the approach we use is inspired by Fiore et al.'s [1999] algebraic view of abstract syntax and variable binding. In particular, both terms and types are constructed as *presheaves*, *i.e.* Type-valued functors on a category of renamings. This is achieved by parameterizing the Coq type of terms with a *type* of its variables, and — through the use of an appropriate library of typeclasses — equipping the resulting type constructor with a functorial structure. We then equip the functor with a monadic structure that encodes substitution.

6.3 Normalization by Evaluation: Methodological and Technical Features

The NbE implementation is the foundation of most of the technical developments of the paper: both the syntactic, progress-and-preservation proof of type soundness and the model construction rely on a variant of the procedure. It is, however, also interesting from the technical, proof-engineering point of view — we leverage the representation of syntax based on presheaves over names to interpret types as presheaves. This has the advantage of baking a lot of the requisite structure of types and their normal forms into the construction, and thus removing a large part of the reasoning based on partial equivalence relation that are necessary in less precise encodings in order to prune the semantic spaces of ill-behaved functions. Compare for instance the work of Allais et al. [2017], whose interpretation of types, while similar at object level, does not require functoriality: while their implementation may be simpler, it requires significantly more post-hoc reasoning, some of which is unnecessary if we interpret the types into spaces with richer structure. Thus, while the particular NbE implementation we consider is relatively simple, and the categorical notions that we utilize are fairly well-known in this context [Fiore 2002], we believe that our formalization goes further towards an NbE that is correct by construction than the state of the art.

6.4 Relational Models and Step-Indexed Logics

The logic used in Section 5 is not the standard Coq logic; in particular, it internalizes the step-indexed methods through the use of the later operator and the Löb induction to define and reason about predicates. Thus we had to choose a framework that allows reasoning in such a logic.

Since the requirements of the model were only related to constructing relations via guarded recursion, we decided to use Polesiuk's IxFree library [Polesiuk 2017], which provides features roughly equivalent to the LSLR logic [Dreyer et al. 2011]. Alternative choices are available, however: one is to work in an axiomatic extension of Coq's type theory, like Sterling and Harper [2018] — although this does come at a price of the final soundness results being expressed in a different type theory. The other obvious candidate is the Iris framework [Jung et al. 2018]: its power to construct solutions of recursive domain equations (via its invariants feature) would be useful in extending the calculus to include higher-order mutable state. We chose against using it, for this project, due to its higher complexity, and the fact that in a simpler logic, such as the one provided by IxFree, we can make a much more direct use of the host logic (*i.e.* Coq's) proof manipulation primitives.

7 RELATED WORK

Of the many calculi that were developed to account for GADTs and similar features, the one most closely related to our work is System F_C [Sulzmann et al. 2007; Weirich et al. 2013]; the calculus developed as an intermediate language for the Glasgow Haskell Compiler. While it accounts for features that we do not handle, such as type functions and equality axioms, which are part of GHC, the treatment of GADTs is much closer to the surface language, through explicitly defined type

constructors and their constructors on term level. The presence of equality constraints leads to similar problems in the syntactic type soundness, and necessitates considering the problem of consistency of the equational theory. However, while the calculus is perfectly well suited as an intermediate language, it is less ideal for the semantic study of the GADTs.

Relational models of higher-kinded polymorphic lambda calculi have been widely studied since the 1990s [Hasegawa 1994; Robinson and Rosolini 1994]. Two examples of interest include the formalized developments of Atkey [2012] and Vytiniotis and Weirich [2010]. Atkey formalizes pure System F_ω, but studies type equality expressible within the system, and inductive types that arise as initial algebras of positive functors. However, the notion of functoriality he uses is external to the calculus, and the encodings of GADTs he obtains through the encoding of Johann and Ghani [2008] does not seem to allow discrimination based on distinct types. Instead, he extends F_ω with additional kinds, including type-level natural numbers — which can express some GADT patterns.

Vytiniotis and Weirich build a relational model of an extension of System F_ω with a single built-in GADT [Vytiniotis and Weirich 2010], and show parametricity properties of this language. We explore a larger class of programs that can contain arbitrary GADTs, including the runtime-type representation type R. The nontermination of their System R_ω when the representation type is extended to include universal quantifiers is likely related to our example in Section 2, although the details of the construction differ somewhat. An interesting feature of their model is the inclusion of syntactic information in the interpretation of types and type equivalence, which is quite different from our two-stage interpretation. However, since the closure under type equivalence is only enforced post-hoc, it is not likely that their construction could be scaled to all type constructors being injective, rather than just the built-in representation GADT.

The most recent line of work on semantics and parametricity of GADTs comes from Johann et al. [Johann and Cagne 2022; Johann and Ghiorzi 2021, 2022; Johann et al. 2021]. These are categorical models for languages where type constructors are considered functorial, and inductive types can be formed out of strictly positive functors, as appropriate initial algebras. Our calculus is somewhat more modest, in that the type constructors have no additional structure; on the other hand, we consider an impredicative system with general recursive types, and thus our approach to obtaining relational models must be somewhat different.

A complementary line of work on the syntax and elaboration of GADTs is that of Dunfield and Krishnaswami [2019], who have given a proof-theoretic account of equality constraints in terms of the Girard–Schroeder-Heister elimination rule for equality [Girard 1992; Schroeder-Heister 1994], which says that $\Gamma, c \equiv_\kappa d \vdash \mathcal{J}$ holds if and only if $\theta^*\Gamma \vdash \theta^*\mathcal{J}$ holds for all substitutions θ in the complete set of unifiers for $c, d :: \kappa$. Unifiability of two constructors is a syntactic and meta-level concept that does not correspond to any behavior of semantic models; indeed, most languages have constructor injectivity as an admissible rule, but it is a special feature of GADTs for these injectivity laws to be *derivable* and thus required in models. Dunfield and Krishnaswami [2019] achieve these laws all at once by incorporating syntactic unifiability into their formal theory; we achieve something analogous in our setting by means of explicit rules. This choice is important because our main results pertain to semantic models, a viewpoint that is not readily accessed from the Girard–Schroeder-Heister perspective.

8 CONCLUSION

In this work, we developed a core calculus that can be used for semantic study of generalized algebraic data types. We show that the calculus is expressive enough to encode many of the commonly used programming patterns, and that it is strictly more expressive than calculi that do not enforce injectivity and discriminability of constant type constructors. We prove that the calculus is type-safe, and build a novel, two-stage relational model that uses a variation on the

normalization-by-evaluation construction to enforce the injectivity and discriminability rules. The model is strong enough to allow for data-abstraction reasoning, the first such for a calculus equipped with GADTs.

Several directions of future work are apparent. First, integrating other advanced language features, such as higher-order mutable state or effect handlers, within our calculus and its model would be an interesting direction that could lead to models that account for virtually all main features of modern functional programming languages; the difficulties in modeling these seem largely orthogonal to those we faced in the current work, so such an extension should be manageable. Second, it seems that even with the inclusion of "inert" step-indexed predicates in the universe, the model is not robust enough to ensure termination of the sub-calculus with restricted injectivity and no recursive types, System $\mathrm{F}_{\omega}^{=i}$ — a property that, we believe, ought to hold. Therefore, we would like to further refine our model, so that it would allow for a more fine-tuned interpretation of non-injective quantifiers, as well as more refined data-abstraction principles. Finally, a possible direction would consider adapting the functorial approach of Johann *et al.* to a setting with a wider space of type constructors. This could allow us to enrich practical programming languages with type constructors that have the mapping action inherently associated with them, including the ability to quantify over such constructs.

ACKNOWLEDGMENTS

We would like to thank a number of our colleagues for discussion and suggestions that helped us develop this work. In particular, Jacques Garrigue helped us with understanding the state of the GADT implementation in OCaml, Piotr Polesiuk suggested the use of the propositional model for inexpressibility of injective constraints, Daniel Gratzer and James McKinna provided useful suggestions regarding the model, and Amin Timany – useful discussions throughout the project. The anonymous reviewers provided many useful comments that helped improve the paper.

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation. Jonathan Sterling is funded by the European Union under the Marie Skłodowska-Curie Actions Postdoctoral Fellowship project TypeSynth: synthetic methods in program verification. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the granting authority can be held responsible for them.

DATA-AVAILABILITY STATEMENT

The formalisation associated with this paper, which covers all the main results except for Theorem 3.2 is available online [Sieczkowski et al. 2023]. For reproduction purposes, we note that the formalization was checked with Coq v.8.16.1.

REFERENCES

Guillaume Allais, James Chapman, Conor McBride, and James McKinna. 2017. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, 195–207. https://doi.org/10.1145/3018610.3018613

Robert Atkey. 2012. Relational Parametricity for Higher Kinds. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France (LIPIcs, Vol. 16)*, Patrick Cégielski and Arnaud Durand (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 46–61. https://doi.org/10.4230/LIPIcs.CSL.2012.46

H. P. Barendregt. 1993. *Lambda Calculi with Types.* Oxford University Press, Inc., USA, 117–309.

Ulrich Berger and Helmut Schwichtenberg. 1991. An Inverse of the Evaluation Functional for Typed lambda-calculus. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July*

15-18, 1991. IEEE Computer Society, 203–211. https://doi.org/10.1109/LICS.1991.151645

Richard S. Bird and Lambert G. L. T. Meertens. 1998. Nested Datatypes. In *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1422)*, Johan Jeuring (Ed.). Springer, 52–67. https://doi.org/10.1007/BFb0054285

James Cheney and Ralf Hinze. 2003. First-Class Phantom Types.

Olivier Danvy. 1996. Type-Directed Partial Evaluation. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 242–257. https://doi.org/10.1145/237721.237784

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical Step-Indexed Logical Relations. *Log. Methods Comput. Sci.* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:16)2011

Jana Dunfield and Neelakantan R. Krishnaswami. 2019. Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed Types. *Proc. ACM Program. Lang.* 3, POPL, Article 9 (jan 2019), 28 pages. https://doi.org/10.1145/3290322

Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Sci. Comput. Program.* 17, 1-3 (1991), 35–75. https://doi.org/10.1016/0167-6423(91)90036-W

Marcelo P. Fiore. 2002. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proceedings of the 4th international ACM SIGPLAN conference on Principles and practice of declarative programming, October 6-8, 2002, Pittsburgh, PA, USA (Affiliated with PLI 2002)*. ACM, 26–37. https://doi.org/10.1145/571157.571161

Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. 1999. Abstract Syntax and Variable Binding. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 193–202. https://doi.org/10.1109/LICS.1999.782615

Jean-Yves Girard. 1992. A Fixpoint Theorem in Linear Logic. Post to Linear Logic mailing list, http://www.seas.upenn.edu/~sweirich/types/archive/1992/msg00030.html..

Ryu Hasegawa. 1994. Categorical Data Types in Parametric Polymorphism. *Math. Struct. Comput. Sci.* 4, 1 (1994), 71–109. https://doi.org/10.1017/S0960129500000372

Patricia Johann and Pierre Cagne. 2022. How Functorial Are (Deep) GADTs? *CoRR* abs/2203.14891 (2022). https://doi.org/10.48550/arXiv.2203.14891 arXiv:2203.14891

Patricia Johann and Neil Ghani. 2008. Foundations for structured programming with GADTs. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 297–308. https://doi.org/10.1145/1328438.1328475

Patricia Johann and Enrico Ghiorzi. 2021. Parametricity for Nested Types and GADTs. *Log. Methods Comput. Sci.* 17, 4 (2021). https://doi.org/10.46298/lmcs-17(4:23)2021

Patricia Johann and Enrico Ghiorzi. 2022. (Deep) induction rules for GADTs. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, Andrei Popescu and Steve Zdancewic (Eds.). ACM, 324–337. https://doi.org/10.1145/3497775.3503680

Patricia Johann, Enrico Ghiorzi, and Daniel Jeffries. 2021. GADTs, Functoriality, Parametricity: Pick Two. In *Proceedings 16th Logical and Semantic Frameworks with Applications, LSFA 2021, Buenos Aires, Argentina (Online), 23rd - 24th July, 2021 (EPTCS, Vol. 357)*, Mauricio Ayala-Rincón and Eduardo Bonelli (Eds.). 77–92. https://doi.org/10.4204/EPTCS.357.6

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, John H. Reppy and Julia Lawall (Eds.). ACM, 50–61. https://doi.org/10.1145/1159803.1159811

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Inf. Comput.* 185, 2 (2003), 182–210. https://doi.org/10.1016/S0890-5401(03)00088-9

Saunders Mac Lane and Ieke Moerdijk. 1992. *Sheaves in geometry and logic: a first introduction to topos theory*. Springer.

Christine Paulin-Mohring. 1993. Inductive Definitions in the system Coq - Rules and Properties. In *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 664)*, Marc Bezem and Jan Friso Groote (Eds.). Springer, 328–345. https://doi.org/10.1007/BFb0037116

Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

Piotr Polesiuk. 2017. IxFree: Step-indexed logical relations in Coq. In *3 rd International Workshop on Coq for Programming Languages (CoqPL)*.

François Pottier and Yann Régis-Gianas. 2006. Stratified type inference for generalized algebraic data types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 232–244. https://doi.org/10.1145/1111037.1111058

Edmund P. Robinson and Giuseppe Rosolini. 1994. Reflexive Graphs and Parametric Polymorphism. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*. IEEE Computer Society, 364–371. https://doi.org/10.1109/LICS.1994.316053

Peter Schroeder-Heister. 1994. Definitional reflection and the completion. In *Extensions of Logic Programming*, Roy Dyckhoff (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 333–347.

Tim Sheard and Emir Pasalic. 2008. Meta-programming With Built-in Type Equality. *Electron. Notes Theor. Comput. Sci.* 199 (2008), 49–65. https://doi.org/10.1016/j.entcs.2007.11.012

Filip Sieczkowski, Sergei Stepanenko, Jonathan Sterling, and Lars Birkedal. 2023. *The Essence of Generalized Algebraic Data Types*. https://doi.org/10.5281/zenodo.10040534

Jonathan Sterling and Robert Harper. 2018. Guarded Computational Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. ACM, New York, NY, USA, 879–888. https://doi.org/10.1145/3209108.3209153

Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, François Pottier and George C. Necula (Eds.). ACM, 53–66. https://doi.org/10.1145/1190315.1190324

Dimitrios Vytiniotis and Stephanie Weirich. 2010. Parametricity, type equality, and higher-order polymorphism. *J. Funct. Program.* 20, 2 (2010), 175–210. https://doi.org/10.1017/S0956796810000079

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, Joseph E. Stoy (Ed.). ACM, 347–359. https://doi.org/10.1145/99370.99404

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*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 275–286. https://doi.org/10.1145/2500365.2500599

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 224–235. https://doi.org/10.1145/604131.604150

Received 2023-07-11; accepted 2023-11-07