Practical Variable-Arity Polymorphism

T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen

PLT @ Northeastern University

Abstract. Just as some functions have uniform behavior over distinct types, other functions have uniform behavior over distinct arities. These variable-arity functions are widely used in scripting languages such as Scheme and Python. Statically typed languages also accommodate modest forms of variable-arity functions, but even ML and Haskell, languages with highly expressive type systems, cannot type check the wide variety of variable-arity functions found in untyped functional languages. Consequently, their standard libraries contain numerous copies of the same function definition with slightly different names.

As part of the Typed Scheme project—an on-going effort to create an explicitly typed sister language for PLT Scheme—we have designed and implemented an expressive type system for variable-arity functions. Our practical validation in the context of our extensive code base confirms the usefulness of the enriched type system.

1 Types for Variable-Arity Functions

For the past two years, Tobin-Hochstadt and Felleisen [1,2] have been developing Typed Scheme, an explicitly and statically typed sister language of PLT Scheme [3]. In many cases, Typed Scheme accommodates existing Scheme programming idioms as much as possible. One remaining obstacle concerns functions of variable arity. Such functions have a long history in programming languages, especially in LISP and Scheme systems where they are widely used for a variety of purposes, ranging from arithmetic operations to list processing. In response, we have augmented Typed Scheme so that its type system can cope with variable-arity functions of many kinds.

Some variadic functions in Scheme are quite simple. For example, the function + takes any number of numeric values and produces their sum. This function, and others like it, could be typed in a system that maps a variable number of arguments into a homogeneous data structure.[1] Other variable-arity functions, however, demand a more sophisticated approach than collecting the extra arguments in such a fashion.

Consider Scheme's *map* function, which takes a function as input as well as an arbitrary number of lists. It then applies the function to the elements of the lists in a pointwise fashion. The function must therefore take precisely as many arguments as the number of lists provided. For example, if the *make-student*

[1] Languages like C, C++, Java, and C# support such variable-arity functions.

function consumes two arguments, a name as a string and a number for a grade, then the expression

(*map make-student* (*list* "Al" "Bob" "Carol") (*list* 87 98 64))

produces a list. We refer to variable-arity functions such as + and *map* as having *uniform* and *non-uniform* polymorphic types, respectively.

For Typed Scheme to be useful to working programmers, its type system must handle this form of polymorphism. Further, although *map* and + are part of the standard library, language implementers cannot arrogate the ability to abstract over the arities of functions. Scheme programmers routinely define such functions, and if they wish to refactor their Scheme programs into Typed Scheme, our language must allow such function definitions.

Of course, our concerns are relevant beyond the confines of Scheme. Variable-arity functions are also useful in statically typed languages, but are barely supported because of a lack of pragmatic approaches. Even the standard libraries of highly expressive typed functional languages contain functions that would benefit from non-uniform variable-arity, but instead are defined via copying of code. For example, the SML Basis Library [4] includes the `ARRAY` and `ARRAY2` signatures, which include functions that differ only in arity. The GHC standard library [5] also features close to a dozen families of functions (such as `zip` and `zipWith`) defined at a variety of arities. We conjecture that if their type systems provided variable-arity polymorphism, Haskell and ML programmers would routinely define such functions, too.

In this paper, we present the first pragmatic and comprehensive approach to variable-arity polymorphism: its design, implementation, and evaluation. Our new version of Typed Scheme can assign types to hundreds of programmer-introduced function definitions with variable arities, something that was simply impossible before. Furthermore, we can now type check library functions such as *map* and *fold-left* without resorting to special tricks or duplication.

In the next two sections, we describe Typed Scheme in general terms and then present the type system for variable-arity functions. In section 4, we introduce a formal model of our variable-arity type system. In section 5 we present preliminary results of our evaluation effort with respect to the PLT Scheme code base and the limitations of our system. In section 6 we discuss related work.

2 Typed Scheme . . .

The goal of our Typed Scheme [2] project is to design a typed sister language for an untyped scripting language in which programmers can transfer programs to the typed world one module at a time. Like PLT Scheme, Typed Scheme is a modular programming language; unlike plain Scheme programs, Typed Scheme programs have explicit type annotations for function and structure definitions that are statically checked. Typed Scheme also supports integration with untyped Scheme code, allowing a typed program to link in untyped modules and vice versa. The mechanism exploits functional contracts [6] to guarantee a generalized type soundness theorem [1].

Typed Scheme supports this gradual refactoring with a type system that accommodates standard Scheme programming idioms with minimal code modification. In principle, Scheme programmers need only annotate structure and function headers with types to move a module to the Typed Scheme world; on occasion, they may also wish to define a type alias to keep type expressions concise. The type system combines true union types, recursive types, first-class polymorphic functions, and the novel discipline of occurrence typing. Additionally, Typed Scheme infers types for instantiations of polymorphic functions, based on locally-available type information.

2.1 Basic Typed Scheme

Scheme programmers typically describe the structure of their data in comments, rather than in executable code. For example, a shape data type might be represented as:

```
;; A shape is either a rectangle or a circle
(define-struct rectangle (l w))   (define-struct circle (r))
```

To accommodate this style in Typed Scheme, programmers can specify true, untagged unions of types:

```
(define-type-alias shape (⋃ rectangle circle))
(define-struct: rectangle ([l : Integer] [w : Integer]))
(define-struct: circle    ([r : Integer]))
```

Typed Scheme also supports explicit recursive types, which, for example, are necessary for typing uses of *cons* pairs in Scheme programs. This allows the specification of both fixed-length heterogeneous lists and arbitrary-length homogeneous lists, or even combinations of the two.

Finally, Typed Scheme introduces *occurrence typing*, which allows the types of variable occurrences to depend on their position in the control flow graph. For example, the program fragment

```
(display "Enter a number to double: ")
(let ([val (read)]) ;; an arbitrary S-expression
  (if (number? val) (display (∗ 2 val))
      (display "That wasn't a number!")))
```

type-checks correctly because the use of $*$ is guarded by the *number?* check.

2.2 Polymorphic Functions and Local Type Inference

Typed Scheme supports first-class polymorphic functions. For example, *list-ref* has the type (∀ (α) ((**Listof** α) **Integer** → α)). It can be defined in Typed Scheme as follows:

```
(: list-ref (∀ (α) ((Listof α) Integer → α)))
(define (list-ref l i)
    (cond [(not (pair? l)) (error "empty list")]
          [(= 0 i) (car l)]
          [else (list-ref (cdr l) (− i 1))]))
```

The example shows two important aspects of polymorphism in Typed Scheme. First, the abstraction over types is explicit in the polymorphic type of *list-ref* but implicit in the function definition. Second, typical uses of polymorphic functions, e.g., *car* and *list-ref*, do not require explicit type instantiation. Instead, the required type instantiations are synthesized from the types of the arguments.

Argument type synthesis uses the local type inference algorithm of Pierce and Turner [7]. It greatly facilitates the use of polymorphic functions and makes conversions from Scheme to Typed Scheme convenient, while dealing with the subtyping present in the rest of the type system in an elegant manner. Furthermore, it ensures that type inference errors are always locally confined, rendering them reasonably comprehensible to programmers.

3 ... with Variable-Arity Functions

A Scheme programmer defines functions with **lambda** or **define**. Both syntactic forms support fixed and variable-arity parameter specifications:

1. (**lambda** (x y) (+ x (∗ y 3))) creates a function of two arguments and (**define** (f x y) (+ x (∗ y 3))) creates the same function and names it f;
2. the function (**lambda** (x y . z) (+ x (*apply* *max* y z))) consumes at least two arguments and otherwise as many as needed;
3. (**define** (g x y . z) (+ x (*apply* *max* y z))) names this function g;
4. (**lambda** z (*apply* + z)) creates a function of arbitrary arity; and
5. (**define** (h . z) (*apply* + z)) is the analogue to this **lambda** expression.

The parameter z in the last four cases is called the *rest parameter.*

The application of a variable-arity function combines any arguments in excess of the number of required parameters into a list. Thus, (g 1 2 3 4) binds x to 1 and y to 2, while z becomes (*list* 3 4) for the evaluation of g's function body. In contrast, (h 1 2 3 4) sets z to (*list* 1 2 3 4).

The *apply* function, used in the examples above, takes a function f, a sequence of fixed arguments $v_1 \dots v_n$, and a list of additional arguments r. If the list r is the value (*list* w_1 ... w_n), then (*apply* f v_1 ... v_n r) is the same as (f v_1 ... v_n w_1 ... w_n). The *apply* function plays a critical role in conjunction with rest arguments.

This section sketches how the revised version of Typed Scheme accommodates variable-arity functions. Our revision focuses on the uses of such functions that accept arbitrarily many arguments. Scheme programmers sometimes use variable-arity functions to simulate optional or keyword arguments. In PLT Scheme, such programs typically employ **case-lambda** [8] or equivalent **define** forms instead.

3.1 Uniform Variable-Arity Functions

Uniform variable-arity functions are those that expect their rest parameter to be a homogeneous list. Consider the following three examples of type signatures:

```
(: + (Integer* → Integer))
(: string-append (String* → String))
(: list (∀ (α) (α* → (Listof α))))
```

The syntax *Type** for the type of rest parameters alludes to the Kleene star for regular expressions. It signals that in addition to the other arguments, the function takes an arbitrary number of arguments of the given base type. The form *Type** is dubbed a *starred pre-type*, because it is not a full-fledged type and may appear only as the last element of a function's domain.

Here is a definition of variable-arity + in Scheme:

```
;; assumes binary-+, a binary addition operator
(define (+ . xs) (if (null? xs) 0 (binary-+ (car xs) (apply + (cdr xs)))))
```

Typing this definition is straightforward. The type assigned to the rest parameter of starred pre-type τ^* in the body of the function is (**Listof** τ), a pre-existing type in Typed Scheme.

3.2 Beyond Uniform Variable-Arity Functions

Not all variable-arity functions assume that their rest parameter is a homogeneous list of values. We can allow heterogeneous rest parameters by finding other constraints. For example, the length of the list assigned to the rest parameter may be connected to the types of other parameters or the returned value.

For example, Scheme's *map* function is not restricted to mapping unary functions over single lists, unlike its counter-parts in ML or Haskell. When *map* receives a function f and n lists, it expects f to accept n arguments. Also, the type of the kth function parameter must match the element type of the kth list.

Scheme's *apply* function provides its own challenges. It is straightforward to type the use of *apply* with a uniform variable-arity function, as in the hypothetical definition of + from section 3.1. If the type of f involves the starred pre-type τ^*, then the list r must have type (**Listof** τ).

However, take the following example taken from the PLT Scheme code base:

```
;; implements a wrapper that prints f's arguments
(define (verbose f)
  (if quiet? f (lambda a (printf "xform-cpp: ~a\n" a) (apply f a))))
```

The intent of the programmer is clear—the result of applying *verbose* to a function f should have the same type as f for *any* function type. With uniform variable-arity functions, we can type the internal **lambda**'s argument a only as a homogeneous list of arbitrary length. Thus, if f has some fixed arity n, then there is no way to statically guarantee that the list of arguments a has n elements, and thus applying the function f to the list a via *apply* may go wrong. Our type system should protect the programmer from arity mismatches, whether through function application or uses of *apply*, while allowing functions like *verbose*.

3.3 Non-uniform Variable-Arity Functions

As of the latest release, Typed Scheme can represent the types of *non-uniform* variable-arity functions. Below are the types for some example functions:

;; *map* is the standard Scheme map
(**:** *map*
  $(\forall\ (\gamma\ \alpha\ \beta\ \ldots)$
    $((\alpha\ \beta\ \ldots_\beta \to \gamma)\ (\textbf{Listof}\ \alpha)\ (\textbf{Listof}\ \beta)\ \ldots_\beta \to (\textbf{Listof}\ \gamma))))$

;; *map-with-funcs* takes any number of functions,
;; and then an appropriate set of arguments, and then produces
;; the results of applying all the functions to the arguments
(**:** *map-with-funcs*
  $(\forall\ (\beta\ \alpha\ \ldots)\ ((\alpha\ \ldots_\alpha \to \beta)^* \to (\alpha\ \ldots_\alpha \to (\textbf{Listof}\ \beta)))))$

Our first key innovation is the possibility to attach ... to the last type variable in the binding position of a $\forall$ type constructor. Such type variables are dubbed *dotted type variables*. Dotted type variables signal that this polymorphic type can be instantiated with an arbitrary number of types.

Next, the body of $\forall$ types with dotted type variables may contain expressions of the form $\tau\ \ldots_\alpha$ for some type τ and a dotted type variable α. These are *dotted pre-types*; they classify non-uniform rest parameters just like starred pre-types classify uniform rest parameters. A dotted pre-type has two parts: the base τ and the bound α. Only dotted type variables can be used as the bound of a dotted pre-type. Since $\forall$-types are nestable and thus multiple dotted type variables may be in scope, dotted pre-types must specify the bound.

When a dotted polymorphic type is instantiated, any dotted pre-types are expanded by copying the base an appropriate number of times and by replacing free instances of the bound in each copy with the corresponding type argument. For example, instantiating *map-with-funcs* as follows:

(**inst** *map-with-funcs* **Number Integer Boolean String**)

results in a value with the type:

$((\textbf{Integer Boolean String} \to \textbf{Number})^* \to$
$(\textbf{Integer Boolean String} \to (\textbf{Listof Number})))$

Typed Scheme also provides local inference of the appropriate type arguments for dotted polymorphic functions, so explicit type instantiation is rarely needed [9]. Thus, the following uses of *map* are successfully inferred at the appropriate types:

(*map not* (*list* #t #f #t))
;; *map* is instantiated (via local type inference) at:
;; $((\textbf{Boolean} \to \textbf{Boolean})\ (\textbf{Listof Boolean}) \to (\textbf{Listof Boolean}))$

(*map make-book* (*list* "Flatland") (*list* "A. Square") (*list* 1884))
;; $((\textbf{String String Integer} \to \textit{book})$
;; $(\textbf{Listof String})\ (\textbf{Listof String})\ (\textbf{Listof Integer}) \to (\textbf{Listof}\ \textit{book}))$

Typed Scheme can also type-check the definitions of non-uniform variable-arity functions:

```
(: fold-left
   (∀ (γ α β ...) ((γ α β ...β → γ) γ (Listof α) (Listof β) ...β → γ)))
(define (fold-left f c as . bss)
  (if (or (null? as) (ormap null? bss)) c
      (apply fold-left (apply f c (car as) (map car bss)) (cdr as)
             (map cdr bss))))
```

The example introduces a definition of *fold-left*. Its type shows that it accepts at least three arguments: a function f; an initial element c; and at least one list *as*. Optionally, *fold-left* consumes another sequence *bss* of lists. For this combination to work out, f must consume as many arguments as there are lists plus one; in addition, the types of these lists must match the types of f's parameters because each list item becomes an argument.

Beyond this, the example illustrates that the rest parameter is treated as if it were a place-holder for a plain list parameter. In this particular case, *bss* is thrice subjected to *map*-style processing. In general, variable-arity functions should be free to process their rest parameters with existing list-processing functions.

The challenge is to assign types to such expressions. On the one hand, list-processing functions expect lists, but the rest parameter has a dotted pre-type. On the other hand, the result of list-processing a rest parameter may flow again into a rest-argument position. While the first obstacle is simple to overcome with a conversion from dotted pre-types to list types, the second one is onerous. Since list-processing functions do not return dotted pre-types but list types, we cannot possibly expect that such list types come with enough information for an automatic conversion.

Thus we use special type rules for the list processing of rest parameters with *map*, *andmap*, and *ormap*. Consider *map*, which returns a list of the same length as the given one and whose component types are in a predictable order. If *xs* is classified by the dotted pre-type $\tau \ldots_{\alpha}$, and f has type $(\tau \to \sigma)$, we classify (*map f xs*) with the dotted pre-type $\sigma \ldots_{\alpha}$. Thus, in the definition of *fold-left* (*map car bss*) is classified as the dotted pre-type $\beta \ldots_{\beta}$ because *car* is instantiated at $((\textbf{Listof}\ \beta) \to \beta)$ and *bss* is classified as the dotted pre-type $(\textbf{Listof}\ \beta) \ldots_{\beta}$.

One way to use such processed rest parameters is in conjunction with *apply*. Specifically, if *apply* is passed a variable-arity function f, then its final argument l, which must be a list, must match up with the rest parameter of f. If the function is a uniform variable-arity procedure and the final argument is a list, typing the use of *apply* is straightforward. If it is a *non-uniform* variable-arity function, the number and types of parameters must match the elements and types of l.

Here is an illustrative example from the definition of *fold-left*:

```
(apply f c (car as) (map car bss))
```

By the type of *fold-left*, f has type $(\gamma\ \alpha\ \beta \ldots_{\beta} \to \gamma)$. The types of c and (*car as*) match the types of the initial parameters to f. Since the *map* application

$$
\begin{array}{rcl}
p & ::= & \texttt{=} \mid \texttt{plus} \mid \texttt{minus} \mid \texttt{mult} \mid \texttt{car} \mid \texttt{cdr} \mid \texttt{null?} \\
v & ::= & n \mid b \mid p \mid \texttt{null}_\tau \mid (\texttt{cons}_\tau\ v\ v) \mid (\lambda\ (\overrightarrow{[x:\tau]})\ e) \mid (\Lambda\ (\overrightarrow{\alpha})\ e) \\
 & \mid & (\Lambda\ (\overrightarrow{\alpha}\ \alpha\ \texttt{...})\ e) \mid (\lambda\ (\overrightarrow{[x:\tau]}\ \texttt{.}\ [x:\tau^*])\ e) \mid (\lambda\ (\overrightarrow{[x:\tau]}\ \texttt{.}\ [x:\tau\ \texttt{...}_\alpha])\ e) \\
e & ::= & v \mid x \mid (e\ \overrightarrow{e}) \mid (\texttt{if}\ e\ e\ e) \mid (\texttt{cons}_\tau\ e\ e) \mid \texttt{error}_\mathrm{L} \\
 & \mid & (\texttt{@}\ e\ \overrightarrow{\tau}) \mid (\texttt{@}\ e\ \overrightarrow{\tau}\ \tau\ \texttt{...}_\alpha) \mid (\texttt{apply}\ e\ \overrightarrow{e}\ e) \\
 & \mid & (\texttt{map}\ e\ e) \mid (\texttt{ormap}\ e\ e) \mid (\texttt{andmap}\ e\ e) \\
\tau & ::= & \textbf{Integer} \mid \textbf{Boolean} \mid \alpha \mid (\textbf{Listof}\ \tau) \mid (\overrightarrow{\tau} \to \tau) \\
 & \mid & (\overrightarrow{\tau}\ \tau^* \to \tau) \mid (\overrightarrow{\tau}\ \tau\ \texttt{...}_\alpha \to \tau) \mid (\forall\ (\overrightarrow{\alpha})\ \tau) \mid (\forall\ (\overrightarrow{\alpha}\ \alpha\ \texttt{...})\ \tau)
\end{array}
$$

Fig. 1. Syntax

has dotted pre-type (**Listof** β) $\ldots_\beta$ and since the rest parameter position of f is bounded by β, we are guaranteed that the length of the list produced by (*map car bss*) matches f's expectations about its rest argument. In short, we use the type system to show that we cannot have an arity mismatch, even in the case of *apply*.

4 A Variable-Arity Type System

The development of our formal model starts from the syntax of a multi-arity version of System F [10], enriched with variable-arity functions. An accompanying technical report [9] contains the full set of type rules as well as a semantics and soundness theorem for this model.

4.1 Syntax

We extend System F with multiple-arity functions at both the type and term level, lists, and uniform rest-argument functions. The use of multiple-arity functions establishes the proper problem context. Lists and uniform rest-argument functions suffice to explain how both kinds of variable-arity functions interact.

The grammar in figure 1 specifies the abstract syntax. We use a syntax close to that of Typed Scheme, including the use of `@` to denote type application. The use of the vector notation $\overrightarrow{e}$ denotes a (possibly empty) sequence of forms (in this case, expressions). In the form $\overrightarrow{e_k}^n$, n indicates the length of the sequence, and the term e_{k_i} is the ith element. The subforms of two sequences of the same length have the same subscript, so $\overrightarrow{e_k}^n$ and $\overrightarrow{\tau_k}^n$ are identically-sized sequences of expressions and types, respectively, whereas $\overrightarrow{e_j}^m$ is unrelated. If all vectors are the same size the sizes are dropped, but the subscripts remain. Otherwise the addition of starred pre-types, dotted type variables, dotted pre-types, and special forms is needed to operate on non-uniform rest arguments.

A *starred pre-type*, which has the form τ^*, is used in the types of uniform variable-arity functions whose rest parameter contains values of type τ. It only appears as the last element in the domain of a function type or as the type of a uniform rest argument.

A *dotted type variable*, which has the form α **...**, serves as a placeholder in a type abstraction. Its presence signals that the type abstraction can be applied to an arbitrary number of types. A dotted type variable can only appear as the last element in the list of parameters to a type abstraction. We call type abstractions that include dotted type variables *dotted type abstractions.*

A *dotted pre-type*, which has the form $\tau \ldots_\alpha$, is a type that is parameterized over a dotted type variable. When a type instantiation associates the dotted type variable α **...** with a sequence $\overrightarrow{\tau_k}^n$ of types, the dotted pre-type $\tau \ldots_\alpha$ is replaced by n copies of τ, where α in the ith copy of τ is replaced with τ_{k_i}. In the syntax, dotted pre-types can appear only in the rightmost position of a function type, as the type of a non-uniform rest argument, or as the last argument to `@`.

In this model the special forms `ormap`, `andmap`, and `map` are restricted to applications involving non-uniform rest arguments, and `apply` is restricted to applications involving rest arguments. In Typed Scheme, they also work for applications involving lists.

4.2 Type System

The type system is an extension of the type system of System F to handle the new linguistic constructs. We start with the changes to the environments and judgments, plus the major changes to the type validity relation. Next we present relations used for dotted types and expressions that have dotted pre-types instead of types. Then we discuss the changes to the standard typing relation, and finally we discuss the metafunctions used to define the new typing judgments.

The environments and judgments used in our type system are similar to those used for System F except as follows:

– The type variable environment (Δ) includes both dotted and non-dotted type variables.
– There is a new class of environments (Σ), which map non-uniform rest parameters to dotted pre-types.
– There is also an additional validity relation $\Delta \rhd \tau \ldots_\alpha$ for dotted pre-types.
– The use of Σ makes typing relation $\Gamma, \Delta, \Sigma \vdash e : \tau$ a five-place relation.
– There is an additional typing relation $\Gamma, \Delta, \Sigma \vdash e \rhd \tau \ldots_\alpha$ for assigning dotted pre-types to expressions.

The type validity relation checks the validity of two forms—types and dotted type variables. The additional rules for establishing type validity of non-uniform variable-arity types are provided below, along with an additional relation which checks the validity of dotted pre-types.

$$\text{TE-DVar}\ \frac{\alpha \ldots \in \Delta}{\Delta \vdash \alpha \ldots} \qquad \text{TE-DFun}\ \frac{\Delta \rhd \tau_r \ldots_\alpha \quad \overrightarrow{\Delta \vdash \tau_j} \quad \Delta \vdash \tau}{\Delta \vdash (\overrightarrow{\tau_j}\ \tau_r \ldots_\alpha \to \tau)} \qquad \text{TE-DAll}\ \frac{\Delta \cup \{\overrightarrow{\alpha_j}, \beta \ldots\} \vdash \tau}{\Delta \vdash (\forall\ (\overrightarrow{\alpha_j}\ \beta \ldots)\ \tau)} \qquad \text{TDE-Pretype}\ \frac{\Delta \vdash \alpha \ldots \quad \Delta \cup \{\alpha\} \vdash \tau}{\Delta \rhd \tau \ldots_\alpha}$$

When validating a dotted pre-type $\tau \ldots_{\alpha}$, the bound α is checked to make sure that it is indeed a valid dotted type variable. Then τ is checked in an environment where the bound is allowed to appear free. It is possible for a dotted pre-type to be nested somewhere within a dotted pre-type over the same bound, e.g.

$$(\forall\ (\alpha\ \ldots)\ ((\alpha\ \ldots_{\alpha} \rightarrow \alpha)\ \ldots_{\alpha} \rightarrow (\alpha\ \ldots_{\alpha} \rightarrow (\textbf{Listof Integer}))))$$

To illustrate how such a type might be used, we instantiate this sample type with the sequence of types **Integer Boolean**:

$$((\textbf{Integer Boolean} \rightarrow \textbf{Integer})\ (\textbf{Integer Boolean} \rightarrow \textbf{Boolean})$$
$$\rightarrow (\textbf{Integer Boolean} \rightarrow (\textbf{Listof Integer})))$$

There are two functions in the domain of the type, each of which corresponds to an element in our sequence. All functions have the same domain—the sequence of types; the *i*th function returns the *i*th type in the sequence.

$$\textsc{TD-Var}\ \frac{\Sigma(x) = \tau \ldots_{\alpha}}{\Gamma, \Delta, \Sigma \vdash x \triangleright \tau \ldots_{\alpha}} \qquad \textsc{TD-Map}\ \frac{\Gamma, \Delta, \Sigma \vdash e_r \triangleright \tau_r \ldots_{\alpha} \qquad \Gamma, \Delta \cup \{\alpha\}, \Sigma \vdash e_f : (\tau_r \rightarrow \tau)}{\Gamma, \Delta, \Sigma \vdash (\texttt{map}\ e_f\ e_r) \triangleright \tau \ldots_{\alpha}}$$

The preceding rules are the typing rules for the two forms of expressions that have dotted pre-types. The TD-Var rule just checks for the variable in Σ. The TD-Map rule assigns a type to a function position. Since the function needs to operate on each element of the sequence represented by e_r, not on the sequence as a whole, the domain of the function's type is the base τ_r instead of the dotted type $\tau_r \ldots_{\alpha}$. This type may include free references to the bound α, however. Therefore, we must check the function in an environment extended with α as a regular type variable.

As expected, most of the typing rules are simple additions of multiple-arity type and term abstractions and lists to System F. For uniform variable-arity functions, the introduction rule treats the rest parameter as a variable whose type is a list of the appropriate type. There is only one elimination rule, which deals with the special form `apply`; other eliminations such as direct application to arguments are handled via the coercion rules.

The type rules in figure 2 concern non-uniform variable-arity functions. These functions also have one introduction and one elimination rule. The rule T-Ormap and its absent counterpart T-Andmap are similar to that of TD-Map in that the dotted pre-type bound of the second argument is allowed free in the type of the first argument. In contrast to uniform variable-arity functions, non-uniform ones cannot be applied directly to arguments in this calculus.

While T-DTAbs, the introduction rule for dotted type abstractions, follows from the rule for normal type abstractions, the elimination rules are quite different. There are two elimination rules: T-DTApp and T-DTAppDots. The former handles type application of a dotted type abstraction where the dotted type variable corresponds to a sequence of types, and the latter deals with the case when the dotted type variable corresponds to a dotted pre-type.

T-DAbs

$$\frac{\overrightarrow{\Delta \vdash \tau_k} \qquad \Delta \triangleright \tau_r \dots_\alpha \qquad \Gamma[\overrightarrow{x_k \mapsto \tau_k}], \Delta, \Sigma[x_r \mapsto \tau_r \dots_\alpha] \vdash e : \tau}{\Gamma, \Delta, \Sigma \vdash (\lambda\ (\overrightarrow{[x_k : \tau_k]}\ . [x_r : \tau_r \dots_\alpha])\ e) : (\overrightarrow{\tau_k}\ \tau_r \dots_\alpha \to \tau)}$$

T-DApply

$$\frac{\begin{array}{c}\Gamma, \Delta, \Sigma \vdash e_f : (\overrightarrow{\tau_k}\ \tau_r \dots_\alpha \to \tau) \\ \overrightarrow{\Gamma, \Delta, \Sigma \vdash e_k : \tau_k} \qquad \Gamma, \Delta, \Sigma \vdash e_r \triangleright \tau_r \dots_\alpha\end{array}}{\Gamma, \Delta, \Sigma \vdash (\texttt{apply}\ e_f\ \overrightarrow{e_k}\ e_r) : \tau}$$

T-Ormap

$$\frac{\begin{array}{c}\Gamma, \Delta, \Sigma \vdash e_r \triangleright \tau_r \dots_\alpha \\ \Gamma, \Delta \cup \{\alpha\}, \Sigma \vdash e_f : (\tau_r \to \mathbf{Boolean})\end{array}}{\Gamma, \Delta, \Sigma \vdash (\texttt{ormap}\ e_f\ e_r) : \mathbf{Boolean}}$$

T-DTAbs

$$\frac{\Gamma, \Delta \cup \{\overrightarrow{\alpha_k}, \beta \dots\}, \Sigma \vdash e : \tau}{\Gamma, \Delta, \Sigma \vdash (\Lambda\ (\overrightarrow{\alpha_k}\ \beta \dots)\ e) : (\forall\ (\overrightarrow{\alpha_k}\ \beta \dots)\ \tau)}$$

T-DTApp

$$\frac{\overrightarrow{\Delta \vdash \tau_j}^n \qquad \overrightarrow{\Delta \vdash \tau_k}^m \qquad \overrightarrow{\beta_k}^m \text{ fresh} \qquad \Gamma, \Delta, \Sigma \vdash e : (\forall\ (\overrightarrow{\alpha_j}^n\ \beta \dots)\ \tau)}{\Gamma, \Delta, \Sigma \vdash (\texttt{@}\ e\ \overrightarrow{\tau_j}^n\ \overrightarrow{\tau_k}^m) : td_\tau(\tau[\overrightarrow{\alpha_j \mapsto \tau_j}^n], \beta, \overrightarrow{\beta_k}^m)[\overrightarrow{\beta_k \mapsto \tau_k}^m]}$$

T-DTAppDots

$$\frac{\overrightarrow{\Delta \vdash \tau_k} \qquad \Delta \triangleright \tau_r \dots_\beta \qquad \Gamma, \Delta, \Sigma \vdash e : (\forall\ (\overrightarrow{\alpha_k}\ \alpha_r \dots)\ \tau)}{\Gamma, \Delta, \Sigma \vdash (\texttt{@}\ e\ \overrightarrow{\tau_k}\ \tau_r \dots_\beta) : sd(\tau[\overrightarrow{\alpha_k \mapsto \tau_k}], \alpha_r, \tau_r, \beta)}$$

Fig. 2. Selected Type Rules

$$\begin{aligned}
&sd(\alpha_r, \alpha_r, \tau_r, \beta) &&= \tau_r \\
&sd(\alpha, \alpha_r, \tau_r, \beta) &&= \alpha \qquad \text{where } \alpha \neq \alpha_r \\
&sd((\overrightarrow{\tau_j}\ \tau'_r \dots_{\alpha_r} \to \tau), \alpha_r, \tau_r, \beta) = \\
&\qquad (\overrightarrow{sd(\tau_j, \alpha_r, \tau_r, \beta)}\ \ sd(\tau'_r, \alpha_r, \tau_r, \beta) \dots_\beta \to sd(\tau, \alpha_r, \tau_r, \beta)) \\
&sd((\overrightarrow{\tau_j}\ \tau'_r \dots_{\alpha} \to \tau), \alpha_r, \tau_r, \beta) = \\
&\qquad (\overrightarrow{sd(\tau_j, \alpha_r, \tau_r, \beta)}\ \ sd(\tau'_r, \alpha_r, \tau_r, \beta) \dots_\alpha \to sd(\tau, \alpha_r, \tau_r, \beta)) \qquad \text{where } \alpha \neq \alpha_r \\
&sd((\forall\ (\overrightarrow{\alpha_j}\ \alpha \dots)\ \tau), \alpha_r, \tau_r, \beta) &&= (\forall\ (\overrightarrow{\alpha_j}\ \alpha \dots)\ \ sd(\tau, \alpha_r, \tau_r, \beta))
\end{aligned}$$

$$\begin{aligned}
&td_\tau((\overrightarrow{\tau_j}^n\ \tau_r \dots_\beta \to \tau), \beta, \overrightarrow{\beta_k}^m) = \\
&\qquad (\overrightarrow{td_\tau(\tau_j, \beta, \overrightarrow{\beta_k}^m)}^n\ \ \overrightarrow{td_\tau(\tau_r, \beta, \overrightarrow{\beta_k}^m)[\beta \mapsto \beta_k]}^m \to td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m)) \\
&td_\tau((\overrightarrow{\tau_j}^n\ \tau_r \dots_\alpha \to \tau), \beta, \overrightarrow{\beta_k}^m) = \\
&\qquad (\overrightarrow{td_\tau(\tau_j, \beta, \overrightarrow{\beta_k}^m)}^n\ \ td_\tau(\tau_r, \beta, \overrightarrow{\beta_k}^m) \dots_\alpha \to td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m)) \qquad \text{where } \alpha \neq \beta
\end{aligned}$$

Fig. 3. Subst-dots and trans-dots

The T-DTAppDots rule is more straightforward, as it is just a substitution rule. Replacing a dotted type variable with a dotted pre-type is more involved than normal type substitution, however, because we need to replace the dotted type variable where it appears as a dotted pre-type bound. The metafunction *sd* performs this substitution. Selected cases of the definition of *sd* appear in figure 3; the remaining clauses perform structural traversals.

The T-DTApp rule must first expand out dotted pre-types that use the dotted type variable before performing the appropriate substitutions. To do this it uses the metafunction td_τ on a sequence of fresh type variables of the appropriate length to expand dotted pre-types that appear in the body of the abstraction's type into a sequence of copies of their base types. These copies are first expanded with td_τ and then in each copy the free occurrences of the bound are replaced with the corresponding fresh type variable. Normal substitution is performed on the result of td_τ, mapping each fresh type variable to its corresponding type argument. The interesting cases of the definition of td_τ also appear in figure 3.

5 Evaluation

Mining the extensive PLT Scheme code base provides significant evidence that variable-arity functions are frequently defined and used; examining a fair number of examples shows that our type system is able to cope with a good portion of these definitions and uses.

5.1 Measurements of Existing Code

A simple pattern-based search of the code base for definitions of variable-arity functions and uses of certain built-in core functions produces the following:

– There are at least 1761 definitions of variable-arity functions.
– There are 488 uses of *map*, *for-each*, *foldl*, *foldr*, *andmap*, and *ormap* with more than the minimum number of arguments.

These numbers demonstrate the need for a type system that deals with variable-arity functions. Programmers use those from the core library at multiple arities. Furthermore, programmers define such functions regularly.

It is this kind of inspection of our code base that inspires a careful investigation of the issue of variable-arity functions. We cannot reasonably ask our programmers to duplicate their code or to duplicate type cases just because our type system does not accommodate their utilization of the expressive power of plain Scheme.

5.2 Evaluation of Examples

Simply counting definitions and uses of variable-arity functions is insufficient. Each definition and use demands a separate inspection in order to validate that our type system can cope with it. This is particularly necessary for function definitions, because our pattern-based search does not indicate whether a definition introduces a uniform or non-uniform variable-arity function.

Uses The sample set for uses of variable-arity functions from the core library covers 30 cases, i.e., 10 randomly-chosen example function applications using each of *map*, *for-each*, and *andmap* with at least two list arguments. For *map*, we are able to type 9 of 10, for *for-each* we are able to type 10 of 10, for *andmap* we are able to type 10 of 10.

In short, our technique is extremely successful for the list-processing functions, checking 29 of the 30 examples. The one failure is due to the use of a list to represent a piece of information that comprises four pieces. In this case, our type system simply does not preserve the length information for the list from the input to *map*.

Definitions. The sample set for definitions of variable-arity functions covers some 120 cases (or some 7%) from the code base. Our findings naturally sort these samples into three categories:

– A majority of the functions can be typed with uniform rest arguments or use variable arity to simulate optional arguments. For the latter, we recommend that programmers rewrite such functions using **case-lambda**.
– Twelve of the 120 inspected definitions are non-uniform and require variable-arity polymorphism. Our type checker can assign types to all of them. Returning to our example in section 3.2, *verbose* can be given the type

$$(\forall\ (\beta\ \alpha\ \ldots)\ ((\alpha\ \ldots_\alpha \to \beta) \to (\alpha\ \ldots_\alpha \to \beta))).$$

– The small remainder cannot be typed using our system.

These inspections demonstrate two important points. First, all of the various ways in which Typed Scheme handles varying numbers of arguments are important for type-checking existing Scheme code. Second, our design choices for variable-arity polymorphism mostly capture the programming style used in practice by working PLT Scheme programmers. In conclusion, we conjecture that our type system can validate more than 95% of the uses of heterogeneous library functions such as *map*, that it can check 70% of the close to 1800 definitions, 10% of which require the heterogeneous version of variable-arity polymorphism.

6 Related Work

Variable-arity functions are nearly ubiquitous in the world of programming languages, but no typed language supports them in a systematic and principled manner. Here we survey existing systems as well as several theoretical efforts.

ANSI C provides "varargs," but the functions that implement this functionality serve as a thin wrapper around direct access to the stack frame. Java [11] and C# are two statically typed languages that have only uniform variable-arity functions, since access occurs via a homogeneous array.

Dzeng and Haynes [12] come close to our goal of providing a practical type system for variable-arity functions. As part of the Infer system for type-checking Scheme [13], they use an encoding of "infinitary tuples" as row types for an ML-like type inference system that handles optional arguments and uniform and non-uniform variable-arity functions.

By comparison to our work, Dzeng and Haynes' system has several limitations. Most importantly, since their system does not support first-class polymorphic functions, they are unable to type many of the definitions of variable-arity functions, such as *map* or *fold*. Additionally, their system requires full type inference to avoid exposing users to the underlying details of row types, and it is also designed around a Hindley-Milner style algorithm. This renders it incompatible with the remainder of the design of Typed Scheme, which is based on a system with subtyping.

Gregor and Järvi [14] propose an extension for variadic templates to C++ for the upcoming C++0x standard. This proposal has been accepted by the C++ standardization committee. Variadic templates provide a basis for implementing non-uniform variable-arity functions in templates. Since the approach is grounded in templates, it is difficult to translate their approach to other languages without template systems. The template approach addresses a simpler problem, since template expansion is a pre-processing step and types are only checked after template expansion. It also significantly complicates the language, since arbitrary computation can be performed during template expansion. Further, the template approach prevents checking of variadic functions at the definition site, meaning that errors in the definition are only caught when the function is used.

Tullsen [15] attempts to bring non-uniform variable-arity functions to Haskell via the Zip Calculus, a type system with restricted dependent types and special kinds that serve as tuple dimensions. This work is theoretical and comes without practical evaluation. The presented limitations of the Zip Calculus imply that it cannot assign a variable-arity type to the definition of `zipWith` (Haskell's name for Scheme's *map*) without further extension, whereas Typed Scheme can do so.

Similarly, McBride [16] and Moggi [17] present restricted forms of dependent typing in which the number of arguments is passed as a parameter to variadic functions. Our system, while not allowing the expression of every dependently-typable program, is simpler than dependent typing, suffices for most examples we have encountered, and does not require an extra function parameter.

7 Conclusion

In this paper, we have presented a design for polymorphic functions with variable arity. Our system accommodates both uniform and non-uniform variadic functions. We also validated our design against existing Scheme and Typed Scheme code. Typed Scheme with variable-arity polymorphism is part of the latest release of PLT Scheme (4.1), available from `http://plt-scheme.org/`.

In closing, we leave the reader with a final observation on the nature of variable-arity polymorphism. Many existing languages allow functions that accept a variable number of arguments, all of a uniform type. Such functions have types of the form $\tau^* \rightarrow \tau$. To accommodate variable-arity polymorphism, however, we must lift this abstraction one level up. For example, given the type ($\forall$ $(\alpha \ldots)$ $(\alpha \ldots_\alpha \rightarrow$ **Boolean**)), the *kind* of this type is simply $\star^* \rightarrow \star$. So we see that *non-uniform* variable arity at the type level is reflected in *uniform* variable arity at the kind level.

References

1. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage Migration: From Scripts to Programs. In: DLS 2006, Companion to OOPSLA, pp. 964–974 (2006)
2. Tobin-Hochstadt, S., Felleisen, M.: The Design and Implementation of Typed Scheme. In: POPL, pp. 395–406 (2008)
3. Flatt, M.: PLT MzScheme: Language Manual. Technical Report PLT-TR2008-1-v4.0.2, PLT Scheme Inc. (2008), `http://www.plt-scheme.org/techreports/`
4. Gansner, E.R., Reppy, J.H.: The Standard ML Basis Library. Cambridge University Press, New York (2002)
5. The GHC Team: The Glasgow Haskell Compiler User's Guide (2008)
6. Findler, R.B., Felleisen, M.: Contracts for Higher-Order Functions. In: ACM SIGPLAN International Conference on Functional Programming, pp. 48–59 (2002)
7. Pierce, B.C., Turner, D.N.: Local Type Inference. ACM Trans. Program. Lang. Syst. 22(1), 1–44 (2000)
8. Dybvig, R.K., Hieb, R.: A new approach to procedures with variable arity. Lisp and Symbolic Computation: An International Journal 3(3) (1990)
9. Strickland, T.S., Tobin-Hochstadt, S., Felleisen, M.: Variable-Arity Polymorphism. Technical Report NU-CCIS-08-03, Northeastern University (2008)
10. Girard, J.Y.: Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In: Fenstad, J.E. (ed.) Proceedings of the Second Scandinavian Logic Symposium, pp. 63–92. North-Holland Publishing Co., Amsterdam (1971)
11. Gosling, J., Joy, B.: The Java Language Specification, 3rd edn. Addison-Wesley, Reading (2005)
12. Dzeng, H., Haynes, C.T.: Type Reconstruction for Variable-Arity Procedures. In: LFP 1994, pp. 239–249. ACM Press, New York (1994)
13. Haynes, C.T.: Infer: A Statically-typed Dialect of Scheme. Technical Report 367, Indiana University (1995)
14. Gregor, D., Järvi, J.: Variadic templates for C++. In: SAC 2007, pp. 1101–1108. ACM Press, New York (2007)
15. Tullsen, M.: The Zip Calculus. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 28–44. Springer, Heidelberg (2000)
16. McBride, C.: Faking it: Simulating Dependent Types in Haskell. J. Funct. Program. 12(5), 375–392 (2002)
17. Moggi, E.: Arity polymorphism and dependent types. In: APPSEM Workshop on Subtyping and Dependent Types in Programming (July 7, 2000) (invited talk)