A verified framework for higher-order uncurrying optimizations

Zaynah Dargaye, Xavier Leroy

Higher-Order and Symbolic Computation manuscript No.
(will be inserted by the editor)

A verified framework for higher-order uncurrying optimizations

Zaynah Dargaye · **Xavier Leroy**

Abstract Function uncurrying is an important optimization for the efficient execution of functional programming languages. This optimization replaces curried functions by uncurried, multiple-argument functions, while preserving the ability to evaluate partial applications. First-order uncurrying (where curried functions are optimized only in the static scopes of their definitions) is well understood and implemented by many compilers, but its extension to higher-order functions (where uncurrying can also be performed on parameters and results of higher-order functions) is challenging. This article develops a generic framework that expresses higher-order uncurrying optimizations as type-directed insertion of coercions, and prove its correctness. The proof uses step-indexed logical relations and was entirely mechanized using the Coq proof assistant.

Keywords Functional programming languages · Compiler optimization · Compiler verification · Semantic preservation · Mechanized verification

1 Introduction

1.1 Currying in functional languages

Most programming languages support n-ary functions, that is, functions that take zero, one or several arguments. Following the models of λ-calculus and set theory, some functional languages provide only unary functions, that is, functions of exactly one argument. This is the case in programming languages such as Standard ML, Caml, and Haskell, and also in specification languages of proof assistants such as HOL and Coq. In these languages, multiple-argument functions are encoded in terms of single-argument functions. One such encoding puts multiple arguments in a tuple or record, which is passed to the function as a single argument, and from which the function recovers the arguments by pattern-matching. For example, the two-argument function taking x and y and returning $x+y$ is encoded as

Zaynah Dargaye
ENSIIE, 1, square de la Résistance, 91025 Évry, France. Author's current address: CEA LIST, Saclay, 91191 Gif-sur-Yvette, France. E-mail: lea-zaynah.dargaye@cea.fr

Xavier Leroy
INRIA Paris-Rocquencourt, Domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France.
E-mail: xavier.leroy@inria.fr

$\lambda p.\ \texttt{let}\ (x,y) = p\ \texttt{in}\ x+y$. Another encoding, called *currying*[1], uses functions that return functions to achieve the effect of passing several arguments. Continuing the example above, the two-argument addition function is encoded as $f = \lambda x.\ \lambda y.\ x+y$; applying it to the arguments 1 and 2 is performed by two successive applications $(f\ 1)\ 2$, also written $f\ 1\ 2$.

The choice between the two encodings is largely a matter of programming style. For example, the Standard ML community favors the use of tuples, while currying is the preferred approach among Caml and Haskell programmers. One technical advantage in favor of currying, however, is the ease with which it expresses *partial applications* of n-ary functions to $m < n$ initial arguments. For example, if $f = \lambda x.\ \lambda y.\ x+y$, the partial application $f\ 1$ computes $\lambda y.\ 1+y$, that is, a one-argument function representing the specialization of f to $x = 1$.

1.2 The run-time cost of currying

Both encodings of n-ary functions entail significant run-time costs if implemented naively. We concentrate on the currying encoding in the following. Consider a simple call-by-value, "eval-apply" strategy (in the terminology of [27]) as embodied in the SECD [23] and CEK [16] abstract machines, for example. In this setting, a total application $f\ 1\ 2$ of a 2-argument curried function f proceeds as follows. First, f is evaluated to a function value $\lambda x.\lambda y.M$. Second, the first argument 1 is evaluated. Third, the function is entered, binding 1 to x, and returns the function closure representing $\lambda y.M\{x \leftarrow 1\}$. Fourth, the second argument 2 is evaluated. Fifth, the intermediate function $\lambda y.M\{x \leftarrow 1\}$ is entered with y bound to 2, and the function body M is evaluated at last. Generalizing over the number of arguments, we see that a total application of a n-ary curried function entails the creation of $n-1$ short-lived intermediate closures, as well as $2n$ control-flow jumps between the caller and the callee. In contrast, an execution model that natively supports n-ary functions would perform only 2 such jumps and avoid the creation of intermediate closures entirely, saving significant run-time costs.

Some of these inefficiencies can be avoided by using a "push-enter" execution model (following again the terminology of [27]) instead of an "eval-apply" model. Examples of "push-enter" models include Krivine's machine [22], the STG abstract machine [30], and the ZINC abstract machine [24]. In these models, a curried application such as $f\ 1\ 2$ is evaluated by first pushing the arguments on a stack, in right-to-left order (first 2, then 1), keeping track of the number of available arguments, then evaluating f to a function value, then entering its code. A prelude to the function checks that enough arguments were provided. If so, the body of the function is evaluated. If not, a function closure representing the partial application is immediately returned to the caller.

The "push-enter" model avoids most of the costs of currying in the "eval-apply" model, especially the creation of intermediate closures. Compared with a native implementation of n-ary functions, some overhead remains, however. For instance, parameters must be passed in memory: hardware registers cannot be used for parameter passing. Also, maintaining the count of available arguments at run-time and testing it on entrance to every n-ary function is costly in execution time. For these reasons, we now turn to another approach: static (compile-time) uncurrying, and concentrate on it in the remainder of this article.

[1] In honor of the logician Haskell B. Curry. Curry himself attributes this technique to Schönfinkel [13], but it was used earlier by Frege according to Cardone and Hindley [10].

1.3 Static uncurrying

Static uncurrying is a compile-time optimization that translates an ML- or Haskell-like surface language of unary functions to an intermediate language featuring n-ary functions without partial applications (i.e. the number of arguments provided at calls must match exactly the arity of the callee). Such an intermediate language can, then, be compiled to efficient machine code, with registers being used for parameter passing and no run-time arity checks on entrance to functions.

We now describe the simple static uncurrying strategy that is implemented in the OCaml native-code compiler and recent versions of the Glasgow Haskell compiler, among others. It detects curried function abstractions that are bound by a `let` (or similar local, top-level or module-level binding construct) and turns them into n-ary functions of the target language[2]:

```
let f = λx.λy. x + y          ⇒    let f = λ(x,y). x + y
```

Within the lexical scope of their bindings, total, curried applications of these functions are turned into single, n-ary applications:

```
in f 1 2                      ⇒    in f(1, 2)
```

Partial applications of these functions, as well as any use of these functions as first-class values, are handled by re-constructing a curried version of the n-ary function. Changing only the "`in`" part of the example above, we have the following translations: [3]

```
in List.map2 f l1 l2          ⇒    in List.map2 (curry2(f), l1, l2)
in List.map (f 3) l3          ⇒    in List.map (curry2(f)(3), l3)
```

The $\texttt{curry}_2$ combinator is defined as $\lambda(f).\,\lambda(x).\,\lambda(y).\,f(x,y)$ in the intermediate language. It takes a function f of arity 2 and returns the equivalent curried function. One such $\texttt{curry}_n$ combinator is defined for each arity n of interest.

There exist several variants of this uncurrying strategy. (The one outlined above is approximately the one used in OCaml.) For instance, the `curry` combinators can be expanded in-line and reduced, at some cost in code size: the partial application `f 3` would then be translated as `let x = 3 in` λ`y. f(x,y)`. Also, the Glasgow Haskell compiler exploits combinators $\texttt{papp}_{n,m} = \lambda(f,x_1,\ldots,x_m).\,\lambda(x_{m+1})\ldots\lambda(x_n).\,f(x_1,\ldots,x_n)$ to speed up the partial application of a n-ary function f to $m < n$ initial arguments.

The performance characteristics of this approach can be summarized as follows. Within the lexical scope of a named definition of a curried function, total applications of the function name are compiled optimally to a n-ary application. All other uses of curried functions (partial applications, passing a curried function as parameter to a higher-order function, storing curried functions in data structures, etc) suffer from the same run-time costs exhibited by the naive "eval-apply" model. However, the first case (total applications of known functions) is very common: Marlow and Peyton Jones [27] report that it amounts for 80% of function calls on a Haskell benchmark suite; the second author observed similar numbers on an OCaml benchmark suite. This uncurrying strategy therefore performs well in practice. Compared with a "push-enter" execution model, it also simplifies the run-time system [27] and is more compatible with the use of standard, currying-unaware compiler back-ends.

[2] Here and throughout this paper, $\lambda(x,y)\ldots$ denotes a two-argument function of the target language, not a function that takes a pair as single argument. Likewise, $f(x,y)$ is an application to two arguments, as in C or Java, not an application to the pair (x,y) as in ML or Haskell.

[3] `List.map` and `List.map2` are the familiar higher-order functions that iterate a function over the elements of one or two lists, respectively.

The correctness of static uncurrying (that the generated intermediate code does not get stuck on an arity mismatch at application-time and computes the same results as the original code) is intuitively clear, but not obvious to prove in full details. As part of her work on the formal verification of a compiler front-end for the MiniML functional language, the first author developed a mechanically-checked proof of correctness for the uncurrying optimization outlined above [14]. The present article extends this work to more aggressive static uncurrying schemes that work across higher-order functions.

1.4 Higher-order static uncurrying

The static uncurrying strategy outlined in section 1.3 is first-order in nature: as soon as a curried function is passed as argument to a higher-order function or used as a first-class value, no uncurrying takes place. This is clearly suboptimal: opportunities for uncurrying across higher-order functions are not uncommon, as the following example demonstrates. Consider the `map2` iterator from OCaml's `List` module, which applies a two-argument function pairwise to the elements of two lists.

```
let rec map2 = λf.λl1.λl2.
  match (l1, l2) with
  | ([], []) -> []
  | (h1::t1, h2::t2) -> f h1 h2 :: map2 f t1 t2
  | (_, _) -> fail
in map2 (λx.λy. x + y) l1 l2
```

First-order uncurrying produces the following intermediate code, where `map2` is uncurried but its `f` parameter remains curried:

```
let rec map2 = λ(f, l1, l2).
  match (l1, l2) with
  | ([], []) -> []
  | (h1::t1, h2::t2) -> f(h1)(h2) :: map2(f, t1, t2)
  | (_, _) -> fail
in map2((λx.λy. x + y), l1, l2)
```

A higher-order uncurrying optimization could uncurry the parameter `f` as well, turning it into a 2-argument function and improving the efficiency of the code:

```
let rec map2 = λ(f, l1, l2).
  match (l1, l2) with
  | ([], []) -> []
  | (h1::t1, h2::t2) -> f(h1, h2) :: map2(f, t1, t2)
  | (_, _) -> fail
in map2((λ(x, y). x + y), l1, l2)
```

What happens if, somewhere else in the program, we need to apply `map2` to a function `f` that has not been uncurried? Just like the $\texttt{curry}_n$ combinators were used previously to resolve mismatches between an uncurried function definition and a curried use site, we can use the $\texttt{uncurry}_n$ combinators

$$\texttt{uncurry}_n = \lambda(f).\lambda(x_1,\ldots,x_n).\ f(x_1)\cdots(x_n)$$

to resolve mismatches between a curried function definition and an uncurried use site. For example, applying `map2` above to a curried function `f` can be done as $\texttt{map2}(\texttt{uncurry}_2(\texttt{f}), \texttt{l1}, \texttt{l2})$.

The `curry` and `uncurry` combinators can be combined together to resolve more complicated mismatches. For example, the term $\lambda(x).\texttt{uncurry}_2(\texttt{curry}_3(f)(x))$ transforms a 3-argument uncurried function f into a semi-curried function that takes one argument, then returns an uncurried function of 2 arguments.

Higher-order uncurrying is a natural extension of the familiar, first-order uncurrying optimizations, but it has not received much attention. The only prior work on higher-order uncurrying that we are aware of is that of Hannan and Hicks [19], who developed a type system to validate a posteriori the correctness of an uncurrying transformation. We are not aware of any compiler for a functional language that implements higher-order uncurrying. One possible reason is that choosing between different possible higher-order uncurryings (for example, to decide whether to uncurry the `f` parameter to the `map2` function above) is difficult and no good heuristics are known. We will not settle this issue in this paper, but rather concentrate on characterizing and proving correct a whole family of possible higher-order uncurrying optimizations.

1.5 This work

In this article, we develop a general framework for higher-order uncurrying optimizations and prove their correctness through a generic semantic preservation argument. Section 2 develops a nondeterministic specification of what constitutes a valid uncurrying of a source program written in a simple, untyped functional language with unary functions. The specification is presented as a non-standard type system instrumented with the generation of uncurried terms. Coercions are introduced in a type-directed manner to solve arity mismatches, generalizing the `curry` and `uncurry` combinators used in earlier examples. To prove that these uncurryings are correct (i.e. preserve the semantics of the source program), we use step-indexed logical relations, a powerful semantic tool introduced by Appel and McAllester [7] and reviewed in section 3. The proof of semantic preservation is presented in section 4. Two applications of our framework are worked out in section 5: one is a very simple correctness proof for the first-order uncurrying optimization outlined in section 1.3; the other is a validating translation that exploits the results of an external uncurrying strategy, after checking them for correctness, to generate the corresponding uncurried code. Issues with non-terminating programs are discussed in section 6. Section 7 discusses related work, followed by concluding remarks and perspectives for future work in section 8.

Our work improves on that of Hannan and Hicks [19] in two ways. First, we show how to automate the insertion of coercions to resolve 'impedance' mismatches between parts of program that make different uncurrying decisions. This was mentioned but left as future work by Hannan and Hicks. Second, our framework is independent of the type system (if any) of the source language, while that of Hannan and Hicks relies on a specific type system (Hindley-Milner polymorphism). Extending their framework to the richer type systems of languages like Haskell or OCaml, or to the even richer type theories found in proof assistants like HOL or Coq[4] is challenging. In contrast, our framework also uses types to capture compile-time arity information, but uses them in the style of soft typing [38]: a

[4] One application initially envisioned for this work is as part of a larger effort to develop and prove correct a compiler for pure, formally verified functional programs directly written in the specification language of Coq.

universal "top" type of fully curried functions can always be used as an escape hatch when contradictory compile-time arity information arises. Our work is therefore applicable to any functional language, statically typed or not.

The formalization and proofs presented in this paper have been mechanically checked using the Coq proof assistant [11,9]. The complete Coq development is available on the Web at `http://gallium.inria.fr/~xleroy/uncurry/`. Taking advantage of this, we only sketch the proofs in this paper, referring the reader to the Coq development for full details.

2 Specification of the uncurrying transformation

2.1 The source language

The source language for the uncurrying transformation is a standard λ-calculus with constants, enriched with a `let` binding and a `letrec` binding for defining recursive functions. While both bindings could be encoded in the λ-calculus fragment of the source language, having them as primitive constructs facilitates uncurrying. We omit other features of functional languages (arithmetic, data types, conditionals, pattern-matching, ...) since these features are orthogonal to the uncurrying optimization.

Constants: $c ::= 0 \mid 1 \mid \ldots$

Terms:		
$a ::= x_n$		variable (as its de Bruijn index n)
$\mid c$		constant
$\mid \lambda.a$		(unary) function abstraction
$\mid a\ a'$		(unary) function application
$\mid$ `let` a `in` a'		local binding
$\mid$ `letrec` $\lambda.a$ `in` a'		recursive binding of a function

Alpha-conversion of bound variables is a delicate issue in most mechanized formalizations of program analyses and transformations [8]. To avoid this issue, we represent variables by de Bruijn indices: x_n is the variable bound by the $(n+1)^{\text{th}}$ enclosing binding construct. Consequently, $\lambda.a$ binds x_0 to the function parameter in the body a, and `let` a `in` a' binds the value of a to x_0 during the evaluation of a'. Consider `letrec` $\lambda.a$ `in` a'. In a, the function parameter is x_0 and the recursive function $\lambda.a$ itself is x_1. In a', x_0 refers to the recursive function $\lambda.a$.

In examples and informal discussions, we will use variable names $f, y, z, \ldots$ for legibility. In standard named notation, our functions are of the shape $\lambda y.a$ and our local bindings are of the shape `let` $y = a$ `in` a' and `letrec` $f = \lambda y.a$ `in` a'.

2.2 The target language

The target language for the uncurrying transformation is similar to the source language, except that it features n-ary function abstractions and applications.

Terms:	$b ::= x_n$		variable (as its de Bruijn index n)
	$\mid c$		constant
	$\mid \lambda^n.b$		n-ary function abstraction
	$\mid b(b_1,\ldots,b_n)$		n-ary function application
	$\mid$ `let` b `in` b'		local binding
	$\mid$ `letrec` $\lambda^n.b$ `in` b'		recursive binding of a n-ary function

The term $\lambda^n.b$ denotes a function of n arguments, which are bound in b to $x_{n-1},\ldots,x_1,x_0$ (in this order). In named syntax, such functions are written $\lambda(y_1,\ldots,y_n).\ b$. The target language naturally supports functions with no arguments (also known as thunks), but the uncurrying transformation will not generate them.

The term $b(b_1,\ldots,b_n)$ applies the function b to the n arguments $b_1,\ldots,b_n$. Partial applications and over-applications are not supported: reduction takes place only if b evaluates to a function of exactly n arguments; evaluation is stuck otherwise. For a recursive function binding `letrec` $\lambda^n.b$ `in` b', variables $x_{n-1},\ldots,x_1,x_0$ are bound in b to the function arguments, and x_n is bound to the recursive function itself.

To guide the uncurrying transformation, we classify terms of the target language according to their *representation types* τ. These types capture information on the arity of functions.

Representation types:

$\tau ::= \top$		constants or fully-curried functions
	$\mid (\tau_1,\ldots,\tau_n) \to \tau$	n-ary functions

Compilation environments:

$\Gamma ::= \tau_0 \ldots \tau_n$	maps de Bruijn indices to representation types

The representation type $(\tau_1,\ldots,\tau_n) \to \tau$ characterizes a function of arity n that expects n arguments conforming to types $\tau_1,\ldots,\tau_n$ and returns a result conforming to τ. For unary functions ($n = 1$), we write $\tau_1 \to \tau_2$ instead of $(\tau_1) \to \tau_2$.

The representation type $\top$ corresponds both to non-function values such as constants and to *fully-curried* functions: functions of arity 1 that expect a fully-curried argument and produce a fully-curried result. In other words, a function of type $\top \to \top$ can be viewed as having type $\top$. (See rule SBASE in section 2.3.) Conversely, it is possible to apply a term b of type $\top$ to one argument of type $\top$, obtaining a result of type $\top$. At run-time, such an application can succeed if b is a (fully-curried) function, or fail if b evaluates to a constant. The subset of target terms that have representation type $\top$ is therefore isomorphic to the source language. It is clear that our type algebra is too weak to guarantee type soundness, but this is not its purpose.

2.3 The uncurrying transformation

Many different uncurrying strategies can be considered, and even more so when uncurrying is performed across higher-order functions. Rather than prove the correctness of one particular strategy, we set out to specify (nondeterministically) a large class of uncurrying strategies, which we prove semantically correct in section 4. The specification is presented as a predicate $\Gamma \vdash a \Longrightarrow b : \tau$ saying that a target term b is a correct uncurrying of the source term a. In this predicate, τ is the desired representation type for b (as we will see shortly, different representation types lead to different uncurryings), and the compilation environment Γ associates representation types to the free variables of a, which are also the free variables of b.

The predicate $\Gamma \vdash a \Longrightarrow b : \tau$ is defined by a set of inference rules reminiscent of a type system. The rules are not syntax-directed, therefore b is not a function of Γ, a, τ but rather of a whole derivation.

$$\frac{\Gamma(n) = \tau}{\Gamma \vdash x_n \Longrightarrow x_n : \tau}\ (\textsc{Tvar}) \qquad \Gamma \vdash c \Longrightarrow c : \top\ (\textsc{Tcst})$$

$$\frac{n > 0 \qquad \tau_n \ldots \tau_1.\Gamma \vdash a \Longrightarrow b : \tau}{\Gamma \vdash \underbrace{\lambda \ldots \lambda}_{n \text{ times}}.a \Longrightarrow \lambda^n.b : (\tau_1, \ldots, \tau_n) \to \tau}\ (\textsc{Tabs})$$

$$\frac{\Gamma \vdash a \Longrightarrow b : \top \qquad \Gamma \vdash a' \Longrightarrow b' : \top}{\Gamma \vdash a\ a' \Longrightarrow b(b') : \top}\ (\textsc{Tapp-}\top)$$

$$\frac{\begin{array}{c} n > 0 \qquad \Gamma \vdash a \Longrightarrow b : (\tau_1, \ldots, \tau_n) \to \tau \\ \Gamma \vdash a_i \Longrightarrow b_i : \tau_i \text{ for } i = 1, \ldots n \end{array}}{\Gamma \vdash a\ a_1\ \ldots\ a_n \Longrightarrow b(b_1, \ldots, b_n) : \tau}\ (\textsc{Tapp-}N)$$

$$\frac{\Gamma \vdash a \Longrightarrow b : \tau \qquad \tau.\Gamma \vdash a' \Longrightarrow b' : \tau'}{\Gamma \vdash \texttt{let}\ a\ \texttt{in}\ a' \Longrightarrow \texttt{let}\ b\ \texttt{in}\ b' : \tau'}\ (\textsc{Tlet})$$

$$\frac{\begin{array}{c} n > 0 \qquad \tau_n \ldots \tau_1.((\tau_1, \ldots, \tau_n) \to \tau).\Gamma \vdash a \Longrightarrow b : \tau \\ ((\tau_1, \ldots, \tau_n) \to \tau).\Gamma \vdash a' \Longrightarrow b' : \tau' \end{array}}{\Gamma \vdash \texttt{letrec}\ \underbrace{\lambda \ldots \lambda}_{n \text{ times}}.a\ \texttt{in}\ a' \Longrightarrow \texttt{letrec}\ \lambda^n.b\ \texttt{in}\ b' : \tau'}\ (\textsc{Tletrec})$$

$$\frac{\Gamma \vdash a \Longrightarrow b : \tau \qquad b : \tau \Longrightarrow b' : \tau'}{\Gamma \vdash a \Longrightarrow b' : \tau'}\ (\textsc{Tcoerce})$$

The rules for variables and constants (TVAR and TCST) are straightforward. The first interesting rule is TABS. At a function type $(\tau_1, \ldots, \tau_n) \to \tau$, a source-level curried abstraction of at least n parameters $\lambda y_1, \ldots, y_n.\ a$ uncurries to the n-ary abstraction $\lambda(y_1, \ldots, y_n).\ b$, provided that a uncurries to b at result type τ under the additional assumptions $y_i : \tau_i$ $(i = 1, \ldots, n)$. For example, the source term $\lambda x.\lambda y.\ y$ becomes the binary function $\lambda(x, y).\ y$ if the desired type is $(\top, \top) \to \top$ but remains a curried function $\lambda(x).\ \lambda(y).\ y$ if the desired type is $\top \to \top \to \top$. Two rules deal with function applications. Rule TAPP-$\top$ matches single applications $a\ a'$ at type $\top$ where the function part a is uncurried to b at type $\top$, meaning that it is a fully-curried function. The argument a' is uncurried to b' at type $\top$ as well and the generated target term is the unary application $b(b')$. The other application rule, TAPP-N, deals with source-level curried applications $a\ a_1\ \ldots\ a_n$ at type τ. The function part a must uncurry to b at some n-ary function type $(\tau_1, \ldots, \tau_n) \to \tau$. In this case, a n-ary application $b(b_1, \ldots, b_n)$ is generated, where b_i is an uncurrying of a_i at type τ_i. The case of `let` bindings (rule TLET) is straightforward. For `letrec` bindings, rule TLETREC combines rules TABS and TLET. The recursive nature of the binding is apparent in the extra hypothesis $x_n : (\tau_1, \ldots, \tau_n) \to \tau$ introduced in the compilation environment when translating the left subterm of the binding. Rule TCOERCE accounts for most of the flexibility of our specification. It enables us to take an uncurrying b obtained at some type τ and turn it into an uncurrying b' at another type τ', provided there exists a *coercion* from τ to τ' that transforms b into b'. (Typical examples of coercions are applications of the `curry` and `uncurry`

combinators mentioned in Introduction.) Such a coercion is written $b : \tau \Longrightarrow b' : \tau'$. A special case of coercion is *subtyping*, written $\tau <: \tau'$, where the type changes but the result of the translation does not ($b' = b$). We now define subtyping and coercion via inference rules, starting with the former.

$$\tau <: \tau \ \text{(SREFL)} \qquad \frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \ \text{(STRANS)}$$

$$\top \to \top <: \top \ \text{(SBASE)} \qquad \frac{\tau'_i <: \tau_i \text{ for } i = 1, \ldots, n \qquad \tau <: \tau'}{(\tau_1, \ldots, \tau_n) \to \tau <: (\tau'_1, \ldots, \tau'_n) \to \tau'} \ \text{(SFUN)}$$

Subtyping is reflexive and transitive. The base case (rule SBASE) corresponds to $\top \to \top$ being a subtype of $\top$. As mentioned in section 2.2, a target term of type $\top \to \top$ is a fully-curried function and can therefore be injected in the type $\top$ of constants or fully-curried functions. Subtyping then extends across function types as usual (rule SFUN): covariantly in the result type and contravariantly in the argument types. The following lemma (used in section 5.1) is an example of complex subtyping.

Lemma 1 *Define the type τ_n of curried functions of n $\top$ arguments by*

$$\tau_n = \underbrace{\top \to \cdots \to \top}_{n \text{ times}} \to \top$$

We have $\tau_n <: \top$ for all n.

Proof The base case $n = 0$ is trivial by rule SREFL. For the inductive case, note that $\tau_{n+1} = \top \to \tau_n$. By induction hypothesis and rules SFUN and SREFL, we have $\top \to \tau_n <: \top \to \top$. By rule SBASE, $\top \to \top <: \top$. We conclude by rule STRANS. □

We now turn to the definition of coercions $b : \tau \Longrightarrow b' : \tau'$.

$$\frac{\tau <: \tau'}{b : \tau \Longrightarrow b : \tau'} \ \text{(CSUB)} \qquad \frac{b : \tau \Longrightarrow b' : \tau' \qquad b' : \tau' \Longrightarrow b'' : \tau''}{b : \tau \Longrightarrow b'' : \tau''} \ \text{(CTRANS)}$$

$$\frac{n > 0}{b : (\tau_1, \ldots, \tau_n) \to \tau \Longrightarrow \mathtt{curry}_n(b) : \tau_1 \to \ldots \to \tau_n \to \tau} \ \text{(CCURRY)}$$

$$\frac{n > 0}{b : \tau_1 \to \ldots \to \tau_n \to \tau \Longrightarrow \mathtt{uncurry}_n(b) : (\tau_1, \ldots, \tau_n) \to \tau} \ \text{(CUNCURRY)}$$

$$\frac{x_{n-i-1} : \tau'_i \Longrightarrow b_i : \tau_i \text{ for } i = 1, \ldots, n \qquad x_n(b_1, \ldots, b_n) : \tau \Longrightarrow b' : \tau'}{b : (\tau_1, \ldots, \tau_n) \to \tau \Longrightarrow (\mathtt{let}\ b\ \mathtt{in}\ \lambda^n.b') : (\tau'_1, \ldots, \tau'_n) \to \tau'} \ \text{(CFUN)}$$

Coercions are defined by three base cases (rules CSUB, CCURRY and CUNCURRY), which can be combined by transitivity (rule CTRANS) or extended across function types (rule CFUN). One base case is when τ is subtype of τ'. Then, the coercion takes place without changing the term ($b' = b$). The other two base cases correspond to applications of the $\mathtt{curry}_n$ or $\mathtt{uncurry}_n$ combinators. There is one such combinator for each arity $n > 0$, defined as:

$$\mathtt{curry}_n = \lambda^1.\ \underbrace{\lambda^1 \ldots \lambda^1}_{n \text{ times}}.\ x_n(x_{n-1}, \ldots, x_0)$$

$$\mathtt{uncurry}_n = \lambda^1.\ \lambda^n.\ x_n(x_{n-1}) \cdots (x_0)$$

Source type τ	Destination type τ'	Term b' after coercion of b
$(\top,\top)\to\top$	$\top$	$\texttt{curry}_2(b)$
$\tau_1\to\tau_2\to\tau$	$(\tau_1,\tau_2)\to\tau$	$\texttt{uncurry}_2(b)$
$\top\to\top$	$((\top,\top)\to\top)\to\top$	$\texttt{let } f = b \texttt{ in } \lambda(x).\ f(\texttt{curry}_2(x))$
$(\tau_1,\tau_2,\tau_3)\to\tau$	$\tau_1\to(\tau_2,\tau_3)\to\tau$	$\texttt{let } f = \texttt{curry}_3(b) \texttt{ in } \lambda(x).\ \texttt{uncurry}_2(f(x))$
$(\tau_1,\tau_2,\tau_3)\to\tau$	$(\tau_1,\tau_2)\to\tau_3\to\tau$	$\texttt{uncurry}_2(\texttt{curry}_3(b))$

Table 1 Some examples of coercions

or in more readable named notation:

$$\begin{aligned}\texttt{curry}_n &= \lambda(f).\ \lambda(y_1)\ldots\lambda(y_n).\ f(y_1,\ldots,y_n)\\ \texttt{uncurry}_n &= \lambda(f).\ \lambda(y_1,\ldots,y_n).\ f(y_1)\cdots(y_n)\end{aligned}$$

As shown by rules CCURRY and CUNCURRY, these combinators mediate between the uncurried function type $(\tau_1,\ldots,\tau_n)\to\tau$ and its curried counterpart $\tau_1\to\ldots\to\tau_n\to\tau$. We could dispense with the $\texttt{curry}_n$ and $\texttt{uncurry}_n$ combinators and expand them at point of use, as in:

$$b : (\tau_1,\tau_2)\to\tau \Longrightarrow (\texttt{let } f = b \texttt{ in } \lambda(x).\lambda(y).\ f(x,y)) : \tau_1\to\tau_2\to\tau$$

replacing a function call by a cheaper `let` binding. However, the `curry` and `uncurry` combinators can be shared across several coercion sites, therefore reducing the overall code size. Finally, rule CFUN enables us to coerce from a function type $(\tau_1,\ldots,\tau_n)\to\tau$ to another function type of the same arity, $(\tau'_1,\ldots,\tau'_n)\to\tau'$. By covariance and contravariance, there must exist coercions from the result type τ to τ', and from each of the argument types τ'_i to τ_i. These coercions are combined in the standard way:

$$\begin{array}{ccccc} b_1:\tau_1 & \ldots & b_n:\tau_n & \to & f(b_1,\ldots,b_n):\tau \\ \Uparrow & & \Uparrow & & \Downarrow \\ y_1:\tau'_1 & \ldots & y_n:\tau'_n & \to & b':\tau' \end{array}$$

(In this diagram, $\Uparrow$ and $\Downarrow$ stand for coercions, and $\to$ denotes functions.) The term generated as the result of the coercion, $\texttt{let } f = b \texttt{ in } \lambda(y_1,\ldots,y_n).\ b'$, first binds b to a variable to avoid recomputing it, then returns the n-ary function appropriate for the destination type. Note that the subtyping rule SFUN for function types is partially redundant with the coercion rule CFUN: SFUN could be removed without changing the set of type pairs (τ,τ') between which coercions exist; however, the terms generated after coercion would be less efficient. For example, coercing b from $\top\to\top\to\top$ to $\top$ produces b if SFUN is available, but $\lambda x.\ b(x)$ if it is not. More generally, systematic use of the subtyping rules avoids generating coercions that are $\beta\eta$-equivalent to the identity function; this is an easy but very profitable optimization. Table 1 shows some examples of coercions, using named notation. The first example combines rule CCURRY and subtyping to go from a 2-argument function to a fully-curried function. The third example illustrates the use of rule CFUN. The last two examples show that combinations of currying and uncurrying can build "semi-uncurried" functions that successively take (in a curried manner) groups of uncurried arguments.

3 An introduction to step-indexed logical relations

Many program transformations can be proved to preserve the semantics of programs using a purely syntactic correspondence between the terms and values appearing during the execution of the original program and those appearing during the execution of the transformed program, then showing that this correspondence is preserved by the dynamic semantics. In the case of first-order uncurrying, the first author developed a Coq proof of semantic preservation that follows this approach [14]. However, this technique does not easily extend to the higher-order case.

For some program transformations, it is preferable to adopt a more semantic viewpoint and reason less on the syntactic shape of functions and more on their "input-output" behavior. In other words, the proof of semantic preservation relies on observational equivalence. For example, the two functions $\lambda x.\, x+x$ and $\lambda x.\, x\times 2$ are equivalent because they map identical integer arguments to identical integer results; therefore, one function can replace the other in any context. In the case of higher-order functional languages, function arguments can themselves be functions. Therefore, we would like to define a notion of equivalence between values such that:

- two constants are related if and only if they are equal;
- two functions are related if and only if they map related arguments to related results.

However, this characterization is not a proper definition because it is circular: for instance, the argument to a function can be the function itself, in the case of a self-application.

One way to avoid this circularity and turn the characterization above into a proper definition is to stratify the language using types. This approach is known as *logical relations* [34]. Taking the simply-typed λ-calculus as an example, we would state that

- two constants are related at a base type ι if and only if they are equal and belong to ι;
- two functions are related at type $\tau_1 \to \tau_2$ if and only if they map arguments related at type τ_1 to results related at type τ_2.

This is a well-founded notion because the definition of the "is related to" relation at type τ involves only the same relation at strict sub-terms of τ. However, this type-based stratification does not apply to all type systems. For example, recursive types, ML/Haskell-style datatypes or ML-style references reintroduce circularity in the characterization of logical relations. A fortiori, the type-based stratification is useless for untyped or weakly-typed languages, such as the source and target languages of our uncurrying transformation.

To circumvent this difficult, Appel and McAllester [7] introduced *step-indexed logical relations*, a variant of Statman's logical relation where the definition is well-founded not by induction on types, but by induction on the number of computational steps performed by the terms under consideration. More precisely, step-indexed logical relations do not directly capture the fact that two terms are observationally equivalent in the absolute, but only the fact that two terms cannot be distinguished in k steps of computation, where k can be chosen arbitrarily large. Taking the call-by-value λ-calculus with constants as an example, two terms M and N are indistinguishable in k steps if and only if:

- for all $j \leq k$, if M reduces to a value v in j steps of β-reduction, there exists a value w such that N reduces to w (in any number of steps) and v and w are indistinguishable in $k-j$ steps.

(We focus here on observational equivalence for terminating terms. Non-termination is discussed in section 6.) Likewise, two values v and w are indistinguishable in k steps if and only if:

- v and w are identical constants, or
- $v = \lambda x.M$, $w = \lambda y.N$, and for all $j < k$ and all arguments v', w' indistinguishable in j steps, the terms $M\{x \leftarrow v'\}$ and $N\{y \leftarrow w'\}$ are indistinguishable in j steps.

In the last case above, note that $M\{x \leftarrow v'\}$ will run for at most j steps. In particular, if v' is a function and M applies it, the function v' cannot run for more than j steps. Therefore, it suffices that v' and w' are indistinguishable for j steps.

The definition of "being indistinguishable" above is well-founded by induction on the number k of steps. In particular, in the case of function values, the arguments v', w' are compared for $j < k$ steps and the results v'', w'' are compared for $j - i$ steps, where $i \leq j$ is the length of the reduction sequence $M\{x \leftarrow v'\} \xrightarrow{*} v''$, and therefore $j - i < k$.

Appel and McAllester [7] and Ahmed [5] use the step-indexed approach to define unary and binary logical relations, leading to PER-style models, for rich type system including recursive types and quantifiers. Acar *et al.* [2] use step-indexed logical relations in an untyped setting to prove a consistency property of a non-deterministic operational semantics. In the next section, we use step-index logical relations in a weakly-typed setting to show the correctness of the uncurrying transformation defined in section 2.3.

4 Semantic preservation

4.1 Dynamic semantics

We first give operational semantics to the source and target languages. To simplify the Coq mechanization, we use environments and function closures instead of substitutions at β-reduction time. The values resulting from computations are therefore either constants c or closures $(\lambda.a)[e]$ or $(\lambda^n.b)[f]$, that is, pairs of a λ-abstraction (unary in the source language, n-ary in the target language) and an evaluation environment e or f. Owing to the use of de Bruijn notation, evaluation environments are simply list of values accessed by position.

	Source language	Target language
Values:	$v ::= c \mid (\lambda.a)[e]$	$w ::= c \mid (\lambda^n.b)[f]$
Evaluation environments:	$e ::= v_0 \dots v_n$	$f ::= w_0 \dots w_n$

Values are defined *coinductively* (that is, as finite or infinite regular terms satisfying the grammar above), so that we can use cyclic closures to represent the values of recursive functions. For example, evaluating `letrec` $\lambda.a$ `in` $\dots$ in an environment e produces the cyclic closure $v = (\lambda.a)[v.e]$, an infinite but regular term. When this closure is invoked, variable x_1 is correctly bound to v, that is, the closure of the function itself. This device was introduced by Landin [23] and is also used in the Categorical Abstract Machine [12]. It is convenient to unify the treatment of recursive and non-recursive functions.

Reduction semantics with environments and function closures can be defined following the $\lambda\sigma$-calculus [1]. However, the Coq mechanization is further simplified if we use natural (big-step) semantics instead of reduction (small-step) semantics.

The dynamic semantics for the source language is therefore of the form $e \vdash a \stackrel{n}{\Rightarrow} v$, meaning "in environment e, term a evaluates in n steps to the value v". This judgment is defined by the inference rules shown in figure 1. The rules are standard except for (1) the use of cyclic closures in the rule for `letrec`, as explained above, and (2) the addition of the index n, which counts the number of β-reductions (of functions and `let`/`letrec` bindings) performed during evaluation. It is easy to check that $e \vdash a \stackrel{n}{\Rightarrow} v$ holds if and only if a reduces

$$\frac{e(n) = v}{e \vdash x_n \overset{0}{\Rightarrow} v} \qquad e \vdash c \overset{0}{\Rightarrow} c \qquad e \vdash \lambda.a \overset{0}{\Rightarrow} (\lambda.a)[e]$$

$$\frac{e \vdash a_1 \overset{i}{\Rightarrow} (\lambda.a')[e'] \quad e \vdash a_2 \overset{j}{\Rightarrow} v_2 \quad v_2.e' \vdash a' \overset{k}{\Rightarrow} v \quad n = i+j+k+1}{e \vdash a_1\ a_2 \overset{n}{\Rightarrow} v}$$

$$\frac{e \vdash a_1 \overset{i}{\Rightarrow} v_1 \quad v_1.e \vdash a_2 \overset{j}{\Rightarrow} v \quad n = i+j+1}{e \vdash \texttt{let}\ a_1\ \texttt{in}\ a_2 \overset{n}{\Rightarrow} v} \qquad \frac{v_1 = (\lambda.a_1)[v_1.e] \quad v_1.e \vdash a_2 \overset{i}{\Rightarrow} v \quad n = i+1}{e \vdash \texttt{letrec}\ \lambda.a_1\ \texttt{in}\ a_2 \overset{n}{\Rightarrow} v}$$

Fig. 1 Indexed natural semantics for the source language

$$\frac{f(n) = w}{f \vdash x_n \Rightarrow w} \qquad f \vdash c \Rightarrow c \qquad f \vdash \lambda^n.b \Rightarrow (\lambda^n.b)[f]$$

$$\frac{f \vdash b \Rightarrow (\lambda^n.b')[f'] \quad f \vdash b_i \Rightarrow w_i \text{ for } i = 1,\ldots,n \quad w_n \ldots w_1.f' \vdash b' \Rightarrow w}{f \vdash b(b_1,\ldots,b_n) \Rightarrow w}$$

$$\frac{f \vdash b_1 \Rightarrow w_1 \quad w_1.f \vdash b_2 \Rightarrow w}{f \vdash \texttt{let}\ b_1\ \texttt{in}\ b_2 \Rightarrow w} \qquad \frac{v_1 = (\lambda^n.b_1)[v_1.e] \quad v_1.e \vdash b_2 \Rightarrow v}{e \vdash \texttt{letrec}\ \lambda^n.b_1\ \texttt{in}\ b_2 \Rightarrow v}$$

Fig. 2 Natural semantics for the target language

to a value related to v by a sequence of exactly n reductions. As explained in section 3, this count of reductions n is crucial in defining logical relations between the source and target languages. The dynamic semantics for the target language is also given in natural semantics, as a predicate $f \vdash b \Rightarrow w$ ("in environment f, the term b evaluates to the value w"), following the inference rules of figure 2. The semantics is not instrumented to count the number of evaluation steps, since this is not needed to define logical relations. In the rule for function applications, note that the number n of arguments provided must exactly match the arity of the function closure being applied, otherwise execution goes wrong.

4.2 Relating values and computations between the source and target languages

Using the step-indexed approach summarized in section 3, we now define the following relations between terms or values of the source and target language that serve as the main invariants in our proof of semantic preservation:

- $v \overset{k}{\approx} w : \tau$ the values v and w are related at representation type τ for up to k computation steps
- $a \,/\, e \overset{k}{\approx} b \,/\, f : \tau$ the source term a in environment e is related to the target term b in environment f, at type τ, for up to k computation steps.
- $a \,/\, e \overset{k}{\approx} w : \tau$ the source term a in environment e is related to the target value w at type τ for up to k computation steps.

We abbreviate "related for up to k computation steps", or equivalently "indistinguishable in k steps", as "k-related". These three relations are specified by the following pseudo-inference rules.

$$c \overset{k}{\approx} c : \top \quad \text{(R1)}$$

$$\frac{\tau = \tau_1 \to \tau_2 \text{ or } \tau = \tau_1 = \tau_2 = \top \qquad \forall j, v, w,\ \ j < k \land v \overset{j}{\approx} w : \tau_1 \implies a \,/\, v.e \overset{j}{\approx} b \,/\, w.f : \tau_2}{(\lambda.a)[e] \overset{k}{\approx} (\lambda^1.b)[f] : \tau} \text{ (R2)}$$

$$\frac{n \geq 2 \qquad \forall j, v, w,\ \ j < k \land v \overset{j}{\approx} w : \tau_1 \implies a \,/\, v.e \overset{j}{\approx} (\lambda^{n-1}.b)[w.f] : (\tau_2, \ldots, \tau_n) \to \tau}{(\lambda.a)[e] \overset{k}{\approx} (\lambda^n.b)[f] : (\tau_1, \ldots, \tau_n) \to \tau} \text{ (R3)}$$

$$\frac{\forall j, v,\ \ j \leq k \land (e \vdash a \overset{j}{\Rightarrow} v) \implies \exists w,\ (f \vdash b \Rightarrow w) \land v \overset{k-j}{\approx} w : \tau}{a \,/\, e \overset{k}{\approx} b \,/\, f : \tau} \text{ (R4)}$$

$$\frac{\forall j, v,\ \ j \leq k \land (e \vdash a \overset{j}{\Rightarrow} v) \implies v \overset{k-j}{\approx} w : \tau}{a \,/\, e \overset{k}{\approx} w : \tau} \text{ (R5)}$$

The notion of k-relatedness between two values is specified by the first three rules (R1), (R2) and (R3). By rule (R1), a constant is related to itself at type $\top$. Rule (R2) relates a source closure $(\lambda.a)[e]$ with a unary target closure $(\lambda^1.b)[e]$. These closures are k-related if and only if for all $j < k$ and j-related argument values v, w, the computations corresponding to the applications of the closures, namely a in environment $v.e$ and b in environment $w.f$, are j-related. The notion of j-relatedness between two computations is defined by rule (R4) and explained below. Concerning types, rule (R2) offers two possibilities: either the closures are related at a unary function type $\tau_1 \to \tau_2$, in which case the argument values are related at type τ_1 and the result values at type τ_2; or the closures, the arguments and the results are all related at type $\top$. These two choices correspond closely to the transformation rules TAPP-$\top$ and TAPP-N (with $n = 1$) in section 2.3. Rule (R3) relates a source closure $(\lambda.a)[e]$ with a n-ary target closure $(\lambda^n.b)[f]$ at an uncurried function type $(\tau_1, \ldots, \tau_n) \to \tau$, where $n \geq 2$. As in rule (R2), we consider j-related argument values v and w, with $j < k$, and computations of the application of $(\lambda.a)[e]$ to v that run for $i \leq j$ steps. However, there is no computation to be performed on the target language side. Therefore, we relate the computation $a \,/\, v.e$ of the source language with the value $(\lambda^{n-1}.b)[w.f]$ or the target language, using the mixed relatedness relation defined by rule (R5) and explained below. This target value $(\lambda^{n-1}.b)[w.f]$ logically represents the application of $(\lambda^n.b)[f]$ to w. Such a partial application is not permitted by the semantics of the target language, but can easily be represented at the level of target values. Rule (R4) defines k-relatedness between two computations: whenever the source computation $a \,/\, e$ terminates on a value v in $j \leq k$ steps, the target computation $b \,/\, f$ must terminate on a value w that is $(k - j)$-related to v. Rule (R5) defines k-relatedness between a computation $a \,/\, e$ of the source language and a value w of the target language. As in rule (R4), we consider evaluations of $a \,/\, e$ that terminate on a value v in $j \leq k$ steps. Then, since there are no computations to perform on the target side, we just require that the result value v is related to the given value w in $k - j$ steps. Rules (R1) to (R5) do not constitute a proper inductive definition by an inference system, because the underlying inference operator is not monotone [3]. Nonetheless, it admits a smallest fixpoint:

Lemma 2 *There exist unique smallest predicates $v \overset{k}{\approx} w : \tau$ and $a \,/\, e \overset{k}{\approx} b \,/\, f : \tau$ and $a \,/\, e \overset{k}{\approx} w : \tau$ that satisfy rules (R1) to (R5).*

Proof Treating rules (R4) and (R5) as equivalences between their premises and their conclusions, and expanding these equivalences in rules (R2) and (R3), we see that $v \overset{k}{\approx} w : \tau$ is uniquely defined in terms of $v' \overset{j}{\approx} w' : \tau'$ for $j < k$, as mentioned in section 3. Existence and uniqueness then follow by Peano induction over k. In Coq, Peano induction — defining $f(n)$ in terms of $\{f(p) \mid p < n\}$ — is not a primitive notion: only structural recursion — defining $f(n+1)$ in terms of $f(n)$ — is primitive. We therefore use Noetherian induction on the well-founded ordering $<$ over the natural numbers [9, chap. 15]. The proof uses the axioms of function extensionality ($f = g$ if $\forall x,\ f(x) = g(x)$) and proof irrelevance (any two proofs of the same proposition are equal). These two axioms are not provable in Coq but consistent with Coq's predicative logic. □

An obvious but important property of k-relatedness is that it is decreasing in k: if two values are indistinguishable in k computation steps, they are *a fortiori* indistinguishable in $k' \leq k$ steps.

Lemma 3 *If $v \overset{k}{\approx} w : \tau$, then $v \overset{k'}{\approx} w : \tau$ for all $k' \leq k$.*

Proof Straightforward by definition of k-relatedness. □

The following lemma gives necessary and sufficient conditions for two trivial computations (of constants, abstractions and variables) to be related.

Lemma 4

1. $c \,/\, e \overset{k}{\approx} c' \,/\, f : \tau \iff c \overset{k}{\approx} c' : \tau \iff c = c' \wedge \tau = \top$
2. $\lambda.a \,/\, e \overset{k}{\approx} \lambda^n.b \,/\, f : \tau \iff (\lambda.a)[e] \overset{k}{\approx} (\lambda^n.b)[f] : \tau$
3. *If $e(n) = v$ and $f(m) = w$, then $x_n \,/\, e \overset{k}{\approx} x_m \,/\, f : \tau \iff v \overset{k}{\approx} w : \tau$.*

Proof Note that the source terms c, $\lambda.a$ and x_n evaluate in exactly zero steps to c, $(\lambda.a)[e]$ and v, respectively. The result then follows from (R4) and (R1). □

The following two lemmas show that related functions map related arguments to related results, as desired. Lemma 5 corresponds to unary "untyped" applications, where the functions are related at type $\top$. Lemma 6 corresponds to n-ary applications, curried on the source side, uncurried on the target side, where the functions are related at a n-ary function type.

Lemma 5 *If $a \,/\, e \overset{k}{\approx} b \,/\, f : \top$ and $a' \,/\, e \overset{k}{\approx} b' \,/\, f : \top$, then $a\ a' \,/\, e \overset{k}{\approx} b(b') \,/\, f : \top$.*

Proof Consider an evaluation $e \vdash a\ a' \overset{j}{\Rightarrow} v$ in $j \leq k$ steps. By inversion, we have $e \vdash a \overset{p}{\Rightarrow} (\lambda.a_1)[e1]$ and $e \vdash a' \overset{q}{\Rightarrow} v'$ and $v'.e_1 \vdash a_1 \overset{r}{\Rightarrow} v$ with $p+q+r+1 = j \leq k$. Exploiting the two hypotheses, it follows that there exist w_f and w' such that

$$\begin{array}{ll} f \vdash b \Rightarrow w_f & (\lambda.a_1)[e_1] \overset{k-p}{\approx} w_f : \top \qquad (1) \\ f \vdash b' \Rightarrow w' & v' \overset{k-q}{\approx} w' : \top \end{array}$$

From (1) and (R2) and (R4), it follows that w_f is of the form $(\lambda^1.b_1)[f_1]$, and moreover there exists w such that

$$w'.f_1 \vdash b_1 \Rightarrow w \qquad v \overset{k-j}{\approx} w : \top$$

Therefore, $f \vdash b(b') \Rightarrow w$ and the expected result follows. □

Lemma 6 *If $n > 0$ and $a \,/\, e \overset{k}{\approx} b \,/\, f : (\tau_1, \ldots, \tau_n) \to \tau$ and $a_i \,/\, e \overset{k}{\approx} b_i \,/\, f : \tau_i$ for $i = 1, \ldots, n$, then $a\ a_1\ \ldots\ a_n \,/\, e \overset{k}{\approx} b(b_1, \ldots, b_n) \,/\, f : \tau$.*

Proof As in the previous proof, consider an evaluation $e \vdash a\ a_1\ \ldots\ a_n \overset{j}{\Rightarrow} v$ in $j \leq k$ steps. By induction over n and inversion of this evaluation, we obtain values v_i, intermediate closures $(\lambda.a'_i)[e_i]$ and counts p_i, q_i such that

$$\begin{array}{ll}
e \vdash a_i \overset{q_i}{\Rightarrow} v_i & (i = 1, \ldots, n) \\
e \vdash a \overset{p_0}{\Rightarrow} (\lambda.a'_1)[e_1] & \\
v_{i-1}.e_{i-1} \vdash a'_{i-1} \overset{p_{i-1}}{\Rightarrow} (\lambda.a'_i)[e_i] & (i = 2, \ldots, n) \\
v_n.e_n \vdash a'_n \overset{p_n}{\Rightarrow} v & \\
j = p_0 + \cdots + p_n + q_1 + \cdots + q_n + (n+1) &
\end{array}$$

Exploiting the first hypothesis, there exists w_f such that

$$f \vdash b \Rightarrow w_f \qquad (\lambda.a'_1)[e_1] \overset{k-p_0}{\approx} w_f : (\tau_1, \ldots, \tau_n) \to \tau \qquad (1)$$

By (R3), it follows that w_f is of the form $(\lambda^n.b')[f']$. A second induction over n, exploiting (1) and the second hypothesis, shows that there exists a list of values $w_1, \ldots, w_n$ and a value w such that

$$\begin{array}{ll}
f \vdash b_i \Rightarrow w_i & (i = 1, \ldots, n) \\
w_n \ldots w_1.f' \vdash b' \Rightarrow w & \\
v \overset{k-j}{\approx} w : \tau &
\end{array}$$

The expected result follows. □

The definition of k-relatedness at uncurried function types $(\tau_1, \ldots, \tau_n) \to \tau$ with $n > 1$ (rule (R3)) is convenient to perform inductions over n, but somewhat mysterious. It is instructive to spell it out in the case $n = 2$. What does $(\lambda.a)[e] \overset{k}{\approx} (\lambda^2.b)[f] : (\tau_1, \tau_2) \to \tau$ entails? Ignoring step counts for the moment, this relation entails two guarantees. The first guarantee is a weak soundness property: applying $(\lambda.a)[e]$ to one suitable value v_1 (suitable meaning related to some value w_1) cannot produce a constant: the result, if it exists, must be a closure that we can further apply to a second argument. The second guarantee is that if we apply $(\lambda.a)[e]$ successively to two arguments v_1 and v_2, which are related to values w_1 and w_2 respectively, and obtain a result v, the application of $(\lambda^2.b)[f]$ to the two arguments w_1 and w_2 succeeds and returns a result w that is related to v. To formalize this discussion and generalize it from $n = 2$ to any $n > 1$, we introduce two predicates $\mathscr{M}$ and $\mathscr{R}$:

- $\mathscr{M}(k, \tau_1 \ldots \tau_n, v)$ captures the fact that the source value v can safely be applied in a curried fashion to n values matching types $\tau_1, \ldots, \tau_n$: none of the intermediate application will return a value other than a closure.
- $\mathscr{R}(k, \tau_1 \ldots \tau_n, v_1 \ldots v_n, w_1 \ldots w_n, v, v', p)$ says that the values v_i and w_i are pairwise related at types τ_i and for decreasing numbers of steps $j_n < \ldots < j_1 < k$. The value v' is the result of applying v successively to $v_1, \ldots, v_n$. The count p is the number of steps that were not consumed by the evaluation of this curried application and remain available to observe the result value v'.

These two predicates are defined by the following inference rules. (ε stands for the empty list.)

$$\mathscr{M}(k,\varepsilon,v)$$

$$\frac{\forall i,j,v,v',w,\ \ i<k \wedge j \le i \wedge v \overset{i}{\approx} w : \tau \wedge v.e \vdash a \overset{j}{\Rightarrow} v' \implies \mathscr{M}(i-j,\overrightarrow{\tau},v')}{\mathscr{M}(k,\tau.\overrightarrow{\tau},(\lambda.a)[e])}$$

$$\mathscr{R}(k,\varepsilon,\varepsilon,\varepsilon,v,v,k)$$

$$\frac{i<k \qquad j \le i \qquad v \overset{i}{\approx} w : \tau \qquad v.e \vdash a \overset{j}{\Rightarrow} v' \qquad \mathscr{R}(i-j,\overrightarrow{\tau},\overrightarrow{v},\overrightarrow{w},v',v'',p)}{\mathscr{R}(k,\tau.\overrightarrow{\tau},v.\overrightarrow{v},w.\overrightarrow{w},(\lambda.a)[e],v'',p)}$$

We can then give an alternate characterization of k-relatedness at function types, which will be useful in proving theorem 11 below.

Lemma 7 *$v \overset{k}{\approx} w : (\tau_1,\ldots,\tau_n) \to \tau$ holds if and only if*

1. *$n > 0$;*
2. *$w = (\lambda^n.b)[f]$ for some b and f;*
3. *$\mathscr{M}(k,\tau_1 \ldots \tau_n,v)$;*
4. *for all $\overrightarrow{v}$, $\overrightarrow{w}$, v' and p,*

$$\mathscr{R}(k,\tau_1 \ldots \tau_n,\overrightarrow{v},\overrightarrow{w},v,v',p) \implies \exists w',\ \ \overleftarrow{w}.f \vdash b \Rightarrow w' \wedge v' \overset{p}{\approx} w' : \tau.$$

(The notation $\overleftarrow{w}$ denotes the list of values $\overrightarrow{w}$ in reverse order.)

Proof By induction over n. Refer to the Coq development for a detailed proof. □

4.3 Correctness of subtyping and coercions

Having defined semantic correspondences between values of the source and target languages, we can now show that subtyping and coercions are sound with respect to these correspondences. For subtyping, we show that any pair of computations related at a type τ are also related at any supertype τ' of τ.

Lemma 8 *If $\tau <: \tau'$, then $a \,/\, e \overset{k}{\approx} b \,/\, f : \tau$ implies $a \,/\, e \overset{k}{\approx} b \,/\, f : \tau'$.*

Proof By (R4), it suffices to show that $v \overset{k}{\approx} w : \tau$ implies $v \overset{k}{\approx} w : \tau'$ for all values v, w. The proof proceeds by induction on the derivation of $\tau <: \tau'$ and case analysis on the last rule used. The cases of rules SREFL and STRANS are trivial. For rule SBASE, we have $\tau = \top \to \top$ and $\tau' = \top$. The result follows immediately from (R2). Finally, the case of rule SFUN follows from an induction on the number n of function parameters. □

We now consider the semantic effect of the $\texttt{curry}_n$ and $\texttt{uncurry}_n$ combinators. If the computations a in e and b in f are related at a n-ary function type $(\tau_1,\ldots,\tau_n) \to \tau$, it is the case that the computations a in e and $\texttt{curry}_n(b)$ in f are related at the corresponding curried function type.

Lemma 9 *If $a \,/\, e \overset{k}{\approx} b \,/\, f : (\tau_1,\ldots,\tau_n) \to \tau$ and $n > 0$, then*

$$a \,/\, e \overset{k}{\approx} \texttt{curry}_n(b) \,/\, f : \tau_1 \to \ldots \to \tau_n \to \tau$$

Proof Consider two closures $(\lambda^n.b_1)[f_1]$ and $(\lambda^n.b_2)[f_2]$ of the target language such that, for all arguments $w_1,\ldots,w_n$ and result w,

$$w_n \ldots w_1.f_1 \vdash b_1 \Rightarrow w \text{ implies } w_n \ldots w_1.f_2 \vdash b_2 \Rightarrow w \tag{1}$$

By induction on n, we first show that

$$v \overset{k}{\approx} (\lambda^n.b_1)[f_1] : (\tau_1,\ldots,\tau_n) \to \tau \text{ implies } v \overset{k}{\approx} (\underbrace{\lambda^1 \ldots \lambda^1}_{n \text{ times}}.b_2)[f_2] : \tau_1 \to \ldots \tau_n \to \tau \tag{2}$$

Exploiting the first hypothesis with (R4), we obtain that if a in e evaluates to v in $j \leq k$ steps, b in f evaluates to a closure $(\lambda^n.b_1)[f_1]$ such that $v \overset{k-j}{\approx} (\lambda^n.b_1)[f_1] : (\tau_1,\ldots,\tau_n) \to \tau$. The application $\texttt{curry}_n(b)$, then, evaluates to the closure $(\underbrace{\lambda^1 \ldots \lambda^1}_{n \text{ times}}.b_2)[f_2]$ with $b_2 = x_n(x_{n-1},\ldots,x_0)$ and $f_2 = (\lambda^n.b_1).f$. It is easy to check that the hypothesis (1) above is satisfied. The expected result follows from (2). □

Unsurprisingly, the $\texttt{uncurry}_n$ combinator enjoys a semantic property that is exactly symmetrical to that of $\texttt{curry}_n$.

Lemma 10 *If $n > 0$ and $a \,/\, e \overset{k}{\approx} b \,/\, f : \tau_1 \to \ldots \to \tau_n \to \tau$, then*

$$a \,/\, e \overset{k}{\approx} \texttt{uncurry}_n(b) \,/\, f : (\tau_1,\ldots,\tau_n) \to \tau.$$

Proof The proof is roughly similar to that of lemma 9, and we omit it. □

We can now prove the main semantic preservation result for coercions: if a target term b in f is related at type τ to a source term a in e, and the coercion of b from type τ to type τ' produces a term b', then a in e and b' in f are related at type τ'.

Theorem 11 *If $b : \tau \Longrightarrow b' : \tau'$, then $a \,/\, e \overset{k}{\approx} b \,/\, f : \tau$ implies $a \,/\, e \overset{k}{\approx} b' \,/\, f : \tau'$.*

Proof The proof proceeds by induction on the derivation of $b : \tau \Longrightarrow b' : \tau'$ and case analysis on the last rule used. The cases of rules CSUB, CCURRY and CUNCURRY correspond to lemmas 8, 9 and 10, respectively. The transitivity case (rule CTRANS) is trivial by induction hypothesis. The difficult case is that of rule CFUN, where the initial type is $(\tau_1,\ldots,\tau_n) \to \tau$ and the final type is $(\tau'_1,\ldots,\tau'_n) \to \tau'$. The premises of the rule are:

- $x_{n-i-1} : \tau'_i \Longrightarrow b_i : \tau_i$ for $i = 1,\ldots,n$
- $x_n(b_1,\ldots,b_n) : \tau \Longrightarrow b' : \tau'$

and the corresponding induction hypotheses hold. We use the alternate characterization of k-relatedness at function types provided by lemma 7, and show the following properties by induction over n and exploitation of the induction hypotheses for the argument types:

- If $\mathscr{M}(k, \tau_1 \ldots \tau_n, v)$, then $\mathscr{M}(k, \tau'_1 \ldots \tau'_n, v)$.
- If $\mathscr{R}(k, \tau'_1 \ldots \tau'_n, v_1 \ldots v_n, w_1 \ldots w_n, v, v', p)$, there exists values $w'_1,\ldots,w'_n$ such that $\mathscr{R}(k, \tau_1 \ldots \tau_n, v_1 \ldots v_n, w'_1 \ldots w'_n, v, v', p)$ and $w_n \ldots w_1.f \vdash b_i \Rightarrow w'_i$ for $i = 1,\ldots,n$.

The result then follows from these two properties and lemma 7. □

4.4 Correctness of the translation

We are now ready to prove the main correctness theorem: if the target term b is an uncurrying of the source term a at type τ in compilation environment Γ, then for any number of steps k, the computation a in e is k-related at τ to the computation b in f, provided the execution environments e and f are k-related at the compilation environment Γ. Relatedness between environments, written $e \overset{k}{\approx} f : \Gamma$, is defined straightforwardly as

- e, Γ and f have the same length n;
- $e(i) \overset{k}{\approx} f(i) : \Gamma(i)$ for all $i = 0, \ldots, n-1$.

Theorem 12 *If $\Gamma \vdash a \Longrightarrow b : \tau$ and $e \overset{k}{\approx} f : \Gamma$, then $a \,/\, e \overset{k}{\approx} b \,/\, f : \tau$.*

Most of the lemmas needed for the proof of theorem 12 have already been proved. To handle the case of function abstractions, we need the three additional lemmas below. (We write $\lambda^{-n-}\lambda$ for n successive function abstractions in the source language.)

Lemma 13 *Assume $\forall k, e, f,\ e \overset{k}{\approx} f : \tau_n \ldots \tau_1.\Gamma \implies a \,/\, e \overset{k}{\approx} b \,/\, f : \tau$. If $n > 0$ and $e \overset{k}{\approx} f : \Gamma$, then $(\lambda^{-n-}\lambda.a)[e] \overset{k}{\approx} (\lambda^n.b)[f] : (\tau_1, \ldots, \tau_n) \to \tau$.*

Proof If $n = 1$, the result follows from rule (R2). If $n > 1$, we first show that $\lambda^{-n-}\lambda.a \,/\, e \overset{k}{\approx} (\lambda^n.b)[f] : (\tau_1, \ldots, \tau_n) \to \tau$ by induction over n, then conclude using rule (R3). □

Lemma 14 *Assume $\forall k, e, f,\ e \overset{k}{\approx} f : \tau_n \ldots \tau_1.\tau_f.\Gamma \implies a \,/\, e \overset{k}{\approx} b \,/\, f : \tau$, where τ_f is the type $(\tau_1, \ldots, \tau_n) \to \tau$. Let $v = (\lambda^{-n-}\lambda.a)[v.e]$ and $w = (\lambda^n.b)[w.f]$ be recursive closures of a and b. If $n > 0$ and $e \overset{k}{\approx} f : \Gamma$, then $v \overset{k}{\approx} w : \tau_f$.*

Proof The proof is similar to that of lemma 13, except that in the base cases we need to assume that $v \overset{j}{\approx} w : \tau_f$ in order to prove that $v.e \overset{j}{\approx} w.f : \tau_f.\Gamma$. Fortunately, this assumption is needed only for $j < k$, therefore breaking circularity. The result thus follows by Peano induction over the number of steps k. □

Proof (of theorem 12) The proof of theorem 12 proceeds by induction on the derivation of $\Gamma \vdash a \Longrightarrow b : \tau$ and case analysis on the last rule used. Cases TVAR and TCST follow from lemma 4, parts 1 and 3. Case TABS follows from lemmas 13 and 4 (part 2). Lemmas 5 and 6 show cases TAPP-$\top$ and TAPP-N, respectively. The case of `let` bindings (TLET) is straightforward. For `letrec` bindings (TLETREC), we use lemma 14. Finally, the case of rule TCOERCE follows from lemma 11. □

As a corollary, we obtain the desired semantic preservation property for whole programs.

Corollary 15 *Let a be a closed term. If $\varepsilon \vdash a \Longrightarrow b : \tau$ and a evaluates a constant ($\varepsilon \vdash a \overset{n}{\Rightarrow} c$ for some n), then b evaluates to the same constant ($\varepsilon \vdash b \Rightarrow c$).*

Proof By theorem 12, $a \,/\, \varepsilon \overset{n}{\approx} b \,/\, \varepsilon : \tau$. Therefore, there exists a value w such that $\varepsilon \vdash b \Rightarrow w$ and $c \overset{0}{\approx} w : \tau$. The latter implies $w = c$ by definition of relatedness. □

5 Applications

This section presents two applications of the general framework developed in sections 2 and 4. The first application is a simple proof of correctness for a first-order uncurrying transformation similar to those used in GHC and OCaml. The second application is a generic translator *cum* validator that exploits the results of any external, untrusted static analysis for higher-order uncurrying after verifying their correctness.

5.1 First-order uncurrying

Following [14], the first-order uncurrying transformation presented here maintains a compile-time environment Θ associating *arity information* to free variables.

Arity information:	$\rho ::= K(n)$	function of known arity n
	$\mid U$	constant or function of unknown arity
Arity environments:	$\Theta ::= \rho_0 \dots \rho_n$	

The compile-time arity of a variable is either $K(n)$, meaning that the value of this variable is an uncurried function of n arguments, or U, meaning that nothing is known about this value and it is assumed to be fully curried. Owing to the first-order nature of the transformation, arguments and results of functions, even of known arity, are assumed to be fully curried and therefore have arity U.

The first-order uncurrying of a source term a in arity environment Θ, written $\mathscr{T}(\Theta, a)$, is the target term b defined by the following recursive equations.

$$
\begin{aligned}
\mathscr{T}(\Theta, x_p) &= \begin{cases} \mathtt{curry}_n(x_p) & \text{if } \Theta(p) = K(n) \text{ and } n > 1; \\ x_p & \text{otherwise} \end{cases} \\
\mathscr{T}(\Theta, c) &= c \\
\mathscr{T}(\Theta, \lambda.a) &= \lambda^1.\ \mathscr{T}(U.\Theta, a) \\
\mathscr{T}(\Theta, x_p\ a_1\ \dots\ a_n) &= x_p(\mathscr{T}(\Theta, a_1), \dots, \mathscr{T}(\Theta, a_n)) \text{ if } \Theta(p) = K(n) \\
\mathscr{T}(\Theta, a\ a') &= \mathscr{T}(\Theta, a)(\mathscr{T}(\Theta, a')) \qquad \text{otherwise} \\
\mathscr{T}(\Theta, \mathtt{let}\ \underbrace{\lambda \dots \lambda}_{n \text{ times}}.a\ \mathtt{in}\ a') &= \mathtt{let}\ \lambda^n.\mathscr{T}(\underbrace{U \dots U}_{n \text{ times}}.\Theta, a)\ \mathtt{in}\ \mathscr{T}(K(n).\Theta, a') \\
\mathscr{T}(\Theta, \mathtt{let}\ a\ \mathtt{in}\ a') &= \mathtt{let}\ \mathscr{T}(\Theta, a)\ \mathtt{in}\ \mathscr{T}(U.\Theta, a') \qquad \text{otherwise} \\
\mathscr{T}(\Theta, \mathtt{letrec}\ \underbrace{\lambda \dots \lambda}_{n \text{ times}}.a\ \mathtt{in}\ a') &= \mathtt{letrec}\ \lambda^n.\mathscr{T}(\underbrace{U \dots U}_{n \text{ times}}.K(n).\Theta, a)\ \mathtt{in}\ \mathscr{T}(K(n).\Theta, a')
\end{aligned}
$$

Uncurrying is performed along the lines mentioned in Introduction. Curried functions of n arguments bound by `let` or `letrec` are transformed into uncurried functions and associated with arity information $K(n)$ within their scope. Anonymous functions are never uncurried. The curried application of a variable x of arity information $K(n)$ to exactly n arguments is turned into an n-ary application. Any other use of a variable x of arity $K(n), n > 1$ is translated to $\mathtt{curry}_n(x)$, therefore recovering a fully-curried function.

Semantic preservation for this uncurrying transformation follows easily from the results of section 4. We first define the obvious translation from arity information and arity environments to representation types and representation environments:

$$\begin{aligned}[\![U]\!] &= \top \\ [\![K(n)]\!] &= \underbrace{(\top,\ldots,\top)}_{n \text{ times}} \to \top \\ [\![\Theta]\!] &= [\![\rho_0]\!]\ldots[\![\rho_n]\!] \text{ if } \theta = \rho_0\ldots\rho_n\end{aligned}$$

Theorem 16 *If $\mathscr{T}(\Theta,a) = b$, then $[\![\Theta]\!] \vdash a \Longrightarrow b : \top$.*

Proof The proof proceeds by structural induction over a and analysis on the corresponding case of the definition of $\mathscr{T}$. We show the non-obvious cases.

Case $a = x_p$. If $\Theta(p) = U$, the result follows from rule TVAR. If $\Theta(p) = K(1)$, rule TVAR shows $[\![\Theta]\!] \vdash x_p \Longrightarrow x_p : \top \to \top$. Since $\top \to \top <: \top$, an application of rule TCOERCE concludes. If $\Theta(p) = K(n)$ with $n > 1$, we also apply TCOERCE to go from $(\top,\ldots,\top) \to \top$ to $\top$, using rule CCURRY first to go to $\top \to \cdots \to \top$, then rule CSUB and lemma 1 to go to type $\top$.

Case $a = \lambda.a'$. Follows from the induction hypothesis, rule TABS (with $n = 1$), and TCOERCE with the trivial coercion from $\top \to \top$ to $\top$. □

Corollary 17 *Let a be a closed term. If $\mathscr{T}(\varepsilon,a) = b$ and a evaluates to a constant ($\varepsilon \vdash a \overset{n}{\Rightarrow} c$ for some n), then b evaluates to the same constant ($\varepsilon \vdash b \Rightarrow c$).*

Proof Follows from theorem 16 and corollary 15. □

5.2 Validating translation

Strategies for performing higher-order uncurrying have not been studied yet, and are likely to build on non-trivial static analyses. Instead of proving the correctness of one such strategy, we now develop and prove correct a generic translator that exploits the results of any external strategy. These results are not trusted: instead, the translator explicitly validates them and raises a compile-time error if they are not correct. This approach is inspired by translation validation [31,29] and proof-carrying code [28,6]. As remarked by Tristan and Leroy [36], a strength of this approach is that it suffices to prove the soundness of the generic validating translation (if it does not fail, the produced code is correct): the external strategy need not be proved correct, and we can experiment with several such strategies without having to re-do any formal proofs.

The results of the external strategy for uncurrying are transmitted to the validating translation as source terms annotated with representation types and explicit coercions. These annotations are underlined in the grammar below.

Annotated source terms:

$$
\begin{array}{rll}
A ::= & x_n & \\
\mid & c & \\
\mid & \lambda^n : \underline{\tau_1 \dots \tau_n}.\ A & \text{curried function of } n \text{ arguments} \\
\mid & A\ A' & \\
\mid & \texttt{let}\ A\ \texttt{in}\ A' & \\
\mid & \texttt{letrec}\ \lambda^n\_.A\colon \underline{(\tau_1 \dots \tau_n) \to \tau}\ \texttt{in}\ A' & \\
\mid & (A : \underline{\tau}) & \text{coercion to representation type } \tau
\end{array}
$$

The external uncurrying strategy is responsible for annotating function parameters and recursive function declarations with their desired representation types. To further reflect the decisions taken by the strategy, curried abstractions are syntactically delimited in annotated terms.

Example 18 Consider the following unannotated source term (a simplified version of the `map2` example from section 1.4):

$$\texttt{let app2} = \lambda f.\lambda y.\lambda z.\ f\ y\ z\ \texttt{in app2}\ (\lambda y.\lambda z.y+z)\ 0\ 1$$

The uncurrying strategy can choose to uncurry `app2`, its f parameter, and the $\lambda y.\lambda z.y+z$ anonymous function, resulting in the following annotated term:

$$
\begin{array}{l}
\texttt{let app2} = \lambda^3(f : (\top, \top) \to \top)(y : \top)(z : \top).\ f\ y\ z \\
\texttt{in app2}\ (\lambda^2(y : \top)(z : \top).\ y+z)\ 0\ 1
\end{array}
$$

If, on the other hand, the strategy chooses to leave the f parameter curried but still uncurry the anonymous function, it must insert an explicit coercion at point of call, obtaining the following annotated term:

$$
\begin{array}{l}
\texttt{let app2} = \lambda^3(f : \top \to \top)(y : \top)(z : \top).\ f\ y\ z \\
\texttt{in app2}\ (\lambda^2(y : \top)(z : \top).\ y+z : \top \to \top)\ 0\ 1
\end{array}
$$

□

Annotated source terms support partial applications and over-applications just like source terms: for a curried function $\lambda^n : \tau_1 \dots \tau_n.\ A$, there is no requirement that it receives n arguments at once in a single curried application. Indeed, the semantics of annotated source terms is exactly that of source terms after erasing the annotations. Erasure of annotations, written $A{\downarrow}$, produces a source term defined as follows:

$$
\begin{aligned}
x_n{\downarrow} &= x_n \\
c{\downarrow} &= c \\
(\lambda^n : \tau_1 \dots \tau_n.\ A){\downarrow} &= \underbrace{\lambda \dots \lambda}_{n \text{ times}}.\ A{\downarrow} \\
(A\ A'){\downarrow} &= A{\downarrow}\ A'{\downarrow} \\
(\texttt{let}\ A\ \texttt{in}\ A'){\downarrow} &= \texttt{let}\ A{\downarrow}\ \texttt{in}\ A'{\downarrow} \\
(\texttt{letrec}\ \lambda^n.A : (\tau_1 \dots \tau_n) \to \tau\ \texttt{in}\ A'){\downarrow} &= \texttt{letrec}\ \underbrace{\lambda \dots \lambda}_{n \text{ times}}.\ A{\downarrow}\ \texttt{in}\ A'{\downarrow} \\
(A : \tau){\downarrow} &= A{\downarrow}
\end{aligned}
$$

$$\mathscr{S}(\top,\top) = \texttt{true} \tag{1}$$
$$\mathscr{S}(\top \to \tau,\top) = \mathscr{S}(\tau,\top) \tag{2}$$
$$\mathscr{S}((\tau_1,\ldots,\tau_n) \to \tau,(\tau'_1,\ldots,\tau'_m) \to \tau') = n = m \wedge \mathscr{S}(\tau'_1,\tau_1) \wedge \ldots \wedge \mathscr{S}(\tau'_n,\tau_n) \wedge \mathscr{S}(\tau,\tau') \tag{3}$$
$$\mathscr{S}(\tau,\tau') = \texttt{false} \text{ otherwise} \tag{4}$$

$$\mathscr{C}(b,\tau,\tau') = \lfloor b \rfloor \text{ if } \mathscr{S}(\tau,\tau') \tag{5}$$
$$\mathscr{C}(b,(\tau_1,\ldots,\tau_n) \to \tau,\top) = \mathscr{C}(b,(\tau_1,\ldots,\tau_n) \to \tau,\top \to \top) \tag{6}$$
$$\mathscr{C}(b,(\tau_1,\ldots,\tau_n) \to \tau,(\tau'_1,\ldots,\tau'_n) \to \tau') = \lfloor \texttt{let } b \texttt{ in } \lambda^n.b' \rfloor \tag{7}$$
$$\text{if } \mathscr{C}(x_{n-i-1},\tau'_i,\tau_i) = \lfloor b_i \rfloor \text{ for } i = 1,\ldots,n$$
$$\text{and } \mathscr{C}(x_n(b_1,\ldots,b_n),\tau,\tau') = \lfloor b' \rfloor$$
$$\mathscr{C}(b,(\tau_1,\ldots,\tau_n) \to \tau,\tau'_1 \to \tau') = \mathscr{C}(\texttt{curry}_n(b),\tau_1 \to \cdots \to \tau_n \to \tau,\tau'_1 \to \tau') \tag{8}$$
$$\text{if } n > 1$$
$$\mathscr{C}(b,\tau_1 \to \tau,(\tau'_1,\ldots,\tau'_n) \to \tau') = \lfloor \texttt{uncurry}_n(b') \rfloor \tag{9}$$
$$\text{if } \mathscr{C}(b,\tau_1 \to \tau,\tau'_1 \to \cdots \to \tau'_n \to \tau') = \lfloor b' \rfloor$$
$$\text{and } n > 1$$
$$\mathscr{C}(b,(\tau_1,\ldots,\tau_n) \to \tau,(\tau'_1,\ldots,\tau'_m) \to \tau') = \lfloor \texttt{uncurry}_m(b') \rfloor \tag{10}$$
$$\text{if } \mathscr{C}(\texttt{curry}_n(b),\ \tau_1 \to \cdots \to \tau_n \to \tau,\ \tau'_1 \to \cdots \to \tau'_m \to \tau') = \lfloor b' \rfloor$$
$$\text{and } n > 1 \text{ and } m > 1 \text{ and } n \neq m$$
$$\mathscr{C}(b,\tau,\tau') = \emptyset \text{ otherwise} \tag{11}$$

Fig. 3 Construction of coercions

The validating translation that we consider in this section takes as input an annotated term A and a representation environment Γ. It is defined in figure 4 as the function $\mathscr{V}(\Gamma,A)$. It checks that the annotations on A are consistent and, if so, returns the representation type τ for A and its translation b (a target term). Option types are used to represent failure: the result of $\mathscr{V}(\Gamma,A)$ is $\lfloor \tau,b \rfloor$ (read: "some τ and b") in case of success and $\emptyset$ (read: "none") in case of failure. Three auxiliary functions, defined in figures 3 and 4, are used:

- $\mathscr{S}(\tau,\tau') = \texttt{false} \mid \texttt{true}$ decides whether τ is a subtype of τ';
- $\mathscr{C}(b,\tau,\tau') = \emptyset \mid \lfloor b' \rfloor$ builds the coercion b' of b from type τ to type τ';
- $\mathscr{A}(\tau,b,(\tau_1,b_1)\ldots(\tau_n,b_n)) = \emptyset \mid \lfloor \tau',b' \rfloor$ type-checks the application of $b : \tau$ to the arguments $b_i : \tau_i$ and returns the uncurried application b' and its type τ'.

The validating translation $\mathscr{V}$ resembles a type-checker for the pseudo type system of section 2.3, augmented with the generation of the translation b. The cases of the definition of $\mathscr{V}$ (figure 4) are straightforward. For applications (case 16), it considers maximal curried applications $A\ A_1\ \cdots\ A_n$ where A is not an application, recursively validates and translates $A,A_1,\ldots,A_n$ to $b,b_1,\ldots,b_n$, then checks (using function $\mathscr{A}$) that the types of the b_i agree with that of b, and groups the b_i into a sequence of k-ary applications as dictated by the type of the function b.

The auxiliary function $\mathscr{S}(\tau,\tau')$ in figure 3 is a decision procedure for the subtyping relation. Its definition can be understood as a syntax-directed variant of the rules defining the $<:$ predicate, where transitivity is used only where strictly necessary.

The most delicate algorithm is function $\mathscr{C}(b,\tau,\tau')$, which determines whether there exists a coercion from type τ to type τ', and if so applies it to b (figure 3). The algorithm

$$
\begin{aligned}
\mathscr{V}(\Gamma, x_n) &= \lfloor \tau, x_n \rfloor \text{ if } \Gamma(n) = \tau && (12)\\
\mathscr{V}(\Gamma, c) &= \lfloor \top, c \rfloor && (13)\\
\mathscr{V}(\Gamma, \lambda^n : \tau_1 \ldots \tau_n.\, A) &= \lfloor (\tau_1, \ldots, \tau_n) \to \tau,\ \lambda^n.\, b \rfloor && (14)\\
&\quad \text{if } n > 0 \text{ and } \mathscr{V}(\tau_n \ldots \tau_1.\Gamma, A) = \lfloor \tau, b \rfloor && (15)\\
\mathscr{V}(\Gamma, A\ A_1\ \cdots\ A_n) &= \mathscr{A}(\tau, b, (\tau_1, b_1) \ldots (\tau_n, b_n)) && (16)\\
&\quad \text{if } A \text{ is not an application}\\
&\quad \text{and } \mathscr{V}(\Gamma, A) = \lfloor \tau, b \rfloor\\
&\quad \text{and } \mathscr{V}(\Gamma, A_i) = \lfloor \tau_i, b_i \rfloor \text{ for } i = 1, \ldots, n\\
\mathscr{V}(\Gamma, (\texttt{let}\ A\ \texttt{in}\ A')) &= \lfloor \tau', (\texttt{let}\ b\ \texttt{in}\ b') \rfloor && (17)\\
&\quad \text{if } \mathscr{V}(\Gamma, A) = \lfloor \tau, b \rfloor\\
&\quad \text{and } \mathscr{V}(\tau.\Gamma, A') = \lfloor \tau', b' \rfloor\\
\mathscr{V}(\Gamma, (\texttt{letrec}\ \lambda^n.A : (\tau_1, \ldots, \tau_n) \to \tau\ \texttt{in}\ A')) &= \lfloor \tau', (\texttt{letrec}\ \lambda^n.b\ \texttt{in}\ b') \rfloor && (18)\\
&\quad \text{if } \mathscr{V}(\tau_n \ldots \tau_1.((\tau_1, \ldots, \tau_n) \to \tau).\Gamma, A) = \lfloor \tau, b \rfloor\\
&\quad \text{and } \mathscr{V}(((\tau_1, \ldots, \tau_n) \to \tau).\Gamma, A') = \lfloor \tau', b' \rfloor\\
\mathscr{V}(\Gamma, (A : \tau)) &= \lfloor \tau, b \rfloor && (19)\\
&\quad \text{if } \mathscr{V}(\Gamma, A) = \lfloor \tau', b' \rfloor\\
&\quad \text{and } \mathscr{C}(b', \tau', \tau) = \lfloor b \rfloor\\
\mathscr{V}(\Gamma, A) &= \emptyset \text{ otherwise} && (20)
\end{aligned}
$$

$$
\begin{aligned}
\mathscr{A}(\tau, b, \varepsilon) &= \lfloor \tau, b \rfloor && (21)\\
\mathscr{A}(\top, b, (\top, b_1).L) &= \mathscr{A}(\top, b(b_1), L) && (22)\\
\mathscr{A}((\tau_1, \ldots, \tau_n) \to \tau,\ b,\ (\tau_1, b_1) \ldots (\tau_n, b_n).L) &= \mathscr{A}(\tau, b(b_1, \ldots, b_n), L) && (23)\\
\mathscr{A}(\tau, b, L) &= \emptyset \text{ otherwise} && (24)
\end{aligned}
$$

Fig. 4 The validating translation

compares the shapes of τ and τ'. In case 6, coercing from a function type to $\top$ reduces to coercing from this function type to $\top \to \top$. Coercion between two function types is handled by cases 7 to 10, depending on the arities of the function types. If the arities match (case 7), the argument and result types are recursively coerced and combined following rule CFUN. Otherwise, the `curry` and `uncurry` combinators are applied as appropriate to reduce functions of 2 or more arguments to unary functions, and coercion is recursively attempted between the resulting unary function types.

Example 19 Consider again the annotated terms given in example 18. Executed on the annotated term

$$
\begin{aligned}
&\texttt{let app2} = \lambda^3(f : (\top, \top) \to \top)(y : \top)(z : \top).\ f\ y\ z\\
&\texttt{in app2}\ (\lambda^2(y : \top)(z : \top).\ y + z)\ 0\ 1
\end{aligned}
$$

and the empty translation environment, the validating translation succeeds and produces the uncurried term

$$\texttt{let app2} = \lambda(f, y, z).\ f(y, z)\ \texttt{in app2}\ ((\lambda(y, z).\ y + z), 0, 1)$$

With the other annotation considered,

$$
\begin{aligned}
&\texttt{let app2} = \lambda^3(f : \top \to \top)(y : \top)(z : \top).\ f\ y\ z\\
&\texttt{in app2}\ (\lambda^2(y : \top)(z : \top).\ y + z : \top \to \top)\ 0\ 1
\end{aligned}
$$

a $\texttt{curry}_2$ coercion from $\top \times \top \to \top$ to $\top \to \top$ is introduced, and we obtain

$$\texttt{let app2} = \lambda(f,y,z)\ f(y)(z)\ \texttt{in app2}\ (\texttt{curry}_2(\lambda(y,z).\ y+z),0,1)$$

Finally, if given the following incorrectly-annotated source term

$$\begin{array}{l}\texttt{let app2} = \lambda^3(f:\top\to\top)(y:\top)(z:\top).\ f\ y\ z\\ \texttt{in app2}\ (\lambda^2(y:\top)(z:\top).\ y+z)\ 0\ 1\end{array}$$

the translation reports an error because the type of the first effective argument to app2, namely $\top \times \top \to \top$, does not match the expected type for its first argument, $\top \to \top$. Case (23) of figure 4 therefore does not apply. □

Concerning termination, $\mathscr{V}$ and $\mathscr{A}$ are syntax-directed and therefore terminate trivially. In the case of $\mathscr{S}(\tau,\tau')$, the sum of the sizes of τ and τ' (i.e. the number of $\to$ constructors in both types) decreases strictly at each recursive call. Termination of $\mathscr{C}(b,\tau,\tau')$ is more elusive. Define the following positive measure for the ordered pair (τ,τ'):

$$\|(\tau,\tau')\| = \|\tau\| + \|\tau'\| + \begin{cases}3 & \text{if } \tau' = \top;\\ 0 & \text{otherwise.}\end{cases}$$

Here, $\|\tau\|$ is the following nonstandard size for a type τ, which gives additional weight to function types of arity 2 or more:

$$\begin{aligned}\|\top\| &= 1\\ \|\tau_1 \to \tau\| &= 1 + \|\tau\| + \|\tau_1\|\\ \|(\tau_1,\tau_2,\ldots,\tau_n) \to \tau\| &= 1 + \|\tau\| + \|\tau_1\| + (2+\|\tau_2\|) + \cdots + (2+\|\tau_n\|)\end{aligned}$$

With these unnatural definitions, we have $\|(\tau_1,\tau_2,\ldots,\tau_n)\to\tau\| > \|\tau_1\to\cdots\to\tau_n\to\tau\|$ if $n \geq 2$. Moreover, $\|(\tau,\top)\| > \|(\tau,\top\to\top)\|$. This guarantees that the measure of the two type arguments of $\mathscr{C}$ strictly decreases at each recursive call, ensuring termination.

We now prove the soundness of the validating translation: if $\mathscr{V}(\Gamma,A)$ succeeds and returns $\lfloor\tau,b\rfloor$, then b is a correct translation of the erasure $A{\downarrow}$ at type τ according to the specification of section 2.3. It follows that b preserves the semantics of $A{\downarrow}$.

Theorem 20 *The validating translation is sound:*

1. *If $\mathscr{S}(\tau,\tau') = \texttt{true}$, then $\tau <: \tau'$ holds.*
2. *If $\mathscr{C}(b,\tau,\tau') = \lfloor b'\rfloor$, then $b : \tau \Longrightarrow b' : \tau'$ holds.*
3. *If $\mathscr{V}(\Gamma,A) = \lfloor\tau,b\rfloor$, then $\Gamma \vdash A{\downarrow} \Longrightarrow b : \tau$.*

Proof The three parts are proved successively by induction on the recursion structure of the corresponding algorithms. For parts 1 and 2, each case of the definition of $\mathscr{S}$ and $\mathscr{C}$ corresponds to a combination of inference rules defining the subtyping and coercion predicates. For example, in case 2 of $\mathscr{S}$, the induction hypothesis shows that $\tau <: \top$, from which we can build the following subtyping derivation:

$$\dfrac{\dfrac{\dfrac{}{\top <: \top}\ (\textsc{Sref})\quad \dfrac{}{\tau <: \top}}{\top\to\tau <: \top\to\top}\ (\textsc{Sfun})\qquad \dfrac{}{\top\to\top <: \top}\ (\textsc{Sbase})}{\top\to\tau <: \top}\ (\textsc{Strans})$$

For part 3, all cases are straightforward except the case of applications. We show that $\mathscr{A}(\tau, b, (\tau_1, b_1) \ldots (\tau_n, b_n)) = \lfloor \tau', b' \rfloor$ and $\Gamma \vdash a \Longrightarrow b : \tau$ and $\Gamma \vdash a_i \Longrightarrow b_i : \tau_i$ for all i imply $\Gamma \vdash a\ a_1\ \cdots\ a_n \Longrightarrow b' : \tau'$, by induction over n, and conclude. □

Corollary 21 *Let A be a closed, annotated term. If $\mathscr{V}(\varepsilon, A) = \lfloor \tau, b \rfloor$ and the erasure $A{\downarrow}$ evaluates to a constant ($\varepsilon \vdash A{\downarrow} \stackrel{n}{\Rightarrow} c$ for some n), then b evaluates to the same constant ($\varepsilon \vdash b \Rightarrow c$).*

Proof By part 3 of lemma 20 and corollary 15. □

Soundness is the essential property of a validating translation such as the $\mathscr{V}$ function. However, some evidence that it works well in practice is also needed: for instance, a validating translation that always fails is correct but useless. In the case of $\mathscr{V}$, we provide evidence of usability in the form of a completeness result with respect to the specification from section 2.3: if there exists a valid uncurrying b of the unannotated source term a according to this specification, or more formally if $\Gamma \vdash a \Longrightarrow b : \tau$ holds, then there exists an annotation A of a such that $\mathscr{V}(\Gamma, A)$ does not fail; indeed, there exists an uncurrying b' such that $\mathscr{V}(\Gamma, A) = \lfloor \tau, b' \rfloor$. In other words, it is always possible for the external uncurrying strategy to annotate the source term a in a way that will be accepted by the validating translation. Note that, in general, the uncurrying b' produced by the validating translation is not identical to the uncurrying b produced by the specification. However, both uncurryings are semantically correct.

Theorem 22 *The validating translation is complete:*

1. *If $\tau <: \tau'$, then $\mathscr{S}(\tau, \tau') = \mathtt{true}$.*
2. *If $b : \tau \Longrightarrow b' : \tau'$, then for all terms b'', $\mathscr{C}(b'', \tau, \tau') \neq \emptyset$.*
3. *If $\Gamma \vdash a \Longrightarrow b : \tau$, there exists an annotated term A and a target term b' such that $A{\downarrow} = a$ and $\mathscr{V}(\Gamma, A) = \lfloor \tau, b' \rfloor$.*

Proof The proof of part 3 is a simple induction on the derivation of $\Gamma \vdash a \Longrightarrow b : \tau$. To construct the annotated term A, we insert coercions in two places: 1- whenever the original derivation uses rule TCOERCE, and 2- when dealing with a curried application $a\ a_1\ \cdots\ a_n$, in which case we insert a trivial coercion around the annotation of a, so that the recognition of maximal curried applications performed by $\mathscr{V}$ stops at a and does not try to decompose it further into applications.

The proof of part 1 is an induction on the derivation of $\tau <: \tau'$. We first need to show that $\mathscr{S}$, viewed as a relation, is reflexive and transitive, which is easy to do by structural induction over types.

Part 2 is the most difficult part of the proof. We first axiomatize the changes in types performed by the various cases of the $\mathscr{C}$ function, using the inference rules given in figure 5. These rules define the predicate $\tau \Longrightarrow \tau'$, read: "$\tau$ can be coerced to τ' according to the algorithm $\mathscr{C}$". A simple induction over the derivation of $\tau \Longrightarrow \tau'$ shows that $\tau \Longrightarrow \tau'$ implies $\forall b, \mathscr{C}(b, \tau, \tau') \neq \emptyset$ as expected. It remains to show that $b : \tau \Longrightarrow b' : \tau'$ implies $\tau \Longrightarrow \tau'$. As in part 1, this is proved by induction over the derivation of $b : \tau \Longrightarrow b' : \tau'$. The only difficult case is rule CTRANS, for which we need to prove some rather delicate transitivity properties:

(4) $\mathscr{S}(\tau_1, \tau_2) = \mathtt{true}$ and $\tau_2 \Longrightarrow \tau_3$ imply $\tau_1 \Longrightarrow \tau_3$.
(5) $\tau_1 \Longrightarrow \tau_2$ and $\mathscr{S}(\tau_2, \tau_3) = \mathtt{true}$ imply $\tau_1 \Longrightarrow \tau_3$.
(6) $\tau_1 \Longrightarrow \tau_2$ and $\tau_2 \Longrightarrow \tau_3$ imply $\tau_1 \Longrightarrow \tau_3$.

$$\frac{\mathscr{S}(\tau,\tau') = \texttt{true}}{\tau \Longrightarrow \tau'} \ (5) \qquad \frac{(\tau_1,\ldots,\tau_n) \to \tau \Longrightarrow \top \to \top}{(\tau_1,\ldots,\tau_n) \to \tau \Longrightarrow \top} \ (6)$$

$$\frac{\tau_i' \Longrightarrow \tau_i \text{ for } i = 1,\ldots,n \qquad \tau \Longrightarrow \tau'}{(\tau_1,\ldots,\tau_n) \to \tau \Longrightarrow (\tau_1',\ldots,\tau_n') \to \tau'} \ (7) \qquad \frac{n > 1 \qquad \tau_1 \to \cdots \to \tau_n \to \tau \Longrightarrow \tau_1' \to \tau'}{(\tau_1,\ldots,\tau_n) \to \tau \Longrightarrow \tau_1' \to \tau'} \ (8)$$

$$\frac{n > 1 \qquad \tau_1 \to \tau \Longrightarrow \tau_1' \to \cdots \to \tau_n' \to \tau'}{\tau_1 \to \tau \Longrightarrow (\tau_1',\ldots,\tau_n') \to \tau'} \ (9)$$

$$\frac{n > 1 \qquad m > 1 \qquad n \neq m \qquad \tau_1 \to \cdots \to \tau_n \to \tau \Longrightarrow \tau_1' \to \cdots \to \tau_m' \to \tau'}{(\tau_1,\ldots,\tau_n) \to \tau \Longrightarrow (\tau_1',\ldots,\tau_m') \to \tau'} \ (10)$$

Fig. 5 Characterization of the pairs of types $\tau \Longrightarrow \tau'$ such that algorithm $\mathscr{C}$ succeeds in producing a coercion from τ to τ'. Each inference rule is annotated by the number of the corresponding case of $\mathscr{C}$ in figure 3.

Parts 4 and 5 are proved by induction on the derivation of $\tau_1 \Longrightarrow \tau_2$. Part 6 is proved by Peano induction on the sum of measures $\|(\tau_1,\tau_2)\| + \|(\tau_2,\tau_3)\|$ and a tedious case analysis on the three types involved. □

As a final note, the Coq formalization models algorithms $\mathscr{S}$, $\mathscr{C}$ and $\mathscr{V}$ as computable functions. These functions can be executed directly within Coq (this is how we obtained the examples of coercions shown in table 1), or automatically translated to efficient OCaml code via Coq's extraction facility [26].

6 Semantic preservation for non-terminating programs

The semantic preservation result proved in section 4 shows that the uncurrying transformation is correct, but only for terminating programs: if the original program diverges, nothing is known about the behavior of its possible uncurryings; they could diverge as well, as expected, but also terminate, or go wrong. This is not a problem in the application scenario that we initially envisioned, namely uncurrying programs extracted from Coq specifications, since the type system of Coq guarantees that all extracted programs are strongly normalizing. Nonetheless, to perform higher-order uncurrying in a general-purpose compiler for a functional language, it would be desirable to show that uncurrying preserves non-termination as well. This raises two difficulties that we now discuss.

6.1 Step-indexed logical relations and divergence

As defined in sections 3 and 4.2 and in the original work of Appel and McAllester [7], the relation $a \,/\, e \overset{k}{\approx} b \,/\, f : \tau$ only states that if $a \,/\, e$ terminates in some number $j \leq k$ of steps, then $b \,/\, f$ terminates in any number of steps. If the computation $a \,/\, e$ diverges, the relation holds vacuously and provides no guarantees on the behavior of the computation $b \,/\, f$. To prove semantic preservation for non-terminating source programs, we would need to strengthen the definition of step-indexed logical relations so that if $a \,/\, e \overset{k}{\approx} b \,/\, f : \tau$ holds for all k and $a \,/\, e$ diverges, then $b \,/\, f$ diverges as well.

Reasoning directly over diverging executions in a step-indexed approach is difficult, because we lack the notion of "number of execution steps" to build inductive definitions and

proofs. However, we can characterize divergence negatively: a computation diverges if, for all k, it neither terminates nor goes wrong within k steps. This characterization is amenable to reasoning by induction over the step count k.

Ahmed [5] shows how this approach leads to the definition of step-indexed relations that entail observational equivalence. Working in the context of a single, strongly-typed functional language equipped with reduction semantics, she first defines the step-indexed relation $a \leq_k a' : \tau$ as "a and a' have type τ and moreover if a evaluates to a value v in $j \leq k$ steps, then a' evaluates in any number of steps to some value v' that is related to v in $k-j$ steps". Then, Ahmed defines the equivalence of two computations $a \cong a' : \tau$ as $(\forall k,\ a \leq_k a' : \tau) \wedge (\forall k,\ a' \leq_k a : \tau)$. If $a \cong a' : \tau$, it is obvious that a evaluates to a value if and only if a' does. Since the terms a and a' are well-typed in a sound type system, divergence is equivalent to not evaluating to a value: by soundness of the type system, a well-typed term cannot go wrong. Therefore, $a \cong a' : \tau$ also implies that a diverges if and only if a' diverges.

This line of reasoning does not immediately apply to our weakly-typed setting, since the type soundness theorem does not hold: a term that does not evaluate could go wrong instead of diverging. Ahmed (personal communication, january 2009) suggested to strengthen the step-indexed relation between computations (rule R4) as follows:

$$\frac{\begin{array}{c}\forall j, v,\ \ j \leq k \wedge (e \vdash a \stackrel{j}{\Rightarrow} v) \implies \exists w,\ (f \vdash b \Rightarrow w) \wedge v \stackrel{k-j}{\approx} w : \tau \\ \forall j, w,\ \ j \leq k \wedge (f \vdash b \stackrel{j}{\Rightarrow} w) \implies \exists v,\ (e \vdash a \Rightarrow v) \wedge v \stackrel{k-j}{\approx} w : \tau \\ \forall j,\ \ j \leq k \wedge (f \vdash b \stackrel{j}{\Rightarrow} \texttt{err}) \implies (e \vdash a \Rightarrow \texttt{err})\end{array}}{a \,/\, e \stackrel{k}{\approx} b \,/\, f : \tau} \quad (\text{R4}')$$

In this definition, $e \vdash a \stackrel{j}{\Rightarrow} \texttt{err}$ means that the computation $a \,/\, e$ goes wrong in j steps, and $e \vdash a \Rightarrow \texttt{err}$ means that it goes wrong in any number of steps. With this definition, if $\forall k,\ a \,/\, e \stackrel{k}{\approx} b \,/\, f : \tau$, we know that if the computation $a \,/\, e$ terminates on a value v, then $b \,/\, f$ terminates on a related value, as before. Moreover, since "terminating on a value", "going wrong" and "diverging" are mutually exclusive behaviors, it is easy to show that if $a \,/\, e$ diverges, so does $b \,/\, f$.

We have not worked out this approach in full details, but believe that it provides an appropriate way to show that program transformations preserve divergence.

6.2 Higher-order uncurrying and divergence

The second difficulty encountered when trying to prove that higher-order uncurrying preserves non-terminating behaviors is that the expected result is false: there exists diverging terms a such that some of their legal uncurryings b terminate. Let ω be a diverging term and consider the following source term a:

$$\texttt{let}\ f = (\lambda x.\ \texttt{let}\ z = \omega\ \texttt{in}\ \lambda y.\ x + y)\ \texttt{in}\ f\ 1$$

This term diverges when evaluated, since the partial application $f\ 1$ causes ω to be evaluated. First-order uncurrying (rule TABS alone) cannot uncurry the function bound to f, because it is not of the shape $\lambda x.\lambda y \ldots$. However, higher-order uncurrying (rule TCOERCE) enables us to produce the following uncurried target term b:

$$\texttt{let}\ f = \texttt{curry}_2(\texttt{uncurry}_2(\lambda x.\ \texttt{let}\ z = \omega\ \texttt{in}\ \lambda y.\ x + y))\ \texttt{in}\ f\ 1$$

Since the composition $\texttt{curry}_2 \circ \texttt{uncurry}_2$ is equivalent to two steps of η-expansion, f is bound to a closure of the form $(\lambda x.\lambda y \ldots)[e]$. The partial application $f\ 1$ therefore does not trigger the evaluation of ω and terminates instead.

An optimization that can turn diverging program into terminating ones is not necessarily bad. However, an optimization that performs uncontrolled η-expansions is generally undesirable: besides non-termination behavior, η-expansion can change time and space complexity, as well as observable effects in functional languages such as ML. To avoid this, we need to restrict uses of the $\texttt{uncurry}_n$ combinators in coercions. (The $\texttt{curry}_n$ combinators alone cannot create η-expansion problems.) One possibility is to enrich representation types τ with trivial function types $(\tau_1, \ldots, \tau_n) \xrightarrow{\emptyset} \tau$. These trivial function types are assigned to functions $\lambda x.a$ that perform no significant computations before returning their results, typically because the body a is a syntactic value such as a λ-abstraction. For instance, $\lambda x.\lambda y.\ x + y$ could be assigned type $\top \xrightarrow{\emptyset} \top \to \top$ because no significant computations are performed between the passing of the two arguments. However, this type cannot be assigned to $\lambda x.\ \texttt{let}\ z = \omega\ \texttt{in}\ \lambda y.\ x + y$, only the less precise type $\top \to \top \to \top$. Trivial function types are of course subtypes of general function types:

$$\tau_1 \xrightarrow{\emptyset} \tau_2 <: \tau_1 \to \tau_2$$

We can use trivial function types to control the coercion rules that introduce `uncurry` and `curry` combinators:

$$b : \tau_1 \xrightarrow{\emptyset} \cdots \xrightarrow{\emptyset} \tau_n \to \tau \Longrightarrow \texttt{uncurry}_n(b) : (\tau_1, \ldots, \tau_n) \to \tau \qquad (n > 0)$$
$$b : (\tau_1, \ldots, \tau_n) \to \tau \Longrightarrow \texttt{curry}_n(b) : \tau_1 \xrightarrow{\emptyset} \cdots \xrightarrow{\emptyset} \tau_n \to \tau \qquad (n > 0)$$

We conjecture that this refinement suffices to ensure that higher-order uncurrying preserves non-termination, as well as observable effects in the case of ML-like languages.

7 Related work

In section 1, we already discussed the first-order uncurrying optimization that is commonly implemented in compilers for functional languages, as well as Hannan and Hick's type system for higher-order uncurrying [19]. We now discuss other related work.

Automatic insertion of coercions to mediate between different data representations is an old idea that has been applied successfully in many areas of programming language research. Examples include the combinations of dynamic and static typing of Thatte [35], Henglein [20], Flanagan [17] and Siek and Taha [33], as well as the unboxing optimizations of Leroy [25] and Henglein and Jørgensen [21]. A feature common to all these works and the present paper is the use of higher-order coercions $\lambda f.\lambda x.\ c'(f(c(x)))$ to lift pairs c, c' of coercions to function types.

Of these earlier works, Henglein's coercion calculus [20] is particularly interesting to us, since it presents efficient algorithms based on constraint solving to infer the placement of coercions in ways that are optimal under reasonable simplifying assumptions. An effective compilation pass performing higher-order uncurrying could probably be obtained by adapting Henglein's algorithms to our algebra of coercions. However, Henglein's base coercions convert between atomic type expressions, while our base coercions (the $\texttt{curry}_n$ and $\texttt{uncurry}_n$ combinators) operate over non-atomic function types. It remains to see what impact this difference has on Henglein's results.

Uncurrying of functions, including across higher-order functions, could also be decided based on the results of Shivers' control-flow analysis [32] or its type-based reformulation by Wells *et al.* [37]. Conservatively, one could uncurry a function definition only if all the call sites to which it flows are curried and match the arity of the function; in this case, no coercions need to be inserted. More aggressively, allowing the insertion of coercions to resolve arity mismatches requires heuristics for placing the coercions that, we believe, are best handled by Henglein's constraint-based algorithms.

8 Conclusions and future work

The framework developed in this article and its generic proof of correctness provides a strong basis to experiment with various higher-order uncurrying strategies without having to prove their correctness every time. Since our framework relies on a "soft" type system, it can be applied to a wide range of source languages, either statically or dynamically typed. Step-indexed logical relations—the key ingredient of our semantic preservation proof—turned out, once more, to be a powerful reasoning technique, providing the general flavor of proofs by denotational semantics while remaining easy to mechanize.

As mentioned in section 1, currying is only one of the two well-known presentations of n-ary functions in the λ-calculus and related programming languages: the other is "tupling", that is, the grouping of n arguments in a single n-tuple argument. Standard ML compilers as well as the OCaml compiler routinely perform first-order elimination of "tupled" functions, turning them into n-ary functions within their syntactic scope. This optimization is known under the name *arity raising*. Hannan and Hicks [18] extend this optimization to the higher-order case. We conjecture that our framework for higher-order uncurrying could easily be extended to deal with tupled functions as well. Representation types τ would be extended with a n-tuple type $\tau_1 \times \cdots \times \tau_n$, with $\top \times \cdots \times \top$ being a subtype of $\top$, and $\texttt{tuplify}_n$ and $\texttt{untuplify}_n$ combinators would be used to mediate between the representation types $(\tau_1, \ldots, \tau_n) \to \tau$ and $\tau_1 \times \cdots \times \tau_n \to \tau$.

The first-order uncurrying optimizations implemented in the Glasgow Haskell and OCaml native-code compilers exploit low-level closure representation tricks to accelerate curried applications of unknown functions [27]. The closure of a curried function $\lambda y_1 \ldots \lambda y_n.\ a$ contains two entry points: one to the uncurried function $f = \lambda(y_1, \ldots, y_n).\ a$ that expects all n arguments at once, another to a piece of generic code corresponding to $\texttt{curry}_n(f)$ and expecting one argument at a time. Moreover, the arity n of the function is also recorded in the closure. This enables the definition of generic combinators for curried applications to n arguments, of the following shape:

```
multappₙ(clos, y₁, ..., yₙ) =
    if clos.arity = n
    then clos.n-ary-entry-point(clos, y₁, ..., yₙ)
    else let clos₁ = clos.unary-entry-point(clos, y₁) in
         let clos₂ = clos₁.unary-entry-point(clos₁, y₂) in
         ...
         closₙ₋₁.unary-entry-point(closₙ₋₁, yₙ)
```

Then, a source-level curried application $f\ a_1\ \ldots\ a_n$, where f is an unknown function, is compiled to $\texttt{multapp}_n(f, a_1, \ldots, a_n)$. The run-time arity test performed by the `multapp` combinators avoids the cost of a fully curried application in the frequent case where arities match. It would be interesting to express this idiom and prove its correctness within our framework.

More generally, uncurrying is a paradigmatic instance of the very general theory of type isomorphisms [15]. Semantically, type isomorphisms have been studied using syntactic equational theories and denotational semantics. Neither of these frameworks extend easily to imperative features such as general references, in contrast with step-indexed relations [4]. Step-indexed relations could, therefore, provide a new angle to reason about type isomorphisms and extend them to functional languages with effects.

Acknowledgements This work was supported in part by Agence Nationale de la Recherche, grant number ANR-05-SSIA-0019. We thank the three anonymous reviewers for their helpful suggestion for improving the presentation of this paper.

References

1. Abadi, M., Cardelli, L., Curien, P.L., Lévy, J.J.: Explicit substitutions. Journal of Functional Programming **1**(4), 375–416 (1991)
2. Acar, U.A., Ahmed, A., Blume, M.: Imperative self-adjusting computation. In: 35th symposium Principles of Programming Languages, pp. 309–322. ACM Press (2008)
3. Aczel, P.: An introduction to inductive definitions. In: J. Barwise (ed.) Handbook of Mathematical Logic, *Studies in Logics and the Foundations of Mathematics*, vol. 90, pp. 739–782. North-Holland (1997)
4. Ahmed, A.J.: Semantics of types for mutable state. Ph.D. thesis, Princeton University (2004)
5. Ahmed, A.J.: Step-indexed syntactic logical relations for recursive and quantified types. In: Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, *Lecture Notes in Computer Science*, vol. 3924, pp. 69–83. Springer (2006)
6. Appel, A.W.: Foundational proof-carrying code. In: Logic in Computer Science 2001, pp. 247–258. IEEE Computer Society Press (2001)
7. Appel, A.W., McAllester, D.A.: An indexed model of recursive types for foundational proof-carrying code. ACM Transactions on Programming Languages and Systems **23**(5), 657–683 (2001)
8. Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The POPLmark challenge. In: Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs), *Lecture Notes in Computer Science*, vol. 3603, pp. 50–65. Springer (2005)
9. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. EATCS Texts in Theoretical Computer Science. Springer (2004)
10. Cardone, F., Hindley, J.R.: History of lambda-calculus and combinatory logic. In: D.M. Gabbay, J. Woods (eds.) Handbook of the History of Logic, volume 5: Logic from Russell to Church. North-Holland (2009)
11. Coq development team: The Coq proof assistant. Software and documentation available at `http://coq.inria.fr/` (1989–2009)
12. Cousineau, G., Curien, P.L., Mauny, M.: The categorical abstract machine. Science of Computer Programming **8**(2), 173–202 (1987)
13. Curry, H.B., Feys, R.: Combinatory Logic, volume I. North-Holland (1958). Third edition 1974
14. Dargaye, Z.: Décurryfication certifiée. In: Journées Francophones des Langages Applicatifs (JFLA'07), pp. 119–134. INRIA (2007)
15. Di Cosmo, R.: Isomorphisms of types: from lambda calculus to information retrieval and language design. Birkhauser (1995)
16. Felleisen, M., Friedman, D.P.: Control operators, the SECD machine and the λ-calculus. In: Formal Description of Programming Concepts III, pp. 131–141. North-Holland (1986)
17. Flanagan, C.: Hybrid type checking. In: 33rd symposium Principles of Programming Languages, pp. 245–256. ACM Press (2006)
18. Hannan, J., Hicks, P.: Higher-order arity raising. In: International Conference on Functional Programming 1998, pp. 27–38. ACM Press (1998)
19. Hannan, J., Hicks, P.: Higher-order uncurrying. Higher-Order and Symbolic Computation **13**(3), 179–216 (2000)
20. Henglein, F.: Dynamic typing: syntax and proof theory. Science of Computer Programming **22**(3), 197–230 (1994)
21. Henglein, F., Jørgensen, J.: Formally optimal boxing. In: 21st symposium Principles of Programming Languages, pp. 213–226. ACM Press (1994)

22. Krivine, J.L.: A call-by-name lambda-calculus machine. Higher-Order and Symbolic Computation **20**(3), 199–207 (2007)
23. Landin, P.J.: The mechanical evaluation of expressions. The Computer Journal **6**, 308–320 (1964)
24. Leroy, X.: The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA (1990)
25. Leroy, X.: Unboxed objects and polymorphic typing. In: 19th symposium Principles of Programming Languages, pp. 177–188. ACM Press (1992)
26. Letouzey, P.: Extraction in Coq: An overview. In: Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008, *Lecture Notes in Computer Science*, vol. 5028, pp. 359–369. Springer (2008)
27. Marlow, S., Peyton Jones, S.: Making a fast curry: push/enter vs. eval/apply for higher-order languages. Journal of Functional Programming **16**(4–5), 375–414 (2006)
28. Necula, G.C.: Proof-carrying code. In: 24th symposium Principles of Programming Languages, pp. 106–119. ACM Press (1997)
29. Necula, G.C.: Translation validation for an optimizing compiler. In: Programming Language Design and Implementation 2000, pp. 83–95. ACM Press (2000)
30. Peyton Jones, S.L.: Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. Journal of Functional Programming **2**(2), 127–202 (1992)
31. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Tools and Algorithms for Construction and Analysis of Systems, TACAS '98, *Lecture Notes in Computer Science*, vol. 1384, pp. 151–166. Springer (1998)
32. Shivers, O.: Control-flow analysis in Scheme. In: Programming Language Design and Implementation 1988, pp. 164–174. ACM Press (1988)
33. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming 2006, pp. 81–92. Technical report TR-2006-06, University of Chicago (2006)
34. Statman, R.: Logical relations and the typed λ-calculus. Information and Control **65**(2/3), 85–97 (1985)
35. Thatte, S.R.: Quasi-static typing. In: 17th symposium Principles of Programming Languages, pp. 367–381. ACM Press (1990)
36. Tristan, J.B., Leroy, X.: Formal verification of translation validators: A case study on instruction scheduling optimizations. In: 35th symposium Principles of Programming Languages, pp. 17–27. ACM Press (2008)
37. Wells, J.B., Dimock, A., Muller, R., Turbak, F.: A calculus for polymorphic and polyvariant flow types. Journal of Functional Programming **12**(3), 183–227 (2002)
38. Wright, A.K., Cartwright, R.: A practical soft type system for Scheme. ACM Transactions on Programming Languages and Systems **19**(1), 87–152 (1997)