



Higher-Order UnCurrying

JOHN HANNAN*
PATRICK HICKS†

hannan@cse.psu.edu

*Department of Computer Science and Engineering, The Pennsylvania State University, University Park,
PA 16802, USA*

Abstract. We present a formal specification of unCurrying for a higher-order, functional language with ML-style let-polymorphism. This specification supports the general unCurrying of functions, even for functions that are passed as arguments or returned as values. The specification also supports partial unCurrying of any consecutive parameters of a function, rather than only unCurrying all of a function's parameters. We present the specification as a deductive system that axiomatizes a judgment relating a source term with an unCurried form of the term. We prove that this system relates only typable terms and that it is correct with respect to an operational semantics. We define a practical algorithm, based on algorithm \mathcal{W} , that implements unCurrying and prove this algorithm sound and complete with respect to the deductive system. This algorithm generates maximally unCurried forms of source terms. These results provide a declarative framework for reasoning about unCurrying and support a richer form of unCurrying than is currently found in compilers for functional languages.

Keywords: semantics-based program analysis, type systems, program transformations

1. Introduction

Most higher-order, functional programming languages support a Curried notation for function definitions, allowing functions to be applied to one argument at a time, rather than to all its arguments at once. This feature allows programmers flexibility in the way they define and use functions, but with the potential of more function calls. Instead of defining a function taking two arguments together, e.g. $f(x, y) = e$, a programmer can define it as a Curried function $f x y = e$ (using notation as found in the language Standard ML). Then, instead of the function being applied to a pair of arguments via one function call, e.g., $f(e_1, e_2)$, the function is applied to the first argument, and then this result is applied to the second, $(f e_1) e_2$, resulting in two function calls. Because these languages are higher-order, this notation is not simply a syntactic variant. Arguments to Curried functions are not necessarily adjacent to one another: The expression $(f e_1)$ could be an argument passed to a function and the corresponding formal parameter could be applied to some e_2 .

As each function call imposes an overhead cost, minimizing the number of function calls (either at compile time or run time) is a useful operation. Various techniques can be applied to programs to accomplish this. A run-time system can recognize when a Curried function is applied to all of its arguments consecutively and generate just one function call. A code generator can also recognize when a Curried function is supplied all its arguments

*Partially supported by NSF CAREER Award #CCR-9502356.

†*Present address:* Brother Boniface Hicks, O.S.B., St. Vincent Archabbey, Latrobe, PA 15650, USA.

and optimize the code. Another alternative is to perform a program transformation that replaces Curried functions of the form $f\ x\ y = e$ by their unCurried form $f(x, y) = e$. This operation, called unCurrying, must be accompanied by corresponding ones for the application of f . Again, because the language is higher-order, this step is not simply the case of replacing all expressions $(f\ e_1\ e_2)$ with $f(e_1, e_2)$, but more generally replacing all expressions $(e_0\ e_1\ e_2)$, where e_0 can evaluate to f , with $e_0(e_1, e_2)$. Furthermore, this operation can only be performed if all possible applications of f have both arguments available together and if all possible values of e_0 are functions that have been unCurried. Generalizing this to functions of $n > 2$ arguments and allowing any sequence of consecutive parameters (and corresponding arguments) to be unCurried adds yet another dimension to the problem. Thus, the operation of unCurrying requires information about all possible uses of a function. We refer to the general unCurrying of functions in a higher-order language as *higher-order UnCurrying*.

The goal of the current work is to provide a study of higher-order unCurrying, including a formal specification, a proof of correctness, and a practical algorithm. We consider the problem of unCurrying more from a theoretical, rather than pragmatic, perspective, studying a simple language based on the λ -calculus extended with a notion of tuples for λ -abstractions and applications, and with let-style polymorphism. We define a deductive system that provides a declarative, high-level specification of unCurrying without resorting to algorithmic detail. We prove the correctness of this specification with respect to an operational semantics for the source (Curried) and target (unCurried) languages. We also give an algorithm, based on Milner's algorithm \mathcal{W} [20], that performs type inference and the unCurry translation. We prove this algorithm correct with respect to the declarative specification of unCurrying.

This work continues our effort to use formal systems and types to specify and verify compilation techniques for functional languages. Types provide a natural way of specifying program properties and type systems provide a mechanism for specifying the constraints between parts of a program and their types. We have studied closure conversion [8], escape analysis [9], arity raising [10], and live variable analysis [12] using types and type systems. These works all have in common the use of deductive systems to specify program analyses and properties, the use of the LF type theory to represent these systems and proofs about these systems, and the use of the Elf programming language to provide experimental implementations and to provide partial machine verification of proofs.

2. Mathematical preliminaries

Our presentation and discussion of type systems and operational semantics as deductive systems utilizes some elements of natural deduction proof theory [6, 25] that is not traditionally found in such work. We review this material here and provide some motivation for our choice of logical foundations for our work.

2.1. Natural deduction and assumptions

Natural deduction consists of deriving a formula from a finite number of assumptions (the context) using a predefined set of inference rules. During the course of constructing

a deduction certain assumptions can be closed or discharged. A proof is a deduction in which all assumptions have been discharged. At least two distinct methods of managing the context have been used.

The first one, used by both Gentzen [6] and Prawitz [25], consists of viewing a deduction as a tree whose nodes are labeled with formulae. One is allowed to tag any set of occurrences of some formula with a number, which also tags the inference rule that simultaneously discharges all the occurrences tagged by that number. We refer to this method as using *implicit contexts* because the set of open assumptions is not explicitly represented, but rather it can be inferred from the structure of the deduction.

The second method consists of maintaining an explicit record of all undischarged assumptions at every node in a deduction. Each node is labeled with an expression $\Gamma \vdash A$ in which A is a formula and Γ is a record of all undischarged assumptions at this point in the deduction. We refer to this method as using *explicit contexts* because each node contains an explicit structure representing the context.

Each of these methods have their advantages. The first one is more natural from a human's perspective and more economical as there is no need to introduce an explicit record at every node. The second method is typically easier to implement efficiently.

2.2. *Implicit type contexts and environments*

The usual presentation of type systems utilizes an explicit type context Γ , which maps variables to types. Contexts represent a set of assumptions about the free variables of a term. Judgments have the form $\Gamma \vdash e : \tau$ in which Γ contains the type information for the free variables of e . Inference rules are usually limited to the form

$$\frac{J_1 \cdot \dots \cdot J_n}{J_0}$$

for $n \geq 0$, in which each J_i is an atomic formula. (I.e., the antecedent of an inference rule is either empty or the conjunction of atomic formulae.) When reasoning about such judgments and their deductions, we must take into account some properties of Γ and its relation to other data structures (such as function closures).

The usual presentation of operational semantics utilizes an environment ρ , which maps variables to values. Values typically consist of at least constants and function closures, and judgments have the form $\rho \vdash e \hookrightarrow v$. Inference rules are again restricted to the form given above. The use of environments can be viewed as an abstraction of explicitly substituting values for bound variables. Relating type deductions and evaluation deductions requires relationships between contexts, environments, and closures.

An alternative presentation, based on natural deduction, replaces explicit contexts with hypothetical assumptions, and uses substitutions instead of environments. Instead of the judgments $\Gamma \vdash e : \tau$ and $\rho \vdash e \hookrightarrow v$, we can use the judgments $e : \tau$ and $e \hookrightarrow v$. Instead of having an explicit environment Γ , we have implicit assumptions (also called open hypotheses). Instead of having an environment and expressions with free variables in a judgment $\rho \vdash e \hookrightarrow_s v$, we ensure that expressions contain no free variables in a judgment $e \hookrightarrow_s v$. To support hypothetical reasoning, we construct inference rules whose antecedents can be

constructed using not only conjunction, but also implication and universal quantification. In particular, a common use of the latter two is in the following form:

$$\forall x . (P(x) \supset P(e))$$

in which P is a predicate on terms. The formula $P(x)$ represents hypothetical assumption about the variable x and the quantification over x ensures the unique identification of the variable x . To construct deductions of such formulae we use the standard rules, implication introduction ($\supset -I$) and universal introduction ($\forall -I$), from natural deduction:

$$\begin{array}{c} (A) \\ \vdots \\ \frac{B}{A \supset B} \quad \frac{A[c/x]}{\forall x.A} \quad c \text{ not free in } A \end{array}$$

The first rule can be read as follows: if from the assumption A we can prove B , then we can prove $A \supset B$ without the assumption A . The second rule can be read as follows: if we can prove a generic instance of A (obtained by replacing free occurrences of x with a new constant c) then we can prove $\forall x.A$.

The equivalence between specifications using explicit contexts or environments and those without is usually a straightforward exercise. For the type systems, we establish a relationship between contexts and open hypotheses. For the operational semantics, we establish a relationship between an environment/expression pair and the corresponding expression obtained by “applying” the environment to the expression. The ability to represent specifications with or without explicit contexts/environments affords us the option of choosing which form of specification to use, depending on our needs. For providing high-level specifications and for reasoning about correctness, specifications without explicit contexts/environments support simpler, more direct explanations and correctness proofs. However, for supporting efficient implementations, contexts and environments are essential. We therefore use both kinds of presentations.

2.3. Substitution property of deductions

Typical presentations of type systems using contexts often need to introduce a substitution lemma to assist in proving properties about the type systems. This lemma typically has the following form:

$$\text{If } \Gamma, x:\tau' \vdash e : \tau \text{ and } \Gamma \vdash e' : \tau' \text{ then } \Gamma \vdash e[e'/x] : \tau.$$

This property is essential for proving the typing of intermediate expressions obtained during evaluation (when substitution is used to implement function calls). Proving this lemma is not particularly difficult, but it does usually require careful consideration of contexts.

Giving specifications (of type systems and operational semantics) in a natural deduction framework with implicit contexts allows us to exploit properties of natural deduction proofs,

obviating the need for an explicit substitution lemma. In particular, we can make use of additional rules from natural deduction: implication elimination ($\supset -E$) and universal elimination ($\forall -E$):

$$\frac{A \supset B \quad A}{B} \quad \frac{\forall x.A}{A[t/x]}$$

which we assume, like the introduction rules, are part of the logic with which we can construct and manipulate deductions. Based on just these rules we automatically obtain the following useful property:

Substitution Property: If $\forall x.\forall y.(J_1 \supset J_2)$ and $J_1[t_1/x, t_2/y]$ (for any terms t_1 and t_2) are derivable then $J_2[t_1/x, t_2/y]$ is also derivable.

To see why this property holds, assume we have a deduction Δ of the formula $\forall x.\forall y.(J_1 \supset J_2)$ and a deduction Δ_1 of formula $J_1[t_1/x, t_2/y]$, then we can construct the following deduction:

$$\frac{\frac{\frac{\forall x.\forall y.(J_1 \supset J_2)}{\forall y.(J_1[t_1/x] \supset J_2[t_1/x])} \forall - E}{J_1[t_1/x, t_2/y] \supset J_2[t_1/x, t_2/y]} \forall - E \quad J_1[t_1/x][t_2/y]}{J_2[t_1/x, t_2/y]} \supset - E$$

We make significant use of this property in the proofs of correctness found in the appendix.

2.4. Deductions as objects

A significant factor motivating our choice of logic and structure of judgments is our interest in representing deductions as objects, allowing us to use mechanical proof assistants to reason about the properties of our type systems and operational semantics. We base this encoding of deductions as objects on the LF type system [14] and have encoded the representation of terms, the specification of deductive systems (including type systems, operational semantics, and the unCurry specification), and proofs of theorems in this logic. We have experimented with and verified (in the case of the proofs) these by implementing these encodings in the Elf language [24]. Specifically we have verified (most of) Theorem 4 using this technique. The LF type system and the Elf language provide a flexible and powerful setting in which to specify and reason about deductive systems. These systems have been used to describe and reason about natural semantics [19] and also compiler verification [13]. Unfortunately, this framework is not particularly well suited to reasoning about algorithms such as the one we give in Section 8, and so the proof of Theorem 5 was performed entirely by hand.

3. Two typed languages

Our specification of typed unCurrying relates terms in a source and a target language, both versions of the typed λ -calculus. The source language is the implicitly typed λ -calculus with let-polymorphism, and the target language generalizes upon this to permit unCurried functions and applications. We do not include products as a primitive construct. Tuples of expressions can only occur in the operand position of an application. We let e and m range over source and target language expressions, respectively. The following grammars define the syntax for these languages.

$$\begin{aligned} e &::= x \mid \lambda x. e \mid e_1 @ e_2 \mid \text{let } x = e_1 \text{ in } e_2 \\ m &::= x \mid \lambda [x_1, \dots, x_k]. m \mid m_1 @ [m_2, \dots, m_k] \mid \text{let } x = m_1 \text{ in } m_2 \end{aligned}$$

We use ‘@’ to denote application, which, as usual, is left associative. In a term $\lambda [x_1, \dots, x_k]. m$ we require all the x_i to be distinct. In the target language, abstractions and applications have an *arity* $k \geq 1$.

The types of the source language are the traditional simple types and type schemas of Standard ML:

$$\begin{aligned} \tau^s &::= \alpha \mid \tau^s \rightarrow \tau^s \\ \sigma^s &::= \tau^s \mid \forall \alpha. \sigma^s \end{aligned}$$

We use the superscript s to denote source language types. We write $\forall \overline{\alpha}_n. \tau^s$ as an abbreviation for $\forall \alpha_1, \dots, \alpha_n. \tau^s$.

The types of the target language generalize these by introducing a notation for unCurried function types.

$$\begin{aligned} \tau^t &::= \alpha \mid [\tau^t, \dots, \tau^t] \rightarrow \tau^t \\ \sigma^t &::= \tau^t \mid \forall \alpha. \sigma^t \end{aligned}$$

We will drop the superscript s and t from types where they can be inferred from the context.

For both source and target types we require the instantiation of type schema σ to type τ , written $\sigma \succeq \tau$, which we define in the usual manner.

Let $\Gamma \triangleright_s e : \tau$ be the judgment for typing expressions in the source language. Its definition is given in figure 1. Let $\Gamma \triangleright_t m : \tau$ be the judgment for typing expressions in the target language. Its definition is given in figure 2. The function $\text{Gen}(\tau, \Gamma)$ returns the type $\forall \alpha_1, \dots, \forall \alpha_n. \tau$ in which $\{\alpha_1, \dots, \alpha_n\} = FV(\tau) - FV(\Gamma)$. We use two rules for handling λ -abstraction to provide a correspondence with the two rules for λ -abstraction found in the specification of unCurrying given in the next section. Observe that

$$\frac{\Gamma \{x_1 : \tau_1\} \cdots \{x_n : \tau_n\} \triangleright_t m : \tau}{\Gamma \triangleright_t \lambda [x_1, \dots, x_n]. m : [\tau_1, \dots, \tau_n] \rightarrow \tau}$$

is a derived inference rule in this system.

We define an operational semantics for each of these languages by axiomatizing judgments of the form $e \hookrightarrow_s v$ and $m \hookrightarrow_t w$. To provide a high-level description for these semantics, we use variable substitution rather than environments to manage the binding

$$\begin{array}{c}
\frac{\Gamma(x) \geq \tau}{\Gamma \triangleright_s x : \tau} (\text{var}_s) \\
\\
\frac{\Gamma \triangleright_s e_1 : \tau_1 \rightarrow \tau \quad \Gamma \triangleright_s e_2 : \tau_1}{\Gamma \triangleright_s e_1 @ e_2 : \tau} (\text{app}_s) \\
\\
\frac{\Gamma\{x : \tau_1\} \triangleright_s e : \tau}{\Gamma \triangleright_s \lambda x. e : \tau_1 \rightarrow \tau} (\text{abs}_s) \\
\\
\frac{\Gamma \triangleright_s e_1 : \tau_1 \quad \sigma_1 = \text{Gen}(\tau_1, \Gamma) \quad \Gamma\{x : \sigma_1\} \triangleright_s e_2 : \tau}{\Gamma \triangleright_s \text{let } x = e_1 \text{ in } e_2 : \tau} (\text{let}_s)
\end{array}$$

Figure 1. The source type inference system.

$$\begin{array}{c}
\frac{\Gamma(x) \geq \tau}{\Gamma \triangleright_t x : \tau} (\text{var}_t) \\
\\
\frac{\Gamma \triangleright_t m : [\tau_1, \dots, \tau_n] \rightarrow \tau \quad \Gamma \triangleright_t m_i : \tau_i \text{ for } i \in [1..n]}{\Gamma \triangleright_t m @ [m_1, \dots, m_n] : \tau} (\text{app}_t) \\
\\
\frac{\Gamma\{x_1 : \tau_1\} \triangleright_t m : \tau}{\Gamma \triangleright_t \lambda [x_1]. m : [\tau_1] \rightarrow \tau} (\text{abs}_{1t}) \\
\\
\frac{\Gamma\{x_1 : \tau_1\} \triangleright_t \lambda [x_2, \dots, x_n]. m : [\tau_2, \dots, \tau_n] \rightarrow \tau \quad n \geq 2}{\Gamma \triangleright_t \lambda [x_1, x_2, \dots, x_n]. m : [\tau_1, \tau_2, \dots, \tau_n] \rightarrow \tau} (\text{abs}_{nt}) \\
\\
\frac{\Gamma \triangleright_t m_1 : \tau_1 \quad \sigma_1 = \text{Gen}(\tau_1, \Gamma) \quad \Gamma\{x : \sigma_1\} \triangleright_t m_2 : \tau}{\Gamma \triangleright_t \text{let } x = m_1 \text{ in } m_2 : \tau} (\text{let}_t)
\end{array}$$

Figure 2. The target type inference system.

$$\begin{array}{c}
\overline{\lambda x. e \hookrightarrow_s \lambda x. e} \\
\\
\frac{e_1 \hookrightarrow_s \lambda x. e' \quad e_2 \hookrightarrow_s v_2 \quad e'[v_2/x] \hookrightarrow_s v}{e_1 @ e_2 \hookrightarrow_s v} \\
\\
\frac{e_1 \hookrightarrow_s v_1 \quad e_2[v_1/x] \hookrightarrow_s v}{\text{let } x = e_1 \text{ in } e_2 \hookrightarrow_s v}
\end{array}$$

Figure 3. Source operational semantics.

of variables to values. Thus, in both languages, values are limited to λ -abstractions. This simplifies correctness proofs as we do not need to reason about function closures. The operational semantics for the two languages are given in figures 3 and 4.

4. Examples

To examine some of the issues of higher-order unCurrying we consider several illustrative examples before giving a specification for unCurrying. Consider the K combinator applied

$$\frac{\overline{\lambda[x_1, \dots, x_k].m} \hookrightarrow_t \lambda[x_1, \dots, x_k].m}{\frac{m \hookrightarrow_t \lambda[x_1, \dots, x_k].m' \quad m_i \hookrightarrow_t w_i \text{ for } i \in [1..k] \quad m'[\overline{w_i/x_i}] \hookrightarrow_t w}{m @ [m_1, \dots, m_k] \hookrightarrow_t w} \quad \frac{m_1 \hookrightarrow_t w_1 \quad m_2[w_1/x] \hookrightarrow_t w}{\text{let } x = m_1 \text{ in } m_2 \hookrightarrow_t w}}$$

Figure 4. Target operational semantics.

to two terms:

$$(\lambda x. \lambda y. x) @ e_1 @ e_2$$

In this trivial example, we can clearly see that the action of unCurrying K requires the tupling of its arguments:

$$(\lambda[x, y]. x) @ [e_1, e_2]$$

When the function being unCurried is immediately applied to its arguments, the process of translating the application is trivial. This first-order form of unCurrying (in which the function to be unCurried is neither passed as an argument nor returned as a result, and is applied to all its Curried arguments) is found in current implementations of functional languages. We refer to this restricted case as *first-order unCurrying*, as the functions being unCurried are essentially used in a first-order setting (though they could have parameters of functional type).

Consider now the slightly more complicated example in which the K combinator is passed as an argument before being applied to its arguments:

$$(\lambda k. k @ e_1 @ e_2) @ (\lambda x. \lambda y. x)$$

This can be unCurried to

$$(\lambda k. k @ [e_1, e_2]) @ [\lambda[x, y]. x].$$

Here, the operation of unCurrying K must be coordinated with the use of formal parameter k : this operation cannot take place if k is applied to only one argument. Simply checking that k is always applied to two arguments (in a Curried fashion) amounts to another form of first-order unCurrying, in which a function named k is defined and then can only be applied to all of its arguments for it to be unCurried.

As a third example, consider the following term:

$$(\lambda g. (g @ (\lambda x. \lambda y. x))) @ (\lambda f. f @ e_1 @ e_2)$$

In this example, the ability to unCurry K relies on ensuring that the argument supplied to g is a function whose parameter is an unCurried function. So unCurrying K yields the

following term:

$$(\lambda g.(g @ [\lambda[x, y].x])) @ [\lambda f.f @ [e_1, e_2]]$$

While these examples are easy and intuitive to understand, the situation of unCurrying quickly becomes unwieldy as demonstrated by the term

$$(\lambda s.\lambda k.s @ (\lambda u.\lambda v.v) @ (\lambda w.w) @ (k @ e_1 @ e_2)) @ S @ K$$

(where S is $\lambda x.\lambda y.\lambda z.(x @ z)(y @ z)$) which can be unCurried to

$$\begin{aligned} &(\lambda[s, k].s @ [\lambda[u, v].v, \lambda w.w, k @ [e_1, e_2]]) \\ &@ [\lambda[x, y, z].x @ [z, y @ [z]], \lambda[x, y].x] \end{aligned}$$

As a final example, consider the term

$$((\lambda g.g @ e_1 @ e_2) @ ((\lambda x.\lambda y.\lambda z.\lambda w.e_3) @ e_4)) @ e_5.$$

We can unCurry this term to

$$((\lambda g.g @ [e_1, e_2]) @ [(\lambda x.\lambda[y, z].\lambda w.e_3) @ [e_4]]) @ [e_5],$$

demonstrating how unCurrying of just some parameters (and not necessarily the first) of a function can be handled.

As these examples illustrate, performing higher-order unCurrying involves generating and solving constraints over the unCurrying properties of expressions (specifically functions and applications). For each λ -abstracted variable and each operand we need to determine whether it can be unCurried and what other abstracted variables and operands must then also be unCurried.

5. Typed unCurrying

To provide an abstract, declarative specification of unCurrying, we define a type system that axiomatizes a relation between a source term, a type, and an unCurried (target language) form of the term. The system is non-deterministic in that it can relate a single source term to many possible unCurried forms of the term. The inference rules follow the structure of a traditional type system for the typed λ -calculus with polymorphic let, but we include unCurry information as part of the types. Just as simple types provide constraints between terms, this unCurrying information provides information about the introduction or use of unCurried functions, and the inference rules use this information to constrain the way unCurried terms can be constructed. We annotate function types with information indicating which parameters and arguments should be unCurried.

5.1. Uncurry types

We introduce sets of annotations, types, and type schemas that we use to specify constraints on terms (via a type system). We use φ , τ , and σ , respectively, to range over annotations, types, and type schemas, respectively.

$$\begin{aligned}\varphi &::= \epsilon \mid \uparrow \\ \tau &::= \alpha \mid \tau \rightarrow_{\varphi} \tau \\ \sigma &::= \tau \mid \forall \alpha. \sigma\end{aligned}$$

The annotation on an arrow is either \uparrow , which indicates that a function's parameter should be unCurried, or ϵ , which indicates that the parameter should not be unCurried. Given a term $\lambda x_1. \lambda x_2. x_2$ of type $\tau_1 \rightarrow \tau_2 \rightarrow \tau_2$, which we unCurry to $\lambda[x_1, x_2]. x_2$, which argument has been unCurried, x_1 or x_2 ? We conceivably could consider either argument as the one unCurried, but we believe the resulting specification of unCurrying is simpler if we choose to view the first argument as being unCurried. Hence we annotate the first function arrow with \uparrow . Thus, given the expression $\lambda x_1. \lambda x_2. x_2$, we will refer to the unCurrying of the parameter x_1 to yield $\lambda[x_1, x_2]. x_2$.

Observe that our choice of annotating arrows implies that for a term to have type $\tau \rightarrow_{\uparrow} \tau'$, the type τ' must be a function type. Thus, not all annotated types will make sense in terms of unCurrying information.

For example, one possible type for the K combinator $\lambda x. \lambda y. x$ is

$$(\tau_1 \rightarrow_{\uparrow} \tau_2 \rightarrow_{\epsilon} \tau_1) \rightarrow_{\uparrow} \tau_3 \rightarrow_{\epsilon} (\tau_1 \rightarrow_{\uparrow} \tau_2 \rightarrow_{\epsilon} \tau_1)$$

indicating the following characteristics: the parameter x should be unCurried (with y to form a tuple) and x should accept as an argument a function unCurried in its first parameter. If K can be given this type in the term

$$(\lambda x. \lambda y. x) @ e_0 @ e_1 @ e_2 @ e_3$$

then we can determine that e_0 must have type $(\tau_1 \rightarrow_{\uparrow} \tau_2 \rightarrow_{\epsilon} \tau_1)$, indicating that its first parameter should be unCurried, the arguments e_0 and e_1 should be tupled (because x and y are being tupled), and the arguments e_2 and e_3 should be tupled (because, operationally, x will be bound to an unCurried function and applied to the values of these two arguments). Let e'_0 be the result of unCurrying e_0 as required. Then we can construct the unCurried term

$$(\lambda[x, y]. x) @ [e'_0, e_1] @ [e_2, e_3].$$

The terms e_1, e_2, e_3 might also undergo some unCurrying, but these transformations are not relevant to this example.

5.2. Two specifications of unCurrying

We present two equivalent definitions of unCurrying, one with and one without explicit contexts. Providing both allows flexibility in reasoning about correctness. (We use one

for reasoning about type correctness and one for reasoning about operational correctness.) Each of these defines a relation between a source term e , an unCurry type τ , and a target term m .

Recall the example $\lambda x_1.\lambda x_2.x_2$, given type $\tau_1 \rightarrow_{\uparrow} \tau_2 \rightarrow_{\epsilon} \tau_2$ to indicate that it should be unCurried to $\lambda[x_1, x_2].x_2$. Now consider the term in the larger context $(\lambda x_1.\lambda x_2.x_2) @ e_1 @ e_2$ unCurried to $(\lambda[x_1, x_2].x_2) @ [e_1, e_2]$. For convenience and consistency with abstraction, we say that the operand e_1 is unCurried in this case. The annotation on the type of the function should provide enough information to direct the unCurrying of both the abstraction and the application. As suggested above, the type annotation supports compositional reasoning about abstractions. However, consider the application $(\lambda x_1.\lambda x_2.x_2) @ e_1$. The operator has annotated type $\tau_1 \rightarrow_{\uparrow} \tau_2 \rightarrow_{\epsilon} \tau_2$, the operand has type τ_1 , and the application itself has type $\tau_2 \rightarrow_{\epsilon} \tau_2$. The annotation ‘ \uparrow ’ is “consumed” by this application, but we cannot immediately unCurry the application at this point because we are not in a context with enough information. We therefore need to maintain the annotation information somewhere until we are in a context in which we can perform the unCurrying. In other words, we need to keep track of which applications should be unCurried and which should not.

To accomplish this we introduce a new form of application to the syntax of target terms:

$$m ::= \dots \mid m @_{\uparrow} m$$

This form of application is used to help construct terms during the unCurrying of applications. A term of the form $(m_0 @_{\uparrow} m_1)$ indicates that operand m_1 should be unCurried. From another perspective, the term $(m_0 @_{\uparrow} m_1)$ represents an application in which the operator m_0 expects a tuple of arguments in which m_1 is just the first. We extend both the type system and operational semantics for the target language by introducing rules for this new form of application.

$$\frac{\Gamma \triangleright_t m : [\tau_1, \tau_2, \dots, \tau_n] \rightarrow \tau \quad \Gamma \triangleright_t m_1 : \tau_1}{\Gamma \triangleright_t m @_{\uparrow} m_1 : [\tau_2, \dots, \tau_n] \rightarrow \tau} \text{ (app}\uparrow_t\text{)}$$

$$\frac{m_0 \hookrightarrow_t \lambda[x_1, x_2, \dots, x_n].m' \quad m_1 \hookrightarrow_t w_1}{m_0 @_{\uparrow} m_1 \hookrightarrow_t \lambda[x_2, \dots, x_n].(m'[w_1/x_1])} \text{ (t-app-}\uparrow\text{)}$$

To define the unCurry translation with explicit contexts we introduce the judgment $\Gamma \triangleright e : \tau \Rightarrow m$, in which τ is an unCurry type. This judgment can be read as “with respect to context Γ , expression e has unCurry type τ and translates (can be unCurried) to expression m .” We axiomatize this judgment by the rules in figure 5. The first rule treats variables straightforwardly, having extended the instantiation operation (‘ \succeq ’) to handle annotated type schemes. Rules (u-app- ϵ) and (u-app- \uparrow) handle the general cases for application. The annotation ‘ \uparrow ’ on the function type (in the premise of rule (u-app- \uparrow)) corresponds to the annotation on the application in the conclusion of that rule. Observe that these two rules ensure that if \uparrow annotates a function arrow then there is another function arrow “to the right.” The next rule, (u-app- $@_{\uparrow}$), unCurries an argument in an application, using (or consuming) the \uparrow -annotation on the application. The unCurried argument is added to the head of the tuple of any previously unCurried arguments (to the right). Thus, arguments are unCurried

$$\begin{array}{c}
\frac{\Gamma(x) \succeq \tau}{\Gamma \triangleright x : \tau \Rightarrow x} \text{(u-var)} \\
\\
\frac{\Gamma \triangleright e_1 : \tau_1 \xrightarrow{\epsilon} \tau_2 \Rightarrow m_1 \quad \Gamma \triangleright e_2 : \tau_1 \Rightarrow m_2}{\Gamma \triangleright e_1 @ e_2 : \tau_2 \Rightarrow m_1 @ [m_2]} \text{(u-app-}\epsilon\text{)} \\
\\
\frac{\Gamma \triangleright e_1 : \tau_1 \xrightarrow{\uparrow} \tau_2 \xrightarrow{\varphi} \tau_3 \Rightarrow m_1 \quad \Gamma \triangleright e_2 : \tau_1 \Rightarrow m_2}{\Gamma \triangleright e_1 @ e_2 : \tau_2 \xrightarrow{\varphi} \tau_3 \Rightarrow m_1 @_{\uparrow} m_2} \text{(u-app-}\uparrow\text{)} \\
\\
\frac{\Gamma \triangleright e_1 @ e_2 : \tau \Rightarrow (m_0 @_{\uparrow} m_1) @ [m_2, \dots, m_k]}{\Gamma \triangleright e_1 @ e_2 : \tau \Rightarrow m_0 @ [m_1, m_2, \dots, m_k]} \text{(u-app-}@_{\uparrow}\text{)} \\
\\
\frac{\Gamma\{x : \tau\} \triangleright e : \tau_1 \Rightarrow m}{\Gamma \triangleright \lambda x.e : \tau \xrightarrow{\epsilon} \tau_1 \Rightarrow \lambda[x].m} \text{(u-abs)} \\
\\
\frac{\Gamma\{x_1 : \tau\} \triangleright \lambda x_2.e : \tau_1 \xrightarrow{\varphi} \tau_2 \Rightarrow \lambda[x_2, \dots, x_k].m}{\Gamma \triangleright \lambda x_1.\lambda x_2.e : \tau \xrightarrow{\uparrow} \tau_1 \xrightarrow{\varphi} \tau_2 \Rightarrow \lambda[x_1, x_2, \dots, x_k].m} \text{(u-abs-}\uparrow\text{)} \\
\\
\frac{\Gamma \triangleright e_1 : \tau_1 \Rightarrow m_1 \quad \sigma_1 = \mathbf{Gen}(\tau_1, \Gamma) \quad \Gamma\{x : \sigma_1\} \triangleright e_2 : \tau \Rightarrow m_2}{\Gamma \triangleright \text{let } x = e_1 \text{ in } e_2 : \tau \Rightarrow \text{let } x = m_1 \text{ in } m_2} \text{(u-let)}
\end{array}$$

Figure 5. The unCurry inference system with contexts.

from right to left. The next rule, (u-abs), handles the general case for λ -abstractions. To type and translate $\lambda x.e$ we simply translate its body. Because no unCurrying of function parameters occurs in this case, the function type receives an ϵ -annotation. The next rule, (u-abs- \uparrow), performs the unCurrying of function parameters, inserting the appropriate \uparrow -annotation in the type. Observe that if $k = 2$ then $\varphi = \epsilon$; if $k > 2$ then $\varphi = \uparrow$. The final rule, (u-let), treats let expressions straightforwardly, without any use of annotations.

To define the unCurry translation without explicit contexts we introduce the judgment $e : \tau \Rightarrow m$, in which τ is an unCurry type, and axiomatize it by the rules in figure 6. Most of the rules are nearly identical to their counterparts in figure 5, except that no context is present. This specification has no rules for translating variables because we use hypothetical assumptions instead. The two rules for translating λ -abstractions introduce assumptions about bound variables. The universal quantifiers ensure the uniqueness of bound variables. Thus the assumption introduced by an implication is the only means for reasoning about a given variable.

The rule for let expressions uses logical implication to encode the property that occurrences of x in e_2 can be typed at distinct instances of τ' . Introducing the assumption

$$\forall \tau'. (e_1 : \tau' \Rightarrow m_1 \supset x : \tau' \Rightarrow x')$$

can be viewed as introducing a new inference rule

$$\frac{e_1 : \tau' \Rightarrow m_1}{x : \tau' \Rightarrow x'}$$

$$\begin{array}{c}
\frac{e_1 : \tau_1 \rightarrow_\epsilon \tau_2 \Rightarrow m_1 \quad e_2 : \tau_1 \Rightarrow m_2}{e_1 @ e_2 : \tau_2 \Rightarrow m_1 @ [m_2]} (\text{u-app-}\epsilon) \\
\\
\frac{e_1 : \tau_1 \rightarrow_{\uparrow} \tau_2 \rightarrow_\varphi \tau_3 \Rightarrow m_1 \quad e_2 : \tau_1 \Rightarrow m_2}{e_1 @ e_2 : \tau_2 \rightarrow_\varphi \tau_3 \Rightarrow m_1 @_{\uparrow} m_2} (\text{u-app-}\uparrow) \\
\\
\frac{e_1 @ e_2 : \tau \Rightarrow (m_0 @_{\uparrow} m_1) @ [m_2, \dots, m_k]}{e_1 @ e_2 : \tau \Rightarrow m_0 @ [m_1, m_2, \dots, m_k]} (\text{u-app-}@_{\uparrow}) \\
\\
\frac{\forall x. \forall x'. (x : \tau \Rightarrow x' \supset e : \tau_1 \Rightarrow m)}{\lambda x. e : \tau \rightarrow_\epsilon \tau_1 \Rightarrow \lambda [x']. m} (\text{u-abs}) \\
\\
\frac{\forall x_1. \forall x'_1. (x_1 : \tau \Rightarrow x'_1 \supset \lambda x_2. e : \tau_1 \rightarrow_\varphi \tau_2 \Rightarrow \lambda [x'_2, \dots, x'_k]. m)}{\lambda x_1. \lambda x_2. e : \tau \rightarrow_{\uparrow} \tau_1 \rightarrow_\varphi \tau_2 \Rightarrow \lambda [x'_1, x'_2, \dots, x'_k]. m} (\text{u-abs-}\uparrow) \\
\\
\frac{e_1 : \tau_1 \Rightarrow m_1 \quad \forall x. \forall x'. ((\forall \tau'. (e_1 : \tau' \Rightarrow m_1 \supset x : \tau' \Rightarrow x')) \supset e_2 : \tau \Rightarrow m_2)}{\text{let } x = e_1 \text{ in } e_2 : \tau \Rightarrow \text{let } x' = m_1 \text{ in } m_2} (\text{u-let})
\end{array}$$

Figure 6. The unCurry inference system without contexts.

for unCurrying x . Since τ' is universally quantified, we can instantiate this rule to distinct instances. This technique simulates the traditional presentation of let-polymorphism in which we generalize τ' to a type schema $\forall \bar{\alpha}_n. \tau'$ and assume this as the type for x when typing e_2 .

Theorem 1. *For all closed terms e , $e : \tau \Rightarrow m$ is derivable iff $\cdot \triangleright e : \tau \Rightarrow m$ is derivable.*

The proof of the theorem requires a generalization to open terms and a correspondence between open hypotheses and contexts.

Both of these specifications allow the translation of terms into ones that contain occurrences of '@_↑', even though such terms are not in our original target language. Some of these terms, such as $\lambda [x]. \lambda [y]. ((x @_{\uparrow} y) @ [y])$, represent unCurrying which has not yet been done. However, other terms, such as $\lambda [x]. \lambda [y]. (x @_{\uparrow} y)$, do not even represent any meaningful unCurrying. The two-phase specification introduced in Section 7 prohibits these latter terms. The algorithm we derive in Section 8 enforces the condition that the translated term have no occurrences of '@_↑'.

These inference systems provide a general specification of unCurrying. For a given source term e , there can exist several τ_i and m_i such that $\Gamma \triangleright e : \tau_i \Rightarrow m_i$. Included among these terms is one in which all function arrows in a deduction are annotated with ϵ , in which case no unCurrying occurs. Our goal in unCurrying is to take a term e and produce a term m that can be used in place of e . If we do not know the context of e (say, for example, during separate compilation) then we must require a translation $e : \tau \Rightarrow m$ in which m and τ contain no '@_↑' annotations. This ensures that the translated term m can be used in the same context in which e can be used.

Because unCurrying is essentially an operation affecting function definitions and applications and this operation preserves the order of evaluation of arguments (assuming a right-to-left evaluation of tupled arguments), extending the unCurry specification to a much

larger language, including one with side effects, does not pose any challenges, except for how to handle separate compilation.

6. Correctness

The specification of unCurrying defines a relation between terms in the source and target languages. We show the correctness of this specification by demonstrating that the relation respects both the type systems and the operational semantics of the two languages: typeable source and target.

In the following, whenever we refer to a judgment J in a proposition, lemma, or theorem, we use J to mean “judgment J is derivable.”

6.1. Type correctness

For our specification of unCurrying to respect the type systems of the source and target languages we must establish two properties, which we call completeness and soundness. By completeness we mean that any (well-typed) source-language term can be unCurried by our system. By soundness we mean that the term resulting from unCurrying is always well-typed.

To establish a relationship between type derivations in the source and target languages and derivations of unCurrying we must provide a relation between annotated and unannotated types. First we define a translation from source-language types to unCurry types.

Definition 1. For every simple type τ , its corresponding unCurry type, $[\tau]_\epsilon$ is obtained by annotating every arrow in τ with ϵ .

We extend this definition to type schemas and type contexts in the natural way: $[\forall \overline{\alpha}_n. \tau]_\epsilon = \forall \overline{\alpha}_n. [\tau]_\epsilon$ and if $\Gamma(x) = \sigma$ then $[\Gamma]_\epsilon(x) = [\sigma]_\epsilon$.

Next we define a translation from unCurry types to source-language and target-language types.

Definition 2. For every unCurry type τ , its erasure $|\tau|$ and its unCurrying $\|\tau\|$ are given by:

$$\begin{aligned}
 |\alpha| &= \alpha \\
 |\tau \rightarrow_\varphi \tau'| &= |\tau| \rightarrow |\tau'| \\
 \|\alpha\| &= \alpha \\
 \|\tau \rightarrow_\epsilon \tau'\| &= \|\tau\| \rightarrow \|\tau'\| \\
 \|\tau \rightarrow_\uparrow \tau'\| &= \mathbf{case} \|\tau'\| \mathbf{of} \\
 &\quad [\tau_1, \dots, \tau_n] \rightarrow \tau_0 \Rightarrow [\|\tau\|, \tau_1, \dots, \tau_n] \rightarrow \tau_0 \\
 &\quad \tau'' \Rightarrow [\|\tau\|] \rightarrow \tau''
 \end{aligned}$$

We extend this definition to type schemas by letting $|\forall\bar{\alpha}.\tau| = \forall\bar{\alpha}.\|\tau\|$ and $\|\forall\bar{\alpha}.\tau\| = \forall\bar{\alpha}.\|\tau\|$. We further extend this to type contexts in the natural way: if $\Gamma(x) = \sigma$ then $|\Gamma|(x) = |\sigma|$ and $\|\Gamma\|(x) = \|\sigma\|$.

For the \uparrow -annotation on a function arrow to be significant in unCurrying, it must occur in a context $\tau \rightarrow_{\uparrow} \tau' \rightarrow_{\varphi} \tau''$. The $\|\cdot\|$ -translation allows more general occurrences so we do not need special constraints on the structure of annotated types and contexts containing these types.

We can now state our theorems of completeness and soundness.

Theorem 2 (Completeness). *If $\Gamma \triangleright_s e : \tau$ then there exists a target term m such that $[\Gamma]_{\varepsilon} \triangleright e : [\tau]_{\varepsilon} \Rightarrow m$.*

The proof, not given here, follows by constructing a deduction that includes no unCurrying (i.e., no occurrences of (u-app- \uparrow), (u-app- $@_{\uparrow}$), or (u-abs- \uparrow)). Thus, every typable source term can be related to some target term.

Theorem 3 (Soundness). *If $\Gamma \triangleright e : \tau \Rightarrow m$ then $|\Gamma| \triangleright_s e : |\tau|$ and $\|\Gamma\| \triangleright_t m : \|\tau\|$.*

The proof can be found in the appendix.

6.2. Operational correctness

To demonstrate the correctness of the unCurrying inference system with respect to the operational semantics, we prove a general statement that accounts for intermediate values that are functions and also for deductions involving \uparrow -annotated types and intermediate unCurried terms of the form $(m_1 @_{\uparrow} m_2)$.

To account for higher-order functions we include as part of the statement of correctness a form of subject reduction for annotated types: if we can derive $e : \tau \Rightarrow m$ and $m \hookrightarrow_t w$ then type τ must be a consistent type for w . In other words, there must be some v such that $v : \tau \Rightarrow w$. In fact, the required v must also be the value of e : the judgment $e \hookrightarrow_s v$ must be derivable. Relating the two values v and w via the unCurrying system also accounts for the case in which v and w contain functions, some of which, in w , are the unCurried versions of functions in v . The absence of environments, contexts, and function closures greatly simplifies the presentation here. Specifically, we do not require a semantic definition of value consistency, as found in some proofs of type consistency [30]. Our definition is purely a syntactic one.

Theorem 4 (Correctness of unCurry inference).

1. *If $e : \tau \Rightarrow m$ and $e \hookrightarrow_s v$ then there exists a term w such that $m \hookrightarrow_t w$ and $v : \tau \Rightarrow w$;*
2. *If $e : \tau \Rightarrow m$ and $m \hookrightarrow_t w$ then there exists a term v such that $e \hookrightarrow_s v$ and $v : \tau \Rightarrow w$.*

The proof can be found in the appendix. Note that the proof of part (2) does not simply proceed by induction on the structure of $m \hookrightarrow_t w$, as this approach fails for the case of applications. If $m = (m_0 @ [m_1, m_2, m_3])$ then e must be of the form $((e_0 @ e_1) @ e_2) @ e_3$ and we cannot simply perform induction involving m_0 and e_0 since we do not immediately

have the unCurry translation between e_0 and m_0 . Instead, we introduce a metric on deductions based on the arity of applications. This metric provides a well-founded order on which to base the induction.

7. A two-phase specification

The task of unCurrying can be described as consisting of two parts: (i) detecting which abstractions and applications can be unCurried and (ii) translating (unCurrying) these abstractions and applications. The two specifications presented in Section 5 intertwine both these parts. Those inference rules specify the constraints required for unCurrying and also relate a term to its unCurried form. This mixing of constraints and translation is what yields a simple specification of unCurrying.

From an algorithmic perspective, however, this approach is troublesome because we typically cannot translate an expression $(e_0 @ e_1 @ e_2)$ to $(e_0 @ [e_1, e_2])$ before solving constraints involving other terms. For example, consider the expression

$$\text{let } f = \lambda x. \lambda y. x \text{ in } (f @ (f @ e_1 @ e_2))$$

We cannot decide whether to unCurry the function $\lambda x. \lambda y. x$ until we have considered all the occurrences of f . Similarly, we cannot determine whether an application can be unCurried until we have considered other applications and the structure of all possible values of f . Hence, we cannot hope to construct an algorithm to perform unCurrying based on a single pass over a program. Instead, we will decompose the task of unCurrying explicitly into phases of inference and translation.

To construct this algorithm we first develop a two-phase specification of unCurrying based on the inference system of figure 5. The original specification performs both type inference and a translation based on annotations. In the two-phase specification we separate these two steps: The first phase axiomatizes the judgment $\Gamma \triangleright_i e : \tau \Rightarrow u$ that specifies that source term e has type τ and can be annotated as term u . UnCurry annotations now annotate both applications and abstractions. The grammar for these annotated terms is given by

$$u ::= x \mid u_1 @_{\varphi} u_2 \mid \lambda_{\varphi} x. u \mid \text{let } x = u_1 \text{ in } u_2$$

The second phase axiomatizes the judgment $u \Rightarrow_i m$ and relates such an annotated term to its translation into a target language term. This phase can be viewed as performing a translation directed by the annotations. The complete specification of this two-phase specification is given in figures 7 and 8.

To specify unCurrying as a two-phase process we need to enforce certain conditions during the inference phase to ensure that translation can proceed properly. We define the following predicates:

$$\begin{aligned} \text{grndApp}(u) &\equiv (u = (u_1 @_{\varphi} u_2)) \supset \varphi = \epsilon \\ \text{grndAbs}(\lambda x_{\varphi}. u) &\equiv (u = (\lambda y_{\varphi'}. u_1)) \vee \varphi = \epsilon \end{aligned}$$

The predicate grndApp ensures that if its argument is an application, then that application is annotated by ' ϵ '. The predicate grndAbs ensures that if the body of a λ -abstraction is

$$\begin{array}{c}
\frac{\Gamma(x) \succeq \tau}{\Gamma \triangleright_i x : \tau \Rightarrow x} \\
\\
\frac{\Gamma \triangleright_i e_1 : \tau \rightarrow_{\varphi} \tau' \Rightarrow u_1 \quad \Gamma \triangleright_i e_2 : \tau \Rightarrow u_2 \quad \text{grndApp}(u_2)}{\Gamma \triangleright_i e_1 @ e_2 : \tau' \Rightarrow u_1 @_{\varphi} u_2} \\
\\
\frac{\Gamma\{x : \tau\} \triangleright_i e : \tau' \Rightarrow u \quad \text{grndAbs}(\lambda_{\varphi} x. u) \quad \text{grndApp}(u)}{\Gamma \triangleright_i \lambda x. e : \tau \rightarrow_{\varphi} \tau' \Rightarrow \lambda_{\varphi} x. u} \\
\\
\frac{\text{grndApp}(u_1) \quad \text{grndApp}(u_2)}{\Gamma \triangleright_i e_1 : \tau_1 \Rightarrow u_1 \quad \sigma_1 = \text{Gen}(\tau_1, \Gamma) \quad \Gamma\{x : \sigma_1\} \triangleright_i e_2 : \tau \Rightarrow u_2} \\
\Gamma \triangleright_i \text{let } x = e_1 \text{ in } e_2 : \tau \Rightarrow \text{let } x = u_1 \text{ in } u_2
\end{array}$$

Figure 7. Phase 1: Inference.

$$\begin{array}{c}
\frac{}{x \Rightarrow_t x} \qquad \frac{u \Rightarrow_t m}{\lambda_{\epsilon} x. u \Rightarrow_t \lambda x. m} \\
\\
\frac{\lambda_{\varphi} x_2. u \Rightarrow_t \lambda[x_2, \dots, x_n]. m}{\lambda_{\uparrow} x_1. \lambda_{\varphi} x_2. u \Rightarrow_t \lambda[x_1, x_2, \dots, x_n]. m} \\
\\
\frac{u_1 \Rightarrow_t m_1 \quad u_2 \Rightarrow_t m_2}{\text{let } x = u_1 \text{ in } u_2 \Rightarrow_t \text{let } x = m_1 \text{ in } m_2} \\
\\
\frac{\text{NoUncApp}(u_1) \quad u_1 \Rightarrow_t m_1 \quad u_2 \Rightarrow_t m_2}{u_1 @_{\varphi} u_2 \Rightarrow_t m_1 @ [m_2]} \\
\\
\frac{u_0 @_{\uparrow} u_1 \Rightarrow_t m_0 @ [m_1, \dots, m_{n-1}] \quad u_n \Rightarrow_t m_n}{(u_0 @_{\uparrow} u_1) @_{\varphi} u_n \Rightarrow_t m_0 @ [m_1, \dots, m_{n-1}, m_n]}
\end{array}$$

Figure 8. Phase 2: Translation.

not another λ -abstraction, then the λ -abstraction is annotated by ‘ ϵ .’ These two predicates prohibit the occurrences of ‘ \uparrow ’ annotations in places where they would be meaningless. Intuitively, these predicates check for conditions when unCurrying *cannot* occur (e.g., if an application is the body of a λ -abstraction, then that application cannot possibly be unCurried). We introduce a predicate for the translation phase:

$$\text{NoUncApp}(u) \equiv \neg(u = (u_1 @_{\uparrow} u_2))$$

This predicate, used for convenience, tests for the case when a term is not an application that is to be unCurried. Using this predicate allows us to distinguish easily between the two cases for translating applications.

Observe that in a deduction of $\Gamma \triangleright_i e_1 @ e_2 : \tau' \Rightarrow u_1 @_{\varphi} u_2$ no constraint is placed on the annotation φ . Only if this deduction occurs in the context of some larger deduction

will φ possibly be constrained. If $(u_1 @_{\varphi} u_2)$ is the “whole” term, then no unCurrying of this application is possible. The inference phase does not explicitly enforce this by requiring $\varphi = \epsilon$. However, the translation phase effectively ignores the annotation on a top-level application since it plays no role in unCurrying. The algorithm presented in the next section shares this behavior.

We can show the equivalence of the original (one-phase) specification and this new two-phase specification.

Lemma 1.

1. If $\Gamma \triangleright e : \tau \Rightarrow m$ and m has no occurrences of ‘ $@_{\uparrow}$ ’ then there exists a term u such that $\Gamma \triangleright_i e : \tau \Rightarrow u$ and $u \Rightarrow_t m$.
2. If $\Gamma \triangleright_i e : \tau \Rightarrow u$ and $u \Rightarrow_t m$ then $\Gamma \triangleright e : \tau \Rightarrow m$.

The condition that m have no occurrences of ‘ $@_{\uparrow}$ ’ allows us to prohibit meaningless terms such as $\lambda x. \lambda y. (x @_{\uparrow} y)$, which are explicitly prohibited by the two-phase specification. The proof requires a relaxation of this condition in which we allow occurrences of ‘ $@_{\uparrow}$ ’ only in terms like $(m_0 @_{\uparrow} m_1) @ [m_2, \dots, m_k]$, which can occur in both specifications.

8. An unCurry algorithm

We have developed and proved correct an algorithm for unCurrying. Part of the algorithm is similar to algorithm \mathcal{W} [20] and has the same complexity. (I.e., on average, the algorithm runs in time proportional to the size of the input expression, but in the worst case it runs in exponential time, relative to the size of the input.)

The algorithm consists essentially of two steps, \mathcal{U} and *translate*, corresponding to the two phases above:

$$\text{unCurry}(\Gamma, e) = \text{let } (\theta, \tau, u) = \mathcal{U}(\Gamma, e) \\ \text{in } \text{translate}(\theta_{\uparrow}(\theta_u))$$

in which θ ranges over substitutions mapping unCurry type and annotation variables to unCurry types and annotations, respectively, and θ_{\uparrow} is the substitution that maps all annotation variables to ‘ \uparrow .’ (We extend the grammars for unCurry annotations to include annotation variables γ .) The application of substitutions to a term u operates on any annotation variables occurring in the term. The use of the substitution θ_{\uparrow} is not strictly necessary, as the function *translate* could simply treat annotation variables as ‘ \uparrow ,’ but including θ_{\uparrow} simplifies the description of *translate*. The definitions of \mathcal{U} and *translate* are given in figures 9 and 10.

The function \mathcal{U} ensures that θ is an idempotent substitution, disallowing cyclic references with an occurs check in the unification algorithm. Intuitively, \mathcal{U} determines where unCurrying *cannot* be done (since this can be checked locally) and sets the corresponding annotations to ϵ . Unification propagates such constraints throughout a term.

The function \mathcal{U} consists of four cases corresponding to the structure of the source term. The cases are almost identical to those of algorithm \mathcal{W} , but \mathcal{U} also generates correct unCurry

$$\begin{aligned}
\mathcal{U}(\Gamma, x) &= \\
&\text{if } x \notin \text{dom}(\Gamma) \text{ then fail} \\
&\text{else let } \forall \overline{\alpha_n}. \tau = \Gamma(x) \\
&\quad \{\overline{\beta_n}\} \text{ be new variables} \\
&\quad \text{in } (ID, \tau[\overline{\beta_n}/\overline{\alpha_n}], x) \\
\mathcal{U}(\Gamma, \lambda x. e) &= \\
&\text{let } \alpha, \gamma \text{ be new variables} \\
&\quad (\theta_1, \tau, u) = \mathcal{U}(\Gamma\{x : \alpha\}, e) \\
&\quad \theta_2 = \text{grndApp}(\theta_1 u) \circ \text{grndAbs}(\lambda_{\gamma} x. (\theta_1 u)) \\
&\text{in } (\theta_2 \circ \theta_1, \alpha \rightarrow_{\gamma} \tau, \lambda_{\gamma} x. u) \\
\mathcal{U}(\Gamma, e_1 @ e_2) &= \\
&\text{let } (\theta_1, \tau_1, u_1) = \mathcal{U}(\Gamma, e_1) \\
&\quad (\theta_2, \tau_2, u_2) = \mathcal{U}(\theta_1 \Gamma, e_2) \\
&\quad \text{let } \alpha, \gamma \text{ be new variables} \\
&\quad \theta_3 = \text{unify}(\theta_2(\theta_1 \tau_1), \theta_2 \tau_2 \rightarrow_{\gamma} \alpha) \\
&\quad \theta_4 = \text{grndApp}(\theta_3 u_2) \\
&\text{in } (\theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1, \alpha, u_1 @_{\gamma} u_2) \\
\mathcal{U}(\Gamma, \text{let } x = e_1 \text{ in } e_2) &= \\
&\text{let } (\theta_1, \tau_1, u_1) = \mathcal{U}(\Gamma, e_1) \\
&\quad \theta'_1 = \text{grndApp}(\theta_1 u_2) \circ \text{grndApp}(\theta_1 u_1) \circ \theta_1 \\
&\quad \sigma_1 = \text{Gen}(\tau_1, \theta'_1(\Gamma)) \\
&\quad (\theta_2, \tau_2, u_2) = \mathcal{U}(\theta'_1(\Gamma\{x \mapsto \forall \overline{\alpha_n}. \tau_1\}), e_2) \\
&\text{in } (\theta_2 \circ \theta'_1, \tau_2, \text{let } x = u_1 \text{ in } u_2)
\end{aligned}$$

Figure 9. The function \mathcal{U} .

annotations on terms and function types. The variable case fails if the variable is not in the domain of Γ , but otherwise the variable's type is instantiated by substituting fresh, unbound type variables for the quantified variable in the type schema. The abstraction case generates a new type variable α for the bound variable and then annotates the body. To ensure the correct annotation on applications, λ -abstractions, and the corresponding function arrows, we employ two auxiliary functions, grndApp and grndAbs . The function grndApp checks if its argument is an application annotated with a variable γ and, if it is, it returns the substitution mapping γ to ϵ . If its argument is an application annotated by ' \uparrow ,' it fails. Otherwise it just returns the identity function. The use of this function in the algorithm corresponds to constraints specified in the inference system: an application occurring as the body of a λ -abstraction or as the operand to an application are annotated with ϵ (and hence their operands will not be unCurried). The function grndAbs checks if the body of a λ -abstraction is another λ -abstraction. If it is not, then the annotation on the λ -abstraction (and corresponding function arrow) is bound to ϵ . Again, the use of this function in the algorithm corresponds to constraints specified in the inference system: a λ -abstraction whose body is not also a λ -abstraction must be annotated by ϵ . The application case uses the function unify to resolve constraints. Again, the use of grndAbs ensures that appropriate

$$\begin{aligned}
& \text{grndApp}(u_1 @_{\gamma} u_2) = ID\{\gamma \mapsto \epsilon\} \\
& \text{grndApp}(u_1 @_{\uparrow} u_2) = \text{fail} \\
& \text{grndApp}(u) = ID \\
\\
& \text{grndAbs}(\lambda_{\gamma} x. \lambda_{\varphi}. u) = ID \\
& \text{grndAbs}(\lambda_{\gamma} x. u) = ID\{\gamma \mapsto \epsilon\} \\
\\
& \text{unify}(\alpha, \tau) = \\
& \quad \text{if } \alpha = \tau \text{ then } ID \\
& \quad \text{else if } \text{occurs}(\alpha, \tau) \text{ then fail else } ID\{\alpha \mapsto \tau\} \\
& \text{unify}(\tau, \alpha) = \text{unify}(\alpha, \tau) \\
& \text{unify}(\tau_1 \rightarrow_{\varphi} \tau_2, \tau'_1 \rightarrow_{\varphi'} \tau'_2) = \\
& \quad \text{let } \theta_1 = \text{unifyann}(\varphi, \varphi) \\
& \quad \quad \theta_2 = \text{unify}(\theta_1 \tau_1, \theta_1 \tau'_1) \\
& \quad \quad \theta_3 = \text{unify}(\theta_2 \tau_2, \theta_2 \tau'_2) \\
& \quad \text{in } \theta_3 \circ \theta_2 \circ \theta_1 \\
\\
& \text{unifyann}(\epsilon, \epsilon) = ID \\
& \text{unifyann}(\uparrow, \uparrow) = ID \\
& \text{unifyann}(\gamma, \varphi) = ID\{\gamma \mapsto \varphi\} \\
& \text{unifyann}(\varphi, \gamma) = ID\{\gamma \mapsto \varphi\} \\
& \text{unifyann}(\varphi, \varphi') = \text{fail} \\
\\
& \text{translate}(x) = x \\
& \text{translate}(\lambda_{\epsilon} x. u) = \lambda x. \text{translate}(u) \\
& \text{translate}(\lambda_{\uparrow} x. u) = \\
& \quad \text{let } \lambda[x_2, \dots, x_n]. m = \text{translate}(u) \\
& \quad \text{in } \lambda[x, x_2, \dots, x_n]. m \\
& \text{translate}(u_0 @_{\uparrow} u_1) @_{\varphi} u_2 = \\
& \quad \text{let } (m @ [m_1, \dots, m_n]) = \text{translate}(u_0 @_{\uparrow} u_1) \\
& \quad \text{in } (m @ [m_1, \dots, m_n, \text{translate}(u_2)]) \\
& \text{translate}(u_1 @_{\varphi} u_2) = \text{translate}(u_1) @ [\text{translate}(u_2)] \\
& \text{translate}(\text{let } x = u_1 \text{ in } u_2) = \\
& \quad \text{let } x = \text{translate}(u_1) \text{ in } \text{translate}(u_2)
\end{aligned}$$

Figure 10. The unCurry algorithm (cont).

annotations are ϵ , just as enforced by the inference system. The case for let expressions can be explained similarly.

The function *unify* (figure 10) takes two unCurry types and returns the most general substitution that unifies the unCurry types. The algorithm for *unify* is nearly identical to the traditional one for types, except it also unifies annotations on function types. Given two types with no occurrences of the \uparrow -annotation, *unify* succeeds exactly when the traditional unification algorithm succeeds on the erased versions of the two types. Failure can occur via an occurs-check (as in traditional unification) or when the annotations on function arrows

cannot be unified. The latter can only happen if the initial context supplied to *unCurry* contained an improper annotation of ‘ \uparrow ’ on a function type.

The final step in the algorithm translates the annotated term into an unCurried (target language) term. The function *translate* examines the annotations on applications and abstractions, and unCurries those that are annotated with ‘ \uparrow ’.

Note that both the algorithm, as outlined above, and the inference systems allow the type of an expression e to change after unCurrying. As previously noted, this can be an unsafe operation if we do not know the context in which e is used (e.g. during separate compilation). To restrict the unCurrying to be local to an expression e (i.e., independent of the context of e) we simply need to extend the substitution θ (returned by \mathcal{U} above) to map all annotation variables occurring in the annotated type τ and types in Γ to ‘ ϵ ’. With regards to the inference systems, this corresponds to considering only deductions in which the judgment at the root of the deduction contains no ‘ \uparrow ’ annotations.

To prove that the algorithm is correct and that it generates maximally unCurried terms, we relate it to the two-phase specification of unCurrying, showing that it is both sound and complete with respect to this specification.

Theorem 5 (*Soundness of unCurry*).

1. If $(\theta, \tau, u) = \mathcal{U}(\Gamma, e)$ then $\theta_{\uparrow}(\theta\Gamma) \triangleright_i e : \theta_{\uparrow}(\theta\tau) \Rightarrow \theta_{\uparrow}(\theta u)$;
2. If $m = \text{translate}(\theta_{\uparrow}(\theta u))$ then $\theta_{\uparrow}(\theta u) \Rightarrow_i m$.

The proof of part (ii) is nearly trivial due to the close similarity between the function ‘ \Rightarrow_i ’ and the inference system of figure 8. The proof of part (i) closely follows a proof of type soundness for Milner’s algorithm \mathcal{W} [20], with additional considerations for unCurry annotations. We outline here the significant steps of this proof.

We first need three auxiliary results.

1. Correctness of the function *unify*. For the proof of soundness we need only show that if $\text{unify}(\tau_1, \tau_2) = \theta$ then $\theta\tau_1 = \theta\tau_2$. For the completeness result below, we must additionally show that θ is the most general unifier of τ_1 and τ_2 .
2. Idempotence. The substitutions returned by \mathcal{U} are idempotent. Furthermore, if θ' is idempotent and $(\theta, \tau, u) = \mathcal{U}(\theta'\Gamma, e)$ then $\theta'\tau = \tau$ and $\theta'u = u$.
3. Substitution property. Applying a substitution to a deduction yields another (valid) deduction: if $\Gamma \triangleright_i e : \tau \Rightarrow u$ is derivable then for any substitution θ , $\theta\Gamma \triangleright_i e : \theta\tau \Rightarrow \theta u$ is derivable.

Each of these is an adaptation of a result used in proving type soundness. Showing each of them is straightforward.

With these results we can prove the soundness of \mathcal{U} by induction on the structure of the expression e . Instead of the statement in the theorem, we prove the following statement:

$$\text{If } (\theta, \tau, u) = \mathcal{U}(\Gamma, e) \text{ then } \theta\Gamma \triangleright_i e : \theta\tau \Rightarrow \theta u.$$

By the substitution property for deductions this implies part (i) of the theorem. We give here just the case for application as an illustrative example. Assume $e = (e_1 @ e_2)$ and

$(\theta, \tau, u) = \mathcal{U}(\Gamma, e_1 @ e_2)$. Then from the definition of \mathcal{U} we have

$$\begin{aligned} (\theta_1, \tau_1, u_1) &= \mathcal{U}(\Gamma, e_1), \\ (\theta_2, \tau_2, u_2) &= \mathcal{U}(\theta_1 \Gamma, e_2), \\ \theta_3 &= \text{unify}(\theta_2(\theta_1 \tau_1), \theta_2 \tau_2 \rightarrow_{\gamma} \alpha) \quad \text{for new variables } \alpha \text{ and } \gamma, \\ \theta_4 &= \text{grndApp}(\theta_3 u_2), \\ \theta &= \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1, \\ \tau &= \alpha, \quad \text{and} \\ u &= u_1 @_{\gamma} u_2. \end{aligned}$$

Applying the inductive hypothesis to each of the first two statements yields some deductions

$$\Xi_1 :: \theta_1 \Gamma \triangleright_i e_1 : \theta_1 \tau_1 \Rightarrow \theta_1 u_1 \quad \text{and} \quad \Xi_2 :: \theta_2(\theta_1 \Gamma) \triangleright_i e_2 : \theta_2 \tau_2 \Rightarrow \theta_2 u_2,$$

respectively. From the idempotency property, $\tau_2 = \theta_1 \tau_2$ and $u_2 = \theta_1 u_2$. Hence Ξ_2 is also a deduction of $\theta_2(\theta_1 \Gamma) \triangleright_i e_2 : \theta_2(\theta_1 \tau_2) \Rightarrow \theta_2(\theta_1 u_2)$.

From the third statement and the correctness of *unify*, θ_3 is a unifier of $\theta_2(\theta_1 \tau_1)$ and $\theta_2 \tau_2 \rightarrow_{\gamma} \alpha$. Because α and γ are new variables introduced after the definition of θ_2 and θ_1 , we also have $\theta_2(\theta_1 \alpha) = \alpha$ and $\theta_2(\theta_1 \gamma) = \gamma$. Hence, θ_3 is a unifier of $\theta_2(\theta_1 \tau_1)$ and $\theta_2(\theta_1(\tau_2 \rightarrow_{\gamma} \alpha))$. This result and the fifth statement imply $\theta(\tau_1) = \theta \tau_2 \rightarrow_{\theta \gamma} \theta \alpha$.

From $\theta_4 = \text{grndApp}(\theta_3 u_2)$, if $\theta_3 u_2 = (u_3 @_{\varphi} u_4)$ then $\theta_4 \varphi = \epsilon$. Hence, the predicate $\text{grndApp}(\theta_{u_2})$ holds.

Applying the substitution property of deductions to Ξ_1 and Ξ_2 we have some deductions

$$\Xi'_1 :: \theta \Gamma \triangleright_i e_1 : \theta \tau_1 \Rightarrow \theta_{u_1} \quad \text{and} \quad \Xi'_2 :: \theta \Gamma \triangleright_i e_2 : \theta \tau_2 \Rightarrow \theta_{u_2}.$$

Because $\theta(\tau_1) = \theta \tau_2 \rightarrow_{\theta \gamma} \theta \alpha$, Ξ'_1 is also a deduction of

$$\theta \Gamma \triangleright_i e_1 : \theta \tau_2 \rightarrow_{\theta \gamma} \theta \alpha \Rightarrow \theta_{u_1}.$$

Hence, we can construct the deduction

$$\frac{\begin{array}{c} \Xi'_1 \\ \theta \Gamma \triangleright_i e_1 : \theta \tau_2 \rightarrow_{\theta \gamma} \theta \alpha \Rightarrow \theta_{u_1} \end{array} \quad \begin{array}{c} \Xi'_2 \\ \theta \Gamma \triangleright_i e_2 : \theta \tau_2 \Rightarrow \theta_{u_2} \quad \text{grndApp}(\theta_{u_2}) \end{array}}{\theta \Gamma \triangleright_i (e_1 @ e_2) : \theta \alpha \Rightarrow \theta(u_1 @_{\gamma} u_2)}$$

which is the one required because $\tau = \alpha$ and $u = (u_1 @_{\gamma} u_2)$ from the sixth and seventh statements above.

The steps followed here are nearly identical to those followed in a proof of type soundness, with the addition of considerations for unCurry annotations. The remaining cases of the proof similarly follow their counterparts in a type soundness proof.

Theorem 6 (Completeness of unCurry). *If $\Gamma \triangleright_i e : \tau \Rightarrow u$ and $u \Rightarrow_i m$ then there exist τ', u', θ , and θ' such that $(\theta', \tau', u') = \mathcal{U}(\Gamma, e)$, $\tau = \theta(\theta' \tau')$, $u = \theta(\theta' u')$, and $\text{translate}(u) = m$.*

$$\begin{array}{c}
\overline{x \sqsubseteq x} \\
\frac{p_0 \sqsubseteq m_0 \cdots p_k \sqsubseteq m_k}{p_0 @ [p_k, \dots, p_1] \sqsubseteq m_0 @ [m_k, \dots, m_1]} \\
\frac{p_0 \sqsubseteq m_0 @ [m_n, \dots, m_{k+1}] \quad p_1 \sqsubseteq m_1 \cdots p_k \sqsubseteq m_k \quad n > k}{p_0 @ [p_k, \dots, p_1] \sqsubseteq m_0 @ [m_n, \dots, m_1]} \\
\frac{p \sqsubseteq m}{\lambda[x_1, \dots, x_k].p \sqsubseteq \lambda[x_1, \dots, x_k].m} \\
\frac{p \sqsubseteq \lambda[x_{k+1}, \dots, x_n].m \quad n > k}{\lambda[x_1, \dots, x_k].p \sqsubseteq \lambda[x_1, \dots, x_n].m} \\
\frac{p_1 \sqsubseteq m_1 \quad p_2 \sqsubseteq m_2}{\text{let } x = p_1 \text{ in } p_2 \sqsubseteq \text{let } x = m_1 \text{ in } m_2}
\end{array}$$

Figure 11. Order on unCurried terms.

The proof is similar to existing proofs of completeness for type inference algorithms [22], with additional consideration given to unCurry annotations.

Completeness demonstrates that the algorithm performs unCurrying whenever possible. We can clarify this idea by introducing an ordering on target terms, corresponding to the degree of unCurrying that occurs in them. This ordering is given in figure 11. Intuitively, $p \sqsubseteq m$ if the terms are both unCurried forms of the same source term, with m unCurried at least as much as p is unCurried. If $p \sqsubseteq m$ and $p \neq m$ then m has fewer abstractions and/or applications. So a maximal term will be more unCurried, meaning applications with more arguments or abstractions with more parameters (than in a less-unCurried term).

We can motivate the rules for applications and abstractions as follows. Given two applications, $p = (p_0 @ [p_k, \dots, p_1])$ and $m = (m_0 @ [m_n, \dots, m_1])$, $p \sqsubseteq m$ if $n \geq k$ and the corresponding subterms of p and m are also related in this order. We find these corresponding subterms by matching up the *rightmost* arguments in each application, and continue working to the left, until we have handled all of p 's arguments. This right-to-left ordering is why we number the operands as we do. If $n = k$ then we have the same number of arguments in each term, and so m_0 and p_0 must be related (the first rule for applications in figure 11). If $n > k$ then m_0 applied to the remaining arguments of m must be related to p_0 . An example illustrates this relationship. Consider the term $(w @ [x] @ [y] @ [z])$ and an unCurried form $(w @ [x, y, z])$. We can prove $(w @ [x] @ [y] @ [z]) \sqsubseteq (w @ [x, y, z])$ by proving $z \sqsubseteq z$ and $(w @ [x] @ [y]) \sqsubseteq (w @ [x, y])$. The latter statement can be proved by proving $y \sqsubseteq y$ and $(w @ [x]) \sqsubseteq (w @ [x])$, which follows from the first rule for applications.

The two rules for abstractions behave in a similar manner, except here we work from left-to-right in matching up corresponding formal parameters. (We use α -conversion to guarantee the names of the corresponding parameters are the same.) Using these rules we can show $\lambda w. \lambda [x, y]. \lambda z. y \sqsubseteq \lambda [w, x, y, z]. y$.

Using this ordering and the previous results, we have the following result.

Theorem 7 (Principal unCurrying). *If $\Gamma \triangleright e : \tau \Rightarrow m$ and m has no occurrences of ‘@_†’, then $\text{unCurry}(\Gamma, e) = m'$ and $m \sqsubseteq m'$.*

The proof follows from the completeness of *unCurry* and the equivalence of the various specifications of unCurrying. Thus, our algorithm computes the best unCurrying possible (relative to our specification in Section 5) for any well-typed term.

9. Untyped unCurrying

We can adapt the unCurrying inference system of Section 5 to treat the untyped λ -calculus. Much of the treatment from the typed setting carries over to the untyped setting, and here we focus on the differences. Obviously, we do not have the simple types and type schemas, but we can introduce a form of type that conveys only the unCurry information. For completeness, we must ensure that every untyped λ -term can be accommodated. To do this we adapt some ideas of partial type inference [7] by introducing a kind of universal type Ω that can be the type of any term. We use this type as a default type when we cannot infer any information about the functional behavior of a term.

We again let τ range over the unCurry types:

$$\tau ::= \Omega \mid \tau \rightarrow_{\varphi} \tau$$

(We add type variables when considering the algorithm for untyped unCurrying.) The source and target languages are similar to those defined in Section 3, including the operational semantics, but without the *let* construct and the type systems.

The inference system includes the rules from figure 6, except for (u-let), plus the additional rules given in figure 12. These new rules ensure that any λ -term can be typed. The resulting system allows us to infer useful unCurrying information, even with the lack of traditional types. Not surprisingly, all of the examples from Section 5, when viewed as untyped terms, can be handled by this new system. Additionally, however, untypable terms (in the simple-type sense) can be non-trivially unCurried if they use functions in a consistent way. As an illustrative example, consider the expression

$$(\text{if } b \text{ then } \lambda x. \lambda y. x + y \text{ else } \lambda x. \lambda y. \lambda z. x + y) n_1 n_2.$$

Our inference system (extended to handle conditionals and integers) can unCurry this term to

$$(\text{if } b \text{ then } \lambda[x, y]. x + y \text{ else } \lambda[x, y]. \lambda z. x + y) [n_1, n_2].$$

$$\frac{e_1 : \Omega \Rightarrow m_1 \quad e_2 : \Omega \Rightarrow m_2}{e_1 @ e_2 : \Omega \Rightarrow (m_1 @ m_2)} (\text{u-app-}\Omega)$$

$$\frac{x : \Omega \Rightarrow x' \quad e : \Omega \Rightarrow m}{\lambda x. e : \Omega \Rightarrow \lambda x'. m} (\text{u-abs-}\Omega)$$

Figure 12. The Ω -rules.

The addition of the Ω -rules allows all untyped terms to be translated to target language terms.

Proposition 1. *For all closed, untyped, source terms e , there exists a type τ and target term m such that $e : \tau \Rightarrow m$ is derivable.*

The proof relies on observing that we can choose $\tau = \Omega$ and $e = m$. Such a judgment is always derivable using just rules (u-app- Ω) and (u-abs- Ω).

We can modify the inference algorithm from the previous section to perform unCurrying. The modifications to the function \mathcal{U} are minor. The type context Γ maps variables to types (not schemas) and so the variable case does not instantiate a schema. The abstraction and application cases, however, remain the same. The let case no longer applies. The major change applies to *unify*, which must now always succeed. We generalize the traditional notion of unification by introducing a general notion of equality, $=_{\Omega}$, extending syntactic equality with the equation

$$\Omega =_{\Omega} \Omega \rightarrow_{\epsilon} \Omega$$

The type $\Omega \rightarrow_{\epsilon} \Omega$ essentially states that no unCurrying occurs in a function. The type Ω provides the same information. We can define a partial order of Ω -types by introducing the inequality

$$\Omega \sqsubseteq_{\Omega} \Omega \rightarrow_{\epsilon} \Omega$$

and show the following properties.

Proposition 2.

1. *If $\tau_1 =_{\Omega} \tau_2$ then there exists a τ' such that $\tau' \sqsubseteq_{\Omega} \tau_1$ and $\tau' \sqsubseteq_{\Omega} \tau_2$.*
2. *If $\Gamma \triangleright e : \tau \Rightarrow m$ and $\tau' \sqsubseteq_{\Omega} \tau$ then $\Gamma \triangleright e : \tau' \Rightarrow m$.*

Exploiting these properties, the function *unify*, given in figure 13, takes two types and returns a unifying substitution modulo the $=_{\Omega}$ -equality. Unlike traditional unification, this function always succeeds when given two types containing no \uparrow -annotations (and no base types). For any such type τ we can always obtain a substitution θ such that $\theta\tau =_{\Omega} \Omega$. If the occurs check succeeds, then, rather than failing, the algorithm forces the offending type and type variable to be equivalent to Ω , indicating that no unCurrying can occur. The function *ground*(τ) simply generates a substitution θ such that $\theta\tau =_{\Omega} \Omega$.

Proposition 3. *If $\theta = \text{unify}(\tau_1, \tau_2)$ then $\theta\tau_1 =_{\Omega} \theta\tau_2$.*

The result of $\mathcal{U}(\Gamma, e)$ is again a triple (θ, τ, u) in which u is an annotated term. We can again use *translate* to produce an unCurried term. This algorithm improves upon Gomard's iterative algorithm of partial type inference [7]. Rather than modifying the unification algorithm, their partial type inference algorithm allows unification to fail. When this happens,

$$\begin{aligned}
& \text{unify}(\Omega, \Omega) = ID \\
& \text{unify}(\Omega, \tau) = \text{ground}(\tau) \\
& \text{unify}(\tau, \Omega) = \text{ground}(\tau) \\
& \text{unify}(\alpha, \tau) = \\
& \quad \text{if } \alpha = \tau \text{ then } ID \\
& \quad \text{else if } \text{occurs}(\alpha, \tau) \text{ then } \text{ground}(\tau) \\
& \quad \quad \text{else } ID\{\alpha \mapsto \tau\} \\
& \text{unify}(\tau, \alpha) = \text{unify}(\alpha, \tau) \\
& \text{unify}(\tau_1 \rightarrow_{\gamma} \tau_2, \tau'_1 \rightarrow_{\gamma'} \tau'_2) = \\
& \quad \text{let } \theta_1 = \text{unifyann}(\gamma, \gamma') \\
& \quad \quad \theta_2 = \text{unify}(\theta_1 \tau_1, \theta_1 \tau'_1) \circ \theta_1 \\
& \quad \quad \theta_3 = \text{unify}(\theta_2 \tau_2, \theta_2 \tau'_2) \circ \theta_2 \\
& \quad \text{in } \theta_3
\end{aligned}$$

Figure 13. The untyped unification algorithm.

their algorithm sets appropriate types to Ω and performs algorithm \mathcal{W} again on the entire term.

This specification of untyped unCurrying does, however, have its limitations. Due to the monomorphic nature of types associated with bound variables, some terms, which can be unCurried by hand, cannot be unCurried by our system. For example, the term

$$(\lambda z.(z @ z @ z)) @ (\lambda x.\lambda y.x) @ e_1 @ e_2$$

can be unCurried (by hand) into

$$(\lambda z.(z @ [z, z])) @ [\lambda[x, y].x] @ [e_1, e_2]$$

but our system cannot infer this. It can only perform the identity translation. The reason for this limitation is that to unCurry $(z @ z @ z)$ into $(z @ [z, z])$ requires distinct types on different occurrences of z . The use of a rank 2 type system appears to overcome this limitation [16].

10. Future work

A number of related issues remain to be explored in conjunction with this work. We briefly outline some of them here.

Practical application. The current work focuses on the formal specification and correctness of unCurrying, isolated from its practical application to real program and its interaction with other program optimizations. One significant issue to address is how to increase the amount of unCurrying that can occur. In our work, to unCurry a function we must also

be able to unCurry all application sites of this function. If just one such site cannot be unCurried] (because not all the arguments are available or because some other function that cannot be unCurried can also be called from that site) then the function cannot be unCurried. This all-or-nothing approach can be relaxed if we allow both the original and unCurried forms of the function to co-exist. This does not require a significant increase in code size as we can simply introduce wrapper functions. For example, if a program originally contains a function definition

```
fun f x y = e
```

then we might rewrite this to

```
fun fc (x,y) = e;
fun f x y = fc(x,y)
```

and use `fc` wherever possible.

We can increase unCurrying by introducing η -expansions and β -reductions. Consider again the original function `f` above. If this occurs in a context such as

```
g(f a)
```

then we could rewrite this via η -expansion as

```
g(fn y => (f a y))
```

allowing us to unCurry the application of `f`. Similarly, by inlining a function call we can increase the possibilities of unCurrying. Given the function definition

```
fun h z = z b
```

and the expression `h(f a)`, inlining (β -reduction) yields the expression `(f a b)` that can be unCurried. While these examples are trivial, they illustrate the effect that simple transformations can have on unCurrying. Further exploration of transformations and their application might yield better practical results for unCurrying.

Completeness of the inference system. While we have shown completeness of our algorithm with respect to the inference system for typed unCurrying, we have not demonstrated a measure of completeness for the inference system itself. To do this, we likely need a denotational description of unCurrying such that for any term e , $\llbracket e \rrbracket$ denotes the set of all typable terms that are unCurried forms of e . Our belief is that our system can specify all possible unCurrying of an expression.

Recursion and other language constructs. The current work treats only the core λ -calculus. The inclusion of additional programming language features is a logical next step. We have already considered the impact of adding recursion and our analysis can easily be extended

to handle this. If $\mu f.e$ is the syntax for a recursive function, then the unCurry translation can be extended with the following rule:

$$\frac{\Gamma\{f : \tau\} \triangleright e : \tau \Rightarrow m}{\Gamma \triangleright \mu f.e : \tau \Rightarrow \mu f.m} \text{ (rec)}$$

Pairs and other data structures are also easily supported. The issue of modules and separate compilation is of pragmatic interest. Currently, we support it by simply requiring that the signature or interface of a module not change. This prohibits unCurrying of functions declared as part of the interface. A possible extension is to provide both the original version of a function and its unCurried form, and provide auxiliary information to the compiler so that other modules, when compiled, can use either form. This relates to some of the practical issues raised above.

Arity raising, lambda lifting and closure conversion. The current work on unCurrying is closely related to other operations or transformations that focus on the structure of functions and their calling conventions and protocols. We have studied arity raising [10] and closure conversion [8], using techniques similar to the ones used for unCurrying. Arity raising and unCurrying both directly address the parameter structure of functions and function calls. Having studied each individually, we would like to study their interaction and joint specification. Lambda lifting [17] is another transformation that has not been given a formal, declarative specification with proofs of correctness. Instead, descriptions of lambda lifting are based on algorithmic descriptions. We would like to give this operation, and the related operations of lambda-dropping [5] and let-floating [23] high-level, declarative specifications, in the same style as the current work. Having a common framework for all these tasks will allow us to study their interaction.

11. Related work

The notions of Currying and unCurrying functions goes back at least to Schönfinkel [26], but is named in honor of Haskell Curry, who also studied it [2–4] in the context of Combinatory Logic. Though unCurrying has been defined for so many years, and it is relevant to functional programming languages, it has received little previous attention as a problem of study. Descriptions of it have been restricted to first-order uses in which a function can be unCurried only if the function is applied to all its (Curried) arguments at once and neither passed as an argument nor returned as a value. Furthermore, these descriptions consist of only an algorithm, and not a declarative specification.

Instances of unCurrying in recent compilers provide limited formal descriptions. The unCurrying operation performed by the Standard ML of New Jersey compiler is only formalized as a basic transformation on terms in an intermediate abstract syntax [1]. Tarditi has observed that the strategy presented there is not guaranteed to unCurry recursive functions of more than two arguments because the strategy is underspecified [28]. Tarditi gives an algorithm for unCurrying, working with the B-normal forms of the TIL compiler [29], but does not provide a precise description of how much unCurrying can be performed, even

though he only considers first-order unCurrying. Leroy has argued that code almost as efficient as that for unCurried functions can be obtained by carefully analyzing the structure of programs and avoiding the overhead of the intermediate function calls/returns [18].

Our work falls under the more general study of compile-time analysis and optimization of higher-order languages. Most of the work in this area is either type-based or flow-based. Other type-based work includes region analysis [31] and closure conversion [21]. Like our work, region analysis uses annotated types to convey properties of programs that can then be exploited by translating the program. They also use deductive systems to give a high-level and declarative specification of the analysis and transformation, from which they separate the inference and translations phases, yielding an efficient algorithm. The work on closure conversion uses types to capture information about the structure of closures, while preserving the typeability of terms. This work serves more to understand the correctness of closure conversion, than to establish new optimizations or translations on programs. All work in this area exploits the notion of type to describe the property of an expression and the constraint that some expressions must have the same type. Unification can be used in algorithms to provide efficient solutions to the constraints generated by type systems.

Work on flow-based analysis includes control-flow analysis [27], set-based analysis [15], and again closure conversion [32]. Flow analysis typically provides more precise information about an expression or program point. Where type systems characterize an expression by a type, flow analysis can characterize an expression by an approximation of the set of values to which the expression can evaluate. This greater detail typically comes at a greater cost in terms of the complexity of the algorithms. For the case of unCurrying, we do not require this detailed information. Types suffice in providing enough information to support the translation.

12. Conclusion

The unCurrying of functions is a conceptually simple idea that has existed for decades and finds application in the compilation of higher-order functional languages. Despite these two facts, only limited algorithms and no rigorous study had previously been offered for it. We have presented a formal specification of higher-order unCurrying and developed a practical algorithm, based on this specification, for unCurrying both typed and untyped λ -terms. These inference systems provide a general framework for reasoning about unCurrying, independent of a particular algorithm. They also support a richer form of unCurrying than is currently found in compilers for functional languages. These specifications also facilitate the correctness proofs for unCurrying algorithms and provide a basis from which related operations on higher-order languages might be studied.

Acknowledgments

We thank David Liben-Nowell and the referees for comments on earlier versions of the paper.

Appendix A: Proofs of some theorems

Proof: The proof is by induction on the structure of $\Xi :: \Gamma \triangleright e : \tau \Rightarrow m$. We demonstrate how to construct deductions $\mathcal{S} :: |\Gamma| \triangleright_s e : |\tau|$ and $\mathcal{T} :: \|\Gamma\| \triangleright_t m : \|\tau\|$. We consider just the cases for λ -abstraction and application, as these are the relevant cases. (The variable case follows trivially and the let-expression case is straightforward.)

1. Ξ is

$$\frac{\begin{array}{c} \Xi_0 \\ \Gamma \triangleright e_0 : \tau \rightarrow_{\epsilon} \tau' \Rightarrow m_0 \end{array} \quad \begin{array}{c} \Xi_1 \\ \Gamma \triangleright e_1 : \tau \Rightarrow m_1 \end{array}}{\Gamma \triangleright e_0 @ e_1 : \tau' \Rightarrow m_0 @ [m_1]} \text{ (u-app-}\epsilon\text{)}$$

By induction on Ξ_0 and Ξ_1 , we have

$$\begin{aligned} \mathcal{S}_0 &:: |\Gamma| \triangleright_s e_0 : |\tau \rightarrow_{\epsilon} \tau'| \\ \mathcal{T}_0 &:: \|\Gamma\| \triangleright_t m_0 : \|\tau \rightarrow_{\epsilon} \tau'\| \\ \mathcal{S}_1 &:: |\Gamma| \triangleright_s e_1 : |\tau| \\ \mathcal{T}_1 &:: \|\Gamma\| \triangleright_t m_1 : \|\tau\| \end{aligned}$$

Observe that $|\tau \rightarrow_{\epsilon} \tau'| = |\tau| \rightarrow |\tau'|$ and $\|\tau \rightarrow_{\epsilon} \tau'\| = \|\tau\| \rightarrow \|\tau'\|$. We can construct \mathcal{S} and \mathcal{T} as

$$\frac{\begin{array}{c} \mathcal{S}_0 \\ |\Gamma| \triangleright_s e_0 : |\tau| \rightarrow |\tau'| \end{array} \quad \begin{array}{c} \mathcal{S}_1 \\ |\Gamma| \triangleright_s e_1 : |\tau| \end{array}}{|\Gamma| \triangleright_s e_0 @ e_1 : |\tau'|} \text{ (app}_s\text{)}$$

and

$$\frac{\begin{array}{c} \mathcal{T}_0 \\ \|\Gamma\| \triangleright_t m_0 : \|\tau\| \rightarrow \|\tau'\| \end{array} \quad \begin{array}{c} \mathcal{T}_1 \\ \|\Gamma\| \triangleright_t m_1 : \|\tau\| \end{array}}{\|\Gamma\| \triangleright_t m_0 @ [m_1] : \|\tau'\|} \text{ (app}_t\text{)}.$$

2. Ξ is

$$\frac{\begin{array}{c} \Xi_0 \\ \Gamma \triangleright e_0 : \tau \rightarrow_{\uparrow} \tau' \rightarrow_{\varphi} \tau'' \Rightarrow m_0 \end{array} \quad \begin{array}{c} \Xi_1 \\ \Gamma \triangleright e_1 : \tau \Rightarrow m_1 \end{array}}{\Gamma \triangleright e_0 @ e_1 : \tau' \rightarrow_{\varphi} \tau'' \Rightarrow m_0 @_{\uparrow} m_1} \text{ (u-app-}\uparrow\text{)}.$$

By induction on Ξ_0 and Ξ_1 , we have

$$\begin{aligned} \mathcal{S}_0 &:: |\Gamma| \triangleright_s e_0 : |\tau \rightarrow_{\uparrow} \tau' \rightarrow_{\varphi} \tau''| \\ \mathcal{T}_0 &:: \|\Gamma\| \triangleright_t m_0 : \|\tau \rightarrow_{\uparrow} \tau' \rightarrow_{\varphi} \tau''\| \\ \mathcal{S}_1 &:: |\Gamma| \triangleright_s e_1 : |\tau| \\ \mathcal{T}_1 &:: \|\Gamma\| \triangleright_t m_1 : \|\tau\| \end{aligned}$$

Observe that $|\tau \rightarrow_{\uparrow} \tau' \rightarrow_{\varphi} \tau''| = |\tau| \rightarrow |\tau' \rightarrow_{\varphi} \tau''|$. Assume $\|\tau' \rightarrow_{\varphi} \tau''\| = [\tau_1, \dots, \tau_n] \rightarrow \tau_0$; then $\|\tau \rightarrow_{\uparrow} \tau' \rightarrow_{\varphi} \tau''\| = [\|\tau\|, \tau_1, \dots, \tau_n] \rightarrow \tau_0$. Hence, we can construct \mathcal{S} and \mathcal{T} as

$$\frac{|\Gamma| \triangleright_s e_0 : |\tau| \rightarrow |\tau' \rightarrow_{\varphi} \tau''| \quad |\Gamma| \triangleright_s e_1 : |\tau|}{|\Gamma| \triangleright_s e_0 @ e_1 : |\tau' \rightarrow_{\varphi} \tau''|} \text{ (app}_s\text{)}$$

and

$$\frac{\|\Gamma\| \triangleright_t m_0 : [\|\tau\|, \tau_1, \dots, \tau_n] \rightarrow \tau_0 \quad \|\Gamma\| \triangleright_t m_1 : \|\tau\|}{\|\Gamma\| \triangleright_t m_0 @_{\uparrow} m_1 : \tau_1, \dots, \tau_n \rightarrow \tau_0} \text{ (app}_{\uparrow t}\text{)}$$

using the rule introduced for handling terms constructed from '@_↑'.

3. Ξ is

$$\frac{\Xi_1 \quad \Gamma\{x : \tau\} \triangleright e_1 : \tau' \Rightarrow m_1}{\Gamma \triangleright \lambda x. e_1 : \tau \rightarrow_{\epsilon} \tau' \Rightarrow \lambda[x]. m_1} \text{ (u-abs)}.$$

By induction on Ξ_1 we have

$$\begin{aligned} \mathcal{S}_1 &:: |\Gamma\{x : \tau\}| \triangleright_s e_1 : |\tau'| \\ \mathcal{T}_1 &:: \|\Gamma\{x : \tau\}\| \triangleright_t m_1 : \|\tau'\| \end{aligned}$$

Observe that $|\Gamma\{x : \tau\}| = |\Gamma\{x : |\tau|\}|$ and $\|\Gamma\{x : \tau\}\| = \|\Gamma\{x : \|\tau\|\}\|$. From \mathcal{S}_1 and \mathcal{T}_1 we can construct \mathcal{S} and \mathcal{T} :

$$\frac{\mathcal{S}_1 \quad |\Gamma\{x : |\tau|\}| \triangleright_s e_1 : |\tau'|}{|\Gamma| \triangleright_s \lambda x. e_1 : |\tau| \rightarrow |\tau'|} \text{ (abs}_s\text{)}$$

and

$$\frac{\mathcal{T}_1 \quad \|\Gamma\{x : \|\tau\|\}\| \triangleright_t m_1 : \|\tau'\|}{\|\Gamma\| \triangleright_t \lambda[x]. m_1 : [\|\tau\|] \rightarrow \|\tau'\|} \text{ (abs}_{1t}\text{)}$$

which are the required deductions because $|\tau| \rightarrow |\tau'| = |\tau \rightarrow_{\epsilon} \tau'|$ and $[\|\tau\|] \rightarrow \|\tau'\| = \|\tau \rightarrow_{\epsilon} \tau'\|$.

4. Ξ is

$$\frac{\Xi_1 \quad \Gamma\{x_1 : \tau\} \triangleright \lambda x_2. e_1 : \tau' \rightarrow_{\varphi} \tau'' \Rightarrow \lambda[x_2, \dots, x_n]. m_1}{\Gamma \triangleright \lambda x_1. \lambda x_2. e_1 : \tau \rightarrow_{\uparrow} \tau' \rightarrow_{\varphi} \tau'' \Rightarrow \lambda[x_1, x_2, \dots, x_n]. m_1} \text{ (u-abs-}\uparrow\text{)}.$$

By induction on Ξ_1 we have

$$\begin{aligned} \mathcal{S}_1 &:: |\Gamma\{x_1 : \tau\}| \triangleright_s \lambda x_2. e_1 : |\tau' \rightarrow_\varphi \tau''| \\ \mathcal{T}_1 &:: \|\Gamma\{x_1 : \tau\}\| \triangleright_t \lambda[x_2, \dots, x_n]. m_1 : \|\tau' \rightarrow_\varphi \tau''\| \end{aligned}$$

Observe that $|\tau \rightarrow_{\uparrow} \tau' \rightarrow_\varphi \tau''| = |\tau| \rightarrow |\tau' \rightarrow_\varphi \tau''|$. Assume $\|\tau' \rightarrow_\varphi \tau''\| = [\tau_1, \dots, \tau_n] \rightarrow \tau_0$; then $\|\tau \rightarrow_{\uparrow} \tau' \rightarrow_\varphi \tau''\| = [\|\tau\|, \tau_1, \dots, \tau_n] \rightarrow \tau_0$. Hence, from \mathcal{S}_1 and \mathcal{T}_1 we can construct \mathcal{S} and \mathcal{T} :

$$\frac{|\Gamma\{x_1 : |\tau|\}| \triangleright_s \lambda x_2. e_1 : |\tau' \rightarrow_\varphi \tau''|}{|\Gamma| \triangleright_s \lambda x_1. \lambda x_2. e_1 : |\tau| \rightarrow |\tau' \rightarrow_\varphi \tau''|} \text{ (abs}_s\text{)}$$

and

$$\frac{\|\Gamma\{x_1 : \|\tau\|\}\| \triangleright_t \lambda[x_2, \dots, x_n]. m_1 : [\tau_1, \dots, \tau_n] \rightarrow \tau_0}{\|\Gamma\| \triangleright_t \lambda[x_1, x_2, \dots, x_n]. m_1 : [\|\tau\|, \tau_1, \dots, \tau_n] \rightarrow \tau_0} \text{ (absn}_t\text{)}.$$

5. Ξ is

$$\frac{\Xi_0 \quad \Gamma \triangleright e_0 @ e_1 : \tau \Rightarrow (m_0 @_{\uparrow} m_1) @ [m_2, \dots, m_n]}{\Gamma \triangleright e_0 @ e_1 : \tau \Rightarrow m_0 @ [m_1, m_2, \dots, m_n]} \text{ (u-app-@}_{\uparrow}\text{)}.$$

By induction on Ξ_0 we have

$$\begin{aligned} \mathcal{S} &:: |\Gamma| \triangleright_s e_0 @ e_1 : |\tau| \\ \mathcal{T}' &:: \|\Gamma\| \triangleright_t (m_0 @_{\uparrow} m_1) @ [m_2, \dots, m_n] : \|\tau\| \end{aligned}$$

Observe that \mathcal{T}' must be of the form

$$\frac{\frac{\|\Gamma\| \triangleright_t m_0 : [\tau_1, \tau_2, \dots, \tau_n] \rightarrow \|\tau\| \quad \|\Gamma\| \triangleright_t m_1 : \tau_1}{\|\Gamma\| \triangleright_t (m_0 @_{\uparrow} m_1) : [\tau_2, \dots, \tau_n] \rightarrow \|\tau\|} \quad \|\Gamma\| \triangleright_t m_i : \tau_i \quad \text{for } i \in 2..n}{\|\Gamma\| \triangleright_t (m_0 @_{\uparrow} m_1) @ [m_2, \dots, m_n] : \|\tau\|}}$$

From this we can construct \mathcal{T} as

$$\frac{\|\Gamma\| \triangleright_t m_0 : [\tau_1, \tau_2, \dots, \tau_n] \rightarrow \|\tau\| \quad \|\Gamma\| \triangleright_t m_i : \tau_i \quad \text{for } i \in 1..n}{\|\Gamma\| \triangleright_t m_0 @ [m_1, m_2, \dots, m_n] : \|\tau\|}. \quad \square$$

Proof: For part (1) we assume given deductions $\Xi :: e : \tau \Rightarrow m$ and $\Pi :: e \hookrightarrow_s v$, and proceed by induction on Π . We demonstrate how to construct deductions $\Delta :: m \hookrightarrow_t w$ and $\Xi' :: v : \tau \Rightarrow w$. We show only the cases for abstraction and application as, again, they are the interesting ones.

1. Π is of the form

$$\overline{\lambda x_1. e_1 \hookrightarrow_s \lambda x_1. e_1.}$$

Then m is of the form $\lambda[x_1, \dots, x_n]. m_1$ (for some $n \geq 1$), we can construct $\Delta :: m \hookrightarrow_t m$ and so $\Xi' = \Xi$.

2. Π is of the form

$$\frac{\begin{array}{ccc} \Pi_0 & \Pi_1 & \Pi_2 \\ e_0 \hookrightarrow_s \lambda x_1. e' & e_1 \hookrightarrow_s v_1 & e'[v_1/x_1] \hookrightarrow_s v \end{array}}{e_0 @ e_1 \hookrightarrow_s v}.$$

Then we have two possible cases for the deduction Ξ .

(a) Ξ is

$$\frac{\begin{array}{ccc} \Xi_0 & & \Xi_1 \\ e_0 : \tau_1 \rightarrow_\epsilon \tau \Rightarrow (m_0 @_{\uparrow} m_1 @_{\uparrow} \dots @_{\uparrow} m_{n-1}) & e_1 : \tau_1 \Rightarrow m_n & \text{(u-app-}\epsilon\text{)} \\ \hline e_0 @ e_1 : \tau \Rightarrow (m_0 @_{\uparrow} m_1 @_{\uparrow} \dots @_{\uparrow} m_{n-1}) @ [m_n] & & \text{(u-app-}@_{\uparrow}) \\ \vdots & & \\ \hline e_0 @ e_1 : \tau \Rightarrow m_0 @ [m_1, \dots, m_n] & & \text{(u-app-}@_{\uparrow}) \end{array}}$$

for $n \geq 1$ and in which ‘ \vdots ’ represents $n - 1$ occurrences of (u-app- $@_{\uparrow}$).

We proceed by induction on the number k of occurrences of rule (u-app- $@_{\uparrow}$) at the root of Ξ .

i) $k = 0$. Then Ξ is

$$\frac{\begin{array}{ccc} \Xi_0 & & \Xi_1 \\ e_0 : \tau_1 \rightarrow_\epsilon \tau \Rightarrow m_0 & e_1 : \tau_1 \Rightarrow m_1 & \text{(u-app-}\epsilon\text{)} \\ \hline e_0 @ e_1 : \tau \Rightarrow (m_0 @ [m_1]) & & \end{array}}$$

By induction on Π_0 and Ξ_0 , and Π_1 and Ξ_1 , we have

$$\begin{array}{l} \Delta_0 :: m_0 \hookrightarrow_t w_0 \\ \Xi'_0 :: \lambda x_1. e' : \tau_1 \rightarrow_\epsilon \tau \Rightarrow w_0 \\ \Delta_1 :: m_1 \hookrightarrow_t w_1 \\ \Xi'_1 :: v_1 : \tau_1 \Rightarrow w_1. \end{array}$$

Then Ξ'_0 must be of the form

$$\frac{\Xi''}{\forall x_1 \forall x'_1 (x_1 : \tau_1 \Rightarrow x'_1 \supset e' : \tau \Rightarrow m')} \text{(u-abs)} \\ \lambda x_1. e' : \tau_1 \rightarrow_\epsilon \tau \Rightarrow \lambda[x'_1]. m'$$

($w_0 = \lambda[x'_1]. m'$). Applying the substitution property (from Section 2) to Ξ'' and Ξ'_1 we can construct

$$\Xi'_2 :: e'[v_1/x_1] : \tau \Rightarrow m'[w_1/x'_1].$$

By induction on Π_2 and Ξ'_2 we then have that there exists a w such that

$$\begin{aligned}\Delta_2 &:: m'[w_1/x'_1] \hookrightarrow_t w \\ \Xi'_2 &:: v : \tau \Rightarrow w.\end{aligned}$$

Hence, Δ is

$$\frac{\begin{array}{ccc} \Delta_0 & \Delta_1 & \Delta_2 \\ m_0 \hookrightarrow_t \lambda[x'_1].m' & m_1 \hookrightarrow_t w_1 & m'[w_1/x'_1] \hookrightarrow_t w \end{array}}{m_0 @ [m_1] \hookrightarrow_t w}.$$

ii) We assume the property holds for some $k \geq 0$ and show that it holds for $k + 1$. In this case, Ξ is

$$\frac{\begin{array}{c} \Xi_0 \\ e_0 @ e_1 : \tau \Rightarrow (m_0 @_{\uparrow} m_1) @ [m_2, \dots, m_n] \end{array}}{e_0 @ e_1 : \tau \Rightarrow m_0 @ [m_1, m_2, \dots, m_n]} \text{ (u-app-@}_{\uparrow}\text{)}$$

in which Ξ_0 has $k (= n - 2)$ occurrences of rule (u-app-@_↑) and so we can apply the induction hypothesis for k (using Ξ_0 and Π), yielding

$$\begin{aligned}\Delta' &:: (m_0 @_{\uparrow} m_1) @ [m_2, \dots, m_n] \hookrightarrow_t w \\ \Xi' &:: v : \tau \Rightarrow w\end{aligned}$$

Then Δ' must be of the form

$$\frac{\begin{array}{ccc} \Delta_2 &:: m_2 \hookrightarrow_t w_2 & \\ & \dots & \\ \Delta_0 & \Delta_1 & \Delta_n \\ m_0 \hookrightarrow_t \lambda[x'_1, x'_2, \dots, x'_n].m' & m_1 \hookrightarrow_t w_1 & m_n \hookrightarrow_t w_n \\ \hline (m_0 @_{\uparrow} m_1) \hookrightarrow_t \lambda[x'_2, \dots, x'_n].(m'[w_1/x'_1]) & \Delta_{n+1} &:: m'' \hookrightarrow_t w \end{array}}{(m_0 @_{\uparrow} m_1) @ [m_2, \dots, m_n] \hookrightarrow_t w}$$

in which $m'' = m'[w_1/x'_1, \dots, w_n/x'_n]$. From this we can construct Δ as

$$\frac{\begin{array}{ccc} \Delta_2 &:: m_2 \hookrightarrow_t w_2 & \\ & \dots & \\ \Delta_0 & \Delta_1 & \Delta_n \\ m_0 \hookrightarrow_t \lambda[x'_1, x'_2, \dots, x'_n].m' & m_1 \hookrightarrow_t w_1 & m_n \hookrightarrow_t w_n \\ \hline m_0 @ [m_1, m_2, \dots, m_n] & \Delta_{n+1} &:: m'' \hookrightarrow_t w \end{array}}{m_0 @ [m_1, m_2, \dots, m_n] \hookrightarrow_t w}$$

(b) Ξ is

$$\frac{\begin{array}{ccc} \Xi_0 & \Xi_1 \\ e_0 : \tau_1 \rightarrow_{\uparrow} \tau \Rightarrow m_0 & e_1 : \tau_1 \Rightarrow m_1 \end{array}}{e_0 @ e_1 : \tau \Rightarrow m_0 @_{\uparrow} m_1} \text{ (u-app-}\uparrow\text{)}$$

By induction on Π_0 and Ξ_0 , and Π_1 and Ξ_1 , we have

$$\begin{aligned}\Delta_0 &:: m_0 \hookrightarrow_t w_0 \\ \Xi'_0 &:: \lambda x_1. e' : \tau_1 \rightarrow_{\uparrow} \tau \Rightarrow w_0 \\ \Delta_1 &:: m_1 \hookrightarrow_t w_1 \\ \Xi'_1 &:: v_1 : \tau_1 \Rightarrow w_1.\end{aligned}$$

Then Ξ'_0 must be of the form

$$\frac{\Xi''}{\frac{\forall x_1 \forall x'_1 (x_1 : \tau_1 \Rightarrow x'_1 \supset \lambda x_2. e'' : \tau \Rightarrow \lambda [x'_2, \dots, x'_n]. m')}{\lambda x_1. \lambda x_2. e'' : \tau_1 \rightarrow_{\uparrow} \tau \Rightarrow \lambda [x'_1, x'_2, \dots, x'_n]. m'}}$$

($e' = \lambda x_2. e''$ and $w_0 = \lambda [x'_1, x'_2, \dots, x'_n]. m'$). Hence, Δ is

$$\frac{\frac{\Delta_0}{m_0 \hookrightarrow_t \lambda [x'_1, x'_2, \dots, x'_n]. m'} \quad \frac{\Delta_1}{m_1 \hookrightarrow_t w_1}}{m_0 @_{\uparrow} m_1 \hookrightarrow_t \lambda [x'_2, \dots, x'_n]. (m' [w_1/x'_1])}$$

Applying the substitution property to Ξ'' and Ξ'_1 we can construct

$$\Xi' :: \lambda x_2. (e'' [v_1/x_1]) : \tau \Rightarrow \lambda [x'_2, \dots, x'_n]. (m' [w_1/x'_1]).$$

This is the required deduction because $v = \lambda x_2. (e'' [v_1/x_1])$ and $w = \lambda [x'_2, \dots, x'_n]. (m' [w_1/x'_1])$.

For the proof of part (2) we begin by introducing a metric on deductions $\Delta :: m \hookrightarrow_t w$. Let $|\Delta| = (k_n, \dots, k_1)$ such that $k_i =$ the number of application rules occurring in Δ in which the application in the consequent has arity i . We write $\Delta < \Delta'$ if $|\Delta| < |\Delta'|$ under the standard lexicographical ordering. Observe that this is a well-founded order.

The proof then proceeds by well-founded induction on $\Delta :: m \hookrightarrow_t w$. We demonstrate how to construct deductions $\Pi :: e \hookrightarrow_s v$ and $\Xi' :: v : \tau \Rightarrow w$.

1. Δ is of the form $\frac{\lambda [x'_1, x'_2, \dots, x'_n]. m_1 \hookrightarrow_t \lambda [x'_1, x'_2, \dots, x'_n]. m_1}{\lambda x_1. e_1}$. Then e is of the form $\lambda x_1. e_1$ and we can construct $\Pi :: e \hookrightarrow_s e$ and so $\Xi' = \Xi$.
2. Δ is of the form

$$\frac{\frac{\Delta_0}{m_0 \hookrightarrow_t \lambda [x'_1, \dots, x'_n]. m'} \quad \frac{\Delta_1}{m_1 \hookrightarrow_t w_1} \cdots \frac{\Delta_n}{m_n \hookrightarrow_t w_n} \quad \frac{\Delta_{n+1}}{m' [w_1/x'_1, \dots, w_n/x'_n] \hookrightarrow_t w}}{m_0 @ [m_1, \dots, m_n] \hookrightarrow_t w}$$

Observe that $\Delta_i < \Delta$ for $0 \leq i \leq n + 1$.

Then we have two cases for the structure of Ξ .

a) Ξ is

$$\frac{\frac{\Xi_0}{e_0 : \tau_1 \rightarrow_{\epsilon} \tau \Rightarrow m_0} \quad \frac{\Xi_1}{e_1 : \tau_1 \Rightarrow m_1}}{e_0 @ e_1 : \tau \Rightarrow (m_0 @ [m_1])} \text{ (u-app-}\epsilon\text{)}$$

Observe that in this case $n = 1$.

By induction on Δ_0 and Ξ_0 , and Δ_1 and Ξ_1 , we have

$$\begin{aligned} \Pi_0 &:: e_0 \hookrightarrow_s v_0 \\ \Xi'_0 &:: v_0 : \tau_1 \rightarrow_{\epsilon} \tau \Rightarrow \lambda [x'_1]. m' \\ \Pi_1 &:: e_1 \hookrightarrow_s v_1 \\ \Xi'_1 &:: v_1 : \tau_1 \Rightarrow w_1. \end{aligned}$$

Then Ξ'_0 must be of the form

$$\frac{\Xi''}{\lambda x_1. e' : \tau_1 \rightarrow_\epsilon \tau \Rightarrow \lambda[x'_1]. m'} \quad (\text{u-abs})$$

($v_0 = \lambda x_1. e'$). Applying the substitution property to Ξ'' and Ξ'_1 we can construct

$$\Xi'_2 :: e'[v_1/x_1] : \tau \Rightarrow m'[w_1/x'_1].$$

By induction on Δ_2 and Ξ'_2 we then have that there exists a v such that

$$\begin{aligned} \Pi_2 &:: e'[v_1/x_1] \hookrightarrow_s v \\ \Xi' &:: v : \tau \Rightarrow w \end{aligned}$$

Hence, Π is

$$\frac{\begin{array}{ccc} \Pi_0 & \Pi_1 & \Pi_2 \\ e_0 \hookrightarrow_s \lambda x_1. e' & e_1 \hookrightarrow_s v_1 & e'[v_1/x_1] \hookrightarrow_s v \end{array}}{e_0 @ e_1 \hookrightarrow_s v}.$$

b) Ξ is

$$\frac{\Xi_1}{e_0 @ e_1 : \tau \Rightarrow (m_0 @_{\uparrow} m_1) @ [m_2, \dots, m_n]} \quad (\text{u-app-}@_{\uparrow})$$

$$e_0 @ e_1 : \tau \Rightarrow m_0 @ [m_1, \dots, m_n]$$

From Δ we can construct Δ' :

$$\frac{\begin{array}{ccc} \Delta_0 & \Delta_1 & m_2 \xrightarrow{\Delta_2} w_2 \cdots \\ m_0 \hookrightarrow_t \lambda[x'_1, \dots, x'_n]. m' & m_1 \hookrightarrow_t w_1 & m_n \xrightarrow{\Delta_n} w_n \\ \hline (m_0 @_{\uparrow} m_1) \hookrightarrow_t \lambda[x'_2, \dots, x'_n]. (m'[w_1/x'_1]) & & m'[w_1/x'_1, \dots, w_n/x'_n] \hookrightarrow_t w \end{array}}{(m_0 @_{\uparrow} m_1) @ [m_2, \dots, m_n] \hookrightarrow_t w} \quad \Delta_{n+1}$$

Observe that $\Delta' < \Delta$ and so by well-founded induction on Δ' (using Ξ_1) we have there exists some v such that

$$\begin{aligned} \Pi &:: e_1 @ e_2 \hookrightarrow_s v \\ \Xi' &:: v : \tau \Rightarrow w. \end{aligned}$$

3. Δ is of the form

$$\frac{\begin{array}{ccc} \Delta_0 & \Delta_1 & \\ m_0 \hookrightarrow_t \lambda[x'_1, x'_2, \dots, x'_n]. m' & m_1 \hookrightarrow_t w_1 & \\ \hline m_0 @_{\uparrow} m_1 \hookrightarrow_t \lambda[x'_2, \dots, x'_n]. (m'[w_1/x'_1]) & & \end{array}}{(t\text{-app-}\uparrow)}.$$

Then Ξ is

$$\frac{\Xi_0 \quad \Xi_1}{e_0 : \tau \rightarrow_{\uparrow} \tau' \rightarrow_{\varphi} \tau'' \Rightarrow m_0 \quad e_1 : \tau \Rightarrow m_1} \quad (\text{u-app-}@_{\uparrow})$$

$$e_0 @ e_1 : \tau' \rightarrow_{\varphi} \tau'' \Rightarrow m_0 @_{\uparrow} m_1$$

By induction on Δ_0 and Ξ_0 , and Δ_1 and Ξ_1 , we have

$$\begin{aligned} \Pi_0 &:: e_0 \hookrightarrow_s v_0 \\ \Xi'_0 &:: v_0 : \tau \rightarrow_{\uparrow} \tau' \rightarrow_{\varphi} \tau'' \Rightarrow \lambda[x'_1, x'_2, \dots, x'_n].m' \\ \Pi_1 &:: e_1 \hookrightarrow_s v_1 \\ \Xi'_1 &:: v_1 : \tau \Rightarrow w_1. \end{aligned}$$

Then Ξ'_0 must be of the form

$$\frac{\Xi'' \quad \forall x_1 \forall x'_1 (x_1 : \tau \Rightarrow x'_1 \supset \lambda x_2. e' : \tau' \rightarrow_{\varphi} \tau'' \Rightarrow \lambda[x'_2, \dots, x'_n].m')}{\lambda x_1. \lambda x_2. e' : \tau \rightarrow_{\uparrow} \tau' \rightarrow_{\varphi} \tau'' \Rightarrow \lambda[x'_1, x'_2, \dots, x'_n].m'}$$

($v_0 = \lambda x_1. \lambda x_2. e'$). Applying the substitution property to Ξ'' and Ξ'_1 we can construct

$$\Xi' :: \lambda x_2. (e'[v_1/x_1]) : \tau' \rightarrow_{\varphi} \tau'' \Rightarrow \lambda[x'_2, \dots, x'_n].(m'[w_1/x'_1])$$

and we can construct Π as

$$\frac{\frac{\Pi_0 \quad \Pi_2}{e_0 \hookrightarrow_s \lambda x_1. \lambda x_2. e' \quad e_1 \hookrightarrow_s v_1 \quad \lambda x_2. (e'[v_1/x_1]) \hookrightarrow_s \lambda x_2. (e'[v_1/x_1])}}{e_0 @ e_1 \hookrightarrow_s \lambda x_2. (e'[v_1/x_1])}}{\quad} \quad \square$$

Note

1. A preliminary version of this paper appeared in the Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages [11].

References

1. Appel, A.W. *Compiling with Continuations*. Cambridge University Press, 1992.
2. Curry, H.B. Grundlagen der Kombinatorischen Logik. *American Journal of Mathematics* **52** (1930) 509–536 and 789–834.
3. Curry, H.B. and Feys, R. *Combinatory Logic, Vol. I*. North Holland, 1958.
4. Curry, H.B., Hindley, J., and Seldin, J.P. *Combinatory Logic, Vol. II*. North Holland, 1972.
5. Danvy, O. and Schultz, U.P. Lambda dropping: Transforming recursive equations into programs with block structure. Technical Report RS-99-27, BRICS, in *Theoretical Computer Science*, Vol. 248, No. 1–2, November 2000.
6. Gentzen, G. Investigations into logical deduction. In *The Collected Papers of Gerhard Gentzen*, M. Szabo (Ed.). North-Holland Publishing Co., 1969, pp. 68–131.
7. Gomard, C.K. Partial type inference for untyped functional programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, 1990, pp. 282–287.
8. Hannan, J. Type systems for closure conversions. In *Participants' Proceedings of the Workshop on Types for Program Analysis*, H.R. Nielson and K.L. Solberg (Eds.), 1995, pp. 48–62.
9. Hannan, J. A type-based escape analysis for functional languages. *Journal of Functional Programming* **8**(3) (1998) 239–273.

10. Hannan, J. and Hicks, P. Higher-order arity raising. In *Proceedings of the Third International Conference on Functional Programming*, P. Hudak and C. Queinnec (Eds.). 1998, pp. 27–38.
11. Hannan, J. and Hicks, P. Higher-order unCurrying. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1998, pp. 1–11.
12. Hannan, J., Hicks, P., and Liben-Nowell, D. A lifetime analysis for higher-order languages. Technical Report CSE-97-014, Penn State University, 1997.
13. Hannan, J. and Pfenning, F. Compiler verification in LF. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, A. Scedrov (Ed.). 1992, pp. 407–418.
14. Harper, R., Honsell, F., and Plotkin, G. A framework for defining logics. *Journal of the ACM* **40**(1) (1993) 143–184. A preliminary version appeared in *Symposium on Logic in Computer Science*, pp. 194–204, June 1987.
15. Heintze, N. Set-based program analysis. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, 1992.
16. Jim, T. Rank 2 type systems and recursive definitions. Technical Memorandum MIT/LCS/TM-531, Massachusetts Institute of Technology, Laboratory for Computer Science, 1995.
17. Johansson, T. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, J.-P. Jouannaud (Ed.). Vol. 201 of Lecture Notes in Computer Science. 1985, pp. 190–203.
18. Leroy, X. The ZINC experiment: An economical implementation of the ML Language. Technical Report 117, INRIA, 1990.
19. Michaylov, S. and Pfenning, F. Natural semantics and some of its meta-theory in Elf. In *Extensions of Logic Programming*, L. Hallnas (Ed.). 1992, pp. 299–344. A preliminary version is available as Technical Report MPI-I-91-211, Max-Planck-Institute for Computer Science, Saarbrücken, Germany, Aug. 1991.
20. Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* **17**(3) (1998) 348–375.
21. Minamide, Y., Morrisett, G., and Harper, R. Typed closure conversion. In *23rd ACM Symposium on Principles of Programming Languages*, St. Petersburg, FL, 1996, pp. 271–283.
22. Nipkow, T. and Prehofer, C. Type reconstruction for type classes. *Journal of Functional Programming* **5**(2) (1995) 201–224.
23. Peyton Jones, S.L., Partain, W., and Santos, A. Let-floating: Moving bindings to give faster programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Languages*, R. Harper (Ed.). 1996, pp. 1–12.
24. Pfenning, F. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, 1989, pp. 313–322.
25. Prawitz, D. *Natural Deduction*. Uppsala: Almqvist & Wiksell, 1965.
26. Schönfinkel, M. Über die Bausteine der Mathematischen Logik. *Mathematische Annalen* **92** (1924) 305–316. English translation in *From Frege to Gödel*, Jan van Heijenoort editor, Harvard University Press, 1967, pp. 355–366.
27. Shivers, O. Control-flow analysis of higher-order languages. Ph.D. Thesis, Carnegie-Mellon University, 1991.
28. Tarditi, D. Design and implementation of code optimizations for a type-directed compiler for standard ML. Ph.D. Thesis, Carnegie Mellon University. Available as CMU CS Technical Report 97-108, 1996.
29. Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, B., and Lee, P. TIL: A Type-directed optimizing compiler for ML. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996, pp. 181–192.
30. Tofte, M. Operational semantics and polymorphic type inference. Ph.D. Thesis, University of Edinburgh, 1987.
31. Tofte, M. and Talpin, J.-P. Implementation of the typed call-by-value λ -calculus using a stack of regions. In *Conf. Rec. 21st ACM Symposium on Principles of Programming Languages*, 1994, pp. 188–201.
32. Wand, M. and Steckler, P. Lightweight closure conversion. *ACM Trans. Program. Lang. Syst.* **19**(1) (1997) 48–86.