

Higher-Order UnCurrying

John Hannan and Patrick Hicks
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802 USA
<http://www.cse.psu.edu/~hannan,~phicks>

In accordance with these definitions, the essential purpose of mathematical logic is the construction of an abstract (or strictly formalized) theory, such that when its fundamental notions are properly interpreted, there ensues an analysis of those universal principles in accordance with which valid thinking goes on.

*Haskell Curry
Professor of Mathematics
Pennsylvania State College
1929*

Abstract

We present a formal specification of unCurrying for a higher-order, functional language with ML-style let-poly-morphism. This specification supports the general unCurrying of functions, even for functions which 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 which 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} , which implements the 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.

1 Introduction

UnCurrying is the transformation of a function $\lambda x.\lambda y.e$ into $\lambda(x,y).e$. While this description captures the essence of the unCurry operation, it does not capture the corresponding

transformations required for the context in which the function occurs. In a higher-order language this context can involve the function as an argument to another function, the function as a value returned by another function, or the function as simply a fragment of a larger function with more parameters, possibly occurring both before and after the parameters x and y . In each case, the eventual use or application of an unCurried function must properly reflect its new form. We refer to unCurrying in such possible contexts as *higher-order UnCurrying*. Algorithms may be provided to perform such unCurrying and any necessitated transformations, but they do not generally provide a suitable framework for studying the problem and proving an implementation correct. We give a formal, declarative specification of higher-order unCurrying in which such reasoning can be done.

We define a deductive system for a higher-order, functional language with ML-style polymorphism, which provides a general description of higher-order unCurrying. We model our approach after that used in the specification of type inference for functional languages. We first introduce *unCurry types*, which generalize simple types by including annotations on function types. These annotations serve a purpose analogous to types in type inference, as they are used to generate constraints about unCurrying among subterms in a term. Additionally, they serve to direct the translation of source terms to unCurried forms. We define a deductive system which infers annotated types for expressions and relates expressions to unCurried forms based on these types. We also give an algorithm, based on Milner's algorithm \mathcal{W} [9], which performs type inference and the unCurry translation. We demonstrate how our algorithm works for several moderately complicated examples. We implemented this algorithm in Standard ML and tested it on these and other, more complicated examples. We prove the correctness of the deductive system with respect to an operational semantics for the source (Curried) and target (unCurried) languages. This proof was performed by hand and simultaneously encoded into the LF type theory [6]. We used the Elf programming system [12] to partially verify the correctness of this encoding.

This work continues our effort to use formal systems to specify and verify compilation techniques for functional languages. In previous work we studied closure conversion [3], escape analysis [4], and live variable analysis [5]. 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 machine verification of proofs.

UnCurrying has received little previous attention as a problem of study. Descriptions of it are restricted to first-order uses in which a function can be unCurried only if it is applied to all its arguments and neither passed as an argument nor returned as a value. Furthermore, these descriptions consist of only an algorithm, and not a declarative specification. In [1], Appel describes the unCurrying operation performed by the Standard ML of New Jersey compiler, giving the basic transformation on terms in an intermediate abstract syntax. Tarditi [13] has observed that the strategy presented there is not guaranteed to unCurry recursive functions of more than two arguments because the strategy is underspecified. Tarditi gives an algorithm for unCurrying, working with the B-normal forms of the TIL compiler [14], but does not provide a precise description of how much unCurrying can be performed, even though he only considers first-order unCurrying. Leroy [7] 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.

The remainder of the paper is organized as follows. In the next section we introduce source and target languages, both based on the typed λ -calculus. We provide traditional type systems and high-level operational semantics for both languages. In Section 3 we define higher-order unCurrying via a deductive system which relates a term with an unCurried form. We give several illustrative examples of the capabilities of this system. In Section 4 we discuss the correctness of the deductive system and outline the structure of its proof. In Section 5 we present an algorithm for unCurrying based on Milner's algorithm \mathcal{W} and outline the proof of its soundness and completeness. In Section 6 we describe a deductive system and algorithm for unCurrying in an untyped language. In Section 7 we discuss extensions and future work, and in Section 8 we summarize our results.

2 Two Typed Languages

Our specification of typed unCurrying relates 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.

$$\begin{aligned} e &::= x \mid \lambda x.e \mid e_1 @ e_2 \mid \text{let } x = e \text{ in } e \\ m &::= x \mid \lambda(x_1, \dots, x_k).m \mid m @ (m, \dots, m) \\ &\quad \mid \text{let } x = m \text{ in } m \end{aligned}$$

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 unCurried language, abstractions and applications have an *arity* $k \geq 1$. If $k = 1$ then we simply write $\lambda x_1.m$ and $m_0 @ m_1$. For notational convenience we use boldface variables to range over tuples. Thus we write $\lambda \mathbf{x}.m$ and $(m_0 @ \mathbf{m})$ letting \mathbf{y} and \mathbf{m} stand for some tuples (x_1, \dots, x_k) ($k \geq 1$) and (m_1, \dots, m_k) ($k \geq 1$), respectively.

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

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

We write $\forall \overline{\alpha}_n. \tau$ as an abbreviation for $\forall \alpha_1 \dots \forall \alpha_n. \tau$.

The types of the target language generalize these by introducing notation for unCurried function types. At the risk of some confusion, we overload τ and σ to range over types and type schemas for these generalized types:

$$\begin{aligned} \tau &::= \alpha \mid (\tau_1 \star \dots \star \tau_n) \rightarrow \tau \\ \sigma &::= \tau \mid \forall \alpha. \sigma \end{aligned}$$

in which $n \geq 1$. The constructor ' \star ' denotes only the tupling of function parameters, which is distinct from a general product constructor ' \times '.

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 $\Gamma \triangleright_s e : \tau$ denote the traditional typing system as given in Figure 1. Let $\Gamma \triangleright_t m : \tau$ denote the variant of this typing system in which types range over unCurried types, as given in Figure 2. In both cases, Γ is a type context mapping term variables to type schemas. We use $FV(\tau)$ to denote the free type variables of a type and generalize this to type contexts, $FV(\Gamma)$.

We define an operational semantics for each of these languages by axiomatizing judgments of the form $e \mapsto_s v$ and $m \mapsto_t w$. To provide a high-level description for these semantics, we use variable substitution rather than environments to manage the binding of variables to values. Thus, in both languages, values are limited to λ -abstractions. This simplifies correctness proofs. The rules are given in Figures 3 and 4.

3 Typed UnCurrying

To provide an abstract, declarative specification of unCurrying, we define a type system which 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 in which unCurried terms can be constructed.

3.1 UnCurry Types

To motivate the form of the judgment and the structure of the inference rules we begin by considering some simple examples of unCurrying and what information needs to be provided to ensure the proper construction of unCurried terms. Consider the K combinator applied 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:

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

$$\begin{array}{c}
\frac{\Gamma(x) \succeq \tau}{\Gamma \triangleright_s x : \tau} \\
\\
\frac{\Gamma \triangleright_s e_1 : \tau_1 \rightarrow \tau \quad \Gamma \triangleright_s e_2 : \tau_1}{\Gamma \triangleright_s e_1 @ e_2 : \tau} \\
\\
\frac{\Gamma\{x : \tau_1\} \triangleright_s e : \tau}{\Gamma \triangleright_s \lambda x.e : \tau_1 \rightarrow \tau} \\
\\
\frac{\Gamma \triangleright_s e_1 : \tau_1 \quad \{\overline{\alpha_n}\} = FV(\tau_1) - FV(\Gamma) \quad \Gamma\{x : \forall \overline{\alpha_n}.\tau_1\} \triangleright_s e_2 : \tau}{\Gamma \triangleright_s \text{let } x = e_1 \text{ in } e_2 : \tau}
\end{array}$$

Figure 1: The Source Type Inference System

$$\begin{array}{c}
\frac{\Gamma(x) \succeq \tau}{\Gamma \triangleright_t x : \tau} \\
\\
\frac{\Gamma \triangleright_t m : (\tau_1 \star \dots \star \tau_n) \rightarrow \tau \quad \Gamma \triangleright_t m_i : \tau_i \text{ for } i \in [1..n]}{\Gamma \triangleright_t m @ (m_1, \dots, m_n) : \tau} \\
\\
\frac{\Gamma\{x_1 : \tau_1, \dots, x_n : \tau_n\} \triangleright_t m : \tau}{\Gamma \triangleright_t \lambda(x_1, \dots, x_n).m : (\tau_1 \star \dots \star \tau_n) \rightarrow \tau} \\
\\
\frac{\Gamma \triangleright_t m_1 : \tau_1 \quad \{\overline{\alpha_n}\} = FV(\tau_1) - FV(\Gamma) \quad \Gamma\{x : \forall \overline{\alpha_n}.\tau_1\} \triangleright_t m_2 : \tau}{\Gamma \triangleright_t \text{let } x = m_1 \text{ in } m_2 : \tau}
\end{array}$$

Figure 2: The Target Type Inference System

When the function being unCurried is immediately applied to its arguments, the process of translating the application is trivial.

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

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

Here, the action of unCurrying K must be communicated to the formal parameter k , and, furthermore, this action 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 argument is unCurried and is applied to two arguments. So unCurrying K yields the following term:

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

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

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

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 these examples illustrate, the kind of information that must be communicated between terms includes:

1. a function has been unCurried;
 2. the formal parameter of a function expects an unCurried function;
 3. the formal parameter of a function expects to be applied to an unCurried function;
- etc.

Complicating matters further, the unCurry operation can tuple any sequence of consecutively occurring parameters of a function. For example, the function

$$\lambda x_1.\lambda x_2.\lambda x_3.\lambda x_4.\lambda x_5.\lambda x_6.e$$

could possibly be unCurried to the term

$$\lambda(x_1, x_2).\lambda x_3.\lambda(x_4, x_5, x_6).e.$$

Given a function $\lambda x_1 \dots \lambda x_k.e$ we refer to the tupling of x_i and x_{i+1} (for $1 \leq i < k$) as the unCurrying of the i^{th} parameter of the function.

Taking the above issues into consideration, we introduce unCurry annotations, types and type schemas. We use φ , τ , and σ , respectively, to range over these.

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

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

Figure 3: Source Operational Semantics

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

Figure 4: Target Operational Semantics

We first informally motivate the structure of types, and then give the deductive system which uses them to specify unCurrying. An annotation is either \uparrow , which indicates that a function's parameter should be unCurried, or ϵ , which indicates that the parameter should not be unCurried. For example, one possible type for the K combinator $\lambda x.\lambda y.x$ is

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

which indicates the following characteristics: the parameter x should be unCurried (with y to form a tuple) and x should accept an unCurried argument. 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, and the arguments e_2 and e_3 should be tupled. 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.

3.2 The Inference System

We axiomatize a relation between a source term e , an unCurry type τ , and a target term m . Similar to the operational semantics, we avoid using an explicit type context in the judgment. Instead we use hypothetical assumptions (via logical implication) to introduce assumptions about variables. Observe that this implies that we do not require a rule for handling variables.

To support the definition of this relation we first extend the syntax of target terms by adding a new form of application:

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

This form of application is used to construct intermediate terms during the unCurrying of applications. A term of the form $(m_0 @_{\uparrow} m_1)$ indicates that m_0 should be applied to a tuple (m_1, m_2) , in which m_2 occurs as the operand in the enclosing application: $((m_0 @_{\uparrow} m_1) @ m_2)$.

To define the unCurry relation we introduce the judgment $e : \tau \Rightarrow m$, in which τ is an unCurry type, and axiomatize it by the rules in Figure 5. We use $'@_{\varphi}'$ to denote $'@'$ when $\varphi = \epsilon$ and $'@_{\uparrow}'$ when $\varphi = \uparrow$. Rule (u-app) handles the general case for application. The annotations on the function type and on the application in the target term must be the same. The annotation φ on the type $\tau \rightarrow_{\varphi} \tau'$ indicates the unCurrying (or lack of unCurrying) for the first argument to a function of this type. Hence, the application in the target term should also convey this information. If $\varphi = \uparrow$ then m_2 will be tupled with the argument applied to $(m_1 @_{\uparrow} m_2)$. The next rule, (u-app- \uparrow), creates a tuple of arguments in an application, using (or consuming) the \uparrow -annotation on the application. The argument m_2 can be either a single term or a tuple (resulting from previous applications of this rule). The next rule, (u-abs), handles the general case for λ -abstractions. To type and translate $\lambda x.e$ we show that from the assumption $x : \tau \Rightarrow x'$ we can derive the type τ' and translate e to m . Because no unCurrying of function parameters occurs, 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. We assume uniqueness of bound variables so an assumption introduced by an implication is the only means for reasoning about a given variable.

The final rule, (u-let), operates on let expressions and 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

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

$$\begin{array}{c}
\frac{e_1 : \tau_1 \rightarrow_{\varphi} \tau \Rightarrow m_1 \quad e_2 : \tau_1 \Rightarrow m_2}{e_1 @ e_2 : \tau \Rightarrow (m_1 @_{\varphi} m_2)} \quad (\text{u-app}) \\
\\
\frac{e_1 @ e_2 : \tau \Rightarrow ((m_0 @_{\uparrow} m_1) @_{\varphi} m_2)}{e_1 @ e_2 : \tau \Rightarrow (m_0 @_{\varphi} (m_1, m_2))} \quad (\text{u-app-}\uparrow) \\
\\
\frac{x : \tau \Rightarrow x' \supset e : \tau_1 \Rightarrow m}{\lambda x. e : \tau \rightarrow_{\epsilon} \tau_1 \Rightarrow \lambda x'. m} \quad (\text{u-abs}) \\
\\
\frac{x : \tau \Rightarrow x' \supset \lambda y. e : \tau_1 \Rightarrow \lambda y. m}{\lambda x. \lambda y. e : \tau \rightarrow_{\uparrow} \tau_1 \Rightarrow \lambda (x', y). m} \quad (\text{u-abs-}\uparrow) \\
\\
\frac{e_1 : \tau_1 \Rightarrow m_1 \quad (\forall \tau'. (e_1 : \tau' \Rightarrow m_1 \supset x : \tau' \Rightarrow x')) \supset e_2 : \tau \Rightarrow m_2}{\text{let } x = e_1 \text{ in } e_2 : \tau \Rightarrow \text{let } x' = m_1 \text{ in } m_2} \quad (\text{u-let})
\end{array}$$

Figure 5: The UnCurry Inference System

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 $\forall \alpha_n. \tau'$ and assume this as the type for x when typing e_2 . Unfortunately, this more familiar presentation cannot be encoded directly into the LF type theory. The rule (u-let), however, can be encoded into LF, encoding bound variables of the source and target languages as bound variables of LF. Michaylov and Pfenning [8] provide further discussion of encoding polymorphic-let typing rules into logical frameworks.

We use logical implication, instead of contexts, in this specification to facilitate reasoning about the structure of deductions constructed from these rules. Logical implication provides a precise and accurate description of the introduction of assumptions required for the specification, without introducing a new data structure (type contexts) to simulate this requirement. We encoded these rules and the operational semantics into the LF type theory, and also encoded the correctness proof of Section 4 into LF. We used the programming language Elf [11] to typecheck all of these encodings. We can give an alternative presentation using type contexts and prove it equivalent to the one given here. However, proving correctness using this lower-level description of unCurrying is significantly more difficult.

This inference system provides a general specification of unCurrying. For a given source term e , there may exist several τ_i, m_i such that $e : \tau_i \Rightarrow m_i$. Included among these is one in which all function arrows in a deduction are annotated with ϵ , in which case $m = e$, corresponding to the identity translation. Our goal in unCurrying is to take a term e and produce a term m which 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 \uparrow -annotations. This ensures that the translated term m can be used in the same context in which e can be used.

3.3 Examples

We present here several examples, including those introduced above, of unCurryings which can be derived using our inference system. For convenience we assume some terms e_i such that $e_i : \tau_i \Rightarrow m_i$, for $i \in [1..5]$.

The following three are straightforward examples from above:

$$(\lambda x. \lambda y. x) @ e_1 @ e_2 : \tau_1 \Rightarrow ((\lambda (x, y). x) @ (m_1, m_2))$$

$$(\lambda k. k @ e_1 @ e_2) @ (\lambda x. \lambda y. x) : \tau_1 \Rightarrow ((\lambda k. k @ (m_1, m_2)) @ \lambda (x, y). x)$$

$$(\lambda g. (g @ (\lambda x. \lambda y. x))) @ (\lambda f. f @ e_1 @ e_2) : \tau_1 \Rightarrow (\lambda g. (g @ \lambda (x, y). x)) @ (\lambda f. f @ (m_1, m_2))$$

The following example from above is not immediately obvious, but upon careful examination can be understood as correct.

$$\begin{aligned}
& (\lambda s. \lambda k. s @ (\lambda u. \lambda v. v) @ (\lambda w. w) @ (k @ e_1 @ e_2)) @ S @ K \\
& \quad : \tau_1 \Rightarrow \\
& \quad (\lambda (s, k). s @ ((\lambda (u, v). v), (\lambda w. w), (k @ (m_1, m_2)))) \\
& \quad @ (\lambda (g, y, z). g @ (z, (y @ z)), \lambda (x, y). x)
\end{aligned}$$

The last example demonstrates the ability to partially unCurry any consecutive parameters to a function. In this case the second parameter of a function is unCurried with the third.

$$\begin{aligned}
& ((\lambda g. g @ e_1 @ e_2) @ ((\lambda x. \lambda y. \lambda z. \lambda w. e_3) @ e_4)) @ e_5 : \tau_3 \Rightarrow \\
& ((\lambda g. g @ (m_1, m_2)) @ ((\lambda x. \lambda (y, z). \lambda w. m_3) @ m_4)) @ m_5
\end{aligned}$$

4 Correctness

We examine the correctness of our unCurry inference system with respect to the type systems and the operational semantics of the two languages.

4.1 Type Correctness

To demonstrate the correctness of the unCurrying inference system with respect to the type systems for the source and target languages, we show that, given a simple restriction, this system derives judgments over exactly the terms typable in these two systems. Additionally, we show that every typable source term can be unCurried.

To facilitate the comparison of the unCurry inference system (which does not use type contexts) and the two type

language systems (which do), we introduce an alternative inference system for unCurrying which uses a type context. This system, which axiomatizes the judgment $\Gamma \triangleright e : \tau \Rightarrow m$, is given in Figure 10 in the Appendix. We can show an equivalence between this system and the one in Figure 5.

Next, we give a translation from simple types and type schemas to unCurry types and type schemas, and also from unCurry types to the types of the source and target languages.

Definition 1 *For every simple type τ , its corresponding unCurry type, $[\tau]_\epsilon$ is given by*

$$\begin{aligned} [\alpha]_\epsilon &= \alpha \\ [\tau_1 \rightarrow \tau_2]_\epsilon &= [\tau_1]_\epsilon \rightarrow_\epsilon [\tau_2]_\epsilon \end{aligned}$$

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 $[\Gamma]_\epsilon(x) = [\sigma]_\epsilon$.

Definition 2 *For every unCurry type τ , its erasure $|\tau|$ and its unCurrying $\|\tau\|$ are given as*

$$\begin{aligned} |\alpha| &= \alpha \\ |\tau_1 \rightarrow_\varphi \tau_2| &= |\tau_1| \rightarrow |\tau_2| \\ |\forall \alpha. \sigma| &= \forall \alpha. |\sigma| \\ \|\alpha\| &= \alpha \\ \|\tau_1 \rightarrow_\epsilon \tau_2\| &= \|\tau_1\| \rightarrow \|\tau_2\| \\ \|\tau_1 \rightarrow_{\uparrow} \tau_2 \rightarrow_\varphi \tau_3\| &= \text{let } \tau'_2 \rightarrow \tau'_3 = \|\tau_2 \rightarrow_\varphi \tau_3\| \\ &\quad \text{in } \|\tau_1\| \star \tau'_2 \rightarrow \tau'_3 \\ \|\forall \alpha. \sigma\| &= \forall \alpha. \|\sigma\| \end{aligned}$$

Observe that the $\|\cdot\|$ -translation assumes the \uparrow -annotation only occurs in a context $\tau_1 \rightarrow_{\uparrow} \tau_2 \rightarrow_\varphi \tau_3$. By considering only relevant deductions, we can ensure that this is a valid assumption. We extend this definition to type contexts in the natural way: if $\Gamma(x) = \sigma$ then $|\Gamma|(x) = |\sigma|$ and $\|\Gamma\|(x) = \|\sigma\|$.

Theorem 1 *If $\Gamma \triangleright_s e : \tau$ then there exists a target term m such that $[\Gamma]_\epsilon \triangleright e : [\tau]_\epsilon \Rightarrow m$*

The proof follows by letting $m = e$ and constructing a deduction which includes no unCurrying. Thus, every typable source term can be unCurried to some target term.

Theorem 2 *If $\Gamma \triangleright e : \tau \Rightarrow m$ and m contains no occurrences of the \uparrow -annotation then $|\Gamma| \triangleright_s e : |\tau|$ and $\|\Gamma\| \triangleright_t m : \|\tau\|$.*

The restriction that m contains no occurrences of the \uparrow -annotation prohibits terms of the form $(m_1 @_{\uparrow} m_2)$ which actually denote intermediate terms in the sense that some tupling is required to form a proper term. Note that a deduction $\Xi :: \Gamma \triangleright e : \tau \Rightarrow m$ may contain sub-deductions involving terms of the form $(m_1 @_{\uparrow} m_2)$. This does not pose a significant problem as we can reason by induction over the subterms m_1 and m_2 and then perform the appropriate tupling. Thus, the inference system only relates terms that are typable in their respective type system.

4.2 Operational Correctness

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

To accomplish this, we first extend the target operational semantics with a rule for treating these intermediate application terms. Because these applications represent functions applied to only some of their arguments, the value of such an application should be a function expecting the remaining arguments, i.e., a partially-applied function. Using this idea, we introduce the following new rule:

$$\frac{m_0 \hookrightarrow_t \lambda(x, \mathbf{y}). m' \quad m_1 \hookrightarrow_t w_1}{(m_0 @_{\uparrow} m_1) \hookrightarrow_t \lambda \mathbf{y}. (m' [w_1/x])} \text{ (t-app-}\uparrow\text{)}$$

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 , may be the unCurried versions of functions in v . The lack of environments, contexts, and closures greatly simplifies the presentation here. Specifically, we do not require a separate definition of value consistency as found in traditional proofs of type consistency.

Theorem 3 (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 of part 1 proceeds by induction on the structure of the deduction for $e \hookrightarrow_s v$. The proof of part 2, however, 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 $\Delta :: m \hookrightarrow_t w$. Let $|\Delta| = (k_n, \dots, k_1)$ such that k_i = the number of application rules (both (t-app) and (t-app- \uparrow)) of arity i occurring in Δ . (Note that all occurrences of (t-app- \uparrow) have arity 1.) 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 $\Delta :: m \hookrightarrow_t w$.

An essential idea used in both parts of the proof involves the structure of the deduction

$$\frac{\Xi}{\lambda x. e : \tau \rightarrow_\epsilon \tau' \Rightarrow \lambda x'. m} \text{ (u-abs)}$$

When encoded into the LF type theory, the deduction Ξ is represented as a function of type

$$\{x : \text{sexp}\} \{x' : \text{texp}\} \{e : \text{sexp}\} \{m : \text{texp}\} \{\tau : \text{tp}\} \{\tau' : \text{tp}\} \\ (x : \tau \Rightarrow x' \rightarrow e : \tau' \Rightarrow m)$$

We assume that source expressions, target expressions, and types are represented as objects of type $sexp$, $texp$, and tp , respectively. The type $\{x : \tau\}\tau'$ denotes a dependent function type which takes an argument v of type τ and returns an object of type $\tau'[v/x]$. Objects of type $e : \tau \Rightarrow w$ represent deductions of the corresponding judgment. The function representing Ξ , when supplied arguments $v : sexp, w : texp, e_0 : sexp, m_0 : texp, \tau_0 : tp, \tau_0 : (v : \tau_0 \Rightarrow w)$, returns an object (deduction) of type $e[v/x] : \tau'_0 \Rightarrow m[w/x']$. This resulting deduction is precisely the one needed by the induction hypothesis for the cases involving rules (s-app) and (t-app) (to relate $e'[v_2/x]$ and $m'[w_2/x]$). Without such a function we would require a substitution lemma to demonstrate the existence of such a deduction.

Encoding the rules into LF (and our experience with the Elf programming language) provided us with insight into how to structure our operational semantics and unCurry system so that the structure of the proof would be more evident and easier to decode in LF. The complete proof, written as a series of declarations in Elf, is about 20 lines. After the Elf system performs type reconstruction, however, the size is much larger. Note that this encoding does not include the reasoning that the proof is complete, only that it is sound. Still, the exercise of encoding the proof was fruitful in that it guided us in the structure of the proof and provided some partial machine verification of the proof.

5 An unCurry Algorithm

We have developed an algorithm for unCurrying based on the inference system presented in Figure 6. The algorithm is similar to algorithm \mathcal{W} and has the same complexity. We implemented this algorithm in Standard ML¹, tested it using all the examples presented in this paper as well as other small, demonstrative test programs and it yielded the expected results in all cases.

For the purpose of establishing and solving constraints on unCurry types and annotations, we extend the grammars for unCurry annotations to include annotation variables γ :

$$\varphi ::= \epsilon \mid \uparrow \mid \gamma$$

We let θ range over substitutions, mapping unCurry type and annotation variables to unCurry types and annotations, respectively. We also introduce annotated terms, which extend source terms by including annotations on $@$'s and λ 's:

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

The algorithm consists essentially of two steps, U and translate :

$$\begin{aligned} \text{unCurry}(\Gamma, e) &= \text{let } (\theta, \tau, u) = U(\Gamma, e) \\ &\quad \text{in } \text{translate}(\theta, u) \end{aligned}$$

The first step corresponds to repeated applications of the (u-abs), (u-app), and (u-let) rules, generating and solving constraints on unCurry types and annotations. In this step, similar to algorithm \mathcal{W} , the function U takes an unCurry type environment (mapping source term variables to unCurry types) and a source term and returns a triple (θ, τ, a) consisting of a substitution θ , an unCurry type τ , and the resulting, annotated term u .

In the second step, the function translate takes a substitution and an annotated term and transforms the term,

as indicated by annotations, into an unCurried form. The transformations applied correspond to the rules (u-abs- \uparrow) and (u-app- \uparrow).

The algorithm ensures that θ is an idempotent substitution, disallowing cyclic references with an occurs check in the unification algorithm. Also note that annotation variables never explicitly take the value \uparrow in the algorithm. This is because it can be assumed that all annotation variables that are still unmapped after applying U to the term can be mapped to \uparrow . This assumption is accounted for in translate .

The function 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 U also generates correct unCurry 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 unCurry type variables for the quantified variable in the type schema. The abstraction case generates a new unCurry type variable α for the bound variable and then annotates the body. The use of grndApp and grndAbs ensure that appropriate annotations are ϵ , just as enforced by the inference system. The application case uses the function unify to resolve constraints. Again, the use of grndAbs ensures that appropriate annotations are ϵ , just as enforced by the inference system.

The function unify takes two unCurry types and returns the most general substitution which 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.

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. These annotations can only be ϵ or variables. (The constant \uparrow is never generated by U .) If an annotation is ϵ then no unCurrying takes place. If an annotation is a variable then this implies that it can be bound to \uparrow , and the corresponding term (either an application or abstraction) is unCurried. (The infix operator \odot inserts an element at the end of a tuple: $(m_1, \dots, m_k) \odot m_{k+1} = (m_1, \dots, m_k, m_{k+1})$.)

To prove that the algorithm is correct (sound) with respect to the inference system presented in Figure 10 and that it generates maximally unCurried terms, we first introduce a new unCurry inference system that more closely parallels the algorithm by separating the steps of annotating terms and translating terms into two deductive systems. (See Figure 11 in the Appendix.) The judgment $\Gamma \triangleright_1 e : \tau \Rightarrow u$ relates a source term e , an unCurry type τ , and an annotated term u . The judgment $u \Rightarrow_{\iota} m$ relates an annotated term u with a target-language term m . These two deductive systems provide a specification of unCurrying equivalent to the one given in Figure 10.

Lemma 1 $\Gamma \triangleright e : \tau \Rightarrow m$ iff $\Gamma \triangleright_1 e : \tau \Rightarrow u$ and $u \Rightarrow_{\iota} m$.

The unCurry algorithm is sound and complete with respect to this new specification.

Theorem 4 (Soundness of unCurry)

1. If $U(\Gamma, e) = (\theta, \tau, u)$ then $\theta\Gamma \triangleright_1 e : \theta\tau \Rightarrow \theta u$.
2. If $m = \text{translate}(\theta, u)$ then $\theta_{\uparrow}\theta u \Rightarrow_{\iota} m$, where θ_{\uparrow} maps all annotation variables to \uparrow .

The proof follows by induction on the structure of terms.

¹available via <http://www.cse.psu.edu/~phicks/uncurry>

$$\begin{array}{l}
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) \\
U(\Gamma, \lambda x. e) = \\
\text{let } \alpha, \gamma \text{ be new variables} \\
\quad (\theta_1, \tau, u) = U(\Gamma\{x : \alpha\}, e) \\
\quad \theta_2 = \text{grndApp}(u) \circ \text{grndAbs}(\lambda \gamma x. u) \\
\text{in } (\theta_2 \circ \theta_1, \alpha \rightarrow_\gamma \tau, \lambda \gamma x. u) \\
U(\Gamma, e_1 @ e_2) = \\
\text{let } (\theta_1, \tau_1, u_1) = U(\Gamma, e_1) \\
\quad (\theta_2, \tau_2, u_2) = U(\theta_1 \Gamma, e_2) \\
\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}(u_2) \\
\text{in } (\theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1, \alpha, u_1 @_\gamma u_2) \\
U(\Gamma, \text{let } x = e_1 \text{ in } e_2) = \\
\text{let } (\theta_1, \tau_1, u_1) = U(\Gamma, e_1) \\
\quad \{\overline{\alpha_n}\} = FV(\tau_1) - FV(\theta_1 \Gamma) \\
\quad (\theta_2, \tau_2, u_2) = 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) \\
\text{grndApp}(u_1 @_\gamma u_2) = ID\{\gamma \mapsto \epsilon\} \\
\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) = \\
\text{if } \alpha = \tau \text{ then } ID \\
\text{else if occurs}(\alpha, \tau) \text{ then fail 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) = \\
\text{let } \theta_1 = \text{unify}(\gamma, \gamma') \\
\quad \theta_2 = \text{unify}(\theta_1 \tau_1, \theta_1 \tau'_1) \circ \theta_1 \\
\quad \theta_3 = \text{unify}(\theta_2 \tau_2, \theta_2 \tau'_2) \circ \theta_2 \\
\text{in } \theta_3 \\
\text{unify}(\epsilon, \epsilon) = ID \\
\text{unify}(\gamma, \gamma') = ID\{\gamma \mapsto \gamma'\} \\
\text{unify}(\epsilon, \gamma) = ID\{\gamma \mapsto \epsilon\} \\
\text{unify}(\gamma, \epsilon) = ID\{\gamma \mapsto \epsilon\} \\
\text{translate}(\theta, x) = x \\
\text{translate}(\theta, \lambda \epsilon x. u) = \lambda x. \text{translate}(\theta, u) \\
\text{translate}(\theta, \lambda \gamma x. u) = \\
\text{if } \theta(\gamma) = \epsilon \text{ then } \lambda x. \text{translate}(\theta, u) \\
\quad \text{else let } \lambda \mathbf{y}. m = \text{translate}(\theta, u) \\
\quad \quad \text{in } \lambda(x, \mathbf{y}). m \\
\text{translate}(\theta, u_0 @_{\gamma'} u_2) = \\
\text{case } u_0 \text{ of} \\
\quad u @_\gamma u_1 \Rightarrow \\
\quad \text{if } \theta(\gamma) = \epsilon \\
\quad \quad \text{then } \text{translate}(\theta, u @_\gamma u_1) @ \text{translate}(\theta, u_2) \\
\quad \quad \text{else let } (m @ \mathbf{m}\mathbf{s}) = \text{translate}(\theta, u @_\gamma u_1) \\
\quad \quad \quad \text{in } (m @ (\mathbf{m}\mathbf{s} \odot \text{translate}(\theta, u_2))) \\
\quad u \Rightarrow \text{translate}(\theta, u) @ \text{translate}(\theta, u_2) \\
\text{translate}(\theta, \text{let } x = u_1 \text{ in } u_2) = \\
\text{let } x = \text{translate}(\theta, u_1) \text{ in } \text{translate}(\theta, u_2)
\end{array}$$

Figure 6: The UnCurry Algorithm

$$\begin{array}{l}
\overline{x \sqsubseteq x} \\
\frac{m_1 \sqsubseteq m'_1 \quad m_2 \sqsubseteq m'_2}{m_1 @ m_2 \sqsubseteq m'_1 @ m'_2} \\
\frac{m_1 @ \mathbf{m}\mathbf{s} \sqsubseteq m'_1 \quad m_2 \sqsubseteq m'_2}{m_1 @ (\mathbf{m}\mathbf{s} \odot m_2) \sqsubseteq m'_1 @ m'_2} \\
\frac{m_1 @ \mathbf{m}\mathbf{s} \sqsubseteq m'_1 @ \mathbf{m}\mathbf{s}' \quad m_2 \sqsubseteq m'_2}{m_1 @ (\mathbf{m}\mathbf{s} \odot m_2) \sqsubseteq m'_1 @ (\mathbf{m}\mathbf{s}' \odot m'_2)} \\
\frac{m \sqsubseteq m'}{\lambda(x_1, \dots, x_k). m \sqsubseteq \lambda(x_1, \dots, x_k). m'} \\
\frac{n > k \quad \lambda(x_{k+1}, \dots, x_n). m \sqsubseteq m'}{\lambda(x_1, \dots, x_n). m \sqsubseteq \lambda(x_1, \dots, x_k). m'}
\end{array}$$

Figure 7: Order on UnCurried Terms

Lemma 2 (Completeness of UnCurry)

1. If $\theta' \Gamma \triangleright_1 e : \tau' \Rightarrow u'$ then there exists a substitution δ such that

$$\begin{array}{l}
U(\Gamma, e) = (\theta, \tau, u), \\
\theta' \Gamma = \delta \theta \Gamma, \\
\tau' = \delta \theta \tau, \text{ and} \\
u' = \delta \theta u.
\end{array}$$

2. If $\theta u = u'$ and $u' \Rightarrow_t m$ then $\text{translate}(\theta, u) = m$.

The proof is similar to the one given in [10].

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 which occurs in them. This ordering is given in Figure 7. Intuitively, $m \sqsubseteq m'$ if the terms are both unCurried forms of the same source term, with m unCurried at least as much as m' is unCurried. For example,

$$\begin{array}{l}
(\lambda(x, y, z). x(y, z)) @ (\lambda(u, v). v, m_1, m_2) \sqsubseteq \\
(\lambda x. \lambda(y, z). x(y, z)) @ (\lambda(u, v). v) @ (m_1, m_2)
\end{array}$$

Using this ordering we have the following result.

Theorem 5 (Principal UnCurrying) *If $\Gamma \triangleright e : \tau \Rightarrow m$ then $\text{unCurry}(\Gamma, e) = m'$ and $m' \sqsubseteq m$.*

The proof follows from the completeness of unCurry. Thus, our algorithm computes the best unCurrying possible for any typed term.

6 Untyped UnCurrying

We can adapt the unCurrying inference system of Section 3 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 which 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 [2] by introducing a kind of universal type Ω which can be the type of any term. We

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

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

Figure 8: The Ω -Rules

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 those defined in Section 2, including the operational semantics, but without the let construct and the type systems.

The inference system includes the rules from Figure 5, except for (u-let), plus the additional rules given in Figure 8. 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 3, 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

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

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 of Figure 6 to perform untyped unCurrying. The modifications to the function 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, $=_{\Omega}$, which extends 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.

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

Figure 9: The Untyped Unification Algorithm

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 9, 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. 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 $\text{ground}(\tau)$ 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 $U(\Gamma, e)$ is again a triple (θ, τ, u) in which u is an annotated term. We can again use translate to produce an unCurried term.

7 Extensions and Future Work

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

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 which are unCurried forms of e .

Recursion and Other Language Constructs. The current work treats only the core λ -calculus. The inclusion of additional programming language features is an important 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:

$$\frac{f : \tau \Rightarrow f' \quad \supset \quad e : \tau \Rightarrow m}{\mu f.e : \tau \Rightarrow \mu f'.m}$$

Interaction with β and η . The current work treats unCurrying in isolation of other program optimizations and translations, in particular in-lining (β -reduction) and η -reduction and expansion. By performing additional transformations to an expression, more unCurrying may become possible. We plan to study the interaction of these transformations with unCurrying.

Lambda Lifting and Closure Conversion. The operations of lambda lifting and closure conversion have properties similar to that of unCurrying and we believe that our inference system and algorithm can be adapted to handle these transformations.

8 Conclusion

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.

Acknowledgments. This work was supported in part by NSF CAREER Award #CCR-9502356. We thank David Liben-Nowell for his comments on earlier versions of the paper.

References

- [1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- [2] Carsten K. Gomard and Peter Sestoft. Globalization and live variables. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 166–177, 1991.
- [3] John Hannan. Type systems for closure conversions. In Hanne Riis Nielson and Kirsten Lackner Solberg, editors, *Participants' Proceedings of the Workshop on Types for Program Analysis*, pages 48–62. Aarhus University, DAIMI PB-493, May 1995.
- [4] John Hannan. A type-based escape analysis for functional languages. *Journal of Functional Programming*, September 1997. To appear.
- [5] John Hannan, Patrick Hicks, and David Liben-Nowell. A lifetime analysis for higher-order languages. Technical Report CSE-97-014, Penn State University, September 1997.
- [6] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. A preliminary version appeared in *Symposium on Logic in Computer Science*, pages 194–204, June 1987.
- [7] Xavier Leroy. The ZINC experiment: An economical implementation of the ML language. Technical Report 117, INRIA, 1990.
- [8] S. Michaylov and F. Pfenning. Natural semantics and some of its meta-theory in Elf. In Lars Hallnäs, editor, *Extensions of Logic Programming*, pages 299–344. Springer-Verlag LNCS 596, 1992. A preliminary version is available as Technical Report MPI-I-91-211, Max-Planck-Institute for Computer Science, Saarbrücken, Germany, August 1991.
- [9] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
- [10] Tobia Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, April 1995.
- [11] Frank Pfenning. An implementation of the Elf core language in Standard ML. Available via ftp over the Internet, September 1991. Send mail to elf-request@cs.cmu.edu for further information.
- [12] Frank Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
- [13] David Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML*. PhD thesis, Carnegie Mellon University, 1996. Available as CMU CS Technical Report 97-108.
- [14] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Bob Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

A UnCurry Deductive Systems using Contexts

$$\begin{array}{c}
\frac{\Gamma(x) \succeq \tau}{\Gamma \triangleright x : \tau \Rightarrow x} \\
\frac{\Gamma \triangleright e_1 : \tau \rightarrow_{\varphi} \tau' \Rightarrow m_1 \quad \Gamma \triangleright e_2 : \tau \Rightarrow m_2}{\Gamma \triangleright e_1 @ e_2 : \tau' \Rightarrow (m_1 @_{\varphi} m_2)} \\
\frac{\Gamma \triangleright e_1 @ e_2 : \tau \Rightarrow (m_0 @_{\uparrow} m_1 @_{\varphi} m_2)}{\Gamma \triangleright e_1 @ e_2 : \tau \Rightarrow (m_0 @_{\varphi} (m_1, m_2))} \\
\frac{\Gamma\{x : \tau\} \triangleright e : \tau' \Rightarrow m}{\Gamma \triangleright \lambda x.e : \tau \rightarrow_{\epsilon} \tau' \Rightarrow \lambda x.m} \\
\frac{\Gamma\{x : \tau\} \triangleright \lambda y.e : \tau' \Rightarrow \lambda y.m}{\Gamma \triangleright \lambda x.\lambda y.e : \tau \rightarrow_{\uparrow} \tau' \Rightarrow \lambda(x, y).m} \\
\frac{\Gamma \triangleright e_1 : \tau_1 \Rightarrow m_1 \quad \{\overline{\alpha}_n\} = FV(\tau_1) - FV(\Gamma) \quad \Gamma\{x : \forall \overline{\alpha}_n.\tau_1\} \triangleright e_2 : \tau \Rightarrow m_2}{\Gamma \triangleright \text{let } x = e_1 \text{ in } e_2 : \tau \Rightarrow \text{let } x = m_1 \text{ in } m_2}
\end{array}$$

Figure 10: Introducing Contexts to the Original UnCurry System

$$\begin{array}{c}
\frac{\Gamma(x) = \varphi \quad \tau = \text{Inst}(\varphi)}{\Gamma \triangleright x : \tau \Rightarrow x} \\
\frac{\Gamma \triangleright e_1 : \tau \rightarrow_{\varphi} \tau' \Rightarrow u_1 \quad \Gamma \triangleright (e_2 @ e_3) : \tau \Rightarrow (u_2 @_{\epsilon} u_3)}{\Gamma \triangleright (e_1 @ (e_2 @ e_3)) : \tau' \Rightarrow (u_1 @_{\varphi} (u_2 @_{\epsilon} u_3))} \\
\frac{\Gamma \triangleright e_1 : \tau \rightarrow_{\varphi} \tau' \Rightarrow u_1 \quad \Gamma \triangleright \lambda x.e_2 : \tau \Rightarrow \lambda_{\varphi'} x.u_2}{\Gamma \triangleright (e_1 @ (\lambda x.e_2)) : \tau' \Rightarrow (u_1 @_{\varphi} (\lambda_{\varphi'} x.u_2))} \\
\frac{\Gamma \triangleright e_1 : \tau \rightarrow_{\varphi} \tau' \Rightarrow u_1 \quad \Gamma \triangleright x : \tau \Rightarrow x}{\Gamma \triangleright (e_1 @ x) : \tau' \Rightarrow (u_1 @_{\varphi} x)} \\
\frac{\Gamma\{x : \tau\} \triangleright (e_1 @ e_2) : \tau' \Rightarrow (u_1 @_{\epsilon} u_2)}{\Gamma \triangleright \lambda x.(e_1 @ e_2) : \tau \rightarrow_{\epsilon} \tau' \Rightarrow \lambda_{\epsilon} x.(u_1 @_{\epsilon} u_2)} \\
\frac{\Gamma\{x : \tau\} \triangleright \lambda y.e : \tau' \Rightarrow \lambda_{\varphi'} y.u}{\Gamma \triangleright \lambda x.\lambda y.e : \tau \rightarrow_{\varphi} \tau' \Rightarrow \lambda_{\varphi} x.\lambda_{\varphi'} y.u} \\
\frac{\Gamma\{x : \tau\} \triangleright y : \tau' \Rightarrow y}{\Gamma \triangleright \lambda x.y : \tau \rightarrow_{\epsilon} \tau' \Rightarrow \lambda_{\epsilon} x.y} \\
\frac{\Gamma \triangleright e_1 : \tau_1 \Rightarrow u_1 \quad \{\overline{\alpha}_n\} = FV(\tau_1) - FV(\Gamma) \quad \Gamma\{x : \forall \overline{\alpha}_n.\tau_1\} \triangleright e_2 : \tau \Rightarrow u_2}{\Gamma \triangleright \text{let } x = e_1 \text{ in } e_2 : \tau \Rightarrow \text{let } x = u_1 \text{ in } u_2} \\
\frac{}{x \Rightarrow_t x} \\
\frac{u \Rightarrow_t m}{\lambda_{\epsilon} x.u \Rightarrow_t \lambda x.m} \\
\frac{\lambda_{\varphi} y.u \Rightarrow_t \lambda y'.m}{\lambda_{\uparrow} x.\lambda_{\varphi} y.u \Rightarrow_t \lambda(x, y').m} \\
\frac{u_2 \Rightarrow_t m_2}{(x @_{\varphi} u_2) \Rightarrow_t (x @ m_2)} \\
\frac{\lambda_{\varphi} x.u_1 \Rightarrow_t m_1 \quad u_2 \Rightarrow_t m_2}{((\lambda_{\varphi} x.u_1) @_{\varphi} u_2) \Rightarrow_t (m_1 @ m_2)} \\
\frac{(u_0 @_{\epsilon} u_1) \Rightarrow_t m \quad u_2 \Rightarrow_t m_2}{((u_0 @_{\epsilon} u_1) @_{\varphi} u_2) \Rightarrow_t (m @ m_2)} \\
\frac{(u_0 @_{\uparrow} u_1) \Rightarrow_t (m_0 @ m) \quad u_2 \Rightarrow_t m_2}{((u_0 @_{\uparrow} u_1) @_{\varphi} u_2) \Rightarrow_t (m_0 @ (m @ m_2))} \\
\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}
\end{array}$$

Figure 11: Splitting UnCurry into Two Systems