Higher-Order Symb Comput (2006) 19:125–162
DOI 10.1007/s10990-006-8611-7

Polymorphic typed defunctionalization and concretization

François Pottier · Nadji Gauthier

Abstract *Defunctionalization* is a program transformation that eliminates functions as first-class values. We show that defunctionalization can be viewed as a *type-preserving* transformation of an extension of System F with *guarded algebraic data types* into itself. We also suggest that defunctionalization is an instance of *concretization*, a more general technique that allows eliminating constructs other than functions. We illustrate this point by presenting two new type-preserving transformations that can be viewed as instances of concretization. One eliminates Rémy-style polymorphic records; the other eliminates the dictionary records introduced by the standard compilation scheme for Haskell's type classes.

Keywords Defunctionalization · Closure conversion · Polymorphism · Type-preserving compilation · Concretization · Polymorphic records · Dictionary records · Type classes

1. Introduction

Defunctionalization is a program transformation that aims to turn a higher-order functional program into a first-order one, that is, to eliminate the use of functions as first-class values. It was discovered in 1972 by Reynolds [34, 35] and, in the realm of logic programming, later rediscovered by Warren [41].

Here is an informal and low-level description of defunctionalization. Under the assumption that the entire source program is available, a distinct tag is associated with every λ-abstraction in the source program, or, in other words, with every code block in the compiled program. Then, a function value is represented by a *closure* composed of the *tag* associated with its code and of a *value environment*. The generic code in charge of function application, which we refer to as *apply* in the following, examines the tag via case analysis and performs a direct jump to the appropriate code block, making the contents of the value environment available to it. Defunctionalization is a close cousin of *closure conversion*, where closures are composed

This is a revised and extended version of an earlier conference paper [30].

F. Pottier · N. Gauthier
INRIA, BP 105, F-78153 Le Chesnay Cedex, France

of a *code pointer* and a value environment. In fact, to a certain extent, closure conversion may be viewed as a particular implementation of defunctionalization, where tags happen to be code pointers, and case analysis of a tag is replaced with a single indirect jump. One reported advantage of defunctionalization over closure conversion is that, due to the idiosyncrasies of branch prediction on modern processors, the cost of an indirect jump may exceed that of a simple case analysis followed by a direct jump. Furthermore, the use of direct jumps opens up numerous opportunities for inlining [4, 45]. One disadvantage of defunctionalization is that it is a whole program transformation, because defining *apply* requires knowing about all of the functions that exist in the program. We come back to this issue in Section 10.

1.1. Closure conversion versus defunctionalization in a typed setting

Even though defunctionalization and closure conversion appear conceptually very close, they differ when viewed as transformations over *typed* programs. Minamide et al. [23] have shown how to view closure conversion as a type-preserving transformation. There, the type of a closure is a pair of a first-order function type (for the code pointer) and of a record type (for the value environment), wrapped together within an *existential* quantifier, so that closures whose value environments have different structure may still receive identical types. Minamide *et al.* deal with both a simply-typed and a type-passing polymorphic λ-calculi. The case of a type-erasure polymorphic λ-calculus has been addressed by Morrisett et al. [25].

Defunctionalization, on the other hand, has been studied mainly in a *simply-typed* setting [1, 26]. There, closures receive *sum* types: closure construction is encoded as injection, that is, the introduction of a sum, while function application is encoded as case analysis, that is, the elimination of a sum. Furthermore, this approach requires some form of *recursive* types. One may use either structural sum types and equirecursive types, that is, the standard type constructors $+$ and μ, or *algebraic data types*, which are nominal, isorecursive sum types.

When the source language features Hindley-Milner-style polymorphism [9, 22], the source program is typically turned into a simply-typed program by applying *monomorphization* prior to defunctionalization [7, 38, 39]. However, monomorphization involves code duplication, whose cost may be difficult to control. Bell et al. [2] propose a combined algorithm that performs on-demand monomorphization during defunctionalization. This may limit the amount of duplication required, but performs identically in the worst case. In source languages with stronger forms of polymorphism, however, monomorphization becomes impossible, because an infinite amount of code duplication might be required. This is the case of ML with polymorphic recursion and of System F [15, 33]. In that case, no type-preserving definition of defunctionalization was known prior to this work. Hanus [18, 19] did design a type system that supports polymorphism and allows typechecking defunctionalized programs; we discuss it after outlining our approach.

1.2. The difficulty with polymorphism

Why is it difficult to define defunctionalization for a typed, polymorphic λ-calculus? The problem lies in the definition of *apply*, the central function that remains in the defunctionalized program, whose task is to perform dispatch based on tags. Its parameters are a closure f and a value *arg*; its task is to simulate the application of the source function encoded by f to the source value encoded by *arg*, and to return its result. In other words, if $[\![e]\!]$ denotes the image of the source expression e through defunctionalization, we intend to define $[\![e_1\, e_2]\!]$

as *apply* $[\![e_1]\!]\,[\![e_2]\!]$. Now, assume that defunctionalization is type-preserving, and that $[\![\tau]\!]$ denotes the image of the source type τ through defunctionalization. Then, if e_1 has type $\tau_1 \to \tau_2$ and e_2 has type τ_1, we find that, for *apply* $[\![e_1]\!]\,[\![e_2]\!]$ to be well-typed, *apply* must have type $[\![\tau_1 \to \tau_2]\!] \to [\![\tau_1]\!] \to [\![\tau_2]\!]$. Furthermore, because e_1 may be arbitrary, this should hold for all types τ_1 and τ_2. The most natural way to satisfy this requirement is to arrange for *apply* to have type $\forall\alpha_1\alpha_2.[\![\alpha_1 \to \alpha_2]\!] \to \alpha_1 \to \alpha_2$ and to ensure that $[\![\cdot]\!]$ commutes with substitution of types for type variables.

Now, what is the code for *apply*? It should be of the form

$$\Lambda\alpha_1.\Lambda\alpha_2.\lambda f : [\![\alpha_1 \to \alpha_2]\!].\lambda arg : \alpha_1.\mathsf{case}\, f\ \mathsf{of}\ \bar{c}$$

where $\bar{c}$ contains one clause for every tag, that is, for every λ-abstraction that appears in the source program. The right-hand side of every such clause is the body of the associated λ-abstraction, renamed so that its formal parameter is *arg*. For the sake of illustration, let us assume that the source program contains the λ-abstractions $\lambda x.x + 1$ and $\lambda x.\mathsf{not}\,x$, whose types are *int* → *int* and *bool* → *bool*, and whose tags are *succ* and *not*, respectively. (These are closed functions, so the corresponding closures have an empty value environment. This does not affect our argument.) Then, $\bar{c}$ should contain the following clauses:

$$\begin{aligned} succ &\mapsto arg + 1 \\ not &\mapsto \mathsf{not}\, arg \end{aligned}$$

However, within System F, these clauses are incompatible: they make different assumptions about the type of *arg*, and produce values of different types. In fact, for *apply* to be well-typed, every λ-abstraction in the source program must produce a value of type α_2, under the assumption that its argument is of type α_1. In the absence of any further hypotheses about α_1 and α_2, this amounts to requiring every λ-abstraction in the source program to have type $\forall\alpha_1\alpha_2.\alpha_1 \to \alpha_2$, which cannot hold in general! This explains why it does not seem possible to define a type-preserving notion of defunctionalization for System F.

1.3. The standard, limited workaround

The workaround that is commonly adopted in the simply-typed case [1, 2, 7, 26, 38, 39] consists in specializing *apply*. Instead of defining a single, polymorphic function, one introduces a family of monomorphic functions, of type $[\![\tau_1 \to \tau_2]\!] \to [\![\tau_1]\!] \to [\![\tau_2]\!]$, where τ_1 and τ_2 range over ground types. The point is that the definition of $apply_{\tau_1\to\tau_2}$ can now list only the tags whose associated λ-abstractions have type $\tau_1 \to \tau_2$. Continuing our example, the definition of $apply_{int\to int}$ should contain a case for *succ*, but none for *not*. Conversely, the definition of $apply_{bool\to bool}$ deals with *not*, but not with *succ*. It is now easy to check that all clauses in the definition of $apply_{\tau_1\to\tau_2}$ are type compatible, so that the function is well-typed. Then, exploiting the fact that e_1 must have a *ground* type of the form $\tau_1 \to \tau_2$, one defines $[\![e_1\, e_2]\!]$ as $apply_{\tau_1\to\tau_2}\,[\![e_1]\!]\,[\![e_2]\!]$. Thus, defunctionalization in a simply-typed setting is not only type-preserving, but also *type-directed*. We note that $[\![\cdot]\!]$ *does not commute* with substitution of types for type variables. Indeed, every distinct arrow type in the source program maps to a distinct algebraic data type in the target program. As a result, there is no natural way of translating non-ground arrow types. These remarks explain why this approach fails in the presence of polymorphism.

1.4. Our solution

In this paper, we suggest another way out of this problem. We keep a single *apply* function, whose type is $\forall\alpha_1\alpha_2.[\![\alpha_1 \rightarrow \alpha_2]\!] \rightarrow \alpha_1 \rightarrow \alpha_2$, as initially suggested above. We also insist that the translation of types should commute with type substitutions, so $[\![\tau_1 \rightarrow \tau_2]\!]$ must be *Arrow* $[\![\tau_1]\!]\,[\![\tau_2]\!]$, for some distinguished, binary algebraic data type constructor *Arrow*. There remains to find a suitable *extension* of System F where the definition of *apply* is well-typed, that is, where every clause does produce a value of type α_2, under the assumption that *arg* is of type α_1. The key insight is that, in order to make this possible, we must acquire further hypotheses about α_1 and α_2. For instance, in the case of the *succ* branch, we might reason as follows. If this branch is taken, then *f* is *succ*, so *succ* has type *Arrow* $\alpha_1\,\alpha_2$. However, we know that the λ-abstraction associated with the tag *succ*, namely $\lambda x.x + 1$, has type *int* → *int*, so it is natural to assign type *Arrow int int* to the data constructor *succ*. Combining these two facts, we find that, if the branch is taken, then we must have *Arrow* $\alpha_1\,\alpha_2 =$ *Arrow int int*, that is, $\alpha_1 =$ *int* and $\alpha_2 =$ *int*. *Under these extra typing hypotheses*, it is possible to prove that, if *arg* has type α_1, then *arg* + 1 has type α_2. Then, by dealing with every clause in an analogous manner, one may establish that *apply* is well-typed.

The ingredients that make this solution possible are simple. First, we need the data constructors *succ* and *not*, which are associated with the algebraic data type *Arrow*, to be assigned the specific types *Arrow int int* and *Arrow bool bool*, respectively. Note that, if *Arrow* was an ordinary algebraic data type, then the nullary data constructors *succ* and *not* would necessarily have type $\forall\alpha_1\alpha_2.$*Arrow* $\alpha_1\,\alpha_2$. Second, when performing case analysis over a value of type *Arrow* $\alpha_1\,\alpha_2$, we need the branch associated with *succ* (resp. *not*) to be typechecked under the extra assumption *Arrow* $\alpha_1\,\alpha_2 =$ *Arrow int int* (resp. *Arrow* $\alpha_1\,\alpha_2 =$ *Arrow bool bool*). Such a mechanism is quite natural: it is reminiscent of the *inductive types* found in the calculus of inductive constructions [28], and is known in a programming-language setting under a variety of names, among which *guarded recursive data types* [44], *first-class phantom types* [8, 20], and *equality-qualified types* [36]. We refer to it as *guarded algebraic data types*. The term *guarded* stems from Xi, Chen, and Chen's observation that this feature may be encoded in terms of recursive types, sum types, and *constrained* (guarded) existential types.

1.5. Hanus' approach

Hanus' [18, 19] typed logic programming language allows "functions" (data constructors, in our terminology) to be assigned arbitrary Hindley-Milner type schemes. This yields expressiveness comparable, at first sight, to that of guarded algebraic data types. In fact, Hanus exploits this expressive power to typecheck defunctionalized programs in the style of Warren [41]. However, Hanus' type system does not deal with pattern matching in a special way: that is, it does not exploit the fact that a successful match provides extra *static* type information. Hanus compensates for this weakness by performing *dynamic* type tests and backtracking when they fail. A programming language equipped with true guarded algebraic data types, on the other hand, statically checks that programs are safe, and requires no dynamic type tests. As a result, Hanus' approach accepts more programs, but offers fewer static guarantees and is more costly at run time.

1.6. Contributions

The main contribution of this paper is a proof that defunctionalization may be viewed as a type-preserving transformation from System F, extended with guarded algebraic data types, into itself. We also observe, but do not explicitly prove, that the same property holds of Hindley and Milner's type system when extended with polymorphic recursion and guarded algebraic data types.

It is interesting to note that, because our version of defunctionalization employs a single, polymorphic *apply* function, it is not type-directed. In other words, although type information in the source program is (of course) used to construct a type derivation for the target program, it does not influence the target program's erasure. Put another way, it is possible to prove that our version of defunctionalization coincides with an untyped version of defunctionalization, up to erasure of all type annotations. This makes it possible to prove that the transformation is meaning-preserving in an *untyped* setting and to lift this result to a typed setting. These (easy) proofs form the paper's second contribution. They appear to be new: indeed, previous proofs [1, 26] were carried out in a simply-typed setting.

Our last contribution is to suggest that defunctionalization is an instance of a more general technique, which we refer to as *concretization*, and which allows eliminating constructs other than functions. To illustrate this point, we define two new type-preserving transformations that can be viewed as instances of concretization. One eliminates *polymorphic records* in the style of Rémy [32] by translating them down to guarded algebraic data types. The other eliminates the dictionary records introduced by the standard compilation scheme for Haskell's *type classes* [17, 40]. This yields a new type-preserving compilation scheme for type classes, whose target type system is an extension of Hindley and Milner's discipline with polymorphic recursion and guarded algebraic data types.

1.7. Road map

The paper is laid out as follows. First, we define an extension of System F with guarded algebraic data types (Section 2), and define defunctionalization of well-typed programs (Section 3). Then, we prove that defunctionalization preserves well-typedness (Section 4). Next, we define defunctionalization of untyped programs, prove that it preserves their meaning, and prove that this result carries over to typed programs (Section 5). Miscellaneous remarks about these results appear in Section 6. Last, we isolate the principle of *concretization* and exploit it to define other type-preserving program transformations (Sections 7–9).

2. The type system

We now define an extension of System F with guarded algebraic data types, which serves both as the source and target language for our version of defunctionalization.

Our presentation of the type system is identical to Xi et al. [44], with a few superficial differences. First, we replace pattern matching with a simple `case` construct, which is sufficient for our purposes. Second, we adopt an implicit introduction style for type variables, so that type variables are not explicitly listed in typing environments, and types or typing environments do not have a notion of well-formedness. Last, we allow data constructors to have arbitrary arity, and we attach labels to their arguments. This allows us not to introduce records as a separate construct.

A *type signature* $\mathcal{T}$ consists of an arbitrary set of *algebraic data type constructors* T, each of which carries a nonnegative arity. The definitions that follow are relative to a type signature. We let α range over a denumerable set of *type variables*. The syntax of *types* is as follows:

$$\tau ::= \alpha \mid \tau \to \tau \mid \forall\alpha.\tau \mid T\ \bar{\tau}$$

Types include type variables, arrow types, universal types, and algebraic data types. In the universal type $\forall\alpha.\tau$, the type variable α is bound within τ. In the algebraic data type $T\ \bar{\tau}$, the length of the vector $\bar{\tau}$ must match the arity of T.

A *constraint* C or D is a conjunction of type equations of the form $\tau = \tau$. An *assignment* is a total mapping of type variables to ground types. An assignment *satisfies* an equation if and only if it maps both of its members to the same ground type; an assignment satisfies a conjunction of equations if and only if it satisfies all of its members. A constraint C *entails* a constraint D (which we write $C \Vdash D$) if and only if every assignment that satisfies C satisfies D. Two constraints are *equivalent* if and only if they entail each other. Constraints serve as hypotheses within typing judgements; entailment allows exploiting them. Entailment is decidable [44].

A *data signature* $\mathcal{D}$ consists of an arbitrary set of *data constructors* K, each of which carries a closed *type scheme* of the form

$$\forall\bar{\alpha}[D].\{\bar{\ell} : \bar{\tau}\} \to T\ \bar{\tau}_1.$$

In such a type scheme, the type variables $\bar{\alpha}$ are bound within D, $\bar{\tau}$, and $\bar{\tau}_1$. The metavariable ℓ ranges over an arbitrary set of *record labels*. We write $\bar{\ell}$ for a vector of distinct record labels and $\bar{\tau}$ for a vector of types. The notation $\bar{\ell} : \bar{\tau}$, defined when $\bar{\ell}$ and $\bar{\tau}$ have the same length, stands for the vector of bindings obtained by associating elements of $\bar{\ell}$, in order, to elements of $\bar{\tau}$. Vectors of bindings are identified up to reordering. (In the following, we employ similar notation for vectors of bindings of the form $\bar{x} : \bar{\tau}$, $\bar{e} : \bar{\tau}$, $\bar{c} : \tau$, and for conjunctions of equations $\bar{\tau}_1 = \bar{\tau}_2$.)

The definitions that follow are relative to (a type signature and) a data signature.

Let x and y range over a denumerable set of *term variables*. The syntax of *expressions e*, also known as *terms*, and of *clauses c* is as follows:

$$\begin{array}{lll}
e ::= & & \textit{— Expressions} \\
& x \mid \lambda x : \tau.e \mid e\ e & \textit{Pure λ-calculus} \\
& \mid \Lambda\alpha.e \mid e\ \tau & \textit{Type abstraction and application} \\
& \mid \mathsf{let}\, x = e\ \mathsf{in}\, e & \textit{Local definitions} \\
& \mid \mathsf{letrec}\, \bar{x} : \bar{\tau} = \bar{e}\ \mathsf{in}\, e & \\
& \mid K\ \bar{\tau}\ \{\bar{\ell} = \bar{e}\} & \textit{Data structures} \\
& \mid \mathsf{case}\, e\ \mathsf{of}\, [\tau]\ \bar{c} & \\
c ::= & K\ \bar{\alpha}\ \{\bar{\ell} = \bar{x}\} \mapsto e & \textit{— Clauses}
\end{array}$$

The language is an extension of the polymorphic λ-calculus [29] with recursive definitions and with constructs for building and inspecting algebraic data structures. (In Section 5, where we present an operational semantics for this programming language, type abstractions and recursive definitions are restricted to values; for the moment, however, this is irrelevant.) In case constructs, the clauses' result type τ is explicitly given, so as to preserve the property that every expression has at most one type, up to equivalence, with respect to a given typing environment. (We do not, however, make use of that property.) We assume that, for some algebraic data type constructor T, every data constructor K associated with T is selected by one and only one clause in $\bar{c}$. In a clause $K\ \bar{\alpha}\ \{\bar{\ell} = \bar{x}\} \mapsto e$, the type variables $\bar{\alpha}$ and the

VAR
$$C, \Gamma \vdash x : \Gamma(x)$$

ABS
$$\frac{C, \Gamma; x : \tau_1 \vdash e : \tau_2 \qquad \mathrm{dom}(\Gamma) = \mathrm{fv}(\lambda x : \tau_1.e)}{C, \Gamma \vdash \lambda x : \tau_1.e : \tau_1 \to \tau_2}$$

APP
$$\frac{C, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad C, \Gamma \vdash e_2 : \tau_1}{C, \Gamma \vdash e_1\, e_2 : \tau_2}$$

TABS
$$\frac{C, \Gamma \vdash e : \tau \qquad \alpha \# C, \Gamma}{C, \Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}$$

TAPP
$$\frac{C, \Gamma \vdash e : \forall\alpha.\tau}{C, \Gamma \vdash e\,\tau_1 : [\alpha \mapsto \tau_1]\tau}$$

LET
$$\frac{C, \Gamma \vdash e_1 : \tau_1 \qquad C, \Gamma; x : \tau_1 \vdash e_2 : \tau_2}{C, \Gamma \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \tau_2}$$

LETREC
$$\frac{C, \Gamma; \bar{x} : \bar{\tau} \vdash \bar{e} : \bar{\tau} \qquad C, \Gamma; \bar{x} : \bar{\tau} \vdash e : \tau}{C, \Gamma \vdash \mathsf{letrec}\, \bar{x} : \bar{\tau} = \bar{e} \,\mathsf{in}\, e : \tau}$$

DATA
$$\frac{K :: \forall\bar{\alpha}[D].\{\bar{\ell} : \bar{\tau}\} \to T\,\bar{\tau}_1 \qquad C \Vdash [\bar{\alpha} \mapsto \bar{\tau}_2]D \qquad C, \Gamma \vdash \bar{e} : [\bar{\alpha} \mapsto \bar{\tau}_2]\bar{\tau}}{C, \Gamma \vdash K\,\bar{\tau}_2\,\{\bar{\ell} = \bar{e}\} : T\,[\bar{\alpha} \mapsto \bar{\tau}_2]\bar{\tau}_1}$$

CASE
$$\frac{C, \Gamma \vdash e : \tau_1 \qquad C, \Gamma \vdash \bar{c} : \tau_1 \to \tau_2}{C, \Gamma \vdash \mathsf{case}\, e \,\mathsf{of}\, [\tau_2]\, \bar{c} : \tau_2}$$

CLAUSE
$$\frac{K :: \forall\bar{\alpha}[D].\{\bar{\ell} : \bar{\tau}\} \to T\,\bar{\tau}_1 \qquad \bar{\alpha} \# C, \Gamma, \bar{\tau}_2, \tau \qquad C \wedge D \wedge \bar{\tau}_1 = \bar{\tau}_2, \Gamma; \bar{x} : \bar{\tau} \vdash e : \tau}{C, \Gamma \vdash K\,\bar{\alpha}\,\{\bar{\ell} = \bar{x}\} \mapsto e : T\,\bar{\tau}_2 \to \tau}$$

CONV
$$\frac{C, \Gamma \vdash e : \tau_1 \qquad C \Vdash \tau_1 = \tau_2}{C, \Gamma \vdash e : \tau_2}$$

WEAKEN
$$\frac{C, \Gamma_1; \Gamma_2 \vdash e : \tau \qquad x \# e}{C, \Gamma_1; x : \tau_1; \Gamma_2 \vdash e : \tau}$$

Fig. 1 The type system

term variables $\bar{x}$ are bound within e. The notation $\{\bar{\ell} = \bar{x}\}$ should be understood as a record pattern.

A *typing environment* Γ is a mapping of term variables to types, typically written as a sequence of bindings of the form $x : \tau$. A *typing judgement* is of the form $C, \Gamma \vdash e : \tau$. We identify typing judgements up to constraint equivalence. A typing judgement is *valid* if and only if it admits a derivation using the rules of Fig. 1.

In TABS, the notation $\alpha \# C, \Gamma$ requires the type variable α not to appear free within C or Γ. In TAPP, $[\alpha \mapsto \tau_1]\tau$ stands for the capture-avoiding substitution of τ_1 for α within τ. All rules but ABS, DATA, CLAUSE, CONV, and WEAKEN are standard (System F) rules. The let construct is introduced only for convenience. It is equivalent to a β-redex, except no type annotation is needed. In particular, it does not involve generalization, as it would in ML.

ABS is standard, except for its second premise, which controls the domain of the typing environment Γ. Thanks to the presence of WEAKEN, this does not cause any loss of expressiveness. This nonstandard presentation makes it possible to define defunctionalization in a concise and elegant way (see ABS in Fig. 2) while ensuring that defunctionalization is not type-directed, a fact which we establish and exploit later on (Lemma 5.7). Still, this is only a minor presentation issue.

DATA'S first premise looks up the type scheme associated with the data constructor K in the current data signature. Its second and third premises check that the constraint D is satisfied and that the arguments $\bar{e}$ have type $\bar{\tau}$, as required by the type scheme. Both of these checks are relative to an instance of the type scheme where the type arguments $\bar{\tau}_2$ are substituted for the quantifiers $\bar{\alpha}$ and to the current hypothesis C.

$$\textsc{Var}\qquad C, \Gamma \vdash x : \Gamma(x) \leadsto x$$

$$\textsc{Abs}\quad \frac{\begin{array}{c} C, \Gamma; x : \tau_1 \vdash e : \tau_2 \leadsto e' \\ \mathrm{dom}(\Gamma) = \mathrm{fv}(\lambda x : \tau_1.e) \\ \bar{\alpha} = \mathrm{ftv}(C, \Gamma, \tau_1, \tau_2) \end{array}}{\begin{array}{c} C, \Gamma \vdash \lambda^m x : \tau_1.e : \tau_1 \rightarrow \tau_2 \leadsto \\ m\,\bar{\alpha}\,\{\Gamma\} \end{array}} \qquad \textsc{App}\quad \frac{\begin{array}{c} C, \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \leadsto e_1' \\ C, \Gamma \vdash e_2 : \tau_1 \leadsto e_2' \end{array}}{\begin{array}{c} C, \Gamma \vdash e_1\,e_2 : \tau_2 \leadsto \\ \mathit{apply}\,[\![\tau_1]\!]\,[\![\tau_2]\!]\,e_1'\,e_2' \end{array}}$$

$$\textsc{TAbs}\quad \frac{C, \Gamma \vdash e : \tau \leadsto e' \qquad \alpha \# C, \Gamma}{C, \Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau \leadsto \Lambda\alpha.e'} \qquad \textsc{TApp}\quad \frac{C, \Gamma \vdash e : \forall\alpha.\tau \leadsto e'}{C, \Gamma \vdash e\,\tau_1 : [\alpha \mapsto \tau_1]\tau \leadsto e'\,[\![\tau_1]\!]}$$

$$\textsc{Let}\quad \frac{\begin{array}{c} C, \Gamma \vdash e_1 : \tau_1 \leadsto e_1' \\ C, \Gamma; x : \tau_1 \vdash e_2 : \tau_2 \leadsto e_2' \end{array}}{\begin{array}{c} C, \Gamma \vdash \mathsf{let}\,x = e_1\,\mathsf{in}\,e_2 : \tau_2 \leadsto \\ \mathsf{let}\,x = e_1'\,\mathsf{in}\,e_2' \end{array}} \qquad \textsc{LetRec}\quad \frac{\begin{array}{c} C, \Gamma; \bar{x} : \bar{\tau} \vdash \bar{e} : \bar{\tau} \leadsto \bar{e}' \\ C, \Gamma; \bar{x} : \bar{\tau} \vdash e : \tau \leadsto e' \end{array}}{\begin{array}{c} C, \Gamma \vdash \mathsf{letrec}\,\bar{x} : \bar{\tau} = \bar{e}\,\mathsf{in}\,e : \tau \leadsto \\ \mathsf{letrec}\,\bar{x} : [\![\bar{\tau}]\!] = \bar{e}'\,\mathsf{in}\,e' \end{array}}$$

$$\textsc{Data}\quad \frac{\begin{array}{c} K :: \forall\bar{\alpha}[D].\{\bar{\ell} : \bar{\tau}\} \rightarrow T\,\bar{\tau}_1 \qquad C \Vdash [\bar{\alpha} \mapsto \bar{\tau}_2]D \\ C, \Gamma \vdash \bar{e} : [\bar{\alpha} \mapsto \bar{\tau}_2]\bar{\tau} \leadsto \bar{e}' \end{array}}{C, \Gamma \vdash K\,\bar{\tau}_2\,\{\bar{\ell} = \bar{e}\} : T\,[\bar{\alpha} \mapsto \bar{\tau}_2]\bar{\tau}_1 \leadsto K\,[\![\bar{\tau}_2]\!]\,\{\bar{\ell} = \bar{e}'\}}$$

$$\textsc{Case}\quad \frac{C, \Gamma \vdash e : \tau_1 \leadsto e' \qquad C, \Gamma \vdash \bar{c} : \tau_1 \rightarrow \tau_2 \leadsto \bar{c}'}{C, \Gamma \vdash \mathsf{case}\,e\,\mathsf{of}\,[\tau_2]\,\bar{c} : \tau_2 \leadsto \mathsf{case}\,e'\,\mathsf{of}\,[[\![\tau_2]\!]]\,\bar{c}'}$$

$$\textsc{Clause}\quad \frac{\begin{array}{c} K :: \forall\bar{\alpha}[D].\{\bar{\ell} : \bar{\tau}\} \rightarrow T\,\bar{\tau}_1 \qquad \bar{\alpha} \# C, \Gamma, \bar{\tau}_2, \tau \\ C \wedge D \wedge \bar{\tau}_1 = \bar{\tau}_2, \Gamma; \bar{x} : \bar{\tau} \vdash e : \tau \leadsto e' \end{array}}{C, \Gamma \vdash K\,\bar{\alpha}\,\{\bar{\ell} = \bar{x}\} \mapsto e : T\,\bar{\tau}_2 \rightarrow \tau \leadsto K\,\bar{\alpha}\,\{\bar{\ell} = \bar{x}\} \mapsto e'}$$

$$\textsc{Conv}\quad \frac{C, \Gamma \vdash e : \tau_1 \leadsto e' \qquad C \Vdash \tau_1 = \tau_2}{C, \Gamma \vdash e : \tau_2 \leadsto e'} \qquad \textsc{Weaken}\quad \frac{C, \Gamma_1; \Gamma_2 \vdash e : \tau \leadsto e' \qquad x \# e}{C, \Gamma_1; x : \tau_1; \Gamma_2 \vdash e : \tau \leadsto e'}$$

Fig. 2 Term translation

CASE's second premise means that each of the clauses in $\bar{c}$ should have type $\tau_1 \rightarrow \tau_2$.

CLAUSE's first premise looks up the type scheme associated with K and implicitly α-converts it so that its universal quantifiers coincide with the type variables $\bar{\alpha}$ introduced by the clause at hand. Its second premise requires these type variables to be fresh, so that they behave as abstract types within the clause's right-hand side e, and do not escape their scope. Its third premise typechecks e under the extra hypothesis $D \wedge \bar{\tau}_1 = \bar{\tau}_2$, which is obtained from the knowledge that the value being examined, which by assumption has type $T\ \bar{\tau}_2$, is an application of K. This extra hypothesis may provide partial or complete information about the type variables $\bar{\alpha}$, making them semi-abstract or concrete.

CONV allows replacing the type τ_1 with the type τ_2, provided they are provably equal under the assumption C. It is analogous to the subtyping rule in a constraint-based type system.

The type system is sound [44]. Although this property is of course essential, it is not explicitly exploited in the present paper. We only make use of the following lemma, which allows weakening a judgement's constraint and replacing its typing environment with an equivalent one. When Γ and Γ' have the same domain, we view $\Gamma' = \Gamma$ as a conjunction of type equations.

Lemma 2.1. *$C, \Gamma \vdash e : \tau$ and $C' \Vdash C \wedge \Gamma' = \Gamma$ imply $C', \Gamma' \vdash e : \tau$.*

3. Defunctionalization

Defunctionalization is a global program transformation: it is necessary that all functions that appear in the source program be known and labeled in a unique manner. Thus, in the following, we consider a fixed term p, which we refer to as the *source program*. We require every λ-abstraction that appears within p to carry a distinct *label* m; we write $\lambda^m x : \tau.e$ for such a labeled abstraction. We require p to be well-typed under the empty constraint true and the empty environment $\varnothing$, and consider a fixed derivation of the judgement $\mathsf{true}, \varnothing \vdash p : \tau_p$. We let $\mathcal{T}$ and $\mathcal{D}$ stand for the type and data signatures under which p is defined.

The transformed program is defined under an extended type signature $\mathcal{T}'$, which contains $\mathcal{T}$ as well as a fresh binary algebraic data type constructor *Arrow*. The effect of the translation on types is particularly simple: the native arrow type constructor is translated to *Arrow*, while all other type formers are preserved.

$$\begin{aligned}
[\![\alpha]\!] &= \alpha \\
[\![\tau_1 \to \tau_2]\!] &= \mathit{Arrow}\,[\![\tau_1]\!]\,[\![\tau_2]\!] \\
[\![\forall\alpha.\tau]\!] &= \forall\alpha.[\![\tau]\!] \\
[\![T\ \bar{\tau}]\!] &= T\ [\![\bar{\tau}]\!]
\end{aligned}$$

This translation function is extended in a compositional manner to vectors of types, typing environments, constraints, type schemes, and data signatures.

The transformed program is defined under a transformed and extended data signature $\mathcal{D}'$, which is set up as follows. First, $\mathcal{D}'$ contains $[\![\mathcal{D}]\!]$. Second, for every λ-abstraction that appears within p and whose typing subderivation ends with

$$C, \Gamma \vdash \lambda^m x : \tau_1.e : \tau_1 \to \tau_2,$$

$\mathcal{D}'$ contains a unary data constructor

$$m :: \forall\bar{\alpha}[[\![C]\!]].\{[\![\Gamma]\!]\} \to \mathit{Arrow}\,[\![\tau_1]\!]\,[\![\tau_2]\!]\,,$$

where $\bar{\alpha}$ stands for the free type variables of the above judgement, that is, $\mathrm{ftv}\,(C, \Gamma, \tau_1, \tau_2)$, ordered in a fixed, arbitrary manner. We point out that $[\![\Gamma]\!]$ is a typing environment, that is, a mapping of term variables to types; we assume that term variables form a subset of record labels, which allows us to view $[\![\Gamma]\!]$ as a mapping of record labels to types. We assume that all variables in the source program p have received fixed (although not necessarily distinct) names, so that turning them into record labels is not a problematic operation.

We may now define a compositional term translation as follows. In the following, let *apply* be a fresh term variable. The translation is defined by a new judgement, of the form $C, \Gamma \vdash e : \tau \rightsquigarrow e'$, whose derivation rules are given in Fig. 2. It is clear that $C, \Gamma \vdash e : \tau \rightsquigarrow e'$ implies $C, \Gamma \vdash e : \tau$. Conversely, given a derivation of $C, \Gamma \vdash e : \tau$, there exists a unique expression e' such that the judgement $C, \Gamma \vdash e : \tau \rightsquigarrow e'$ is the conclusion of a derivation of the same shape. We refer to e' as the image of e through defunctionalization. In the following, we refer to the image of p through defunctionalization as p'. It is obtained from the derivation of true, $\varnothing \vdash p : \tau_p$ that was fixed above.

The only two interesting rules in the definition of the translation are ABS and APP. Indeed, all other rules preserve the structure of the expression at hand, using the type translation defined above to deal with type annotations. ABS translates every λ-abstraction to an injection, making closure allocation explicit. The data constructor (or, in other words, the closure's tag) is m, the unique label that was assigned to this λ-abstraction. Its type arguments, $\bar{\alpha}$, are all of the type variables that appear free in the typing judgement. (By convention, these must be ordered in the same way as in the type scheme associated with the data constructor m in the data signature $\mathcal{D}'$.) Its value arguments form a record that stores the values currently associated with all of the term variables that are bound by the environment Γ. We write $\{\Gamma\}$ as a short-hand for $\{y = y\}_{y \in \mathrm{dom}(\Gamma)}$, where the left-hand y is interpreted as a record label, while the right-hand y is a term variable. This is the closure's value environment. ABS's second premise requires $\mathrm{dom}(\Gamma)$ to coincide with $fv(\lambda x : \tau_1.e)$, so the variables whose values are saved in the closure are exactly the function's free variables. As announced in the introduction, APP translates function applications into invocations of *apply*.

To complete the definition of the program transformation, there remains to wrap the term p' within an appropriate definition of *apply*. Let τ_{apply} stand for

$$\forall \alpha_1.\forall \alpha_2.\mathit{Arrow}\,\alpha_1\,\alpha_2 \to \alpha_1 \to \alpha_2.$$

Let f and arg be fresh term variables. Let α_1 and α_2 be fresh type variables. Then, the translation of the source program p, which we write $[\![p]\!]$, is the target program

$$\begin{array}{l}
\mathsf{letrec}\,\mathit{apply} : \tau_{apply} = \\
\quad \Lambda\alpha_1.\Lambda\alpha_2. \\
\qquad \lambda f : \mathit{Arrow}\,\alpha_1\,\alpha_2. \\
\qquad\quad \lambda arg : \alpha_1. \\
\qquad\qquad \mathsf{case}\,f\ \mathsf{of}\,[\alpha_2]\,\bar{c}_p \\
\mathsf{in}\,p',
\end{array}$$

where, for every λ-abstraction that appears within p and whose translation subderivation ends with

ABS

$$\frac{C, \Gamma; x : \tau_1 \vdash e : \tau_2 \rightsquigarrow e' \qquad \mathrm{dom}(\Gamma) = \mathrm{fv}(\lambda x : \tau_1 . e) \quad \bar{\alpha} = \mathrm{ftv}(C, \Gamma, \tau_1, \tau_2)}{C, \Gamma \vdash \lambda^m x : \tau_1 . e : \tau_1 \to \tau_2 \rightsquigarrow m\bar{\alpha}\{\Gamma\}}$$

the vector $\bar{c}_p$ contains the clause

$$m\,\bar{\alpha}\,\{\Gamma\} \mapsto \mathsf{let}\,x = arg\,\mathsf{in}\,e'.$$

As announced in the introduction, *apply* examines the closure's tag in order to determine which code to execute. The clause associated with the tag m reintroduces the type and term variables, namely $\bar{\alpha}$, Γ, and x, that must be in scope for the function's code, namely e', to make sense. Again, the type variables $\bar{\alpha}$ must be ordered in the same way as in the type scheme associated with m, and we write $\{\Gamma\}$ as a short-hand for $\{y = y\}_{y \in \mathrm{dom}(\Gamma)}$.

Our definition of defunctionalization is now complete. Although, for the sake of simplicity, we have identified the source and target languages, it is easy to check that every defunctionalized program is first-order, as desired. Indeed, all function applications in such a program are double applications of *apply*, a letrec-bound, binary function.

Example. Figure 3 contains a sample program, whose defunctionalized version appears in Fig. 4. For the sake of simplicity, this program does not make use of guarded algebraic data types: it is a System F program. We believe that choosing a sample program that does make use of guarded algebraic data types would provide no additional insights.

The program, inspired by Banerjee et al. [1], defines a very simple implementation of sets as characteristic functions, then builds the singleton set $\{1\}$ and tests whether 2 is a member of it. It makes use of a polymorphic equality function $=$ of type $\forall\alpha.\alpha \rightarrow \alpha \rightarrow \mathit{bool}$ and of the Boolean "or" combinator $||$ of type $\mathit{bool} \rightarrow \mathit{bool} \rightarrow \mathit{bool}$. Applications of these two primitive operations are not affected by the translation.

The empty set *empty* has type $\forall\alpha.\alpha \rightarrow \mathit{bool}$. The insertion function *insert* has type $\forall\alpha.\alpha \rightarrow (\alpha \rightarrow \mathit{bool}) \rightarrow (\alpha \rightarrow \mathit{bool})$. The complete program has type *bool*. Its defunctionalized counterpart is defined under the data signature that appears at the top of Fig. 4, where $\mathit{set}\,\alpha$ stands for $\mathit{Arrow}\,\alpha\,\mathit{bool}$.

The type scheme associated with m_i specifies the structure of the value environment found in every closure tagged m_i, as well as the type of the function that every such closure encodes. Closures formed using m_1 or m_2 carry an empty value environment, because they encode closed functions. On the other hand, closures formed using m_3 or m_4 carry a nonempty value environment, because the corresponding λ-abstractions have free term variables. The type schemes associated with m_1 and m_4 are similar to those usually assigned to the data

Fig. 3 A sample code fragment

```
let $empty = \Lambda\alpha.\lambda^{m_1}x : \alpha.$false in
let $insert = \Lambda\alpha.\lambda^{m_2}x : \alpha.\lambda^{m_3}s : \alpha \rightarrow bool.$
                 $\lambda^{m_4}y : \alpha.(=\ \alpha\,x\,y)\ ||\ (s\,y)$ in
$insert\ int\ 1\ (empty\ int)\ 2$
```

Fig. 4 The defunctionalized code fragment

```
$m_1$ :: $\forall\alpha[\text{true}].\{\} \rightarrow set\,\alpha$
$m_2$ :: $\forall\alpha[\text{true}].\{\} \rightarrow Arrow\,\alpha\,(Arrow\,(set\,\alpha)\,(set\,\alpha))$
$m_3$ :: $\forall\alpha[\text{true}].\{x : \alpha\} \rightarrow Arrow\,(set\,\alpha)\,(set\,\alpha)$
$m_4$ :: $\forall\alpha[\text{true}].\{x : \alpha;\ s : set\,\alpha\} \rightarrow set\,\alpha$

letrec $apply : \forall\alpha_1.\forall\alpha_2.Arrow\,\alpha_1\,\alpha_2 \rightarrow \alpha_1 \rightarrow \alpha_2 =$
  $\Lambda\alpha_1.\Lambda\alpha_2.$
    $\lambda f : Arrow\,\alpha_1\,\alpha_2.$
      $\lambda arg : \alpha_1.$
        case $f$ of $[\alpha_2]$
        | $m_1\,\alpha\,\{\} \mapsto$ let $x = arg$ in false
        | $m_2\,\alpha\,\{\} \mapsto$ let $x = arg$ in $m_3\,\alpha\,\{x\}$
        | $m_3\,\alpha\,\{x\} \mapsto$ let $s = arg$ in $m_4\,\alpha\,\{x; s\}$
        | $m_4\,\alpha\,\{x; s\} \mapsto$ let $y = arg$ in
                    $(=\ \alpha\,x\,y)\ ||\ (apply\,\alpha\,bool\,s\,y)$
in
let $empty = \Lambda\alpha.m_1\,\alpha\,\{\}$ in
let $insert = \Lambda\alpha.m_2\,\alpha\,\{\}$ in
$apply\,(apply\,(apply\,(insert\,int)\,1)\,(empty\,int))\,2$
```

constructors *nil* and *cons*, which makes apparent the fact that sets built using *empty* and *insert* become lists after defunctionalization.

The defunctionalized program forms the remainder of Fig. 4. As before, we use punning, that is, we write $\{x\}$ for $\{x = x\}$ and $\{x; s\}$ for $\{x = x; s = s\}$. For the sake of brevity, we omit the type arguments to *apply* in the last line. Most of the code is straightforward, but it is perhaps worth explaining why every clause in the definition of *apply* is well-typed. Let us consider, for instance, the clause associated with m_4. Because the type scheme associated with m_4 is

$$\forall\alpha[\mathsf{true}].\{x : \alpha;\ s : set\,\alpha\} \to Arrow\,\alpha\,bool\,,$$

the clause's right-hand side is typechecked under the extra hypothesis $\alpha = \alpha_1 \wedge bool = \alpha_2$ and under a typing environment that ends with $arg : \alpha_1; x : \alpha; s : set\,\alpha$. After binding y to *arg*, the typing environment ends with $x : \alpha; s : set\,\alpha; y : \alpha_1$. Thus, y has type α_1, which by hypothesis equals α. Hence, by CONV, y has type α. It is then straightforward to check that the expression $(=\alpha\ x\ y)\ ||\ (apply\,\alpha\ bool\,s\ y)$ has type *bool*. Furthermore, by hypothesis, *bool* equals α_2, so the clause's right-hand side has the expected type α_2. All other clauses may be successfully typechecked in a similar manner: although not all of them have type *bool*, all have type α_2. Lemma 4.2 carries out the proof in the general case.

4. Type preservation

We now prove that defunctionalization, as defined in Section 3, preserves types. As illustrated by the above example, the proof is not difficult.

In the following statements, for the sake of brevity, we write *apply*, *f*, and *arg* for the bindings $apply : \tau_{apply}$, $f : Arrow\,\alpha_1\,\alpha_2$, and $arg : \alpha_1$, respectively. We use this notation in λ-abstractions and in typing environments.

Our first lemma states that if an expression e is well-typed, then its image through defunctionalization e' must be well-typed as well, under a constraint, a typing environment, and a type given by the type translation. Of course, the typing environment must be extended with a binding for *apply*, which is used in the translation of applications.

Lemma 4.1. *$C, \Gamma \vdash e : \tau \leadsto e'$ implies $[\![C]\!], apply; [\![\Gamma]\!] \vdash e' : [\![\tau]\!]$.*

Proof: By structural induction on the derivation of $C, \Gamma \vdash e : \tau \leadsto e'$. In each case, we use the notations of Fig. 2. We explicitly deal with value abstraction and application only; all other cases are straightforward.

○ *Case* ABS. The rule's conclusion is

$$C, \Gamma \vdash \lambda^m x : \tau_1.e : \tau_1 \to \tau_2 \leadsto m\,\bar{\alpha}\,\{\Gamma\} \qquad (1)$$

Its last premise is

$$\bar{\alpha} = \mathrm{ftv}(C, \Gamma, \tau_1, \tau_2) \qquad (2)$$

By (1), (2), and by definition of the data signature $\mathcal{D}'$, we have

$$m :: \forall\bar{\alpha}[\llbracket C\rrbracket].\{\llbracket\Gamma\rrbracket\} \rightarrow \mathit{Arrow}\,\llbracket\tau_1\rrbracket\,\llbracket\tau_2\rrbracket \qquad (3)$$

By reflexivity of entailment, we have

$$\llbracket C\rrbracket \Vdash \llbracket C\rrbracket \qquad (4)$$

By VAR, for every $y \in \mathrm{dom}(\Gamma)$, we have

$$\llbracket C\rrbracket, \mathit{apply}; \llbracket\Gamma\rrbracket \vdash y : \llbracket\Gamma\rrbracket(y) \qquad (5)$$

Applying DATA to (3), (4), and (5), and exploiting the fact that the types $\mathit{Arrow}\,\llbracket\tau_1\rrbracket\,\llbracket\tau_2\rrbracket$ and $\llbracket\tau_1 \rightarrow \tau_2\rrbracket$ coincide, we find

$$\llbracket C\rrbracket, \mathit{apply}; \llbracket\Gamma\rrbracket \vdash m\,\bar{\alpha}\,\{\Gamma\} : \llbracket\tau_1 \rightarrow \tau_2\rrbracket.$$

○ *Case* APP. The rule's conclusion is

$$C, \Gamma \vdash e_1\,e_2 : \tau_2 \rightsquigarrow \mathit{apply}\,\llbracket\tau_1\rrbracket\,\llbracket\tau_2\rrbracket\,e_1'\,e_2'.$$

Its first premise is

$$C, \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \rightsquigarrow e_1' \qquad (1)$$

Its second premise is

$$C, \Gamma \vdash e_2 : \tau_1 \rightsquigarrow e_2' \qquad (2)$$

Applying the induction hypothesis to (1) yields

$$\llbracket C\rrbracket, \mathit{apply}; \llbracket\Gamma\rrbracket \vdash e_1' : \llbracket\tau_1 \rightarrow \tau_2\rrbracket \qquad (3)$$

Applying it to (2) yields

$$\llbracket C\rrbracket, \mathit{apply}; \llbracket\Gamma\rrbracket \vdash e_2' : \llbracket\tau_1\rrbracket \qquad (4)$$

Furthermore, by VAR, by definition of τ_{apply}, by TAPP, and by exploiting the fact that $\mathit{Arrow}\,\llbracket\tau_1\rrbracket\,\llbracket\tau_2\rrbracket$ equals $\llbracket\tau_1 \rightarrow \tau_2\rrbracket$, we find

$$\llbracket C\rrbracket, \mathit{apply}; \llbracket\Gamma\rrbracket \vdash \mathit{apply}\,\llbracket\tau_1\rrbracket\,\llbracket\tau_2\rrbracket : \llbracket\tau_1 \rightarrow \tau_2\rrbracket \rightarrow \llbracket\tau_1\rrbracket \rightarrow \llbracket\tau_2\rrbracket \qquad (5)$$

APP, (3), (4), (5), imply $\llbracket C\rrbracket, \mathit{apply}; \llbracket\Gamma\rrbracket \vdash \mathit{apply}\,\llbracket\tau_1\rrbracket\,\llbracket\tau_2\rrbracket\,e_1'\,e_2' : \llbracket\tau_2\rrbracket$. □

The second lemma states that *apply* itself is well-typed and has type τ_{apply}, as desired. Because *apply* is recursive, this assertion holds under the binding *apply* : τ_{apply}.

Lemma 4.2. $\mathsf{true}, \mathit{apply} \vdash \Lambda\alpha_1\alpha_2.\lambda f.\lambda arg.\mathsf{case}\,f\,\mathsf{of}\,[\alpha_2]\,\bar{c}_p : \tau_{apply}$.

Proof: We must prove that every clause in $\bar{c}_p$ is well-typed. Thus, let us consider a λ-abstraction that appears within p and whose translation subderivation ends with

$$\textsc{Abs}\quad \frac{C, \Gamma; x : \tau_1 \vdash e : \tau_2 \rightsquigarrow e' \qquad \mathrm{dom}(\Gamma) = \mathrm{fv}(\lambda x : \tau_1 \,.\, e) \quad \bar{\alpha} = \mathrm{ftv}(C, \Gamma, \tau_1, \tau_2)}{C, \Gamma \vdash \lambda^m x : \tau_1 \,.\, e : \tau_1 \to \tau_2 \rightsquigarrow m\bar{\alpha}\{\Gamma\}}.$$

Applying Lemma 4.1 to the first premise yields

$$[\![C]\!], \mathit{apply}; [\![\Gamma]\!]; x : [\![\tau_1]\!] \vdash e' : [\![\tau_2]\!],$$

which, by Lemma 2.1, CONV, and WEAKEN, implies

$$[\![C]\!] \wedge [\![\tau_1]\!] = \alpha_1 \wedge [\![\tau_2]\!] = \alpha_2, \mathit{apply}; \mathit{arg}; [\![\Gamma]\!]; x : \alpha_1 \vdash e' : \alpha_2.$$

Using LET, this leads to

$$[\![C]\!] \wedge [\![\tau_1]\!] = \alpha_1 \wedge [\![\tau_2]\!] = \alpha_2, \mathit{apply}; \mathit{arg}; [\![\Gamma]\!] \vdash \mathsf{let}\, x = \mathit{arg}\, \mathsf{in}\, e' : \alpha_2 \qquad (1)$$

By definition of the data signature $\mathcal{D}'$, we have

$$m :: \forall\bar{\alpha}[[\![C]\!]].\{[\![\Gamma]\!]\} \to \mathit{Arrow}\, [\![\tau_1]\!]\, [\![\tau_2]\!] \qquad (2)$$

By construction, we have

$$\bar{\alpha} \,\#\, \alpha_1\alpha_2 \qquad (3)$$

Applying CLAUSE to (1), (2), and (3) yields

$$\mathsf{true}, \mathit{apply}; \mathit{arg} \vdash m\, \bar{\alpha}\, \{\Gamma\} \mapsto \mathsf{let}\, x = \mathit{arg}\, \mathsf{in}\, e' : \mathit{Arrow}\, \alpha_1\, \alpha_2 \to \alpha_2.$$

Now, because this holds for every λ-abstraction that appears within p, and by definition of $\bar{c}_p$, we have established

$$\mathsf{true}, \mathit{apply}; \mathit{arg} \vdash \bar{c}_p : \mathit{Arrow}\, \alpha_1\, \alpha_2 \to \alpha_2.$$

The result follows by WEAKEN, CASE, ABS, and TABS. □

It is now easy to conclude that the image of the source program p under defunctionalization is well-typed.

Theorem 4.3. $\mathsf{true}, \varnothing \vdash [\![p]\!] : [\![\tau_p]\!]$.

Proof: Applying Lemma 4.1 to the judgement $\mathsf{true}, \varnothing \vdash p : \tau_p \rightsquigarrow p'$ yields $\mathsf{true}, \mathit{apply} \vdash p' : [\![\tau_p]\!]$. The result follows from this and from Lemma 4.2 by LETREC. □

5. Meaning preservation

We now prove that defunctionalization preserves the meaning of programs, as defined by a call-by-value operational semantics. In order to define such a semantics, we will require the right-hand sides of letrec definitions to be values. Furthermore, to ensure that type abstractions and applications do not influence reduction, we will require the bodies of type abstractions to be values. The language is otherwise identical to the one studied in Sections 2–4.

We believe that it would be straightforward to carry out a similar proof in a call-by-name or call-by-need setting, but have not attempted to do so.

Because our notion of defunctionalization is not type-directed, we are able to proceed in two steps, as follows. First, we define defunctionalization of *untyped* programs, and prove that it preserves meaning. Second, exploiting the fact that the two notions of defunctionalization coincide modulo type erasure, we easily lift this result back to a typed setting.

This approach appears more general than those found in previous works, which assumed a simply-typed calculus and carried out proofs based on logical relations [1, 26]. Proofs based on logical relations proceed by induction on typing derivations. We believe that, because defunctionalization can be defined in an untyped setting, its semantic correctness should also be proved in such a setting. Indeed, an untyped correctness statement is much more general, and needs not be proved again if, for some reason, the type system evolves. Also, its proof is somewhat more concise, since terms are not cluttered with type annotations.

5.1. Untyped defunctionalization

Syntax. The syntax of the untyped calculus appears in Fig. 5. It is the type-free counterpart of the typed language presented in Section 2, with a few amendments that help define a call-by-value operational semantics.

We introduce a new syntactic class of *references*, ranged over by r. References never appear in source programs, but arise during reduction. They represent memory addresses, and help model recursive definitions, which can allocate cyclic data structures in memory. References are immutable: once a memory location has been allocated and initialized, its contents cannot be modified.

We also introduce a new syntactic class of *values*, ranged over by v. Values include functions and data constructor applications. We require the right-hand sides of letrec definitions to be values: this restriction is standard in a call-by-value setting.

A *store* S is a partial mapping of references to closed values. A *configuration* S/e is a pair of a store S and a closed expression e. As usual, the references in the domain of the store S are considered bound in the configuration S/e.

Fig. 5 Syntax of the untyped calculus

$$
\begin{array}{rcl}
e & ::= & r \\
 & \mid & x \\
 & \mid & \lambda x.e \\
 & \mid & e\,e \\
 & \mid & \mathsf{let}\, x = e \,\mathsf{in}\, e \\
 & \mid & \mathsf{letrec}\, \bar{x} = \bar{v} \,\mathsf{in}\, e \\
 & \mid & K\,\{\bar{\ell} = \bar{e}\} \\
 & \mid & \mathsf{case}\, e \,\mathsf{of}\, \bar{c} \\
c & ::= & K\,\{\bar{\ell} = \bar{x}\} \mapsto e
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & \lambda x.e \\
 & \mid & K\,\{\bar{\ell} = \bar{w}\} \\
w & ::= & r \\
 & \mid & x \\
S & ::= & \bar{r} = \bar{v} \\
E & ::= & \Box\, e \\
 & \mid & r\, \Box \\
 & \mid & \mathsf{let}\, x = \Box \,\mathsf{in}\, e \\
 & \mid & K\,\{\ell = \Box; \bar{\ell} = \bar{e}\} \\
 & \mid & \mathsf{case}\, \Box \,\mathsf{of}\, \bar{c}
\end{array}
$$

Fig. 6 Semantics of the untyped calculus

$$\begin{array}{rl}
S\,/\,v & \to S; r \;=\; v\,/\,r \\
& \quad \text{if } r \mathrel{\#} S \\[1ex]
S\,/\,r\,r' & \to S\,/\,[x \mapsto r']e \\
& \quad \text{if } S(r) = \lambda x.e \\[1ex]
S\,/\,\mathsf{let}\, x = r \,\mathsf{in}\, e & \to S\,/\,[x \mapsto r]e \\[1ex]
S\,/\,\mathsf{case}\, r \,\mathsf{of}\, (K\,\{\bar{\ell} = \bar{x}\} \mapsto e) \mid \bar{c} & \to S\,/\,[\bar{x} \mapsto \bar{r}]e \\
& \quad \text{if } S(r) = K\,\{\bar{\ell} = \bar{r}\} \\[1ex]
S\,/\,\mathsf{letrec}\, \bar{x} = \bar{v} \,\mathsf{in}\, e & \to S; \bar{r} \;=\; [\bar{x} \mapsto \bar{r}]\bar{v}\,/\,[\bar{x} \mapsto \bar{r}]e \\
& \quad \text{if } \bar{r} \mathrel{\#} S \\[1ex]
S\,/\,E[e] & \to S'\,/\,E[e'] \\
& \quad \text{if } S\,/\,e \to S'\,/\,e'
\end{array}$$

In order to simplify the semantics and to make it more faithful to an actual implementation, we require values to be *flat*: that is, for a data constructor application to be a value, its arguments must be of the form w, where w ranges over references and variables. This does not restrict the expressive power of the letrec construct, because more complex values can be expressed as collections of flat values.

Semantics. The operational semantics, a rewriting system on closed configurations, is defined in Fig. 6.

The first reduction rule models memory allocation: a value v reduces to a fresh reference r, together with a new binding of r to v in the store. (We write $r \mathrel{\#} S$ when r appears neither in the domain or in the image of S.) Thus, in this semantics, only references are meant to be irreducible. An irreducible configuration S/e, where e is not a reference, represents a runtime error.

The next three rules are standard reduction rules for β-redexes, let-redexes, and case analysis. A common pattern is for the left-hand side to mention a reference r, which is interpreted by looking up its value $S(r)$ in the store.

The next rule deals with letrec constructs simply by allocating new store bindings, where the bound variables $\bar{x}$ are replaced with fresh references $\bar{r}$.

The last rule allows reduction under an evaluation context E.

Defunctionalization. We may now define untyped defunctionalization. As in the typed case, we consider a fixed closed program p, in which every λ-abstraction carries a unique tag. We require that no reference appears in p. We do not, however, require p to be well-typed. The translation of expressions is defined, in an inductive manner, in Fig. 7. All cases are trivial, except Abs, which translates every λ-abstraction to a closure allocation, and App, which translates every function application to an invocation of *apply*. Note that the relation $\leadsto$ is in fact a function, defined for all subexpressions of p. We write p' for the image of p through it. Then, the complete defunctionalized program $[\![p]\!]$ is

$$\mathsf{letrec}\, apply = \lambda f.\lambda arg.\mathsf{case} f \,\mathsf{of}\, \bar{c}_p \,\mathsf{in}\, p',$$

where, for every abstraction of the form $\lambda^m x.e$ that appears within p, $\bar{c}_p$ contains the clause

$$m\{\mathrm{fv}(\lambda x \cdot e)\} \mapsto let x = arg \,\mathsf{in}\, e,$$

with e' defined by $e \leadsto e'$.

$$\textsc{Ref}\quad r \rightsquigarrow r \qquad \textsc{Var}\quad x \rightsquigarrow x \qquad \textsc{Abs}\quad \lambda^m x.e \rightsquigarrow m\,\{\mathrm{fv}(\lambda x.e)\} \qquad \textsc{App}\quad \frac{e_1 \rightsquigarrow e_1' \qquad e_2 \rightsquigarrow e_2'}{e_1\,e_2 \rightsquigarrow \mathit{apply}\; e_1'\; e_2'}$$

$$\textsc{Let}\quad \frac{e_1 \rightsquigarrow e_1' \qquad e_2 \rightsquigarrow e_2'}{\begin{array}{c}\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \\ \rightsquigarrow \mathsf{let}\, x = e_1' \,\mathsf{in}\, e_2'\end{array}} \qquad \textsc{LetRec}\quad \frac{\bar{v} \rightsquigarrow \bar{v}' \qquad e \rightsquigarrow e'}{\begin{array}{c}\mathsf{letrec}\, \bar{x} = \bar{v} \,\mathsf{in}\, e \\ \rightsquigarrow \mathsf{letrec}\, \bar{x} = \bar{v}' \,\mathsf{in}\, e'\end{array}}$$

$$\textsc{Data}\quad \frac{\bar{e} \rightsquigarrow \bar{e}'}{K\,\{\bar{\ell} = \bar{e}\} \rightsquigarrow K\,\{\bar{\ell} = \bar{e}'\}} \qquad \textsc{Case}\quad \frac{e \rightsquigarrow e' \qquad \bar{c} \rightsquigarrow \bar{c}'}{\mathsf{case}\, e \,\mathsf{of}\, \bar{c} \rightsquigarrow \mathsf{case}\, e' \,\mathsf{of}\, \bar{c}'}$$

$$\textsc{Clause}\quad \frac{e \rightsquigarrow e'}{K\,\{\bar{\ell} = \bar{x}\} \mapsto e \rightsquigarrow K\,\{\bar{\ell} = \bar{x}\} \mapsto e'}$$

Fig. 7 Untyped term translation

5.2. Untyped meaning preservation

To establish that untyped defunctionalization preserves meaning, we exhibit a simulation between closed source configurations and their defunctionalized versions. The translation relation $\rightsquigarrow$ cannot play this role, because it is not stable under substitution. For instance, the translation of the function $\lambda^m x.y\,x$, which has a free variable y, is the one-field closure $m\,\{y = y\}$, whereas the translation of the function $\lambda^m x.r\,x$, which is obtained by substituting r for y, would be the zero-field closure $m\,\{\}$. Instead, we must define a simulation relation $\succsim$ that relates $\lambda^m x.r\,x$ with $m\,\{y = r\}$, so as to be stable under substitution of references for program variables.

In the following, let r_{apply} denote a fixed, distinguished reference. For the sake of brevity, we will also write r_{apply} for the store binding $r_{\mathit{apply}} = \lambda f.\lambda \mathit{arg}.\mathsf{case} f \,\mathsf{of}\, [\mathit{apply} \mapsto r_{\mathit{apply}}]\bar{c}_p$.

We define $\succsim$ by the same rules that specify $\rightsquigarrow$ (that is, by the rules of Fig. 7, where every occurrence of $\rightsquigarrow$ is replaced with $\succsim$), except Abs and App are replaced with SimAbs and SimApp (Fig. 8). In SimAbs, we write $\{[\bar{y} \mapsto \bar{r}]\bar{x}\}$ for $\{\bar{x} = [\bar{y} \mapsto \bar{r}]\bar{x}\}$. The relation $\succsim$ is extended to stores and closed configurations by SimStore and SimConfig, also in Fig. 8. Note that the right-hand store may contain garbage bindings (written $\bar{r}_1 = \bar{v}_1$) introduced by the reduction of the transformed program.

It is immediate to check that $\succsim$ extends $\rightsquigarrow$ in the following sense.

Lemma 5.1. *$e \rightsquigarrow e'$ implies $e \succsim [\mathit{apply} \mapsto r_{\mathit{apply}}]e'$.*

Furthermore, $\succsim$ is closed under substitution, as desired.

Fig. 8 Extra rules for the simulation relation

$$\textsc{SimAbs}\quad \frac{\lambda^m x.e \rightsquigarrow m\,\{\bar{x}\} \qquad \bar{y} \subseteq \bar{x}}{\lambda x.[\bar{y} \mapsto \bar{r}]e \succsim m\,\{[\bar{y} \mapsto \bar{r}]\bar{x}\}} \qquad \textsc{SimApp}\quad \frac{e_1 \succsim e_1' \qquad e_2 \succsim e_2'}{e_1\,e_2 \succsim r_{\mathit{apply}}\; e_1'\; e_2'}$$

$$\textsc{SimStore}\quad \frac{r_{\mathit{apply}} \# \bar{r} \# \bar{r}_1 \qquad \bar{v} \succsim \bar{v}'}{\bar{r} = \bar{v} \succsim r_{\mathit{apply}}; \bar{r} = \bar{v}'; \bar{r}_1 = \bar{v}_1} \qquad \textsc{SimConfig}\quad \frac{S \succsim S' \qquad e \succsim e'}{S \,/\, e \succsim S' \,/\, e'}$$

Lemma 5.2. *$e \succsim e'$ implies $[x \mapsto r]e \succsim [x \mapsto r]e'$.*

Last, $\succsim$ preserves values.

Lemma 5.3. *If $v \succsim e'$ holds, then e' is a value.*

We are now ready to prove that $\succsim$ is a simulation.

Lemma 5.4 (Simulation). *The following diagram commutes:*

$$\begin{tikzcd}
S_1 \,/\, e_1 \arrow[r] \arrow[d, "\succsim"'] & S_2 \,/\, e_2 \arrow[d, dotted, "\succsim"] \\
S_1' \,/\, e_1' \arrow[r, dotted, "+"] & S_2' \,/\, e_2'
\end{tikzcd}$$

Proof: By induction on the derivation of $S_1/e_1 \to S_2/e_2$. We give only the case of β-reduction, which is the most interesting one, and the case of reference allocation, which is straightforward but involves a minor α-conversion argument.

○ *Case* (β-reduction). In this case, e_1 is $r\ r'$, $S_1(r)$ is $\lambda x.e$, and e_2 is $[x \mapsto r']e$. By SIMCONFIG, SIMAPP and REF, the hypothesis $S_1/r\ r' \succsim S_1'/e_1'$ implies that e_1' must be

$$r_{apply}\ r\ r'.$$

By SIMCONFIG, SIMSTORE, and SIMABS, this hypothesis also implies that S_1' contains the binding r_{apply} and that $S_1'(r)$ is of the form

$$m\ \{[\bar{y} \mapsto \bar{r}]\bar{x}\},$$

where, for some expression e_m, we have

$$\begin{aligned} \bar{x} &= \mathrm{fv}(\lambda x.e_m) && (1) \\ \bar{y} &\subseteq \bar{x} && (2) \\ e &= [\bar{y} \mapsto \bar{r}]e_m && (3) \end{aligned}$$

Because we are dealing with closed configurations, e_2 is closed, which implies that e has no free variables other than x. By (3), this implies that the free variables of e_m form a subset of $\bar{y} \cup \{x\}$. By (1), this yields $\bar{x} \subseteq \bar{y}$. By (2), this means that $\bar{x}$ and $\bar{y}$ coincide, so we may write $\{[\bar{y} \mapsto \bar{r}]\bar{x}\}$ as $\{\bar{x} = \bar{r}\}$.

Now, by definition, $\bar{c}_p$ contains the clause $m\ \{\bar{x}\} \mapsto \mathsf{let}\, x = arg \,\mathsf{in}\, e'_m$, where e'_m is defined by $e_m \rightsquigarrow e'_m$. This allows us to build the following reduction sequence:

$$\begin{array}{ll}
 & S'_1/e'_1 \\
= & S'_1/r_{apply}\ r\ r' \\
\to & S'_1/(\lambda arg.\mathsf{case}\ r\ \mathsf{of}\ \bar{c}_p)\ r' \\
\to & S'_1; r'' = (\lambda arg.\mathsf{case}\ r\ \mathsf{of}\ \bar{c}_p)/r''\ r' \\
 & \text{with } r'' \,\#\, S'_1 \\
\to & S''_1/\mathsf{case}\ r\ \mathsf{of}\ (m\ \{\bar{x}\} \mapsto \mathsf{let}\, x = r' \,\mathsf{in}\, [apply \mapsto r_{apply}]e'_m) \mid \ldots \\
 & \text{with } S''_1 = S'_1; r'' = (\lambda arg.\mathsf{case}\ r\ \mathsf{of}\ \bar{c}_p) \\
\to & S''_1/\mathsf{let}\, x = r' \,\mathsf{in}\, [\bar{x} \mapsto \bar{r}][apply \mapsto r_{apply}]e'_m \\
 & \text{because } S'_1(r) \text{ is } m\ \{\bar{x} = \bar{r}\} \text{ and, by (1), } x \notin \bar{x} \text{ holds} \\
\to & S''_1/[x \mapsto r'][\bar{x} \mapsto \bar{r}][apply \mapsto r_{apply}]e'_m
\end{array}$$

There remains to verify that the simulation holds. By Lemma 5.1, we have $e_m \succsim [apply \mapsto r_{apply}]e'_m$. By Lemma 5.2, this implies

$$[x \mapsto r'][\bar{x} \mapsto \bar{r}]e_m \succsim [x \mapsto r'][\bar{x} \mapsto \bar{r}][apply \mapsto r_{apply}]e'_m,$$

which, by (3) and by equality of $\bar{y}$ and $\bar{x}$, may be written

$$e_2 \succsim [x \mapsto r'][\bar{x} \mapsto \bar{r}][apply \mapsto r_{apply}]e'_m.$$

The result follows by SIMCONFIG.

○ *Case* (reference allocation). In this case, e_1 is a value v_1, S_2 is $(S_1; r_1 = v_1)$, and e_2 is r_1, with $r_1 \,\#\, S_1$. Since configurations are considered equal modulo consistent renamings of references, we can require, without loss of generality, $r_1 \,\#\, S'_1$. Furthermore, the hypothesis $S_1/v_1 \succsim S'_1/e'_1$ implies that e'_1 is a value v'_1. We can then close the diagram as follows:

$$\begin{tikzcd}
S_1 / v_1 \arrow[r] \arrow[d, "\succsim"'] & S_1; r_1 = v_1 / r_1 \arrow[d, "\succsim"] \\
S'_1 / v'_1 \arrow[r] & S'_1; r_1 = v'_1 / r_1
\end{tikzcd}$$

□

Given a closed expression e, let us write $e \Uparrow$ if and only if the configuration $\emptyset/e$ admits an infinite reduction sequence; let us write $e \Downarrow$ if and only if $\emptyset/e$ reduces to a configuration whose right-hand component is a value. Then, the fact that defunctionalization preserves meaning, in an untyped setting, is stated by the next theorem.

Theorem 5.5. *$p \Uparrow$ implies $[\![p]\!] \Uparrow$. $p \Downarrow$ implies $[\![p]\!] \Downarrow$.*

Proof: To begin, let us recall that, by definition, $[\![p]\!]$ is

$$\mathsf{letrec}\ apply = \lambda f.\lambda arg.\mathsf{case} f\ \mathsf{of}\ \bar{c}_p\ \mathsf{in}\ p'.$$

Thus, the configuration $\emptyset/[\![p]\!]$ reduces, in one step, to

$$r_{apply}/[apply \mapsto r_{apply}]p'.$$

Furthermore, by Lemma 5.1, SIMSTORE, and SIMCONFIG, we have

$$\emptyset/p \gtrsim r_{apply}/[apply \mapsto r_{apply}]p'.$$

Now, assume p diverges. Then, $\emptyset/p$ admits an infinite reduction sequence. By Lemma 5.4, so does $r_{apply}/[apply \mapsto r_{apply}]p'$, hence so does $\emptyset/[\![p]\!]$, which proves that $[\![p]\!]$ diverges.

Last, assume p converges to a configuration of the form S/v. By the same argument as above, $\emptyset/[\![p]\!]$ must then reduce to a configuration that simulates S/v. By Lemma 5.3, the right-hand component of that configuration must be a value, hence $[\![p]\!]$ converges. □

The theorem states that defunctionalization preserves the termination behavior of the program. It does not apply to programs that go wrong; however, they are of little interest, since, in a realistic setting, they should be ruled out by some sound type system. Of course, if desired, it would be easy to prove that defunctionalization also preserves the property of going wrong.

5.3. Typed meaning preservation

We now sketch how the meaning preservation result may be lifted, if desired, to a typed setting. (The task is simple enough that it does not, in our opinion, warrant a detailed development.) Naturally, we consider the type system presented in Section 2; however, any other type system would do just as well, provided it is powerful enough to encode typed defunctionalization and has a type erasure semantics.

We begin by restricting the typed language defined in Section 2 so as to reflect the restrictions imposed on the untyped language at the beginning of Section 5. That is, we restrict letrec definitions to values, where values now include λ-abstractions, Λ-abstractions, and data constructor applications. Furthermore, we restrict type abstraction to values, that is, we replace the construct $\Lambda\alpha.e$ with $\Lambda\alpha.v$. Indeed, we do not wish Λ-abstractions to suspend computation, because they are erased when going down to the untyped language.

Next, we define a typed operational semantics for the language, which is identical to the untyped semantics of Section 5, except that type information is kept track of.

The two semantics are related by a simple *type erasure* property, stated as follows. Given a typed expression e, let $\lfloor e \rfloor$ be its untyped counterpart, obtained by erasing all type information. Then, we have:

Lemma 5.6. *Let* true, $\varnothing \vdash p : \tau_p$. *Then,* $p \Uparrow$ *is equivalent to* $\lfloor p \rfloor \Uparrow$, *and* $p \Downarrow$ *is equivalent to* $\lfloor p \rfloor \Downarrow$.

Last, we have insisted earlier that our version of defunctionalization is not type-directed. In other words, it commutes with type erasure. This is stated by the following lemma, whose proof is straightforward:

Lemma 5.7. *Let* true, $\varnothing \vdash p : \tau_p$. *Then,* $[\![\lfloor p \rfloor]\!]$ *is* $\lfloor [\![p]\!] \rfloor$.

Using Theorem 5.5 as well as the previous two Lemmas, it is now straightforward to establish that typed defunctionalization also preserves convergence and divergence. To conclude, types do not help establish the correctness of defunctionalization; on the contrary, we believe it is pleasant to get them out of the way, because an untyped correctness statement is stronger. Of course, our detour through an untyped semantics is not mandatory; if desired, one could carry out our simulation proof directly in a typed setting.

6. Remarks

6.1. Recursion

It is worth noting that recursive or mutually recursive functions in the source program do not cause any difficulty. Indeed, a set of mutually recursive bindings whose right-hand sides are λ-abstractions is mapped to a set of mutually recursive bindings whose right-hand sides are closures, that is, applications of data constructors to variables—in other words, values. Even under a call-by-value evaluation regime, mutually recursive definitions of values make perfect sense: see the semantics given in Section 5.1. Our treatment of mutually recursive function definitions corresponds to Morrisett and Harper's `fixpack`-based extension to closure conversion [24].

6.2. Optimizations

The reader may notice that the simply-typed version of defunctionalization described in the introduction is more efficient than the one we have presented, because specializing *apply* with respect to the ground types τ_1 and τ_2 means dispatching among fewer cases. If dispatch is implemented by a tree of binary comparisons, as in SmallEiffel [45], this leads to faster code. In fact, specialization is a simple way of exploiting the flow information provided for the source program by the type system. Our version of defunctionalization is naïve, and includes no such optimization. It is straightforward, however, to perform specialization in a similar way. Indeed, if τ_1 and τ_2 are arbitrary (not necessarily ground) types, whose free type variables are $\bar{\alpha}$, then one may define a specialized function $apply_{\exists\bar{\alpha}.\tau_1\to\tau_2}$, whose type is $\forall\bar{\alpha}.[\![\tau_1 \to \tau_2]\!] \to [\![\tau_1]\!] \to [\![\tau_2]\!]$, and whose code is identical to that of *apply*, except it contains branches *only* for the tags corresponding to source functions whose type is an instance of $\tau_1 \to \tau_2$. Branches for all other tags can be omitted, or, equivalently, can have right-hand sides that consist of a special expression *dead*, which is well-typed only under the false constraint. The resulting program is still well-typed, because, in each of these branches, inconsistent typing hypotheses are available, allowing false to be proved. More details about this mechanism are given by Xi [42], who points out that a type system equipped with guarded algebraic data types supports identification and elimination of dead branches. Thus, whereas, in the simply-typed case, type-based specialization was mandatory in order to achieve type preservation, it is now optional, but still possible.

Another source of inefficiency in our presentation of defunctionalization is our naïve treatment of multiple-argument functions. Indeed, we have adopted the view that all functions are unary. As a result, applying a (curried) function to multiple arguments causes the allocation of several intermediate closures, which immediately become garbage. In practice, it is possible to address this issue by defining yet more versions of *apply*, specialized for 2, 3, ... arguments, and to use these specialized versions at every call site where multiple arguments are available at once.

The specialization techniques described in the previous two paragraphs may be combined, without compromising our type preservation result. This yields defunctionalized programs containing many highly specialized versions of *apply*, each of which typically has few branches. Thus, this approach may allow producing reasonably small dispatch tables, by exploiting type information only, instead of relying on a separate closure analysis.

In a realistic implementation, one should also give special treatment to calls to known (let-bound) functions. There is no point in going through *apply* for such calls.

6.3. Defunctionalization as a programming idiom

Defunctionalization may be viewed not only as a compilation technique, but also as a tool that helps programmers transform programs and reason about them. Danvy and Nielsen [11] have pointed out that it is an inverse of Church's encoding, which means that it allows reasoning in terms of data structures instead of higher-order functions. This is nicely illustrated by the case of the *sprintf* function, whose type is notoriously difficult to express in Hindley and Milner's type system, because the value of its first argument dictates the number and types of its remaining arguments. Danvy [10] suggested a clever way of expressing *sprintf* in a Hindley-Milner setting by encoding format specifiers as first-class functions. More recently, Xi et al. [44] showed that *sprintf* may be expressed in a more direct style, where format specifiers are data structures, using guarded algebraic data types. We point out that the latter code is but a defunctionalized version of the former: it could, in principle, have been derived from it in a systematic manner. Thus, type-preserving defunctionalization turns clever continuation-based programs that work around the limitations of Hindley and Milner's type system back into first-order programs. Put another way, extending Hindley and Milner's type system with polymorphic recursion and guarded algebraic data types makes it possible to write more programs in direct style.

7. Concretization

So far, we have concentrated on getting rid of first-class *functions*, by translating λ-abstraction into injection (sum introduction) and function application into case analysis (sum elimination). Thus, an *abstract* object—a function—is given a *concrete* representation—a tag, applied to a number of parameters.

However, no specific property of functions is exploited by this scheme. In fact, the same transformation could be applied to virtually *any source language construct*, by translating its introduction form(s) into injection and its elimination form(s) into case analysis. Regardless of the intended meaning of the objects introduced by this construct, they can be represented using tags, which are interpreted at elimination sites.

This simple and general idea can be, and has been, exploited in many ways. It does not seem to have a name of its own yet: we refer to it as *concretization*. Our contribution is the (informal) remark that, using guarded algebraic data types, any program transformation that is an instance of concretization can be made type-preserving.

In the following, we illustrate this point by studying two such instances. One (Section 8) eliminates Rémy-style polymorphic records by translating them down to guarded algebraic data types. The other (Section 9) is a new compilation scheme for Haskell's type classes. Neither of the underlying untyped encodings is new: indeed, Zendra et al.'s [45] compilation scheme for SmallEiffel is similar to our encoding of polymorphic records, while

Furuse's [13] compilation scheme for G'Caml is similar to our encoding of type classes. The type-preserving versions that we present appear to be novel.

Like defunctionalization, instances of concretization appear, at first sight, to be whole program transformations. We come back to this issue in Section 10.

8. A new compilation scheme for polymorphic records

A typed programming language is said to have *polymorphic records*, or, equivalently, *records with polymorphic access*, if it allows defining a function that is applicable to every record value containing a field named ℓ, *regardless of which other fields are also present*, and returns the value of that field.

As a counter-example, a type system equipped with simple structural record types of the form $\{\bar{\ell} : \bar{\tau}\}$ does *not* have polymorphic records. Indeed, in such a system, a function that projects field ℓ out of its record argument must be written $\lambda x : \{\ell : \tau;\ \bar{\ell} : \bar{\tau}\}.(x.\ell)$, where $\bar{\ell}$ is a fixed, arbitrary set of field labels. Thus, it can be applied only to records that contain *exactly* the fields ℓ and $\bar{\ell}$. It is not applicable to *all* records containing *at least* field ℓ.

8.1. A type system with polymorphic records

There are several ways of adding support for polymorphic records to a type system. One approach consists in introducing subtyping [6], so that every record value containing *at least* field ℓ has type $\{\ell : \tau\}$ for some type τ. Another consists in introducing rows [32, 31], so that every record value containing *at least* field ℓ has type $\{\ell : \mathit{Pre}\ \tau;\ \rho\}$ for some type τ and some row ρ. Here, we follow the second approach, so as to avoid studying the combination of subtyping and guarded algebraic data types, which is more involved [37, 43].

Thus, we now extend the programming language of Section 2 with polymorphic records. We provide new syntax for creating and accessing polymorphic records:

$$e ::= \cdots \mid \{\!\{\bar{\ell} = \bar{e}\}\!\} \mid e\,\#\,\ell$$

The grammar of types found in Section 2 is extended with a new type constructor for polymorphic records and with syntax for rows:

$$\begin{aligned}\tau &::= \cdots \mid \{\!\{\rho\}\!\}\\ \rho &::= \alpha \mid (\ell : \theta;\ \rho) \mid \partial\theta\\ \theta &::= \mathit{Pre}\ \tau \mid \mathit{Abs}\end{aligned}$$

In fact, this grammar is slightly too permissive, and must be restricted using kinds. We omit this aspect; for details, the reader is referred to previous treatments of rows [31, 32]. If ρ is a row, then $\{\!\{\rho\}\!\}$ is a polymorphic record type. Although rows are finite terms, they denote total functions from field labels to types. The row $(\ell : \theta;\ \rho)$ denotes the function that maps the label ℓ to the field type θ and that coincides with the row ρ at other labels. The row $\partial\theta$ denotes the function that maps every label to the type θ. To support this intuition, rows are identified modulo the equations $(\ell_1 : \theta_1;\ (\ell_2 : \theta_2;\ \rho)) = (\ell_2 : \theta_2;\ (\ell_1 : \theta_1;\ \rho))$ and $(\ell : \theta;\ \partial\theta) = \partial\theta$. Since rows denote *total* functions, the domain of a row does not tell which fields of a record are defined or undefined. The unary field type constructor *Pre* and the nullary field type constructor *Abs* are introduced for this purpose; they stand for *present* and *absent*, respectively.

$$\textsc{Record}\quad \frac{C, \Gamma \vdash \bar{e} : \bar{\tau}}{C, \Gamma \vdash \{\!\{\bar{\ell} = \bar{e}\}\!\} : \{\!\{\bar{\ell} : \mathit{Pre}\,\bar{\tau};\ \partial \mathit{Abs}\}\!\}} \qquad \textsc{Proj}\quad \frac{C, \Gamma \vdash e : \{\!\{\ell : \mathit{Pre}\,\tau;\ \rho\}\!\}}{C, \Gamma \vdash e\#\ell : \tau}$$

Fig. 9 Typing rules for polymorphic records

Next, new rules for typechecking polymorphic records are introduced (Fig. 9). Record states that a record whose fields are $\ell_1, \ldots, \ell_n$ must have a type of the form $\{\!\{\ell_1 : \mathit{Pre}\,\tau_1;\ \ldots;\ \ell_n : \mathit{Pre}\,\tau_n;\ \partial \mathit{Abs}\}\!\}$. This type involves a row that maps every ℓ_i to $\mathit{Pre}\,\tau_i$ and that maps every other field label to *Abs*. (In the rule's conclusion, we abuse notation and apply the unary type constructor *Pre*, pointwise, to a vector of types, yielding another vector of types.) Proj states that projecting field ℓ out of the record e is permitted if the row that describes e maps ℓ to a type of the form $\mathit{Pre}\,\tau$ (as opposed to *Abs*). The remainder of the row, known as ρ, is irrelevant: the rule doesn't care which other fields are present.

The modified type system offers polymorphic record access: indeed, the polymorphic function $\Lambda\alpha.\Lambda\beta.\lambda x : \{\!\{\ell : \mathit{Pre}\,\alpha;\ \beta\}\!\}.(x \# \ell)$ is applicable, after suitable type applications, to every record value that contains a field named ℓ, regardless of which other fields are also present.

8.2. Compiling polymorphic records

Compiling standard records is easy. Let field labels be ordered in some arbitrary, fixed manner. Every standard record type $\{\bar{\ell} : \bar{\tau}\}$ may be written, in a canonical way, under the form $\{\ell_0 : \tau_0;\ \ldots;\ \ell_{n-1} : \tau_{n-1}\}$, where $\ell_0 < \cdots < \ell_{n-1}$. A value of such a type can then be represented in memory as a block of n words, with the contents of field ℓ_i located at offset i in the block. This compilation scheme only assumes that every value fits in one memory word, regardless of its type.

Compiling polymorphic records is more difficult. The above polymorphic access function must be able to extract the field ℓ out of a record value that must indeed contain a field named ℓ, but whose structure is otherwise unknown. Thus, it is impossible to arrange for the field to be found at a statically determined offset.

One solution, followed by Ohori [27] and by Gaster and Jones [14], consists in parameterizing every function with extra arguments, representing the offsets of the fields that the function needs to access.

Another approach consists in letting every record value carry information that allows mapping field labels to integer offsets. This approach has been widely investigated in the related setting of *dynamic dispatch* for object-oriented languages. The information carried by every record or object can be a complex data structure [46] or an atomic tag. The SmallEiffel compiler [45], for instance, attaches a tag to every object, and implements dynamic dispatch via case analysis over tags. This compilation scheme is an instance of concretization, applied to objects. In the following, we describe a type-preserving version of this encoding, applied to polymorphic records.

One should point out that, even though our transformation eliminates polymorphic records, it does not eliminate rows: typechecking translated programs still requires rows. This is unsurprising: it is well-known that type-preserving compilation requires target languages with rich type systems. Indeed, because compiling down to a low-level language obscures some of a program's structure, proving that a translated program is still type-correct often requires the type system of the target language to be at least as elaborate as that of the source language, and sometimes even more [25].

8.3. Concretizing polymorphic records

We now sketch how polymorphic records may be compiled away via concretization. We do not prove that the encoding is semantics-preserving, but do prove that it is type-preserving. As evidenced by its use in the SmallEiffel compiler [45], the untyped version of this encoding is not new. To the best of our knowledge, the recognition that this transformation is an instance of concretization, and can be made type-preserving, is novel.

We require every polymorphic record creation expression that appears in the source program to carry a distinct label m; we write $\{\!\{\bar{\ell} = \bar{e}\}\!\}^m$ for such an expression.

The transformed program is defined under a type signature extended with a fresh unary algebraic data type constructor *Record*. The type translation is simple: the native polymorphic record type constructor $\{\!\{\cdot\}\!\}$ is translated to *Record*, while all other type formers are preserved.

$$[\![\{\!\{\rho\}\!\}]\!] = \mathit{Record}\ [\![\rho]\!]$$

The type constructor *Record* is parameterized by a row.

The transformed program is defined under a transformed and extended data signature. As in Section 3, the new data signature $\mathcal{D}'$ contains the translated original data signature $[\![\mathcal{D}]\!]$. Furthermore, for every polymorphic record creation expression that appears in the source program and whose typing subderivation ends with

$$C, \Gamma \vdash \{\!\{\bar{\ell} = \bar{e}\}\!\}^m : \{\!\{\bar{\ell} : \mathit{Pre}\,\bar{\tau};\ \partial \mathit{Abs}\}\!\},$$

the new data signature $\mathcal{D}'$ contains a unary data constructor

$$m :: \forall \bar{\alpha}[[\![C]\!]].\{\bar{\ell} : [\![\bar{\tau}]\!]\} \rightarrow \mathit{Record}\,(\bar{\ell} : \mathit{Pre}\,[\![\bar{\tau}]\!];\ \partial \mathit{Abs}),$$

where $\bar{\alpha}$ stands for $\mathrm{ftv}(C, \bar{\tau})$, ordered in a fixed, arbitrary manner. In other words, the data constructor m maps a tuple of values, whose types are $[\![\bar{\tau}]\!]$, to (the encoding of) a polymorphic record, whose type is $[\![\{\!\{\bar{\ell} : \mathit{Pre}\,\bar{\tau};\ \partial \mathit{Abs}\}\!\}]\!]$. That is, a polymorphic record is represented as a tuple, carrying a tag that identifies its layout. To access such a value, one must first examine its tag, which reveals how the tuple is laid out; it is then possible to access each of its components at a statically known offset.

To precisely define this process, we introduce, for every field label ℓ, a set of clauses $\bar{c}_\ell$, which reflects the analysis required to access field ℓ. This set consists of two distinct families of clauses.

First, $\bar{c}_\ell$ contains all clauses of the form

$$m\,\bar{\alpha}\,\{\bar{\ell} = \bar{x}\} \mapsto x,$$

where m, $\bar{\alpha}$, and $\bar{\ell}$ are as above, ℓ is a member of $\bar{\ell}$, and x is the variable associated with ℓ in the vector of bindings $\bar{\ell} = \bar{x}$. Note that field ℓ is stored at a statically known offset in a tuple of the form $\{\bar{\ell} = \cdot\}$.

Second, $\bar{c}_\ell$ contains all clauses of the form

$$m\,\bar{\alpha}\,\{\bar{\ell} = \bar{x}\} \mapsto \mathit{dead},$$

where m, $\bar{\alpha}$, and $\bar{\ell}$ are as above, ℓ is *not* a member of $\bar{\ell}$, and the special expression *dead* is well-typed only under the false constraint. As we shall see, these clauses are provably dead, which means that, in the final compiled code, they will be omitted. We include them because our type system requires case analyses to be exhaustive (Section 2).

The clauses $\bar{c}_\ell$ are intended to encode exactly the polymorphic record access operation: that is, they map the encoding of a polymorphic record to the contents of its ℓ field. Thus, for the encoding to be type-preserving, we need to prove that these clauses, when applied to an argument of type $[\![\{\!\{\ell : \mathit{Pre}\,\tau;\ \rho\}\!\}]\!]$, yield a result of type $[\![\tau]\!]$ (see PROJ in Fig. 9). This must hold for arbitrary τ and ρ. This fact is stated by the following lemma.

Lemma 8.1. $\mathsf{true}, \varnothing \vdash \bar{c}_\ell : \mathit{Record}\ (\ell : \mathit{Pre}\,[\![\tau]\!];\ [\![\rho]\!]) \rightarrow [\![\tau]\!]$.

Proof: We must prove that every clause of the form $m\ \bar{\alpha}\ \{\bar{\ell} = \bar{x}\} \mapsto \bar{x}.\ell$ has type $\mathit{Record}\ (\ell : \mathit{Pre}\,[\![\tau]\!];\ [\![\rho]\!]) \rightarrow [\![\tau]\!]$, where τ and ρ are arbitrary. (We let $\bar{x}.\ell$ stand for x if $\ell = x$ is a member of the vector of bindings $\bar{\ell} = \bar{x}$ and for *dead* otherwise.) By definition of $\mathcal{D}'$ and by CLAUSE, this reduces to proving

$$[\![C]\!] \wedge (\bar{\ell} : \mathit{Pre}\,[\![\bar{\tau}]\!];\ \partial \mathit{Abs}) = (\ell : \mathit{Pre}\,[\![\tau]\!];\ [\![\rho]\!]), \bar{x} : [\![\bar{\tau}]\!] \vdash \bar{x}.\ell : [\![\tau]\!],$$

where C and $\bar{\tau}$ are given by the existence of the typing judgement $C, \Gamma \vdash \{\!\{\bar{\ell} = \bar{e}\}\!\}^m : \{\!\{\bar{\ell} : \mathit{Pre}\,\bar{\tau};\ \partial \mathit{Abs}\}\!\}$ in the typing derivation of the source program. We now distinguish two cases: either the constraint $[\![C]\!] \wedge (\bar{\ell} : \mathit{Pre}\,[\![\bar{\tau}]\!];\ \partial \mathit{Abs}) = (\ell : \mathit{Pre}\,[\![\tau]\!];\ [\![\rho]\!])$ is satisfiable, or it is not.

- ∘ It is satisfiable. Then, $\ell \in \bar{\ell}$ must hold, because this constraint would otherwise entail the unsatisfiable type equation $\mathit{Abs} = \mathit{Pre}\,[\![\tau]\!]$. Thus, $\bar{x}.\ell$ stands for x, with $\ell = x$ a member of the vector of bindings $\bar{\ell} = \bar{x}$. By VAR, x has type $[\![\bar{\tau}.\ell]\!]$, where $\bar{\tau}.\ell$ stands for the type found at index ℓ in the vector $\bar{\tau}$. Furthermore, because ℓ is a member of $\bar{\ell}$, the constraint $(\bar{\ell} : \mathit{Pre}\,[\![\bar{\tau}]\!];\ \partial \mathit{Abs}) = (\ell : \mathit{Pre}\,[\![\tau]\!];\ [\![\rho]\!])$ entails the equation $[\![\bar{\tau}.\ell]\!] = [\![\tau]\!]$. Thus, by CONV, x has type $[\![\tau]\!]$.
- ∘ It is unsatisfiable. (The situations where this constraint is unsatisfiable are discussed in Section 8.4.) Then, $\bar{x}.\ell$ stands for *dead*, which admits any type under a false constraint. □

We can now complete the definition of the compilation scheme. A compositional term translation is defined by a judgement of the form $C, \Gamma \vdash e : \tau \rightsquigarrow e'$, whose most important two derivation rules appear in Fig. 10. RECORD translates a polymorphic record $\{\!\{\bar{\ell} = \bar{e}\}\!\}^m$ to an application of the tag m to a tuple $\{\bar{\ell} = \bar{e}'\}$. PROJ translates an access to field ℓ of a polymorphic record into an analysis of the tag, followed by an access to the underlying tuple, as expressed by the clauses $\bar{c}_\ell$. All other rules (omitted) preserve the structure of the expression at hand, using the type translation defined above to deal with type annotations. The present translation is somewhat simpler than the one studied in Section 3, because there is no need for a recursive *apply*-like function.

It is straightforward to check that the translation is type-preserving:

Fig. 10 Translation rules for polymorphic records

RECORD
$$\frac{C, \Gamma \vdash \bar{e} : \bar{\tau} \rightsquigarrow \bar{e}' \qquad \bar{\alpha} = \mathrm{ftv}(C, \bar{\tau})}{C, \Gamma \vdash \{\!\{\bar{\ell} = \bar{e}\}\!\}^m : \{\!\{\bar{\ell} : \mathit{Pre}\,\bar{\tau};\ \partial \mathit{Abs}\}\!\} \rightsquigarrow m\ \bar{\alpha}\ \{\bar{\ell} = \bar{e}'\}}$$

PROJ
$$\frac{C, \Gamma \vdash e : \{\!\{\ell : \mathit{Pre}\,\tau;\ \rho\}\!\} \rightsquigarrow e'}{C, \Gamma \vdash e\#\ell : \tau \rightsquigarrow \mathsf{case}\ e'\ \mathsf{of}\ [[\![\tau]\!]]\ \bar{c}_\ell}$$

Theorem 8.2. *$C, \Gamma \vdash e : \tau \rightsquigarrow e'$ implies $[\![C]\!], [\![\Gamma]\!] \vdash e' : [\![\tau]\!]$.*

Proof: By structural induction on the derivation of $C, \Gamma \vdash e : \tau \rightsquigarrow e'$. In each case, we use the notations of Fig. 10.

- *Case* RECORD. Applying the induction hypothesis to the first premise yields $[\![C]\!], [\![\Gamma]\!] \vdash \bar{e}' : [\![\bar{\tau}]\!]$. By definition of the data signature $\mathcal{D}'$ and by DATA, this implies

$$[\![C]\!], [\![\Gamma]\!] \vdash m\,\bar{\alpha}\,\{\bar{\ell} = \bar{e}'\} : Record\ (\bar{\ell} : Pre\,[\![\bar{\tau}]\!];\ \partial Abs).$$

- *Case* PROJ. Applying the induction hypothesis to the premise yields

$$[\![C]\!], [\![\Gamma]\!] \vdash e' : Record\ (\ell : Pre\,[\![\tau]\!];\ [\![\rho]\!]).$$

  Furthermore, Lemma 8.1, Lemma 2.1, and WEAKEN yield

$$[\![C]\!], [\![\Gamma]\!] \vdash \bar{c}_\ell : Record\ (\ell : Pre\,[\![\tau]\!];\ [\![\rho]\!]) \rightarrow [\![\tau]\!].$$

  By CASE, these imply $[\![C]\!], [\![\Gamma]\!] \vdash \mathsf{case}\, e'\, \mathsf{of}\, [[\![\tau]\!]]\, \bar{c}_\ell : [\![\tau]\!]$. □

Example. The code fragment in Fig. 11 is meant to illustrate the main features of the compilation scheme. The function *create* allocates a polymorphic record that holds a single field, named *pos*, whose value is the parameter x. Interestingly, the type of x, namely α, is abstract at this point, and this flexibility is exploited in the remainder of the code fragment, where *create* is applied to *int* and to *bool* at two distinct call sites. The function *bump* expects a polymorphic record p that contains at least one field named *pos*. It extracts the value of that field, increments it, and creates a new record. The function *pair* allocates a pair, represented as a polymorphic record with fields *left* and *right*. The body of the program consists of a pair of applications of *bump* and *create*.

The translation of this code fragment is shown in Fig. 12. Every polymorphic record creation expression has been replaced with an application of a tag (m_1, m_2, or m_3) to an appropriate number of type and value parameters.

The polymorphic record access expression $p\,\#pos$ has been replaced with a case analysis over p. All left-hand sides in this case analysis involve the binding $pos = x$. However, the fields other than *pos* may differ. In the branch for m_1, *pos* is the only field in the tuple, so this branch really means "fetch the field at offset 0 in this tuple". In the branch for m_3, there is also a *color* field, so, assuming (for example's sake) that fields are laid out in memory in

```
let $create = \Lambda\alpha.\lambda x : \alpha.\{\!\{pos = x\}\!\}^{m_1}$ in
let $bump = \Lambda\beta.\lambda p : \{\!\{pos : Pre\ int;\ \beta\}\!\}$.
   let $x = p\#pos$ in
   $create\ int\ (x + 1)$ in
let $pair = \Lambda\gamma_1.\Lambda\gamma_2.\lambda x_1 : \gamma_1.\lambda x_2 : \gamma_2.\{\!\{left = x_1;\ right = x_2\}\!\}^{m_2}$ in
$pair$
   $\{\!\{pos : Pre\ int;\ \partial Abs\}\!\}$
   $\{\!\{pos : Pre\ bool;\ \partial Abs\}\!\}$
   $(bump\ (color : Pre\ int;\ \partial Abs)\ \{\!\{color = 1;\ pos = 0\}\!\}^{m_3})$
   $(create\ bool\ true)$
```

Fig. 11 A sample code fragment

```
$m_1$ :: $\forall\alpha.\{pos : \alpha\} \rightarrow Record\ (pos : Pre\,\alpha;\ \partial Abs)$
$m_2$ :: $\forall\gamma_1\gamma_2.\{left : \gamma_1;\ right : \gamma_2\} \rightarrow$
                $Record\ (left : Pre\,\gamma_1;\ right : Pre\,\gamma_2;\ \partial Abs)$
$m_3$ :: $\{color, pos : int\} \rightarrow Record\ (color, pos : Pre\ int;\ \partial Abs)$

let $create = \Lambda\alpha.\lambda x : \alpha.(m_1\,\alpha\,\{pos = x\})$ in
let $bump = \Lambda\beta.\lambda p : Record\ (pos : Pre\ int;\ \beta)$.
  let $x$ = case $p$ of $[int]$
             | $m_1\,\alpha\,\{pos = x\} \mapsto x$
             | $m_3\,\{color = y;\ pos = x\} \mapsto x$
  in
  $create\ int\ (x + 1)$ in
let $pair$ =
  $\Lambda\gamma_1.\Lambda\gamma_2.\lambda x_1 : \gamma_1.\lambda x_2 : \gamma_2.(m_2\,\gamma_1\gamma_2\,\{left = x_1;\ right = x_2\})$ in
$pair$
  $(Record\ (pos : Pre\ int;\ \partial Abs))$
  $(Record\ (pos : Pre\ bool;\ \partial Abs))$
  $(bump\ (color : Pre\ int;\ \partial Abs)\ (m_3\ \{color = 1;\ pos = 0\}))$
  $(create\ bool\ true)$
```

Fig. 12 The translated code fragment

alphabetical order, this branch really means "fetch the field at offset 1 in this tuple". In other words, examining the tag allows telling at which offset the *pos* field is stored.

The case analysis does not explicitly contain a branch for m_2. One could add such a branch, but it would be provably dead. Indeed, p has type *Record* (*pos* : *Pre int*; β), while m_2 builds values of type *Record* (*left* : *Pre* γ_1; *right* : *Pre* γ_2; ∂Abs). These types are not unifiable, so no value that carries the tag m_2 can ever be substituted for p. This is discussed more thoroughly in Section 8.4.

8.4. Remarks

Eliminating dead clauses. For every type τ and row ρ, the proof of Lemma 8.1 partitions $\bar{c}_\ell$ into two subsets, consisting respectively of the clauses where $[\![C]\!] \wedge (\bar{\ell} : Pre\,[\![\bar{\tau}]\!];\ \partial Abs) = (\ell : Pre\,[\![\tau]\!];\ [\![\rho]\!])$ is satisfiable and of the clauses where it is not. The clauses in the second group are provably dead. Indeed, when this constraint is unsatisfiable, the type of the polymorphic record created at site m does not match the type of the polymorphic record that is being accessed. Thus, the latter cannot carry the tag m. In practice, a compiler can and will detect and eliminate such dead clauses [42]. Thus, only the first subset of $\bar{c}_\ell$ will appear in the compiled code; yet, we have a static guarantee that the case analysis cannot fail.

According to the proof of Lemma 8.1, if a clause in $\bar{c}_\ell$ is live, then $\ell \in \bar{\ell}$ must hold, that is, the polymorphic record created at site m must have a field named ℓ. It is interesting to note, however, that the converse is not true: it is possible for a clause to satisfy $\ell \in \bar{\ell}$ and to nevertheless be provably dead. Indeed, for the above constraint to be unsatisfiable, it is sufficient for some field (possibly other than ℓ) to be known to be present at the access site and absent at the creation site, or conversely, or to be present at both sites, but with incompatible types. Thus, there are plenty of ways to prove that a clause is dead, and, thereby, to reduce the number of cases that must be considered at each polymorphic record access site.

Setting up fewer clauses. Although, for simplicity, we have associated a distinct tag m with every polymorphic record creation site, it is really only necessary to distinguish between polymorphic records with distinct *domains*. In other words, one *could* translate all source expressions of the form $\{\!\{\bar\ell = \bar e\}\!\}$, where $\bar\ell$ is fixed, to applications of a single data constructor $m^{\bar\ell}$, whose type scheme is

$$m^{\bar\ell} :: \forall\bar\alpha[\mathsf{true}].\{\bar\ell : \bar\alpha\} \to \mathit{Record}\ (\bar\ell : \mathit{Pre}\,\bar\alpha;\ \partial\mathit{Abs})\,.$$

Notice that the constraint $[\![C]\!]$ has been replaced with true and that the types $[\![\bar\tau]\!]$ have been abstracted away and replaced with a vector of type variables $\bar\alpha$. This allows $m^{\bar\ell}$ to be used at every creation site for a polymorphic record of domain $\bar\ell$.

If this suggestion was followed, then there would be fewer branches to select from when accessing a polymorphic record. In particular, when the domain of the record is statically known at the access site (that is, when its type does not involve a row variable), then only one branch would remain, so polymorphic access would effectively be compiled down to standard record access. In other words, one would be able to guarantee that, at access sites that do not exploit the extra expressiveness offered by polymorphic records, no time penalty is paid for their use.

Unfortunately, at first sight, this suggestion appears at odds with the idea put forth in the previous paragraph. Indeed, assigning less specific type schemes to the data constructors prevents valuable type information from being carried from polymorphic record creation sites to access sites. As a result, some clauses, which could have been proven dead using this information, may now appear potentially live. In other words, at access sites where the domain of the polymorphic record is not statically known, we may end up with branches that could have been discarded if the data constructors had been assigned more specific type schemes.

Combining the two optimizations. Fortunately, the ideas discussed in the previous two paragraphs can really be made compatible, if we make a minor extension of the type system. Let $\bar\ell$ be a fixed set of field labels. The idea is to introduce a single data constructor $m^{\bar\ell}$, as in the previous paragraph, and nevertheless to assign it a very specific type scheme, as follows. Assume that the source program contains k polymorphic record creation sites of domain $\bar\ell$. Let their typing subderivations end with

$$C_i, \Gamma_i \vdash \{\!\{\bar\ell = \bar e_i\}\!\} : \{\!\{\bar\ell : \mathit{Pre}\,\bar\tau_i;\ \partial\mathit{Abs}\}\!\},$$

where i ranges over $\{1, \ldots, k\}$. Let $\bar\alpha_i$ stand for $\mathrm{ftv}(C_i, \bar\tau_i)$. Then, the type scheme ascribed to $m^{\bar\ell}$ is

$$m^{\bar\ell} :: \bigwedge_{i=1}^{k} \forall\bar\alpha_i[[\![C_i]\!]].\{\bar\ell : [\![\bar\tau_i]\!]\} \to \mathit{Record}\ (\bar\ell : \mathit{Pre}\,[\![\bar\tau_i]\!];\ \partial\mathit{Abs})\,.$$

This is exactly the type-theoretic *intersection* of the type schemes that would be individually assigned to the k data constructors $m_1^{\bar\ell}, \ldots, m_k^{\bar\ell}$ if we followed the definitions in Section 8.3 and allocated a distinct data constructor for each polymorphic record creation site. Clearly, this is potentially much more specific than the type scheme that was initially suggested for $m^{\bar\ell}$. In fact, it is as specific as possible, while still meeting the requirement that applications of $m^{\bar\ell}$ at all sites should be well-typed.

The intersection type constructor, however, is not part of the type system studied so far, so we must extend it. The required extension is extremely simple: we extend the syntax of *constraints* by letting constraints consist of conjunctions *and disjunctions* of type equations. (With this extension, entailment checks become more expensive, but remain decidable.) Nothing else needs be modified! Then, the above intersection of type schemes may be written as a single type scheme, as follows:

$$\forall \bar{\alpha}\bar{\alpha}_1 \ldots \bar{\alpha}_k \left[\bigvee_{i=1}^{k} (\llbracket C_i \rrbracket \wedge \llbracket \bar{\tau}_i \rrbracket = \bar{\alpha}) \right] . \{\bar{\ell} : \bar{\alpha}\} \rightarrow \mathit{Record}\ (\bar{\ell} : \mathit{Pre}\, \bar{\alpha};\ \partial \mathit{Abs})$$

We let the reader check the following two facts. First, the translation is still type-preserving. Second, at every polymorphic record access site, the clause associated with $m^{\bar{\ell}}$ is provably dead if and only if all of the clauses that would be associated with $m_1^{\bar{\ell}}, \ldots, m_k^{\bar{\ell}}$ in the approach of Section 8.3 are provably dead. In other words, we have met our goal of setting up fewer clauses and nevertheless allowing as many dead clauses as possible to be detected.

9. A new compilation scheme for Haskell's type classes

We now exploit concretization to eliminate the *dictionary records* introduced by the standard compilation scheme for Haskell's type classes [17, 40]. Where the standard encoding creates a dictionary record, we create a data constructor application. Where the standard encoding accesses a dictionary record, we perform case analysis. This leads to a new compilation scheme for type classes, where dictionaries are represented as algebraic data structures, instead of as records of functions. Thus, we propose a rather different "explanation" of type classes.

This new encoding is also type-preserving. Its target type system is an extension of Hindley and Milner's discipline with polymorphic recursion and guarded algebraic data types.

We cannot recall in this paper the definition of type classes, nor the full details of their standard compilation scheme. Both are explained by Hall et al. [16, 17]. We assume that the reader is familiar at least with type classes, although perhaps not with their encoding.

The standard encoding of type classes and our new encoding share much of their structure. We begin with an overview of this common architecture. Then, we describe the specific aspects of the standard encoding and those of our own encoding. We borrow much of our notation from Hall et al. [16].

As an illustration, we use a few basic examples, given in Fig. 13. They are definitions for two classes, *Eq* and *Ord*, and for a few instances of these classes.

9.1. Architecture of a type class encoding

To begin, every encoding must define how dictionaries are represented at runtime. To do so, one must define within the target type system, for every class name κ, a unary type constructor (or type abbreviation), also written κ. In other words, whereas, in the Haskell language, every class name κ acts as a unary predicate over types (the formula $\kappa\ \tau$ asserts that the type τ is a member of the class κ), in the target language, κ acts as a unary type operator (the type $\kappa\ \tau$ describes values that form *evidence* that the type τ is a member of the class κ). In the following, we also refer to evidence values as *dictionaries*.

```
class  Eq a where
  (==) :: a → a → Bool

instance  Eq Int where
  (==) = eqInt
          where eqInt :: Int → Int → Bool
                eqInt = ...
instance  (Eq a) ⇒ Eq [a] where
  (==) = eqList (==)
          where eqList :: (a → a → Bool) → [a] → [a] → Bool
                eqList = ...

class  (Eq a) ⇒ Ord a where
  (<) :: a → a → Bool
  (≤) :: a → a → Bool

instance  Ord Int where
  ...
instance  (Ord a) ⇒ Ord [a] where
  ...
```

Fig. 13 Sample class and instance declarations

Second, in every encoding, a class declaration of the general form

$$\begin{array}{ll}\text{class } (\kappa_1\ \alpha, \ldots, \kappa_m\ \alpha) \to \kappa\ \alpha & \qquad \textit{(class-decl)} \\ \text{where } var_1 : \tau_1, \ldots, var_n : \tau_n & \end{array}$$

is translated into m superclass accessors and n method accessors. A superclass accessor has type $\forall\alpha.\kappa\ \alpha \to \kappa_i\ \alpha$, for some $i \in \{1, \ldots, m\}$. Its role is to turn evidence for the formula $\kappa\ \alpha$ into evidence for the formula $\kappa_i\ \alpha$, where κ_i is a superclass of κ. A method accessor has type $\forall\alpha.\kappa\ \alpha \to \tau_j$, for some $j \in \{1, \ldots, n\}$. Its role is to turn evidence for the formula $\kappa\ \alpha$ into an implementation for method var_j at type α.

For instance, the class declarations in Fig. 13 must give rise to three method accessors and one superclass accessor, whose types are:

$$\begin{array}{rl}(==) & :: \forall\alpha.Eq\ \alpha \to \alpha \to \alpha \to Bool \\ getEqFromOrd & :: \forall\alpha.Ord\ \alpha \to Eq\ \alpha \\ (<) & :: \forall\alpha.Ord\ \alpha \to \alpha \to \alpha \to Bool \\ (\leq) & :: \forall\alpha.Ord\ \alpha \to \alpha \to \alpha \to Bool\end{array}$$

Third, in every encoding, an instance declaration of the general form

$$\begin{array}{ll}\text{instance } (\kappa'_1\ \tau'_1, \ldots, \kappa'_l\ \tau'_l) \to \kappa\ \tau & \qquad \textit{(inst-decl)} \\ \text{where } var_1 = exp_1, \ldots, var_n = exp_n & \end{array}$$

must be translated into a value of type

$$\forall\alpha_1 \cdots \alpha_k.\kappa'_1\ \tau'_1 \to \cdots \to \kappa'_l\ \tau'_l \to \kappa\ \tau.$$

Here, $\alpha_1, \ldots, \alpha_k$ are the type variables that occur free in the instance declaration; for more details, see Hall et al. [16]. In other words, a nonparameterized instance declaration ($l = 0$) is translated to a dictionary. A parameterized instance declaration ($l > 0$) is translated to a function that accepts l dictionaries and produces a new dictionary. In the following, we speak of "parameterized dictionaries," regardless of whether l is zero or nonzero.

9.2. The standard encoding

In the standard encoding, dictionaries are records containing (pointers to) superclass dictionaries and method implementations. Thus, in the translated program, κ is defined as an abbreviation for a suitable record type. Hall et al. [16] use unlabeled records, that is, tuples, and write $\langle\cdot\rangle$ for the tuple type constructor.

For instance, if the declarations in Fig. 13 are part of the source program, then, in the transformed program, $Eq\ \alpha$ is defined as an abbreviation for the tuple type $\langle \alpha \rightarrow \alpha \rightarrow Bool\rangle$, because a dictionary for $Eq\ \alpha$ consists of an implementation for the method $(==)$ at type α. Similarly, $Ord\ \alpha$ is defined as an abbreviation for $\langle Eq\ \alpha, \alpha \rightarrow \alpha \rightarrow Bool, \alpha \rightarrow \alpha \rightarrow Bool\rangle$, because a dictionary for $Ord\ \alpha$ consists of a dictionary for $Eq\ \alpha$ and of implementations for the methods $(<)$ and $(\leq)$ at type α.

Second, superclass and method accessors are simply defined as projections out of a dictionary tuple; see Hall et al. [16]. For instance, *getEqFromOrd* is defined as $\lambda\mathbf{dOrd}.\pi_1^3\ \mathbf{dOrd}$. That is, *getEqFromOrd* extracts the first component of its argument, which is expected to be a three-component tuple of type $Ord\ \alpha$.

Third, the parameterized dictionary value associated with an instance declaration of the form *(inst-decl)* is defined as follows:

$$\begin{array}{l}\Lambda\alpha_1 \ldots \alpha_k.\\ \quad \lambda\mathbf{dvar}_1 : \kappa_1'\ \tau_1'. \ldots \lambda\mathbf{dvar}_l : \kappa_l'\ \tau_l'.\\ \quad\quad \langle \mathbf{dexp}_1, \ldots, \mathbf{dexp}_m, \mathbf{exp}_1, \ldots, \mathbf{exp}_n\rangle\end{array}$$

Here, every $\mathbf{dexp}_i$ builds a dictionary of type $\kappa_i\ \tau$, where $\kappa_1, \ldots, \kappa_m$ are the superclasses of κ, while every $\mathbf{exp}_j$ is the translation of exp_j and is an implementation for the method var_j. Both $\mathbf{dexp}_i$ and $\mathbf{exp}_j$ may contain free occurrences of the variables $\mathbf{dvar}_1, \ldots, \mathbf{dvar}_l$.

9.3. The new encoding

As announced earlier, the basic idea underlying the new encoding is to eliminate dictionary records by replacing them with data constructor applications. In other words, the new encoding could be viewed as the composition of two phases, where the first phase is the standard encoding and the second phase eliminates the dictionary records introduced by the previous phase. To avoid complicating matters, we do not insist on such a view; instead, we give a direct definition of the new encoding.

To begin, dictionaries are algebraic data structures. Thus, in the translated program, κ is defined as a unary algebraic data type constructor. In other words, we associate a distinct parameterized algebraic data type with every class name.

Second, superclass and method accessors must perform case analysis over their argument. Indeed, because κ is now an algebraic data type constructor, the only way of exploiting a value of type $\kappa\ \alpha$ is to perform case analysis over its tag. Thus, the superclass and method accessors associated with a class declaration of the form *(class-decl)* must be defined in the following style:

$$\begin{aligned}\mathsf{let\ rec}\ \mathbf{dvar}_i &= \Lambda\alpha.\lambda\mathbf{dvar} : \kappa\ \alpha.\mathsf{case}\ \mathbf{dvar}\ \mathsf{of}\ [\kappa_i\ \alpha]\ \ldots\\ \mathbf{var}_j &= \Lambda\alpha.\lambda\mathbf{dvar} : \kappa\ \alpha.\mathsf{case}\ \mathbf{dvar}\ \mathsf{of}\ [\tau_j]\ \ldots\end{aligned}$$

We momentarily omit the clauses of the case constructs and write ellipses in their place. We cannot yet define these clauses, because we have not yet declared the data constructors associated with the algebraic data type constructor κ. We shall do so shortly.

Third, every dictionary record is concretized, that is, replaced with an application of a fresh data constructor to $\mathbf{dvar}_1, \ldots, \mathbf{dvar}_l$, which, by definition, include the free variables of the original dictionary record $\langle \mathbf{dexp}_1, \ldots, \mathbf{dexp}_m, \mathbf{exp}_1, \ldots, \mathbf{exp}_n \rangle$.

Thus, to every instance declaration of the form *(inst-decl)*, we associate a unique data constructor I, whose type scheme is

$$I :: \forall \alpha_1 \ldots \alpha_k.\langle \kappa'_1\ \tau'_1, \ldots, \kappa'_l\ \tau'_l \rangle \rightarrow \kappa\ \tau.$$

This definition makes κ a *guarded* algebraic data type, as opposed to an ordinary algebraic data type, because τ is not a type variable.

The parameterized dictionary value associated with this instance declaration is then simply I, that is,

$$\begin{aligned}&\Lambda\alpha_1 \ldots \alpha_k.\\ &\quad \lambda\mathbf{dvar}_1 : \kappa'_1\ \tau'_1. \ldots \lambda\mathbf{dvar}_l : \kappa'_l\ \tau'_l.\\ &\qquad I\ \alpha_1\ \ldots\ \alpha_k\ \langle \mathbf{dvar}_1, \ldots, \mathbf{dvar}_l \rangle\end{aligned}$$

Notice how the dictionary record $\langle \mathbf{dexp}_1, \ldots, \mathbf{dexp}_m, \mathbf{exp}_1, \ldots, \mathbf{exp}_n \rangle$, which appeared in the standard encoding, has been replaced with an application of I.

We have associated the data constructor I with the algebraic data type κ. Thus, we must now add a clause for I to each of the superclass and method accessors for the class κ. More precisely, to the definition of the i-th superclass accessor, where $i \in \{1, \ldots, m\}$, we add the clause

$$I\ \alpha_1\ \ldots\ \alpha_k\ \langle \mathbf{dvar}_1, \ldots, \mathbf{dvar}_l \rangle \rightarrow \mathbf{dexp}_i,$$

where, exactly as in the standard encoding, $\mathbf{dexp}_i$ is evidence for $\kappa_i\ \tau$. Furthermore, to the definition of the j-th method accessor, where $j \in \{1, \ldots, n\}$, we add the clause

$$I\ \alpha_1\ \ldots\ \alpha_k\ \langle \mathbf{dvar}_1, \ldots, \mathbf{dvar}_l \rangle \rightarrow \mathbf{exp}_j,$$

where, exactly as in the standard encoding, $\mathbf{exp}_j$ is the translation of exp_j and is an implementation for the method var_j.

The definition of the new encoding is now complete. Indeed, we have explained all of the differences with respect to the standard encoding; the rest of the machinery is shared by both encodings [16]. In the interest of brevity, we do not include a proof that the new encoding is type-preserving; it is, however, straightforward. Proving that *both* encodings are semantics-preserving would also be a good thing. To the best of our knowledge, the standard encoding has never been proven sound. We do not address this issue here.

9.4. Example

The instance declarations of Fig. 13 give rise to the following data constructors:

$$\begin{aligned} \mathit{EqInt} &:: \mathit{Eq\ Int} \\ \mathit{EqList} &:: \forall\alpha.\mathit{Eq}\ \alpha \rightarrow \mathit{Eq}\ [\alpha] \\ \mathit{OrdInt} &:: \mathit{Ord\ Int} \\ \mathit{OrdList} &:: \forall\alpha.\mathit{Ord}\ \alpha \rightarrow \mathit{Ord}\ [\alpha] \end{aligned}$$

where *Eq* and *Ord* are unary guarded algebraic data type constructors.

The definition of the method accessor (==) is

$$\begin{aligned} &(==)\ :: Eq\, a \rightarrow a \rightarrow a \rightarrow Bool \\ &(==)\ \ EqInt = eqInt\ \text{where}\, eqInt = \cdots \\ &(==)\ \ (EqListdEq) = eqList\,((==)\,dEq)\ \text{where}\ eqList = \cdots \end{aligned}$$

(We have used pattern matching instead of an explicit case construct, and have elided the definitions of *eqInt* and *eqList*, as in Fig. 13.) It is straightforward to check that this case analysis is well-typed. In the first clause, the type variable *a* is known to be *Int*, and the right-hand side has type *Int* → *Int* → *Bool*, so this clause has the desired type. In the second clause, the type variable *a* is known to be [*b*], and *dEq* to have type *Eq b*, for some unknown type *b*. Thus, the right-hand side has type [*b*] → [*b*] → *Bool*, and this clause also has the desired type. Again, by exploiting a guarded algebraic data type, we are able to perform case analyses whose branches have seemingly incompatible types.

The definition of the superclass accessor *getEqFromOrd* is

$$\begin{aligned} &\mathit{getEqFromOrd} :: \mathit{Ord}\ \mathrm{a} \rightarrow \mathit{Eq\ a} \\ &\mathit{getEqFromOrd\ OrdInt} = \mathit{EqInt} \\ &\mathit{getEqFromOrd}\ (\mathit{OrdList\ dOrd}) = \mathit{EqList}\ (\mathit{getEqFromOrd\ dOrd}) \end{aligned}$$

Again, it is straightforward to check that it is well-typed.

9.5. Remarks

Proof terms. In this new compilation scheme, dictionaries are represented as *terms* built exclusively out of the data constructors associated with instance declarations. In fact, they may be viewed as *proof terms*. Indeed, an instance declaration may be viewed as a deduction rule. A collection of such declarations forms a deduction system, which allows deriving formulas of the form $\kappa\ \tau$. Dictionaries, in our encoding, are isomorphic to derivations in this deduction system. For instance, a dictionary for *Eq* [[*Int*]] is *EqList* (*EqList EqInt*), reflecting the deductions that must be made to establish that the type [[*Int*]] is a member of the type class *Eq*. In other words, in our encoding, a dictionary is not a record of methods, but only a proof that such methods exist. The proof term is *interpreted*, to produce an actual method, only when required, that is, at method invocation time. These proof terms are somewhat reminiscent of Sheard's *witness* terms [36], which also involve guarded algebraic data types.

An interesting feature of the new encoding is that, since dictionaries are proof terms, they may be analyzed and deconstructed. For instance, it is possible, in the target language, to implement a function of type $\forall\alpha.\mathit{Eq}\ [\alpha] \rightarrow \mathit{Eq}\ \alpha$. Its definition is:

getEqAFromEqListA :: *Eq [a]* $\rightarrow$ *Eq a*
getEqAFromEqListA (*EqList dEq*) = *dEq*

It is worth noting that this case analysis is exhaustive: no branches for data constructors other than *EqList* are necessary. Thus, the function *EqAFromEqListA* does not involve a dynamic check, and cannot fail. This is guaranteed by the fact that Haskell disallows so-called *overlapping instances*. Indeed, Haskell requires that all derivations whose conclusion is of the form *Eq [τ]* begin with an application of the same instance declaration [16]. Thus, all dictionary values whose type is of the form *Eq [τ]* must be applications of *EqList*.

In the standard encoding, *EqAFromEqListA* cannot be defined. Indeed, a dictionary record of type *Eq [α]* consists of an implementation for the method (==) at type *[α]*, represented as a function closure. Even though, by construction, this closure must contain, as part of its value environment, a pointer to a dictionary record of type *Eq* α, extracting it is not permitted by the type discipline. This fact has had implications on the design of Haskell. Indeed, in Haskell, the constraint *Eq* α implies *Eq [α]*, but the converse implication is deemed not to hold, because the standard compilation scheme cannot support it. Our compilation scheme, on the other hand, supports both implications, and would allow *Eq* α and *Eq [α]* to be considered equivalent formulæ by the Haskell typechecker. This would allow the typechecker to find more proofs of an assertion like *Eq* α, that is, more ways of building a dictionary for such an assertion.

Flow graphs. A transformation that appears similar to our encoding is performed, in an untyped setting, by Furuse [13]. Furuse compiles a source language equipped with a form of overloading down to a target language where dictionaries are explicit and are represented by so-called *flow graphs.* These flow graphs appear to correspond to our "proof terms."

One minor difference is that Furuse's flow graphs are possibly cyclic, whereas the proof terms that arise out of our encoding must be finite, because the definition of Haskell requires the types that appear in the premises of an instance declaration to be strict subterms of the type mentioned in its conclusion.

We conjecture that, using guarded algebraic data types, it is possible to define a type-preserving version of Furuse's encoding.

Flat variants of the encodings. The standard encoding of type classes represents dictionaries as *nested* records, where the nesting of dictionaries reflects the class hierarchy. In this scheme, the time required to access a method is proportional to the depth of the hierarchy. One can imagine a variant of the standard scheme, where dictionaries are represented as *flat* polymorphic records, so that superclass accessors become identity functions. In this scheme, dictionary construction may be more costly, but access is presumably faster, especially when the class hierarchy is deep. Our compilation scheme also admits such a variant.

The encoding as a programming idiom. Our encoding of type classes can also be viewed as a programming idiom and explicitly exploited by programmers, if the programming language that they are using lacks type classes, but features guarded algebraic data types. In fact, instances of this idiom have already appeared in the literature. For instance, Cheney and Hinze's [8] version of *equal*, which relies on a guarded algebraic data type constructor *Rep*, is essentially our encoding of the method (==), provided integers, characters, lists, and pairs are declared to be instances of the class *Eq*. Similarly, Xi et al. [44] version of *val2string* corresponds to an encoding of Haskell's standard *show* method. The data constructors *TYint*, *TYtup*, and *TYfun* correspond to instance declarations stating that integers, pairs, and functions are members of the class *Show*. However, Xi et al. *TYtyp*, which also allows proof terms (that is, values of type *TY a*) to be converted to string representations, cannot appear through our encoding. Indeed, dictionaries are invisible for the Haskell programmer: there is no type of dictionaries, so it is impossible to declare that this type is itself an instance of *Show*. Thus, directly programming in terms of guarded algebraic data types, instead of programming in Haskell with type classes, is more clumsy, but offers the equivalent of "first-class dictionaries."

Haskell allows instance declarations for a single class to be split over several program units, whereas most programming languages equipped with algebraic data types only provide a monolithic `case` construct, preventing case analyses from being split over several program modules. This significantly reduces the attractiveness of our encoding as a programming idiom. However, one could design programming languages that allow `case` analyses to be modularly defined. We come back to this issue in Section 10.

10. Conclusion

We have defined a type-preserving version of defunctionalization for polymorphic type systems. The transformation relies on polymorphic recursion and on guarded algebraic data types. We have shown that semantic correctness (that is, meaning preservation) can be established in an untyped setting. Last, we have pointed out that a common principle, which we refer to as concretization, is at the heart of several program transformations, including defunctionalization and encodings of polymorphic records and of type classes. We have proved that guarded algebraic data types allow defining type-preserving versions of these transformations.

This paper is intended both as a study of concretization in a typed setting and as an illustration of the usefulness of guarded algebraic data types. The techniques that we describe can be of help not only to authors of type-preserving compilers, but also to programmers whose programming language offers guarded algebraic data types.

Defunctionalization rearranges code in a non-local manner. Indeed, the function bodies, which could be arbitrarily spread throughout the source program, are gathered inside the definition of *apply* in the translated program. Thus, the body of *apply* potentially contains code from arbitrary source program modules. For this reason, defunctionalization appears to be a whole program transformation. This is in contrast with closure conversion, which is a modular transformation, because its definition is compositional: it transforms expressions into expressions in a purely local manner. The same holds of concretization: the encodings of polymorphic records and of type classes presented in Section 8 and Section 9 appear to be whole program transformations, whereas the standard encodings of dynamic dispatch and of type classes, which employ code pointers instead of tags, are not. Does that diminish the attractiveness of the program transformations presented in this paper?

Our answer is that defunctionalization (or concretization) is not *inherently* a whole program transformation. It is usually *considered* one because it is usually defined for target languages that do not allow the definitions of dispatch functions, like *apply*, to be split over several independent modules. In that case, *apply* must be defined in a monolithic manner, so the transformation cannot be defined in a modular way. Conversely, for defunctionalization (or concretization) to become modular, it is necessary, and sufficient, to adopt a richer target language, where function definitions can be split over several program modules.

This point is little known, but not new: more than two decades ago, Warren [41] noted that defunctionalization *is* indeed a modular transformation when the target language is Prolog, because Prolog allows placing the clauses that define the *apply* predicate inside several independent modules.

If we choose to remain within the realm of functional programming, then the target language must allow the definition of a (recursive) function to be split over several independent modules. Such a facility is currently offered by programming languages with multi-methods [3, 5, 12, 21]. In these languages, concretization *is* modular, and could lead to extremely useful programming idioms. For this reason, we believe the design of a typed programming language that combines incremental function definitions with a form of guarded algebraic data types to be a very interesting direction for future research.

Acknowledgments The authors wish to thank Greg Pettyjohn, Simon Peyton Jones, and the anonymous referees for numerous useful comments and suggestions.

References

1. Banerjee, A., Heintze, N., Riecke, J.G.: Design and correctness of program transformations based on control-flow analysis. In: International Symposium on Theoretical Aspects of Computer Software (TACS), vol. 2215 of Lecture Notes in Computer Science, pp. 420–447 (2001)
2. Bell, J.M., Bellegarde, F., Hook, J.: Type-driven defunctionalization. In: ACM International Conference on Functional Programming (ICFP) (1997)
3. Bonniot, D.: Type-checking multi-methods in ML (a modular approach). In: Workshop on Foundations of Object-Oriented Languages (FOOL) (2002)
4. Boquist, U.: Code optimisation techniques for lazy functional languages. Ph.D. thesis, Chalmers University of Technology (1999)
5. Bourdoncle, F., Merz, S.: Type checking higher-order polymorphic multi-methods. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 302–315 (1997)
6. Cardelli, L.: A semantics of multiple inheritance. Information and Computation **76**(2/3), 138–164 (1988)
7. Cejtin, H., Jagannathan, S., Weeks, S.: Flow-directed closure conversion for typed languages. In: European Symposium on Programming (ESOP), vol. 1782 of Lecture Notes in Computer Science, pp. 56–71 (2000)
8. Cheney, J., Hinze, R.: First-class phantom types. Technical Report 1901, Cornell University (2003)
9. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 207–212 (1982)
10. Danvy, O.: Functional unparsing. Journal of Functional Programming **8**(6), 621–625 (1998)
11. Danvy, O., Nielsen, L.R.: Defunctionalization at work. In: ACM International Conference on Principles and Practice of Declarative Programming (PPDP), pp. 162–174 (2001)
12. Frey, A.: Approche algébrique du typage d'un langage à la ML avec objets, sous-typage et multi-méthodes. Ph.D. thesis, École des Mines de Paris (2004)
13. Furuse, J.: Extensional polymorphism by flow graph dispatching. In: Asian Symposium on Programming Languages and Systems, vol. 2895 of Lecture Notes in Computer Science (2003)
14. Gaster, B.R., Jones, M.P.: A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham (1996)
15. Girard, J.-Y.: Interprétation fonctionnelle et élimination des coupures de larithmétique dordre supérieur. Thése d'état, Université Paris 7 (1972)
16. Hall, C., Hammond, K., Peyton Jones, S., Wadler, P.: Type classes in Haskell. In: Sannella D. (ed.): European Symposium on Programming (ESOP), vol. 788 of Lecture Notes in Computer Science, pp. 241–256 (1994)
17. Hall, C., Hammond, K., Peyton Jones, S., Wadler, P.: Type classes in Haskell. ACM Transactions on Programming Languages and Systems **18**(2), 109–138 (1996)
18. Hanus, M.: Horn clause specifications with polymorphic types. Ph.D. thesis, Fachbereich Informatik, Universität Dortmund (1988)
19. Hanus, M.: Horn clause programs with polymorphic types: Semantics and resolution. In: International Joint Conference on Theory and Practice of Software Development (TAPSOFT), vol. 352 of Lecture Notes in Computer Science, pp. 225–240 (1989)
20. Hinze, R.: Fun with phantom types. In: Gibbons J., de Moor O. (eds.) The Fun of Programming. Palgrave Macmillan, pp. 245–262 (2003)

21. Millstein, T., Chambers, C.: Modular statically typed multimethods. Information and Computation **175**(1), 76–118 (2002)
22. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17**(3), 348–375 (1978)
23. Minamide, Y., Morrisett, G., Harper, R.: Typed closure conversion. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 271–283 (1996)
24. Morrisett, G., Harper, R.: Typed closure conversion for recursively-defined functions (extended abstract). In: International Workshop on Higher Order Operational Techniques in Semantics (HOOTS), vol. 10 of Electronic Notes in Theoretical Computer Science (1998)
25. Morrisett, G., Walker, D., Crary, K., Glew, N.: From system F to typed assembly language. ACM Transactions on Programming Languages and Systems **21**(3), 528–569 (1999)
26. Nielsen, L.R.: A denotational investigation of defunctionalization. Technical Report RS-00-47, BRICS (2000)
27. Ohori, A.: A polymorphic record calculus and its compilation. ACM Transactions on Programming Languages and Systems **17**(6), 844–895 (1995)
28. Paulin-Mohring, C.: Inductive definitions in the system Coq: rules and properties. Research Report RR1992-49, ENS Lyon (1992)
29. Pierce, B.C.: Types and Programming Languages. MIT Press, (2002)
30. Pottier, F., Gauthier, N.: Polymorphic typed defunctionalization. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 89–98 (2004)
31. Pottier, F., Rémy, D.: The essence of ML type inference. In: Pierce B. C. (ed.), Advanced Topics in Types and Programming Languages. MIT Press, Chapt. 10, pp. 389–489 (2005)
32. Rémy, D.: Type inference for records in a natural extension of ML. In: Gunter C. A., Mitchell J. C. (eds), Theoretical Aspects of Object-Oriented Programming. Types, Semantics and Language Design. MIT Press (1994)
33. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: Information Processing vol. 83, pp. 513–523 (1983)
34. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. Higher-Order and Symbolic Computation **11**(4), 363–397 (1998)
35. Reynolds, J.C.: Definitional interpreters revisited. Higher-Order and Symbolic Computation **11**(4), 355–361 (1998)
36. Sheard, T.: Languages of the future. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 116–119 (2004)
37. Simonet, V., Pottier, F.: Constraint-based type inference for guarded algebraic data types. Research Report 5462, INRIA (2005)
38. Tolmach, A.: Combining closure conversion with closure analysis using algebraic types. In: Workshop on Types in Compilation (TIC) (1997)
39. Tolmach, A., Oliva, D.P.: From ML to Ada: Strongly-typed language interoperability via source translation. Journal of Functional Programming **8**(4), 367–412 (1998)
40. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 60–76 (1989)
41. Warren, D.H.D.: Higher-order extensions to PROLOG: are they needed?. In: Hayes J. E., Michie D., Pao Y.-H. (eds.) Machine Intelligence 10. Ellis Horwood, pp. 441–454 (1982)
42. Xi, H.: Dead code elimination through dependent types. In: International Workshop on Practical Aspects of Declarative Languages (PADL), vol. 1551 of Lecture Notes in Computer Science, pp. 228–242 (1999)
43. Xi, H.: Applied type system. In: TYPES 2003, vol. 3085 of Lecture Notes in Computer Science, pp. 394–408 (2004)
44. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 224–235 (2003)
45. Zendra, O., Colnet, D., Collin, S.: Efficient dynamic dispatch without virtual function tables. the SmallEiffel compiler. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 125–141 (1997)
46. Zibin, Y., Gil, Y.: Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 142–160 (2002)