Polymorphic Typed Defunctionalization

François Pottier*
Francois.Pottier@inria.fr

Nadji Gauthier*
Nadji.Gauthier@inria.fr

Abstract

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. Its purpose is thus identical to that of *closure conversion*. It differs from closure conversion, however, by storing a *tag*, instead of a code pointer, within every closure. Defunctionalization has been used both as a reasoning tool and as a compilation technique.

Defunctionalization is commonly defined and studied in the setting of a simply-typed λ-calculus, where it is shown that semantics and well-typedness are preserved. It has been observed that, in the setting of a polymorphic type system, such as ML or System F, defunctionalization is not type-preserving. In this paper, we show that extending System F with *guarded algebraic data types* allows recovering type preservation. This result allows adding defunctionalization to the toolbox of type-preserving compiler writers.

1 Introduction

Defunctionalization, due to Reynolds [13, 14], 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. Let us begin with a rough, machine-oriented description of it. Under the assumption that the entire source program is available, a distinct tag is associated with every λ-abstraction, or, in other words, with every code block. 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, performs case analysis on the tag and jumps to the associated code block, making the contents of the value environment available to it. The reader may notice that defunctionalization is a close cousin of *closure conversion*. In fact, to a certain extent, closure conversion may be viewed as a particular implementation of defunctionalization, whereby 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.

*INRIA, BP 105, F-78153 Le Chesnay Cedex, France.

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, Morrisett, and Harper [8] 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 and of a record type, packed within an *existential* type, so that closures whose value environments have different structure may still receive identical types. Minamide *et al.* deal with both simply-typed and type-passing, polymorphic λ-calculi. The case of a type-erasure polymorphic λ-calculus has been addressed in [10]. Defunctionalization, on the other hand, has been studied mainly in a *simply-typed* setting [11, 1]. There, closures receive *sum* types: closure construction becomes injection, while the case analysis involved by application becomes elimination. (If one wishes to avoid recursive types, one must in fact employ *algebraic data types*, rather than anonymous sum types.) When the source language is ML, the source program is typically turned into a simply-typed program by applying *monomorphization* prior to defunctionalization [15, 16, 3]. However, monomorphization involves code duplication, whose cost may be difficult to control. Bell, Bellegarde, and Hook [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. When the source language is ML with polymorphic recursion or System F, monomorphization becomes impossible, because an infinite amount of code duplication might be required. In that case, no type-preserving definition of defunctionalization was known to date.

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 $\mathit{apply}\, [\![e_1]\!]\, [\![e_2]\!]$. 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 $\mathit{apply}\, [\![e_1]\!]\, [\![e_2]\!]$ 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 $int \to int$ and $bool \to bool$, 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, the definition of *apply* should contain the following clauses:

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

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.

The standard, limited workaround The workaround commonly adopted in the simply-typed case [2, 15, 16, 3, 11, 1] consists in specializing *apply*. Instead of defining a single, polymorphic function, one introduces a family of monomorphic functions, indexed by ground types τ_1 and τ_2, each member of which has type $[\![\tau_1 \to \tau_2]\!] \to [\![\tau_1]\!] \to [\![\tau_2]\!]$. The point is that the definition of $apply_{\tau_1 \to \tau_2}$ lists 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 $[\![\cdot]\!]$ *no longer commutes* with substitution of types for type variables. Indeed, every distinct arrow type in the source program must now map 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 the approach fails in the presence of polymorphism.

Our solution In the present paper, we suggest another way out of this problem. We keep a single *apply* function, whose type is $\forall\alpha_1\alpha_2.[\![\alpha_1 \to \alpha_2]\!] \to \alpha_1 \to \alpha_2$, as initially suggested above. We also insist that the translation of types should commute with type substitutions, so $[\![\tau_1 \to \tau_2]\!]$ 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 $Arrow\, \alpha_1\, \alpha_2$, that is, it encodes a source function of type $\alpha_1 \to \alpha_2$. However, we know that the λ-abstraction associated with the tag *succ*, namely $\lambda x.x + 1$, has type $int \to int$, 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 should be possible to prove that $arg + 1$ has type α_2 under the assumption that *arg* has type α_1. Then, by dealing with every clause in an analogous manner, it should be possible to 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 types $Arrow\, int\, int$ and $Arrow\, bool\, bool$, respectively. Please note that, if *Arrow* was a standard (ML-style) 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 [12], and is known in a programming-language setting as *guarded recursive data types* [18] or as *first-class phantom types* [4]. We refer to it as *guarded algebraic data types*. The term *guarded* stems from Xi, Chen, and Chen's observation that they may be encoded in terms of recursive types, sum types, and *constrained* existential types.

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 ML, 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, type information in the source program is used to construct a type derivation for the target program, but does not influence the latter's structure. 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 first prove that the transformation is meaning-preserving in an untyped setting, then lift this result to the typed setting. These proofs form the paper's second contribution. They appear to be new:

indeed, previous proofs [11, 1] were carried out in a simply-typed setting.

Road map The paper is laid out as follows. Section 2 defines an extension of System F with guarded algebraic data types. Section 3 defines defunctionalization of well-typed programs. In Section 4, we prove that defunctionalization preserves well-typedness. In Section 5, we define defunctionalization of untyped programs, prove that it preserves their meaning, and prove that this result carries over to defunctionalization of typed programs. Section 6 contains some closing remarks.

2 The type system

In this section, we 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, Chen, and Chen's [18], with a couple of 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.

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* and r range over an arbitrary set of *record labels*. The syntax of *types* is as follows:

$$\begin{array}{rcl} \tau & ::= & \alpha \\ & \mid & \tau \to \tau \\ & \mid & \forall \alpha.\tau \\ & \mid & \{\bar{r} : \bar{\tau}\} \\ & \mid & T\,\bar{\tau} \end{array}$$

Types include type variables, arrow types, universal types, record types, and algebraic data types. In the universal type $\forall \alpha.\tau$, the type variable α is bound within τ. In the record type $\{\bar{r} : \bar{\tau}\}$, $\bar{r}$ must be a vector of distinct record labels, while $\bar{\tau}$ is a vector of types of the same length. We write $\bar{r} : \bar{\tau}$ for the vector of bindings obtained by associating elements of $\bar{r}$, 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 $\bar{x} : \bar{\tau}$, $\bar{e} : \bar{\tau}$, $\bar{c} : \bar{\tau}$, and for conjunctions of equations $\bar{\tau}_1 = \bar{\tau}_2$.) 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 from 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; see *e.g.* [18].

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 $\forall \bar{\alpha}[D].\bar{\tau} \to T\,\bar{\tau}_1$. In such a type scheme, the type variables $\bar{\alpha}$ are bound within D, $\bar{\tau}$, and $\bar{\tau}_1$. The length of the vector $\bar{\tau}$ is the arity of the data constructor K. 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:

$$\begin{array}{rcl} e & ::= & x \\ & \mid & \lambda x : \tau.e \\ & \mid & e\,e \\ & \mid & \Lambda \alpha.e \\ & \mid & e\,\tau \\ & \mid & \mathsf{let}\, x = e\, \mathsf{in}\, e \\ & \mid & \mathsf{letrec}\, \bar{x} : \bar{\tau} = \bar{e}\, \mathsf{in}\, e \\ & \mid & \{\bar{r} = \bar{e}\} \\ & \mid & e.r \\ & \mid & K\,\bar{\tau}\,\bar{e} \\ & \mid & \mathsf{case}\, e\, \mathsf{of}\, [\tau]\bar{c} \\ c & ::= & K\,\bar{\alpha}\,\bar{x} \mapsto e \end{array}$$

The language is an extension of the polymorphic λ-calculus with recursive definitions, constructs for creating and accessing records, and 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.) The injection construct $K\,\bar{\tau}\,\bar{e}$ requires the data constructor K to be fully applied. Thus, partial applications of K are not valid expressions: they must be encoded via η-expansion. This choice simplifies the definition of defunctionalization, because it means that only λ-abstractions have arrow types. 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 $K\,\bar{\alpha}\,\bar{x} \mapsto e$, the type variables $\bar{\alpha}$ and the term variables $\bar{x}$ are bound within e.

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 Figure 1. In TAbs, the notation $\alpha \# C, \Gamma$ requires the type variable α not to appear free within C or Γ. All rules but Data, Clause, Conv, and Weaken are standard (System F) rules. 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 in fact relative to an instance of the type scheme where the type arguments $\bar{\tau}_2$ are substituted for the quantifiers $\bar{\alpha}$, as well as to the current hypothesis C. Data may be viewed as a combination of the standard rules for (constrained) type application and value application. Clause's first premise looks up the type scheme associated with K and 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,

$$\textsc{Var}\ \ C, \Gamma \vdash x : \Gamma(x) \qquad \textsc{Abs}\ \ \frac{C, \Gamma; x : \tau_1 \vdash e : \tau_2}{C, \Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2} \qquad \textsc{App}\ \ \frac{C, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad C, \Gamma \vdash e_2 : \tau_1}{C, \Gamma \vdash e_1\, e_2 : \tau_2}$$

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

$$\textsc{Let}\ \ \frac{C, \Gamma \vdash e_1 : \tau_1 \qquad C, \Gamma; x : \tau_1 \vdash e_2 : \tau_2}{C, \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau_2} \qquad \textsc{LetRec}\ \ \frac{C, \Gamma; \bar{x} : \bar{\tau} \vdash \bar{e} : \bar{\tau} \qquad C, \Gamma; \bar{x} : \bar{\tau} \vdash e : \tau}{C, \Gamma \vdash \mathsf{letrec}\, \bar{x} : \bar{\tau} = \bar{e}\, \mathsf{in}\, e : \tau}$$

$$\textsc{Record}\ \ \frac{C, \Gamma \vdash \bar{e} : \bar{\tau}}{C, \Gamma \vdash \{\bar{r} = \bar{e}\} : \{\bar{r} : \bar{\tau}\}}$$

$$\textsc{Proj}\ \ \frac{C, \Gamma \vdash e : \{r : \tau; \bar{r} : \bar{\tau}\}}{C, \Gamma \vdash e.r : \tau} \qquad \textsc{Data}\ \ \frac{K :: \forall \bar{\alpha}[D] . \bar{\tau} \to T\, \bar{\tau}_1 \quad C \Vdash [\bar{\alpha} \mapsto \bar{\tau}_2]D \quad C, \Gamma \vdash \bar{e} : [\bar{\alpha} \mapsto \bar{\tau}_2]\bar{\tau}}{C, \Gamma \vdash K\, \bar{\tau}_2\, \bar{e} : T\, [\bar{\alpha} \mapsto \bar{\tau}_2]\bar{\tau}_1}$$

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

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

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

Figure 1: The type system

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 $\bar{\alpha}$, in effect making them semi-abstract or concrete. Conv allows replacing the 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 presence of Weaken is perhaps surprising, since this rule is admissible. It is intended as a hint to the defunctionalization algorithm not to include the value of x within closures allocated inside e; see Section 3.

The type system is sound [18]. 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 $C' \Vdash C$ and $C' \Vdash \Gamma' = \Gamma$ 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.

In the derivation of $\mathsf{true}, \varnothing \vdash p : \tau_p$, we require every instance of Abs whose conclusion is of the form $C, \Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2$ to satisfy $\mathrm{dom}(\Gamma) = \mathrm{fv}(\lambda x . e)$. Thanks to the presence of Weaken, this assumption does not cause any loss of generality. This restriction ensures that defunctionalization is independent of the manner in which Weaken is employed in the type derivation. This in turn ensures that our notion of defunctionalization is not type-directed, a fact which we establish and exploit later on (Lemma 5.6). The restriction is otherwise inessential.

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.

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

The type translation function extends 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 defined 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 from term variables to types; we assume that term variables form a subset of record labels, which allows us to form the record type $\{[\![\Gamma]\!]\}$.

$$\textsc{Var}\quad C, \Gamma \vdash x : \Gamma(x) \rightsquigarrow x$$

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

$$\textsc{App}\quad \frac{C, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \rightsquigarrow e_1' \qquad C, \Gamma \vdash e_2 : \tau_1 \rightsquigarrow e_2'}{C, \Gamma \vdash e_1\, e_2 : \tau_2 \rightsquigarrow \mathit{apply}\, [\![\tau_1]\!]\, [\![\tau_2]\!]\, e_1'\, e_2'}$$

$$\textsc{TAbs}\quad \frac{C, \Gamma \vdash e : \tau \rightsquigarrow e' \qquad \alpha \# C, \Gamma}{C, \Gamma \vdash \Lambda\alpha . e : \forall \alpha . \tau \rightsquigarrow \Lambda\alpha . e'}$$

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

$$\textsc{Let}\quad \frac{C, \Gamma \vdash e_1 : \tau_1 \rightsquigarrow e_1' \qquad C, \Gamma; x : \tau_1 \vdash e_2 : \tau_2 \rightsquigarrow e_2'}{C, \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau_2 \rightsquigarrow \mathsf{let}\, x = e_1'\, \mathsf{in}\, e_2'}$$

$$\textsc{LetRec}\quad \frac{C, \Gamma; \bar{x} : \bar{\tau} \vdash \bar{e} : \bar{\tau} \rightsquigarrow \bar{e}' \qquad C, \Gamma; \bar{x} : \bar{\tau} \vdash e : \tau \rightsquigarrow e'}{C, \Gamma \vdash \mathsf{letrec}\, \bar{x} : \bar{\tau} = \bar{e}\, \mathsf{in}\, e : \tau \rightsquigarrow \mathsf{letrec}\, \bar{x} : [\![\bar{\tau}]\!] = \bar{e}'\, \mathsf{in}\, e'}$$

$$\textsc{Record}\quad \frac{C, \Gamma \vdash \bar{e} : \bar{\tau} \rightsquigarrow \bar{e}'}{C, \Gamma \vdash \{\bar{r} = \bar{e}\} : \{\bar{r} : \bar{\tau}\} \rightsquigarrow \{\bar{r} = \bar{e}'\}}$$

$$\textsc{Proj}\quad \frac{C, \Gamma \vdash e : \{r : \tau; \bar{r} : \bar{\tau}\} \rightsquigarrow e'}{C, \Gamma \vdash e.r : \tau \rightsquigarrow e'.r}$$

$$\textsc{Data}\quad \frac{K :: \forall \bar{\alpha}[D] . \bar{\tau} \to T\, \bar{\tau}_1 \qquad C \Vdash [\bar{\alpha} \mapsto \bar{\tau}_2]D \qquad C, \Gamma \vdash \bar{e} : [\bar{\alpha} \mapsto \bar{\tau}_2]\bar{\tau} \rightsquigarrow \bar{e}'}{C, \Gamma \vdash K\, \bar{\tau}_2\, \bar{e} : T\, [\bar{\alpha} \mapsto \bar{\tau}_2]\bar{\tau}_1 \rightsquigarrow K\, [\![\bar{\tau}_2]\!]\, \bar{e}'}$$

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

$$\textsc{Clause}\quad \frac{K :: \forall \bar{\alpha}[D] . \bar{\tau} \to T\, \bar{\tau}_1 \qquad \bar{\alpha} \# C, \Gamma, \bar{\tau}_2, \tau \qquad C \wedge D \wedge \bar{\tau}_1 = \bar{\tau}_2, \Gamma; \bar{x} : \bar{\tau} \vdash e : \tau \rightsquigarrow e'}{C, \Gamma \vdash K\, \bar{\alpha}\, \bar{x} \mapsto e : T\, \bar{\tau}_2 \to \tau \rightsquigarrow K\, \bar{\alpha}\, \bar{x} \mapsto e'}$$

$$\textsc{Conv}\quad \frac{C, \Gamma \vdash e : \tau_1 \rightsquigarrow e' \qquad C \Vdash \tau_1 = \tau_2}{C, \Gamma \vdash e : \tau_2 \rightsquigarrow e'}$$

$$\textsc{Weaken}\quad \frac{C, \Gamma_1; \Gamma_2 \vdash e : \tau \rightsquigarrow e' \qquad x \# e}{C, \Gamma_1; x : \tau_1; \Gamma_2 \vdash e : \tau \rightsquigarrow e'}$$

Figure 2: Term translation

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 Figure 2. It is immediate to check 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 $\mathsf{true}, \varnothing \vdash p : \tau_p$ 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 argument is 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 the record term $\{y = y\}_{y \in \mathrm{dom}(\Gamma)}$, where the left-hand y is interpreted as a record label, while the right-hand y is a term variable. This record is the closure's value environment. One might think that it is inefficient to save all of the term variables in Γ into the closure, rather than only those that appear free in $\lambda x : \tau_1 . e$. However, if Weaken is used eagerly in the original type derivation, these must in fact coincide, so no efficiency is lost. This trick simplifies our notation. 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

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

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

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

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

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

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 re-introduces 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.) We write $\{\Gamma\}$ as a short-hand for the record pattern $\{y = y\}_{y \in \mathrm{dom}(\Gamma)}$. The use of pattern matching is not, strictly speaking, part of our language: we write $K\, \bar{\alpha}\, \{\bar{r} = \bar{x}\} \mapsto e$ as syntactic sugar for

$K\,\bar{\alpha}\,env \mapsto \mathsf{let}\,\bar{x} = env.\bar{r}\,\mathsf{in}\,e$, where we use vector notation to succinctly represent multiple let definitions.

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 must be double applications of *apply*, a letrec-bound, binary function.

Example For the sake of illustration, we give a short example program together with its defunctionalized version. The program, inspired from [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 $\forall\alpha.\alpha \to \alpha \to bool$ and of the Boolean "or" combinator $||$ of type $bool \to bool \to bool$. Applications of these two primitive operations are not affected by the translation.

$$\begin{array}{l}\mathsf{let}\ empty = \Lambda\alpha.\lambda^{m_1}x : \alpha.\mathsf{false}\ \mathsf{in}\\ \mathsf{let}\ insert = \Lambda\alpha.\lambda^{m_2}x : \alpha.\lambda^{m_3}s : \alpha \to bool.\\ \qquad\qquad \lambda^{m_4}y : \alpha.(=\,\alpha\,x\,y)\ ||\ (s\,y)\ \mathsf{in}\\ insert\ int\,1\,(empty\ int)\,2\end{array}$$

The empty set *empty* has type $\forall\alpha.\alpha \to bool$. The insertion function *insert* has type $\forall\alpha.\alpha \to (\alpha \to bool) \to (\alpha \to bool)$. The complete program has type *bool*. Its defunctionalized counterpart is defined under the following data signature, where $set\,\alpha$ stands for $Arrow\,\alpha\,bool$.

$$\begin{array}{lcl} m_1 & :: & \forall\alpha[\mathsf{true}].\{\} \to set\,\alpha\\ m_2 & :: & \forall\alpha[\mathsf{true}].\{\} \to Arrow\,\alpha\,(Arrow\,(set\,\alpha)\,(set\,\alpha))\\ m_3 & :: & \forall\alpha[\mathsf{true}].\{x : \alpha\} \to Arrow\,(set\,\alpha)\,(set\,\alpha)\\ m_4 & :: & \forall\alpha[\mathsf{true}].\{x : \alpha; s : set\,\alpha\} \to set\,\alpha\end{array}$$

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 constructors *nil* and *cons*, which makes apparent the fact that sets built using *empty* and *insert* become lists after defunctionalization. The defunctionalized program is given below:

$$\begin{array}{l}\mathsf{letrec}\ apply : \tau_{apply} =\\ \quad \Lambda\alpha_1.\Lambda\alpha_2.\\ \quad\quad \lambda f : Arrow\,\alpha_1\,\alpha_2.\\ \quad\quad\quad \lambda arg : \alpha_1.\\ \quad\quad\quad\quad \mathsf{case}\ f\ \mathsf{of}\\ \quad\quad\quad\quad\ |\ \ m_1\,\alpha\,\{\} \mapsto \mathsf{let}\ x = arg\ \mathsf{in}\ \mathsf{false}\\ \quad\quad\quad\quad\ |\ \ m_2\,\alpha\,\{\} \mapsto \mathsf{let}\ x = arg\ \mathsf{in}\ m_3\,\alpha\,\{x\}\\ \quad\quad\quad\quad\ |\ \ m_3\,\alpha\,\{x\} \mapsto \mathsf{let}\ s = arg\ \mathsf{in}\ m_4\,\alpha\,\{x; s\}\\ \quad\quad\quad\quad\ |\ \ m_4\,\alpha\,\{x; s\} \mapsto \mathsf{let}\ y = arg\ \mathsf{in}\\ \quad\quad\quad\quad\qquad\qquad (=\,\alpha\,x\,y)\ ||\ (apply\,\alpha\,bool\,s\,y)\\ \quad\quad\quad\quad [\alpha_2]\\ \mathsf{in}\\ \mathsf{let}\ empty = \Lambda\alpha.m_1\,\alpha\,\{\}\ \mathsf{in}\\ \mathsf{let}\ insert = \Lambda\alpha.m_2\,\alpha\,\{\}\ \mathsf{in}\\ apply\,(apply\,(apply\,(insert\ int)\,1)\,(empty\ int))\,2\end{array}$$

As before, we use punning, that is, we write $\{x\}$ for the pattern or expression $\{x = x\}$ and $\{x; s\}$ for $\{x = x; s = s\}$. For the sake of brevity, we have omitted 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*. However, 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, 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 *$C, \Gamma \vdash e : \tau \rightsquigarrow e'$ implies $[\![C]\!], apply; [\![\Gamma]\!] \vdash e' : [\![\tau]\!]$.*

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

∘ *Case* Abs. The rule's conclusion is $C, \Gamma \vdash \lambda^m x : \tau_1.e : \tau_1 \to \tau_2 \rightsquigarrow m\,\bar{\alpha}\,\{\Gamma\}$ (**1**). Its premises are $C, \Gamma; x : \tau_1 \vdash e : \tau_2 \rightsquigarrow e'$ (**2**) and $\bar{\alpha} = \mathrm{ftv}(C, \Gamma, \tau_1, \tau_2)$ (**3**). By (1), (3), and by definition of the data signature $\mathcal{D}'$, we have $m :: \forall\bar{\alpha}[[\![C]\!]].\{[\![\Gamma]\!]\} \to Arrow\,[\![\tau_1]\!]\,[\![\tau_2]\!]$ (**4**). Furthermore, by reflexivity of entailment, we have $[\![C]\!] \Vdash [\![C]\!]$ (**5**). Last, by Var and Record, we have $[\![C]\!], apply; [\![\Gamma]\!] \vdash \{\Gamma\} : \{[\![\Gamma]\!]\}$ (**6**). Applying Data to (4), (5), and (6), we find $[\![C]\!], apply; [\![\Gamma]\!] \vdash m\,\bar{\alpha}\,\{\Gamma\} : Arrow\,[\![\tau_1]\!]\,[\![\tau_2]\!]$, that is, $[\![C]\!], apply; [\![\Gamma]\!] \vdash m\,\bar{\alpha}\,\{\Gamma\} : [\![\tau_1 \to \tau_2]\!]$.

∘ *Case* App. The rule's conclusion is $C, \Gamma \vdash e_1\,e_2 : \tau_2 \rightsquigarrow apply\,[\![\tau_1]\!]\,[\![\tau_2]\!]\,e_1'\,e_2'$. Its premises are $C, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \rightsquigarrow e_1'$ (**1**) and $C, \Gamma \vdash e_2 : \tau_1 \rightsquigarrow e_2'$ (**2**). Var yields $[\![C]\!], apply; [\![\Gamma]\!] \vdash apply : \tau_{apply}$. By definition of τ_{apply} and by TApp, this implies $[\![C]\!], apply; [\![\Gamma]\!] \vdash apply\,[\![\tau_1]\!]\,[\![\tau_2]\!] : Arrow\,[\![\tau_1]\!]\,[\![\tau_2]\!] \to [\![\tau_1]\!] \to [\![\tau_2]\!]$, that is, $[\![C]\!], apply; [\![\Gamma]\!] \vdash apply\,[\![\tau_1]\!]\,[\![\tau_2]\!] : [\![\tau_1 \to \tau_2]\!] \to [\![\tau_1]\!] \to [\![\tau_2]\!]$ (**3**). Furthermore, applying the induction hypothesis to (1) and (2) yields $[\![C]\!], apply; [\![\Gamma]\!] \vdash e_1' : [\![\tau_1 \to \tau_2]\!]$ (**4**) and $[\![C]\!], apply; [\![\Gamma]\!] \vdash e_2' : [\![\tau_1]\!]$ (**5**). By App, (3), (4), and (5) imply $[\![C]\!], apply; [\![\Gamma]\!] \vdash apply\,[\![\tau_1]\!]\,[\![\tau_2]\!]\,e_1'\,e_2' : [\![\tau_2]\!]$. □

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 $apply : \tau_{apply}$.

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

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 enriched typing subderivation ends with

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

By applying Lemma 4.1 to the first premise, we obtain $[\![C]\!], apply; [\![\Gamma]\!]; x : [\![\tau_1]\!] \vdash e' : [\![\tau_2]\!]$. Then, Lemma 2.1, Conv, and Weaken yield $[\![C]\!] \wedge [\![\tau_1]\!] = \alpha_1 \wedge [\![\tau_2]\!] = \alpha_2, apply; arg; [\![\Gamma]\!]; x : \alpha_1 \vdash e' : \alpha_2$. By Let, this implies $[\![C]\!] \wedge [\![\tau_1]\!] = \alpha_1 \wedge [\![\tau_2]\!] = \alpha_2, apply; arg; [\![\Gamma]\!] \vdash \mathsf{let}\, x = arg\, \mathsf{in}\, e' : \alpha_2$ (**1**). Now, by definition of the data signature $\mathcal{D}'$, we have $m :: \forall\bar{\alpha}[[\![C]\!]].\{[\![\Gamma]\!]\} \to Arrow\, [\![\tau_1]\!]\, [\![\tau_2]\!]$ (**2**). Last, by construction, we have $\bar{\alpha} \# \alpha_1\alpha_2$ (**3**). Applying Clause to (1), (2), and (3) yields $\mathsf{true}, apply; arg \vdash m\, \bar{\alpha}\, \{\Gamma\} \mapsto \mathsf{let}\, x = arg\, \mathsf{in}\, e' : Arrow\, \alpha_1\, \alpha_2 \to \alpha_2$. Now, because this holds for every λ-abstraction that appears within p, and by definition of $\bar{c}_p$, we have established $\mathsf{true}, apply; arg \vdash \bar{c}_p : Arrow\, \alpha_1\, \alpha_2 \to \alpha_2$. 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 4.1 $\mathsf{true}, \varnothing \vdash [\![p]\!] : [\![\tau_p]\!]$.

Proof. Applying Lemma 4.1 to the judgement $\mathsf{true}, \varnothing \vdash p : \tau_p \rightsquigarrow p'$ yields $\mathsf{true}, apply \vdash p' : [\![\tau_p]\!]$ (**1**). The result follows from Lemma 4.2 and from (1) by LetRec. □

5 Meaning preservation

We now prove that defunctionalization preserves the meaning of programs, as defined by a call-by-value operational semantics. 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. We emphasize that working in an untyped setting makes such a statement particularly simple and general. 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 [11, 1].

In the following, we slightly amend the definition of the programming language in three ways. First, we introduce two new syntactic classes, namely *values* v and *values or variables* w. The former class contains functions, records, and algebraic data structures, while the latter includes variables in addition. We restrict the right-hand sides of letrec definitions to be values: this restriction is standard in a call-by-value setting. Second, we place λ-, let-, or case-bound term variables, written x, and letrec-bound term variables, written X, in distinct syntactic classes. This choice, while not essential, simplifies some α-conversion arguments. We refer to an expression that does not have any free variables of the former (resp. latter) class as x-closed (resp. X-closed). Last, we restrict the syntax of programs by requiring, in many places, values or variables w instead of arbitrary expressions e. This design, which is reminiscent of Flanagan *et al.*'s *A*-normal forms [7], does not incur any loss of expressiveness, but simplifies our proofs by making $\mathsf{let}\, x = []\, \mathsf{in}\, e$ the only evaluation context. Because the amendments described in this paragraph have nothing to do with the type preservation result presented in the previous sections, we have chosen not to introduce them earlier on.

5.1 Untyped defunctionalization

We first define the untyped language. It is the type free counterpart of the typed language presented in Section 2, with the amendments described above.

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

Next, we define a call-by-value, small-step operational semantics. The objects of reduction are not expressions, but *configurations* of the form $S \,/\, e$, where a *store* S is a set of bindings of (distinct) letrec-class term variables to x-closed values. As usual, the term variables in the domain of the store S are considered bound in the configuration $S \,/\, e$. In fact, the notation $S \,/\, e$ may be viewed as a shorthand for $\mathsf{letrec}\, S\, \mathsf{in}\, e$. The use of a store allows describing the dynamics of letrec definitions in a straightforward manner. The operational semantics, a rewriting system on closed configurations, is defined by the following standard rules.

$$\begin{array}{rcl} S \,/\, (\lambda x.e)\, v & \to & S \,/\, [x \mapsto v]e \\ S \,/\, \mathsf{let}\, x = v\, \mathsf{in}\, e & \to & S \,/\, [x \mapsto v]e \\ S \,/\, \{r = v; \bar{r} = \bar{v}\}.r & \to & S \,/\, v \\ S \,/\, \mathsf{case}\, K\, \bar{v}\, \mathsf{of}\, (K\, \bar{x} \mapsto e) \mid \bar{c} & \to & S \,/\, [\bar{x} \mapsto \bar{v}]e \\ S \,/\, \mathsf{letrec}\, \bar{X} = \bar{v}\, \mathsf{in}\, e & \to & S; \bar{X} = \bar{v} \,/\, e \text{ if } \bar{X} \# S \\ S; X = v \,/\, X & \to & S; X = v \,/\, v \end{array}$$

$$\frac{S_1 \,/\, e_1 \to S_2 \,/\, e_2}{S_1 \,/\, \mathsf{let}\, x = e_1\, \mathsf{in}\, e \to S_2 \,/\, \mathsf{let}\, x = e_2\, \mathsf{in}\, e}$$

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 do not, however, require p to be well-typed. The translation of expressions is defined, in an inductive manner, in Figure 3. All cases are trivial, except Abs, which translates λ-abstraction to closure construction, and App, which translates function application to invocations of *apply*. Please note that the relation $\rightsquigarrow$ 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 defined as

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

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

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

where e' is defined by $e \rightsquigarrow e'$.

$$\textsc{Var}\quad x \rightsquigarrow x \qquad \textsc{Abs}\quad \lambda^m x.e \rightsquigarrow m\,\{\mathrm{fv}(\lambda x.e)\}$$

$$\textsc{App}\quad \frac{e_1 \rightsquigarrow e_1' \qquad e_2 \rightsquigarrow e_2'}{e_1\,e_2 \rightsquigarrow \mathit{apply}\,e_1'\,e_2'} \qquad \textsc{Let}\quad \frac{e_1 \rightsquigarrow e_1' \qquad e_2 \rightsquigarrow e_2'}{\mathsf{let}\,x = e_1\,\mathsf{in}\,e_2 \rightsquigarrow \mathsf{let}\,x = e_1'\,\mathsf{in}\,e_2'}$$

$$\textsc{LetRec}\quad \frac{\bar{v} \rightsquigarrow \bar{v}' \qquad e \rightsquigarrow e'}{\mathsf{letrec}\,\bar{X} = \bar{v}\,\mathsf{in}\,e \rightsquigarrow \mathsf{letrec}\,\bar{X} = \bar{v}'\,\mathsf{in}\,e'} \qquad \textsc{Record}\quad \frac{\bar{w} \rightsquigarrow \bar{w}'}{\{\bar{r} = \bar{w}\} \rightsquigarrow \{\bar{r} = \bar{w}'\}}$$

$$\textsc{Proj}\quad \frac{w \rightsquigarrow w'}{w.r \rightsquigarrow w'.r} \qquad \textsc{Data}\quad \frac{\bar{w} \rightsquigarrow \bar{w}'}{K\,\bar{w} \rightsquigarrow K\,\bar{w}'}$$

$$\textsc{Case}\quad \frac{w \rightsquigarrow w' \qquad \bar{c} \rightsquigarrow \bar{c}'}{\mathsf{case}\,w\,\mathsf{of}\,\bar{c} \rightsquigarrow \mathsf{case}\,w'\,\mathsf{of}\,\bar{c}'} \qquad \textsc{Clause}\quad \frac{e \rightsquigarrow e'}{K\,\bar{x} \mapsto e \rightsquigarrow K\,\bar{x} \mapsto e'}$$

Figure 3: Untyped term translation

5.2 Untyped meaning preservation

In order to establish that untyped defunctionalization preserves meaning, we exhibit a simulation between closed source configurations and their defunctionalized versions. Roughly speaking, the desired simulation is the closure of the translation $\rightsquigarrow$ under substitution of (x-closed) values for (x-class) program variables. More precisely, let us define the relation $\gtrsim$ by the same rules that define $\rightsquigarrow$ (that is, by the rules of Figure 3, where every occurrence is of $\rightsquigarrow$ is replaced with $\gtrsim$), extended with the following new rule, which subsumes Abs:

$$\textsc{SimAbs}\quad \frac{\lambda^m x.e \rightsquigarrow m\,\{\bar{y}\} \qquad \bar{x} \subseteq \bar{y} \qquad \bar{v}, \bar{v}'\,x\text{-closed} \qquad \bar{v} \gtrsim \bar{v}'}{\lambda x.[\bar{x} \mapsto \bar{v}]e \gtrsim m\,\{[\bar{x} \mapsto \bar{v}']\bar{y}\}}$$

The notation $\{[\bar{x} \mapsto \bar{v}']\bar{y}\}$ stands for the record expression $\{\bar{y} = [\bar{x} \mapsto \bar{v}']\bar{y}\}$. In short, SimAbs extends Abs by translating not only functions *that appear in the source program*, but also the functions *that appear at runtime*, where some of the free variables have been instantiated with runtime values, that is, x-closed values. It is immediate to check that $\gtrsim$ extends $\rightsquigarrow$, as stated by the following lemma.

Lemma 5.1 *$e \rightsquigarrow e'$ implies $e \gtrsim e'$.*

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

Lemma 5.2 *Let $e \gtrsim e'$ and $v \gtrsim v'$, where v and v' are x-closed. Then, $[x \mapsto v]e \gtrsim [x \mapsto v']e'$ holds.*

Last, $\gtrsim$ preserves values.

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

We extend $\gtrsim$ to stores and to (closed) configurations as follows:

$$\textsc{Store}\quad \frac{\mathit{apply} \notin \bar{X} \qquad \bar{v} \gtrsim \bar{v}'}{\bar{X} = \bar{v} \gtrsim \mathit{apply}; \bar{X} = \bar{v}'} \qquad \textsc{Config}\quad \frac{S \gtrsim S' \qquad e \gtrsim e'}{S \,/\, e \gtrsim S' \,/\, e'}$$

Above, and in the following, we use *apply* to stand for the store binding $\mathit{apply} = \lambda f.\lambda \mathit{arg}.\mathsf{case}\,f\,\mathsf{of}\,\bar{c}_p$. We are now ready to prove that $\gtrsim$ is a simulation.

Lemma 5.4 (Simulation) *This diagram commutes:*

$$\begin{array}{ccc} S_1 \,/\, e_1 & \xrightarrow{\gtrsim} & S_1' \,/\, e_1' \\ \downarrow & & \vdots \\ S_2 \,/\, e_2 & \xrightarrow{\gtrsim} & S_2' \,/\, e_2' \end{array} \quad (\vdots = {}^{+})$$

Proof. By induction on the derivation of $S_1 \,/\, e_1 \to S_2 \,/\, e_2$. We give only the most interesting case, namely that of β-reduction.

∘ *Case* e_1 is $(\lambda x.e)\,v$ and e_2 is $[x \mapsto v]e$. Examining the hypothesis $S_1 \,/\, (\lambda x.e)\,v \gtrsim S_1' \,/\, e_1'$, we find that e_1' must be of the form $\mathit{apply}\,(m\,\{[\bar{x} \mapsto \bar{v}']\bar{y}\})\,v'$, where $\bar{y}$ is $\mathrm{fv}(\lambda x.e_m)$ (**1**), $\bar{x} \subseteq \bar{y}$ holds (**2**), e is $[\bar{x} \mapsto \bar{v}]e_m$ (**3**), $\bar{v}$ and $\bar{v}'$ are x-closed and satisfy $\bar{v} \gtrsim \bar{v}'$ (**4**), and $v \gtrsim v'$ holds (**5**).

Because we are dealing with closed configurations, e_1 is x-closed, which implies that e has no free (x-class) variables other than x. Given (3), this implies that the free variables of e_m form a subset of $\bar{x} \cup \{x\}$. Given (1), this yields $\bar{y} \subseteq \bar{x}$. Given (2), $\bar{y}$ and $\bar{x}$ must coincide, so we may write $\{[\bar{x} \mapsto \bar{v}']\bar{y}\}$ as $\{\bar{y} = \bar{v}'\}$.

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

$$\begin{array}{ll} & S_1' \,/\, e_1' \\ = & S_1' \,/\, \mathit{apply}\,(m\,\{\bar{y} = \bar{v}'\})\,v' \\ \to^+ & S_1' \,/\, \mathsf{case}\,m\,\{\bar{y} = \bar{v}'\}\,\mathsf{of}\,(m\,\{\bar{y}\} \mapsto \mathsf{let}\,x = v'\,\mathsf{in}\,e_m') \mid \ldots \\ \to & S_1' \,/\, \mathsf{let}\,x = v'\,\mathsf{in}\,[\bar{y} \mapsto \bar{v}']e_m' \\ & \text{since } v' \text{ and } \bar{v}' \text{ are } x\text{-closed and, by (1), } x \notin \bar{y} \text{ holds} \\ \to & S_1' \,/\, [x \mapsto v'][\bar{y} \mapsto \bar{v}']e_m' \end{array}$$

There remains to verify that the simulation holds. By Lemma 5.1, we have $e_m \gtrsim e_m'$. By (4), (5), and Lemma 5.2, this implies $[x \mapsto v][\bar{y} \mapsto \bar{v}]e_m \gtrsim [x \mapsto v'][\bar{y} \mapsto \bar{v}']e_m'$, that is, $e_2 \gtrsim [x \mapsto v'][\bar{y} \mapsto \bar{v}']e_m'$. The result follows by Config. □

Given a closed expression e, we write $e \Uparrow$ if and only if the configuration $\emptyset \,/\, e$ admits an infinite reduction sequence; we 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 5.1 *$p \Uparrow$ implies $[\![p]\!] \Uparrow$. $p \Downarrow$ implies $[\![p]\!] \Downarrow$.*

Proof. To begin, let us notice that, by definition, $[\![p]\!]$ is $\mathsf{letrec}\ \mathit{apply}\ \mathsf{in}\ p'$. Thus, the configuration $\emptyset \,/\, [\![p]\!]$ reduces, in one step, to $\mathit{apply} \,/\, p'$. Furthermore, by Lemma 5.1, STORE, and CONFIG, $\emptyset \,/\, p \gtrsim \mathit{apply} \,/\, p'$ holds.

Now, assume p diverges. Then, $\emptyset \,/\, p$ admits an infinite reduction sequence. By Lemma 5.4, so does $\mathit{apply} \,/\, p'$, 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, $[\![p]\!]$ 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 to a value. □

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 possible 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. 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 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 5.5 *Let* $\mathsf{true}, \varnothing \vdash p : \tau_p$. *Then,* $p \Uparrow$ *implies* $\lfloor p \rfloor \Uparrow$, *and* $p \Downarrow$ *implies* $\lfloor p \rfloor \Downarrow$.

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 5.6 *Let* $\mathsf{true}, \varnothing \vdash p : \tau_p$. *Then,* $[\![\lfloor p \rfloor]\!]$ *is* $\lfloor [\![p]\!] \rfloor$.

Using Theorem 5.1 as well as the previous two Lemmas, it is now straightforward to establish the correctness of typed defunctionalization. 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, so as to obtain a stronger result (Theorem 5.1).

6 Discussion

It is worth noting that recursive or mutually recursive functions in the source program do not cause any extra 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 records of 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. The strict functional language Objective Caml implements such a semantics. Our treatment of mutually recursive function definitions corresponds to Morrisett and Harper's `fixpack`-based extension to closure conversion [9].

The reader may notice that the simply-typed version of defunctionalization described in the introduction is more efficient than the one presented in this paper, because specializing *apply* with respect to ground types τ_1 and τ_2 allows setting up smaller dispatch tables. 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 (non-ground) types, whose free type variables are $\bar{\alpha}$, then one may define a specialized function $\mathit{apply}_{\exists\bar{\alpha}.\tau_1 \to \tau_2}$, 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 $\tau_1 \to \tau_2$. The resulting program is still well-typed: indeed, as pointed out by Xi [17], a type system equipped with guarded algebraic data types supports identification and elimination of dead branches: they are the branches whose typing hypothesis is inconsistent. Thus, whereas, in the simply-typed case, type-based specialization was mandatory in order to achieve type preservation, it is now optional.

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 $2, 3, \ldots$ 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.

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 [6] 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 ML, because the value of its first argument dictates the number and types of its remaining arguments. Danvy [5] suggested a clever way of expressing *sprintf* in ML by encoding format specifiers as first-class functions. More recently, Xi, Chen, and Chen [18] showed that *sprintf* may be expressed in a more

direct style, whereby format specifiers are data structures, in an extension of ML with 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 may be viewed as a tool to turn existing ML programs that use clever continuation-based tricks to work around the limitations of ML's type system back into perhaps more natural, first-order programs, expressed in an extension of ML with guarded algebraic data types.

References

[1] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In *International Symposium on Theoretical Aspects of Computer Software (TACS)*, volume 2215 of *Lecture Notes in Computer Science*, pages 420–447. Springer Verlag, October 2001.

[2] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In *ACM International Conference on Functional Programming (ICFP)*, August 1997.

[3] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *European Symposium on Programming (ESOP)*, volume 1782 of *Lecture Notes in Computer Science*, pages 56–71. Springer Verlag, March 2000.

[4] James Cheney and Ralf Hinze. First-class phantom types. Technical Report 1901, Cornell University, 2003.

[5] Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, November 1998.

[6] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *ACM International Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 162–174, September 2001.

[7] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 237–247, 1993.

[8] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–283, January 1996.

[9] Greg Morrisett and Robert Harper. Typed closure conversion for recursively-defined functions (extended abstract). In *International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1998.

[10] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[11] Lasse R. Nielsen. A denotational investigation of defunctionalization. Technical Report RS-00-47, BRICS, December 2000.

[12] Christine Paulin-Mohring. Inductive definitions in the system coq: Rules and properties. Research Report RR1992-49, ENS Lyon, 1992.

[13] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, December 1998.

[14] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, December 1998.

[15] Andrew Tolmach. Combining closure conversion with closure analysis using algebraic types. In *Workshop on Types in Compilation (TIC)*, June 1997.

[16] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

[17] Hongwei Xi. Dead code elimination through dependent types. In *International Workshop on Practical Aspects of Declarative Languages (PADL)*, volume 1551 of *Lecture Notes in Computer Science*, pages 228–242. Springer Verlag, January 1999.

[18] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2003.