Variable-Arity Polymorphism

T. Stephen Strickland Sam Tobin-Hochstadt Matthias Felleisen

PLT @ Northeastern University

{sstrickl,samth,matthias}@ccs.neu.edu

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 many languages, such as Scheme, Python, and other scripting languages. Statically typed languages, such as C and Java, accommodate modest forms of variable-arity "rest" arguments via homogeneous data structures (plus casts). Even languages with highly expressive type systems such as ML or Haskell, however, cannot type check the typical variety of variable-arity functions found in untyped functional languages; instead their standard libraries contain numerous copies of the same function definition, with slightly different names.

As part of the Typed Scheme project—an effort to design an explicitly typed sister language for PLT Scheme—we have designed, implemented, and evaluated a highly expressive type system for variable-arity functions. In order to make the type system convenient to use for Scheme programmers who wish to convert untyped modules, the implementation of the type system comes with a local type inference algorithm that restores types for polymorphic applications whenever possible. Our practical validation of the type system has confirmed for a small sample that the type system should now support most use cases in the extensive PLT Scheme code base.

1. Types for Variable-Arity Functions

For the past two years, Tobin-Hochstadt and Felleisen (2006, 2008) have been developing Typed Scheme, an explicitly and statically typed sister language of PLT Scheme (Flatt 2008). In many cases, Typed Scheme accommodates existing Scheme programming idiom 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 revised Typed Scheme's type system so that it can cope 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 to a homogeneous data structure.[1] A fair number of other variable-arity functions, however, demand a far more sophisticated approach than allocating rest arguments in homogeneous data structures.

Consider the *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 stepwise fashion. The function must therefore take precisely as many arguments as the number of lists provided. For example, if the *make-student* function takes 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 of students. We refer to variable-arity functions such as + and *map* as having *uniform* and *non-uniform* rest parameter types, respectively.

For Typed Scheme to be useful to working programmers, its type system must handle all of these cases. Further, although *map* and + are part of the standard library, language implementors cannot arrogate the ability to abstract over the arities of functions. Scheme programmers routinely define such functions, and if any of them 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 present in numerous other typed languages, such as C, C++, and Java. Even highly expressive typed functional languages offer such functions in their libraries, but do so via copying of code. For example, the SML Basis Library (Gansner and Reppy 2002) includes the ARRAY_MAP and ARRAY_MAP2 signatures, which could be abstracted into a single code component using our system. The GHC (The GHC Team 2008) standard library also features close to a dozen families of functions defined at a variety of arities, such as `zipWith`, `zipWith3`, `zipWith4` and so on.[2] In fact, the `zipWith` function, in full generality, simply *is* the standard *map* function from Scheme, as described above.

This paper is organized as follows. In the next two sections, we describe Typed Scheme, first in general terms and then the type system for variable-arity functions; the examples involve the definition and use of such functions with both uniform and non-uniform rest parameter types. In section 4, we introduce a formal model of our variable-arity type system. In section 5, we describe how our innovations are integrated with the rest of Typed Scheme and sketch a local type inference algorithm. In section 6 we present some preliminary results concerning a comprehensive evaluation effort with respect to our code base and the limitations of our system. In section 7 we discuss related work, and in section 8 we conclude with a unifying perspective.

2. Typed Scheme . . .

The goal of our Typed Scheme project is to design a typed sister language for an untyped scripting language so that programmers can transfer programs into the typed world on a module-by-module basis. Like PLT Scheme, Typed Scheme is a modular programming language; unlike plain Scheme programs, Typed Scheme programs come with explicit type annotations for functions and structures that are statically checked. Typed Scheme also provides integration with untyped PLT Scheme code, allowing a typed program to link in untyped modules and vice versa. The mechanism exploits functional contracts (Findler and Felleisen 2002) to guarantee a gen-

[1] Such systems are found in C, C++, Java, C# and many other languages.

[2] The full list of such functions is: `zipWith`, `zip`, `unzip`, `liftA`, `liftM`, `typeOf`, `typeOfDefault`, `gcast` and `dataCast`.

eralized type soundness theorem (Tobin-Hochstadt and Felleisen 2006). Interlanguage refactoring can therefore proceed gradually.

Typed Scheme supports this gradual refactoring with a type system that accommodates the standard Scheme programming idioms without (much) code modification. In principle, Scheme programmers need only annotate structure and function headers with types to move a module into the Typed Scheme world; on occasion, they may also have to define a type alias to keep type expressions concise. For this purpose, 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

Good 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 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. 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.[3] For example, *list-ref* has the type $(\forall\ (\alpha)\ ((\textbf{Listof}\ \alpha)\ \textbf{Integer} \rightarrow \alpha))$. 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 illustrates 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 explicitly type instantiation. Instead, the required type instantiations are synthesized from the types of the arguments.

[3] Such functions are not always parametric, because occurrence typing can be used to examine the arguments.

Argument type synthesis uses the local type inference algorithm of Pierce and Turner (2000). 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 permit fixed and variable-arity parameter specifications:

1. (**lambda** (*x y z*) (+ *x* (∗ *y z*))) creates a function of three arguments (and one result) and (**define** (*f x y z*) (+ *x* (∗ *y z*))) 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*))) is another way of defining such a function;
4. (**lambda** *x* (*apply* + *x*)) creates a function of an arbitrary number of arguments; and
5. (**define** (*h . x*) (*apply* + *x*)) is the equivalent 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).

Schemers use variable-arity functions for many situations, including those that really call for different mechanisms. Thus, while programs in plain Scheme employ variable-arity functions to encode functions with optional and keyword arguments, PLT Scheme programs tend to replace such uses with **case-lambda**, **opt-lambda**, **lambda/kw** or equivalent **define** forms.

This section sketches how the revised version of Typed Scheme accommodates variable-arity functions. Our revision focuses on the standard uses of such functions, not encodings of other mechanisms.

3.1 Uniform Variable-Arity Functions

Uniform variable-arity functions expect their rest parameter to be a homogeneous list. Consider the following four examples:

```
(: + (Integer* → Integer))
(: − (Integer 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 hypothetical 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), and it maps to an already-existing type in Typed Scheme. Thus, no further work is needed to handle uses of such rest parameters.

3.2 Approximating Non-Uniform Variable-Arity Functions

Non-uniform variable-arity functions do not assume that their rest parameter is a homogeneous list of values. Often though, homogeneity is traded for some other assumptions concerning the length of the argument list, its relationship to the required arguments, and even its connection to the results of the function.

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 values. Also, the type of the kth function parameter must match the type of the kth list.

Prior to our recent revision, Typed Scheme approximated the types of functions like *map* as follows:

```
(: map (∀ (C A B)
          (case-lambda
           [((A → C) (Listof A)) (Listof C)]
           [((A B → C) (Listof A) (Listof B)) (Listof C)])))
```

This **case-lambda** type contains a series of clauses which consist of a list of domain types and a range type. Here the domain lists of the clause must be of distinct lengths. When a function with a **case-lambda** type is applied, the number of arguments to the function determines which clause is used as the type of the function for that application site.

So the above (simplistic) type for *map* provides two different types at which it can be applied: the type that corresponds to mapping a unary function and the one for mapping a binary function. This means that in this following program fragment

```
(define-struct: book ([name : String] [author : String]
                      [year : Integer]))

(map not (list #t #f #t))
(map = (list 1 20 300) (list 10 20 30))
(map make-book (list "Flatland") (list "A. Square")
               (list 1884))
```

the first two uses of *map* successfully type-check, but the third doesn't.

While **case-lambda** is a good stop-gap measure for providing types for functions like *map*, repeated requests for extending and re-extending such types for widely-used functions in Scheme provided the incentive to search for a better solution.

3.3 Non-Uniform Variable-Arity Functions

Now Typed Scheme can represent such non-uniform variable-arity functions natively. Below are the types for some example functions:

```
;; map is the standard Scheme map
(: map
 (∀ (γ α β ...)
  ((α β ..._β → γ) (Listof α) (Listof β) ..._β → (Listof γ))))

;; app takes a function and a sequence of arguments,
;; and performs the application
(: app (∀ (β α ...) ((α ..._α → β) α ..._α → β)))

;; 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
 (∀ (β α ...) ((α ..._α → β)* → (α ..._α → (Listof β)))))
```

The key innovation is the possibility to attach ... to the last type variable in the binding position of a ∀ type constructor. Such type variables are dubbed *dotted type variables*. Dotted type variables signal that polymorphic types can be instantiated at an arbitrary number of types.

Next, the body of ∀ 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. Also, if a dotted type variable α appears as a type, it must show up within the base of a dotted pre-type whose bound is α.

When a dotted polymorphic type is instantiated, any dotted pre-types are expanded by copying the base the 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* with the sequence of types

Number Integer Boolean String

results in the type:

((**Integer Boolean String** → **Number**)* →
 (**Integer Boolean String** → (**Listof Number**)))

Typed Scheme also provides local inference of the appropriate type arguments for dotted polymorphic functions, so direct type instantiation is rarely needed. The example uses of *map* from section 3.2 can be successfully inferred at the appropriate types:

```
(map not (list #t #f #t))
;; map is instantiated (via local type inference) at:
;; ((Boolean → Boolean)
;;  (Listof Boolean)
;;  → (Listof Boolean))

(map = (list 1 20 300) (list 10 20 30))
;; ((Integer Integer → Boolean)
;;  (Listof Integer) (Listof Integer)
;;  → (Listof Boolean))

(map make-book (list "Flatland") (list "A. Square")
               (list 1884))
;; ((String String Integer → book)
;;  (Listof String) (Listof String) (Listof Integer)
;;  → (Listof book))
```

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

```
(: fold-left
  (∀ (γ α β ..._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 reasonably complete definition of *fold-left*. Its type shows that at least three items: a rest-argument function f; an "identity" element c for the fold; 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 eventually becomes an argument.

Beyond this, the example illustrates how 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 function.

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 relatively easy to overcome with a conversion from dotted pre-types to list types, the second one is ominous. After all, list-processing functions don't return dotted pre-types but list types, and we can't possibly expect that such list types come with enough information.

Our pragmatic compromise is to allow some list processing with certain functions. For example, functions like *map* return a list of the same length as the given list and the component types are in a predictable order. Thus, if *xs* is classified by the dotted pre-type $\tau \ldots_\alpha$, and f has type $(\tau \to \sigma)$, then (*map f xs*) is classified by the dotted pre-type $\sigma \ldots_\alpha$. For example, in the definition of *fold-left* (*map car bss*) is classified as the dotted pre-type $\beta \ldots_\beta$ because *car* is instantiated at ((**Listof** β) $\to \beta$) and *bss* is classified as the dotted pre-type (**Listof** β) $\ldots_\beta$.

Contrast this behavior with functions that arbitrarily mangle the given lists. One obvious example is *filter*, which shortens the list. Because we don't have any guarantees about these results, we can't possibly predict how the result type relates to a dotted pre-type and therefore the results of such expressions cannot be used in conjunction with rest parameters.

One way to use processed rest parameters is in conjunction with *apply*. Specifically, if *apply* consumes 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, then 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

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

from the definition of *fold-left*, f has type (γ α β $\ldots_\beta \to \gamma$). The types of *c* and (*car as*) match the types of the initial parameters to f. Since the *map* application has 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 the length that f expects its rest argument to be. In short, we have just used the type system to show that we don't have an arity mismatch in this example.

The next section demonstrates how a type system can both deal with rest arguments in a reasonably flexible manner and have the right properties. After that, we show how to integrate it into a practical system and that it is practical.

4. A Variable-Arity Type System

For the purpose of this paper, Typed Scheme is an implementation of System F (Girard 1971; Reynolds 1974), a λ calculus with first-class parametric polymorphism. We therefore present the theoretical essence of our result as an extension of this system. We start with the syntax of a multi-arity version of System F, enriched with variable-arity functions, and then present both a type system and a semantics. The section concludes with a 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 uniform and non-uniform variable-arity functions interact.

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

Figure 1. Syntax

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 will have the same subscript, so $\overrightarrow{e_k}^n$ and $\overrightarrow{\tau_k}^n$ are similarly-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. Other than notational differences, the major changes to the syntax are the addition of starred pre-types, dotted type variables, dotted pre-types, and special forms that are 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 τ **...**$_\alpha$, 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 τ **...**$_\alpha$ 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 only appear in the rightmost position of a function type, as the type of a non-uniform rest argument, or as the last argument to `@`.

The special forms `any`, `all`, and `lift` are the same as Scheme's *ormap*, *andmap*, and *map*, but they are restricted to applications involving non-uniform rest arguments. The forms `apply` * and `apply` correspond to Scheme's *apply* on uniform and non-uniform variable-arity functions respectively. Typed Scheme does not use different names for these functions when used on rest arguments, but the model distinguishes them to simplify the presentation and the soundness proof.

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. We then outline 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 show the metafunctions used in some of the new typing judgments and discuss their details.

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 $\Delta \triangleright \tau \ldots_\alpha$ for dotted pre-types.
- The typing relation $\Gamma, \Delta, \Sigma \vdash e : \tau$ is a five-place relation with the addition of Σ.
- There is an additional typing relation $\Gamma, \Delta, \Sigma \vdash e \triangleright \tau \ldots_\alpha$ for assigning dotted pre-types to expressions.

$$\text{TE-BOOL}\ \ \Delta \vdash \textbf{Boolean} \qquad \text{TE-NUM}\ \ \Delta \vdash \textbf{Integer}$$

$$\text{TE-VAR}\ \ \frac{\alpha \in \Delta}{\Delta \vdash \alpha} \qquad \text{TE-DVAR}\ \ \frac{\alpha \ldots \in \Delta}{\Delta \vdash \alpha \ldots}$$

$$\text{TE-FUN}\ \ \frac{\overrightarrow{\Delta \vdash \tau_j} \qquad \Delta \vdash \tau}{\Delta \vdash (\overrightarrow{\tau_j} \to \tau)} \qquad \text{TE-DFUN}\ \ \frac{\overrightarrow{\Delta \vdash \tau_j} \qquad \Delta \triangleright \tau_r \ldots_\alpha \qquad \Delta \vdash \tau}{\Delta \vdash (\overrightarrow{\tau_j}\ \tau_r \ldots_\alpha \to \tau)}$$

$$\text{TE-ALL}\ \ \frac{\Delta \cup \{\overrightarrow{\alpha_j}\} \vdash \tau}{\Delta \vdash (\forall\ (\overrightarrow{\alpha_j})\ \tau)} \qquad \text{TE-DALL}\ \ \frac{\Delta \cup \{\overrightarrow{\alpha_j}, \beta \ldots\} \vdash \tau}{\Delta \vdash (\forall\ (\overrightarrow{\alpha_j}\ \beta \ldots)\ \tau)}$$

$$\text{TE-SFUN}\ \ \frac{\overrightarrow{\Delta \vdash \tau_j} \qquad \Delta \vdash \tau_s \qquad \Delta \vdash \tau}{\Delta \vdash (\overrightarrow{\tau_j}\ \tau_s^* \to \tau)} \qquad \text{TE-LIST}\ \ \frac{\Delta \vdash \tau}{\Delta \vdash (\textbf{Listof}\ \tau)}$$

Figure 2. Type validity

The rules for establishing type validity are provided in figure 2. This relation checks the validity of two forms—types and dotted type variables—and is straightforward. The interesting relation checks the validity of dotted pre-types:

$$\text{TDE-PRETYPE}\ \ \frac{\Delta \vdash \alpha \ldots \qquad \Delta \cup \{\alpha\} \vdash \tau}{\Delta \triangleright \tau \ldots_\alpha}$$

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.

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

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

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

There are three 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 ith function returns the ith type in the sequence.

$$\text{TD-VAR}\ \ \frac{\Sigma(x) = \tau \ldots_\alpha}{\Gamma, \Delta, \Sigma \vdash x \triangleright \tau \ldots_\alpha}$$

$$\text{TD-LIFT}\ \ \frac{\Gamma, \Delta, \Sigma \vdash e_r \triangleright \tau_r \ldots_\alpha \qquad \Gamma, \Delta \cup \{\alpha\}, \Sigma \vdash e_f : (\tau_r \to \tau)}{\Gamma, \Delta, \Sigma \vdash (\texttt{lift}\ e_f\ e_r) \triangleright \tau \ldots_\alpha}$$

Figure 3. Type rules for dotted pre-types

The typing rules for the two forms of expressions that have dotted pre-types are given in figure 3. TD-VAR is straightforward, because it just checks for the variable in Σ. TD-LIFT 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.

$$\text{TC-TODFUN}\ \ \frac{\Gamma, \Delta, \Sigma \vdash e : (\overrightarrow{\tau_j}\ \tau_s^* \to \tau) \qquad \Delta \vdash \alpha \ldots \qquad \Delta - \{\alpha\} \vdash \tau_s}{\Gamma, \Delta, \Sigma \vdash e : (\overrightarrow{\tau_j}\ \tau_s \ldots_\alpha \to \tau)}$$

$$\text{TC-TOFUN}\ \ \frac{\Gamma, \Delta, \Sigma \vdash e : (\overrightarrow{\tau_j}\ \tau_s^* \to \tau)}{\Gamma, \Delta, \Sigma \vdash e : (\overrightarrow{\tau_j} \to \tau)} \qquad \text{TC-UNROLL}\ \ \frac{\Gamma, \Delta, \Sigma \vdash e : (\overrightarrow{\tau_j}\ \tau_s^* \to \tau)}{\Gamma, \Delta, \Sigma \vdash e : (\overrightarrow{\tau_j}\ \tau_s\ \tau_s^* \to \tau)}$$

Figure 4. Coercion from uniform variable-arity types

Before covering the other changes to the typing rules, we first look at how uniform variable-arity functions can be used in different contexts, since this shapes our typing judgments for such functions. If we have an expression of uniform variable-arity type $(\overrightarrow{\tau_j}\ \tau_s^* \to \tau)$, then it can be treated as a different type in the following three ways (figure 4): as a function with only the fixed parameters; as a function with an extra fixed parameter via unrolling of the rest parameter; and as a function where the rest parameter is classified as a dotted type variable.

In TC-TODFUN the type of the uniform rest parameter is not allowed to contain the bound of the resulting non-uniform rest parameter as a free type variable.[4]

$$\begin{aligned}
\delta_\tau(\texttt{=}) &= (\textbf{Integer Integer} \to \textbf{Boolean}) \\
\delta_\tau(\texttt{plus}) &= (\textbf{Integer}^* \to \textbf{Integer}) \\
\delta_\tau(\texttt{minus}) &= (\textbf{Integer Integer}^* \to \textbf{Integer}) \\
\delta_\tau(\texttt{mult}) &= (\textbf{Integer}^* \to \textbf{Integer}) \\
\delta_\tau(\texttt{car}) &= (\forall\ (\alpha)\ ((\textbf{Listof}\ \alpha) \to \alpha)) \\
\delta_\tau(\texttt{cdr}) &= (\forall\ (\alpha)\ ((\textbf{Listof}\ \alpha) \to (\textbf{Listof}\ \alpha))) \\
\delta_\tau(\texttt{null?}) &= (\forall\ (\alpha)\ ((\textbf{Listof}\ \alpha) \to \textbf{Boolean}))
\end{aligned}$$

Figure 6. Types of primitives

As expected, most of the typing rules in figure 5 are straightforward additions of multiple-arity type and term abstractions and lists to System F. The types of primitives are provided in figure 6.

[4] We are investigating whether this restriction can be dropped.

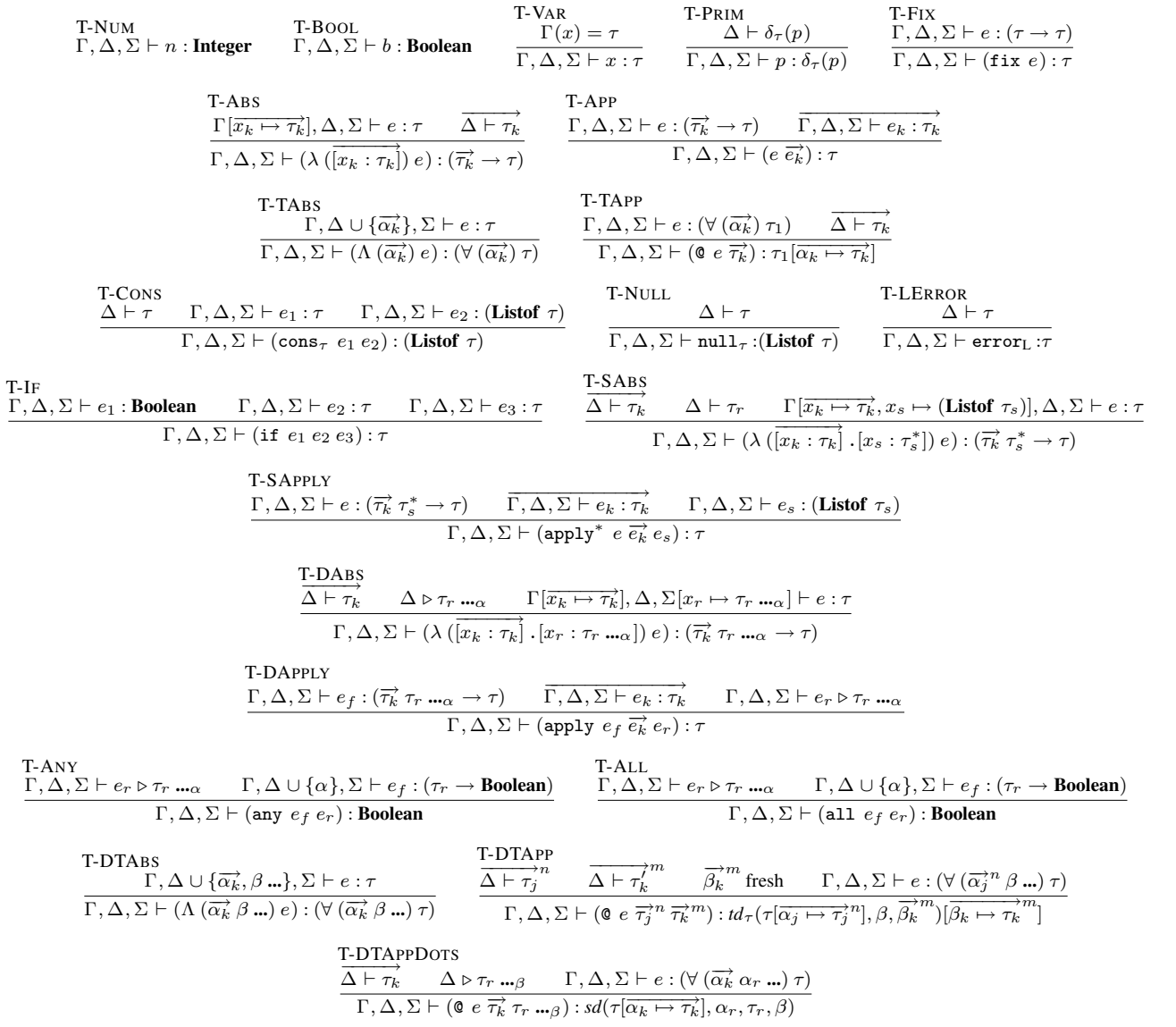

Figure 5. Type rules

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.

Non-uniform variable-arity functions also have one introduction and one elimination rule, and these rules differ from those for uniform variable-arity functions in the expected ways. The rules T-Any and T-All are similar to that of TD-Lift 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 T-TAbs, 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 (possibly empty) sequence of types, and the latter deals with the case when the dotted type variable corresponds to a dotted pre-type.

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, however, than normal type substitution because we need to replace the dotted type variable where it appears as a dotted pre-type bound. Figure 7 describes the metafunction *sd*, which performs this substitution.

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

$$
\begin{array}{lcll}
sd(\textbf{Integer}, \alpha_r, \tau_r, \beta) & = & \textbf{Integer} & \\
sd(\textbf{Boolean}, \alpha_r, \tau_r, \beta) & = & \textbf{Boolean} & \\
sd(\alpha_r, \alpha_r, \tau_r, \beta) & = & \tau_r & \\
sd(\alpha, \alpha_r, \tau_r, \beta) & = & \alpha & \text{where } \alpha \neq \alpha_r \\
sd((\textbf{Listof } \tau), \alpha_r, \tau_r, \beta) & = & (\textbf{Listof } sd(\tau, \alpha_r, \tau_r, \beta)) & \\
sd((\overrightarrow{\tau_j} \to \tau), \alpha_r, \tau_r, \beta) & = & (\overrightarrow{sd(\tau_j, \alpha_r, \tau_r, \beta)} \to sd(\tau, \alpha_r, \tau_r, \beta)) & \\
sd((\overrightarrow{\tau_j}\ \tau'_r \textbf{ ...}_{\alpha_r} \to \tau), \alpha_r, \tau_r, \beta) & = & (\overrightarrow{sd(\tau_j, \alpha_r, \tau_r, \beta)}\ sd(\tau'_r, \alpha_r, \tau_r, \beta) \textbf{ ...}_{\beta} \to sd(\tau, \alpha_r, \tau_r, \beta)) & \\
sd((\overrightarrow{\tau_j}\ \tau'_r \textbf{ ...}_{\alpha} \to \tau), \alpha_r, \tau_r, \beta) & = & (\overrightarrow{sd(\tau_j, \alpha_r, \tau_r, \beta)}\ sd(\tau'_r, \alpha_r, \tau_r, \beta) \textbf{ ...}_{\alpha} \to sd(\tau, \alpha_r, \tau_r, \beta)) & \text{where } \alpha \neq \alpha_r \\
sd((\forall\ (\overrightarrow{\alpha_j})\ \tau), \alpha_r, \tau_r, \beta) & = & (\forall\ (\overrightarrow{\alpha_j})\ sd(\tau, \alpha_r, \tau_r, \beta)) & \\
sd((\forall\ (\overrightarrow{\alpha_j}\ \alpha \textbf{ ...})\ \tau), \alpha_r, \tau_r, \beta) & = & (\forall\ (\overrightarrow{\alpha_j}\ \alpha \textbf{ ...})\ sd(\tau, \alpha_r, \tau_r, \beta)) & \\
sd((\overrightarrow{\tau_j}\ \tau_s^* \to \tau), \alpha_r, \tau_r, \beta) & = & (\overrightarrow{sd(\tau_j, \alpha_r, \tau_r, \beta)}\ sd(\tau_s, \alpha_r, \tau_r, \beta)^* \to sd(\tau, \alpha_r, \tau_r, \beta)) &
\end{array}
$$

Figure 7. Substituting dotted pre-types for dotted variables

$$
\begin{array}{lcll}
td_\tau(\textbf{Integer}, \beta, \overrightarrow{\beta_k}^m) & = & \textbf{Integer} & \\
td_\tau(\textbf{Boolean}, \beta, \overrightarrow{\beta_k}^m) & = & \textbf{Boolean} & \\
td_\tau(\alpha, \beta, \overrightarrow{\beta_k}^m) & = & \alpha & \\
td_\tau((\textbf{Listof } \tau), \beta, \overrightarrow{\beta_k}^m) & = & (\textbf{Listof } td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m)) & \\
td_\tau((\overrightarrow{\tau_j}^n \to \tau), \beta, \overrightarrow{\beta_k}^m) & = & (\overrightarrow{td_\tau(\tau_j, \beta, \overrightarrow{\beta_k}^m)}^n \to td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m)) & \\
td_\tau((\overrightarrow{\tau_j}^n\ \tau_r \textbf{ ...}_\beta \to \tau), \beta, \overrightarrow{\beta_k}^m) & = & (\overrightarrow{td_\tau(\tau_j, \beta, \overrightarrow{\beta_k}^m)}^n\ td_\tau(\tau_r, \beta, \overrightarrow{\beta_k}^m)[\beta \mapsto \beta_k] \to td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m)) & \\
td_\tau((\overrightarrow{\tau_j}^n\ \tau_r \textbf{ ...}_\alpha \to \tau), \beta, \overrightarrow{\beta_k}^m) & = & (\overrightarrow{td_\tau(\tau_j, \beta, \overrightarrow{\beta_k}^m)}^n\ td_\tau(\tau_r, \beta, \overrightarrow{\beta_k}^m) \textbf{ ...}_\alpha \to td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m)) & \text{where } \alpha \neq \beta \\
td_\tau((\forall\ (\overrightarrow{\alpha_j}^n)\ \tau), \beta, \overrightarrow{\beta_k}^m) & = & (\forall\ (\overrightarrow{\alpha_j}^n)\ td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m)) & \\
td_\tau((\forall\ (\overrightarrow{\alpha_j}^n\ \alpha_r \textbf{ ...})\ \tau), \beta, \overrightarrow{\beta_k}^m) & = & (\forall\ (\overrightarrow{\alpha_j}^n\ \alpha_r \textbf{ ...})\ td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m)) & \\
td_\tau((\overrightarrow{\tau_j}^n\ \tau_s^* \to \tau), \beta, \overrightarrow{\beta_k}^m) & = & (\overrightarrow{td_\tau(\tau_j, \beta, \overrightarrow{\beta_k}^m)}^n\ td_\tau(\tau_s, \beta, \overrightarrow{\beta_k}^m)^* \to td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m)) &
\end{array}
$$

Figure 8. Expanding dots in types

$$
\begin{array}{lll}
E & ::= & [\,] \mid (\overrightarrow{v}\ [\,]\ \overrightarrow{e}) \mid (@\ [\,]\ \overrightarrow{\tau}) \\
 & \mid & (\texttt{if}\ [\,]\ e\ e) \mid (\texttt{cons}_\tau\ [\,]\ e) \mid (\texttt{cons}_\tau\ v\ [\,]) \\
 & \mid & (\texttt{apply}^*\ \overrightarrow{v}\ [\,]\ \overrightarrow{e})
\end{array}
$$

$$
\begin{array}{ll}
E[(\texttt{car null}_\tau)] & \hookrightarrow E[\texttt{error}_\text{L}] \\
E[(\texttt{cdr null}_\tau)] & \hookrightarrow E[\texttt{error}_\text{L}] \\
E[(@\ (\Lambda\ (\overrightarrow{\alpha_k})\ e)\ \overrightarrow{\tau_k})] & \hookrightarrow E[e[\overrightarrow{\alpha_k \mapsto \tau_k}]] \\
E[(@\ (\Lambda\ (\overrightarrow{\alpha_j}^n\ \beta \textbf{ ...})\ e)\ \overrightarrow{\tau_j}^n\ \overrightarrow{\tau_k}^m)] & \hookrightarrow \\
\multicolumn{2}{l}{\quad E[td_e(e, \beta, \overrightarrow{\beta_k}^m)[\overrightarrow{\alpha_j \mapsto \tau_j}^n, \overrightarrow{\beta_k \mapsto \tau_k}^m]]} \\
\multicolumn{2}{l}{\quad \text{where } \overrightarrow{\beta_k}^m \text{ fresh}} \\
E[((\lambda\ (\overrightarrow{[x_k : \tau_k]})\ e)\ \overrightarrow{v_k})] & \hookrightarrow E[e[\overrightarrow{x_k \mapsto v_k}]] \\
E[((\lambda\ (\overrightarrow{[x_j : \tau_j]}^n\ .\ [x_s : \tau_s^*])\ e)\ \overrightarrow{v_j}^n\ \overrightarrow{v_k}^m)] & \hookrightarrow \\
\multicolumn{2}{l}{\quad E[e[\overrightarrow{x_j \mapsto v_j}^n,} \\
\multicolumn{2}{l}{\qquad x_s \mapsto (\texttt{cons}_{\tau_s}\ v_{k_1}\ (\texttt{cons}_{\tau_s}\ \cdots (\texttt{cons}_{\tau_s}\ v_{k_m}\ \texttt{null}_{\tau_s})\cdots)))] } \\
E[(\texttt{apply}^*\ (\lambda\ (\overrightarrow{[x_j : \tau_j]}^n\ .\ [x_s : \tau_s^*])\ e)\ \overrightarrow{v_j}^n\ \overrightarrow{v_k}^m\ v_s)] & \hookrightarrow \\
\multicolumn{2}{l}{\quad E[e[\overrightarrow{x_j \mapsto v_j}^n,} \\
\multicolumn{2}{l}{\qquad x_s \mapsto (\texttt{cons}_{\tau_s}\ v_{k_1}\ (\texttt{cons}_{\tau_s}\ \cdots (\texttt{cons}_{\tau_s}\ v_{k_m}\ v_s)\cdots))]]}
\end{array}
$$

Figure 9. Selected Reduction Rules

each fresh type variable to its corresponding type argument. The definition of td_τ is given in figure 8.

4.3 Dynamic Semantics

In order to prove the given type system sound, we introduce a reduction semantics. The reduction contexts and interesting reduction rules are given in figure 9.

Even here, most of these definitions are obvious extensions of those for System F. Application of a multiple-arity function or a multiple-arity type abstraction requires substituting for all bound variables. Application of a starred function constructs a single list from all the arguments that correspond to the rest parameter. Using the `apply` * function is similar, but the last argument, which must be a list, forms the base of the rest argument list.

When applying a dotted type abstraction, all subforms dealing with expressions and types in that dotted type variable are expanded through the use of td_e. This explains why there are no reduction rules corresponding to any expressions that involve dotted pre-types; they are always expanded away.

The semantics relies on three metafunctions:

$td_e(e, \beta, \overrightarrow{\beta_k}^m)$ This metafunction expands expressions that are not within a λ expression whose rest argument has a dotted pre-type with bound β.

$td_x(e, \beta, \overrightarrow{\beta_k}^m, X)$ This metafunction expands expressions that appear within λ expressions whose rest argument has bound β and that expand to a single expression. X is a mapping from rest variables to m fresh variables.

$td_d(e, \beta, \overrightarrow{\beta_k}^m, X)$ This metafunction expands expressions that appear within λ expressions whose rest argument has bound β and that may expand into multiple expressions (i.e. dotted rest parameters and `lift`).

Selected cases of these metafunctions are specified in figure 10. The other parts of the function definitions are either straightforward structural recursion or are similar to the cases presented here.

For td_e and td_x, dotted pre-types in type applications that are bound by the dotted type variable are expanded appropriately to form type applications with only type arguments. Similarly, non-uniform variable-arity λ expressions have their parameter lists ex-

$$
\begin{array}{lcl}
td_e((\texttt{@}\ e\ \overrightarrow{\tau_j}^n\ \tau_r\ \texttt{...}_\beta), \beta, \overrightarrow{\beta_k}^m) & = & (\texttt{@}\ td_e(e, \beta, \overrightarrow{\beta_k}^m)\ \overrightarrow{td_\tau(\tau_j, \beta, \overrightarrow{\beta_k}^m)}^n\ \overrightarrow{td_\tau(\tau_r, \beta, \overrightarrow{\beta_k}^m)[\beta \mapsto \beta_k]}^m) \\
td_e((\lambda\ (\overrightarrow{[x_j : \tau_j]}^n\ .[x_r : \tau_r\ \texttt{...}_\beta])\ e), \beta, \overrightarrow{\beta_k}^m) & = & (\lambda\ (\overrightarrow{[x_j : td_\tau(\tau_j, \beta, \overrightarrow{\beta_k}^m)]}^n\ \overrightarrow{[x_k : \tau'_r[\beta \mapsto \beta_k]]}^m)\ td_x(e, \beta, \overrightarrow{\beta_k}^m, [x_r \mapsto \overrightarrow{x_k}^m])) \\
\quad \text{where } \tau'_r = td_\tau(\tau_r, \beta, \overrightarrow{\beta_k}^m) & & \\
\quad \text{and } \overrightarrow{x_k}^m \text{ fresh} & &
\end{array}
$$

$$
\begin{array}{lcl}
td_x((\texttt{@}\ e\ \overrightarrow{\tau_j}^n\ \tau_r\ \texttt{...}_\beta), \beta, \overrightarrow{\beta_k}^m, X) & = & (\texttt{@}\ td_x(e, \beta, \overrightarrow{\beta_k}^m, X)\ \overrightarrow{td_\tau(\tau_j, \beta, \overrightarrow{\beta_k}^m)}^n\ \overrightarrow{td_\tau(\tau_r, \beta, \overrightarrow{\beta_k}^m)[\beta \mapsto \beta_k]}^m) \\
td_x((\lambda\ (\overrightarrow{[x_j : \tau_j]}^n\ .[x_r : \tau_r\ \texttt{...}_\beta])\ e), \beta, \overrightarrow{\beta_k}^m, X) & = & (\lambda\ (\overrightarrow{[x_j : td_\tau(\tau_j, \beta, \overrightarrow{\beta_k}^m)]}^n\ \overrightarrow{[x_k : \tau'_r[\beta \mapsto \beta_k]]}^m)\ td_x(e, \beta, \overrightarrow{\beta_k}^m, X[x_r \mapsto \overrightarrow{x_k}^m])) \\
\quad \text{where } \tau'_r = td_\tau(\tau_r, \beta, \overrightarrow{\beta_k}^m) & & \\
\quad \text{and } \overrightarrow{x_k}^m \text{ fresh} & & \\
td_x((\texttt{apply}\ e\ \overrightarrow{e_j}^n\ e_r), \beta, \overrightarrow{\beta_k}^m, X) & = & \text{case } td_d(e_r, \beta, \overrightarrow{\beta_k}^m, X) \text{ of } \begin{array}{lcl} e'_r & \Rightarrow & (\texttt{apply}\ e'\ \overrightarrow{e'_j}^n\ e'_r) \\ \overrightarrow{e_k}^m & \Rightarrow & (e'\ \overrightarrow{e'_j}^n\ \overrightarrow{e_k}^m) \end{array} \\
\quad \text{where } e' = td_x(e, \beta, \overrightarrow{\beta_k}^m, X) & & \\
\quad \text{and } \overrightarrow{e'_j}^n = \overrightarrow{td_x(e_j, \beta, \overrightarrow{\beta_k}^m, X)}^n & & \\
td_x((\texttt{any}\ e\ e_r), \beta, \overrightarrow{\beta_k}^m, X) & = & \\
\quad \text{case } td_d(e_r, \beta, \overrightarrow{\beta_k}^m, X) \text{ of } \begin{array}{lcl} e'_r & \Rightarrow & (\texttt{any}\ e'\ e'_r) \\ \overrightarrow{e_k}^m & \Rightarrow & (\texttt{if}\ (e'[\beta \mapsto \beta_{k_1}]\ e_{k_1})\ \texttt{\#t}\ (\texttt{if}\ \text{——}\ (e'[\beta \mapsto \beta_{k_m}]\ e_{k_m}))) \end{array} & \\
\quad \text{where } e' = td_x(e, \beta, \overrightarrow{\beta_k}^m, X) & &
\end{array}
$$

$$
\begin{array}{lcl}
td_d((\texttt{lift}\ e\ e_r), \beta, \overrightarrow{\beta_k}^m, X) & = & \text{case } td_d(e_r, \beta, \overrightarrow{\beta_k}^m, X) \text{ of } \begin{array}{lcl} e'_r & \Rightarrow & (\texttt{lift}\ e'\ e'_r) \\ \overrightarrow{e_k}^m & \Rightarrow & \overrightarrow{(e'[\beta \mapsto \beta_k]\ e_k)}^m \end{array} \\
\quad \text{where } e' = td_x(e, \beta, \overrightarrow{\beta_k}^m, X) & & \\
td_d(x, \beta, \overrightarrow{\beta_k}^m, X) & = & X(x) \\
\quad \text{where } x \in \text{dom}(X) & & \\
td_d(x, \beta, \overrightarrow{\beta_k}^m, X) & = & x \\
\quad \text{where } x \notin \text{dom}(X) & &
\end{array}
$$

Figure 10. Selected rules for expansion of expressions

panded to replace the rest parameter with the appropriate number of fresh parameters.

When the last argument to an `any` form expands into multiple expressions, td_x expands it into a sequence of `if` expressions that are equivalent to the appropriate `or` expressions. Likewise td_x expands `all` to `if` expressions that are equivalent to `and`s and `apply` into direct function application.

The rest parameters that are removed by td_e and td_x are expanded into the appropriate fresh parameters by td_d. td_d also expands `lift` expressions whose last argument expands into multiple expressions into sequences of function application.

4.4 Soundness

Our soundness theorem for our model recognizes only stuck states involving list operations ($\texttt{error}_\text{L}$).

Theorem 4.1 (Soundness). *If* $\vdash e : \tau$, *then either*

(i) $e \hookrightarrow^* v$ *and* $\vdash v : \tau$,
(ii) $e \Uparrow$, *or*
(iii) $e \hookrightarrow^* E[\texttt{error}_L]$ *for some* E.

We prove soundness via progress and preservation in the style of Wright and Felleisen (1994). Those lemmas are not interesting in themselves, but they lead to the following interesting lemmas involving the metafunctions used to expand expressions and types. These lemmas use the following environments for typing the expanded expressions:

- $\Gamma' = [x \mapsto td_\tau(\Gamma(x), \beta, \overrightarrow{\beta_k}^m) \mid x \in \text{dom}(\Gamma)]$
- $\Delta' = (\Delta - \{\beta\ \texttt{...}\}) \cup \{\overrightarrow{\beta_k}^m\}$
- $\Sigma' = [x \mapsto \tau'\ \texttt{...}_\alpha \mid x \in \text{dom}(\Sigma) \wedge \Sigma(x) = \tau\ \texttt{...}_\alpha]$
  where $\tau' = td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m)$ and $\alpha \neq \beta$.

These definitions ensure that dotted types are appropriately expanded in the environments. Replacement of rest parameters with dotted pre-types with the appropriately typed parameters are handled in the individual lemmas.

Lemma 4.2 (td_e Preserves Types). *If* $\Gamma, \Delta, \Sigma \vdash e : \tau$, *then* $\Gamma', \Delta', \Sigma' \vdash td_e(e, \beta, \overrightarrow{\beta_k}^m) : td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m)$.

Lemma 4.3 (td_x Preserves Types). *If*
$\Gamma, \Delta, \Sigma[\overrightarrow{x_r \mapsto \tau_r\ \texttt{...}_\beta}^n] \vdash e : \tau$, *then*
$\Gamma'', \Delta', \Sigma' \vdash td_x(e, \beta, \overrightarrow{\beta_k}^m, X) : td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m)$ *where*
$\Gamma'' = \Gamma'[\overrightarrow{x_{r_k} \mapsto td_\tau(\tau_r, \beta, \overrightarrow{\beta_k}^m)[\beta \mapsto \beta_k]}^n\ ]$
and $X = [x_r \mapsto \overrightarrow{x_{r_k}}^m]$.

Lemma 4.4 (td_d Expands Dotted Pre-Types over β). *If*
$\Gamma, \Delta, \Sigma[\overrightarrow{x_r \mapsto \tau_r\ \texttt{...}_\beta}^n] \vdash e \triangleright \tau\ \texttt{...}_\beta$, *then*
$\Gamma'', \Delta', \Sigma' \vdash e'_k : td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m)[\beta \mapsto \beta_k]$ *where*
$\Gamma'' = \Gamma'[\overrightarrow{x_{r_k} \mapsto td_\tau(\tau_r, \beta, \overrightarrow{\beta_k}^m)[\beta \mapsto \beta_k]}^n\ ]$
and $X = [x_r \mapsto \overrightarrow{x_{r_k}}^m]$.

Lemma 4.5 (td_d Preserves Dotted Pre-Types over $\alpha \neq \beta$). *If*
$\Gamma, \Delta, \Sigma[\overrightarrow{x_r \mapsto \tau_r\ \texttt{...}_\beta}^n] \vdash e \triangleright \tau\ \texttt{...}_\alpha$, *then*
$\Gamma'', \Delta', \Sigma' \vdash td_d(e, \beta, \overrightarrow{\beta_k}^m, X) \triangleright td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m)\ \texttt{...}_\alpha$ *where*
$\Gamma'' = \Gamma'[\overrightarrow{x_{r_k} \mapsto td_\tau(\tau_r, \beta, \overrightarrow{\beta_k}^m)[\beta \mapsto \beta_k]}^n\ ]$
and $X = [x_r \mapsto \overrightarrow{x_{r_k}}^m]$.

Proof. Induction on the type derivation tree. We also need the following equivalences:

1. $td_\tau(td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m), \alpha, \overrightarrow{\alpha_j}^n) = td_\tau(td_\tau(\tau, \alpha, \overrightarrow{\alpha_j}^n), \beta, \overrightarrow{\beta_k}^m)$
2. $td_\tau(\tau[\overrightarrow{\alpha_j \mapsto \tau_j}^n], \beta, \overrightarrow{\beta_k}^m) =$
   $td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m)[\overrightarrow{\alpha_j \mapsto td_\tau(\tau_j, \beta, \overrightarrow{\beta_k}^m)}^n\ ]$
3. $td_\tau(sd(\tau, \alpha_r, \tau_r, \beta), \beta, \overrightarrow{\beta_k}^m) =$
   $\tau'[\overrightarrow{\alpha_k \mapsto td_\tau(\tau_r, \beta, \overrightarrow{\beta_k}^m)[\beta \mapsto \beta_k]}^m\ ]$
   where $\tau' = td_\tau(td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m), \alpha_r, \overrightarrow{\alpha_k}^m)$ and $\overrightarrow{\alpha_k}^m$ are fresh.
4. $td_\tau(sd(\tau, \alpha_r, \tau_r, \alpha), \beta, \overrightarrow{\beta_k}^m) =$
   $sd(td_\tau(\tau, \beta, \overrightarrow{\beta_k}^m), \alpha_r, td_\tau(\tau_r, \beta, \overrightarrow{\beta_k}^m), \alpha)$
   where $\alpha \neq \beta$.

These equivalences are proven via induction on the structure of types. Equivalence 3 is only used where $\Delta - \{\alpha_r\} \vdash \tau$ and equivalence 4 is only used where $\Delta - \{\beta\} \vdash \tau$; i.e. when either α_r or β only appears in Δ as a dotted type variable, not also as a regular type variable. □

5. Inference and Integration

Translating the calculus into a viable extension of Typed Scheme requires two engineering steps. First, the addition of starred and dotted type variables interacts with the local type inference system of Typed Scheme. Second, Typed Scheme supports several language constructs that benefit from the type calculus, in particular case-based **lambda** expressions and the multiple value return mechanism.

5.1 Local Type Inference

To maintain the convenience of Typed Scheme for programmers, we must integrate variable-arity polymorphism into the framework of local type inference (Pierce and Turner 2000), while preserving the desirable properties of both. This means that it must be possible to use variable-arity functions such as *map* without necessarily explicitly instantiating them at appropriate types (for most cases). Further, the inference algorithm must handle all of the features described in section 4.

Unfortunately, lack of space prevents us from presenting a formal description of our extension to to Pierce and Turner's system. Instead, we sketch the two most important changes required to handle variable-arity polymorphism. First, the constraint generation mechanism must create constraints for type relationships involving dotted pre-types. Second, the resulting substitution must handle the presence of dotted pre-types in the result type of functions.

Constraint Generation The heart of the Pierce and Turner algorithm generates subtyping constraints on the type variables in the type of a polymorphic function. All constraints are of the form of an upper and lower bound for each type variable.[5] For example, if we have the application (*map add1* (*list* 1 2 3)), where *map* has the type $(\forall\ (\alpha\ \beta)\ ((\alpha \to \beta)\ (\textbf{Listof}\ \alpha) \to (\textbf{Listof}\ \beta)))$, then we would generate the constraints

$$\alpha = \textbf{Integer}, \textbf{Integer} <: \beta$$

From these constraints, we can generate a substitution that maps both α and β to **Integer**, giving a result type of (**Listof Integer**) .

When variable-arity polymorphism is introduced, such simple constraints no longer suffice. For example, in this world, the type for *map* is

$$(\forall\ (\gamma\ \alpha\ \beta \ldots)\ ((\alpha\ \beta \ldots_\beta \to \gamma)\ (\textbf{Listof}\ \alpha)\ (\textbf{Listof}\ \beta) \ldots_\beta \to (\textbf{Listof}\ \gamma))))$$

[5] We use $=$ when the bounds are the same, and omit trivial bounds.

meaning that the application

(*map modulo* (*list* 10 20 30) (*list* 5 2 3))

requires that β be expanded into a single instance of **Integer**.

To this end, we extend our language of constraints. Instead of merely providing bounds for each type variable, a dotted type variable can be constrained to map to a sequence of new type variables, each with its own constraints. In the above example, we would say that β is mapped to a singleton list of type variables $[\beta_1]$, which has the constraint $\beta_1 = \textbf{Integer}$. This is in addition to the standard constraints generated by this application, which include $\alpha = \textbf{Integer}$ and $\textbf{Integer} <: \gamma$.

Once such constraint sets are generated, they need to be combined and normalized. For example, the constraints for the first argument in the above expression are $\alpha <: \textbf{Integer}$, $\textbf{Integer} <: \gamma$ and $\beta_1 <: \textbf{Integer}$, where β_1 is generated from β. The only constraint from the second argument is $\textbf{Integer} <: \alpha$. Finally, the constraint from the third argument is $\textbf{Integer} <: \beta_1$. Combining the constraints for α and γ is straightforward, and follows the original algorithm. For the sequence of type variables generated for β (here just β_1), we combine them pairwise, giving the constraints for the entire application above.

In addition, we need to handle combinations of uniform and non-uniform variable-arity functions. Recall that the type of $+$ is $(\textbf{Integer}^* \to \textbf{Integer})$. Then inference for the application

(*map* + (*list* 1 2 3) (*list* 10 20 30) (*list* 100 200 300))

generates the constraint

$$(\textbf{Integer}^* \to \textbf{Integer}) <: (\alpha\ \beta \ldots_\beta \to \gamma)$$

This yields the same constraints for α and γ as above, but for β, the constraint set must record that the number of types for β is not yet fixed and that all types substituted for β are bounded above by **Integer**. We record this as a constraint directly on β. When the constraints for the three list arguments are generated, we generate precisely two new type variables for β. Since the bounds match appropriately, this results in the final constraints of $\beta_1 = \textbf{Integer}$ and $\beta_2 = \textbf{Integer}$, and no constraints on β since the number of arguments has been fixed.

Generating Substitutions Once the full set of constraints is generated for an application, we solve them, i.e., we derive a substitution from the constraint set. The substitution is then applied to the result type of the operand. In the above examples with *map*, the substitution is simply $\gamma \mapsto$ (**Listof Integer**) . For other function types, however, the substitution may be more complex. For example, given a function

$$(\textbf{:}\ f\ (\forall\ (\alpha \ldots)\ ((\alpha \ldots_\alpha \to \textbf{Integer}) \to (\alpha \ldots_\alpha \to \textbf{Integer})))$$

the application (*f modulo*) generates $\alpha_1 <: \textbf{Integer}$ and $\alpha_2 <: \textbf{Integer}$, which satisfy

$$(\alpha \ldots_\alpha \to \textbf{Integer}) <: (\textbf{Integer Integer} \to \textbf{Integer})\ .$$

The resulting substitution should replace the dotted pre-type $\alpha \ldots_\alpha$ with the sequence **Integer Integer**. To represent such a substitution, we extend our notion of a substitution to allow a single type variable to map to *multiple* types. Here, the resulting substitution is $\alpha \mapsto [\textbf{Integer}, \textbf{Integer}]$.

Again, the presence of uniform rest arguments complicates the picture. Consider $(f\ -)$, where $- : (\textbf{Integer Integer}^* \to \textbf{Integer})$. The constraint set contains $\textbf{Integer} <: \alpha_1$ and $\textbf{Integer} <: \alpha$, with no upper bound on the number of arguments. Therefore, the substitution should produce the result type $(\textbf{Integer Integer}^* \to \textbf{Integer})$. We represent such a substitution as

$$\alpha \mapsto [\textbf{Integer}, \textbf{Integer}^*]$$

Given these extensions to the original local type inference algorithm, we are able to infer the types for all applications of variable-arity polymorphic functions, provided that the types of the actual argument expressions are sufficient to determine the result type.

5.2 Integrating with Typed Scheme

The calculus presented in section 4 is obviously not the whole of Typed Scheme. Several constructs of the full language, such as multiple values and **case-lambda**, interact in interesting ways with variable-arity polymorphism.

***Functions with* case-lambda** In addition to variable-arity functions, PLT Scheme also supports the **case-lambda** construct, which creates functions that have different bodies for different arities. For example, given this definition

```
(define f (case-lambda [(x)   1]
                       [(x y) 2]))
```

(*f* 'a) yields 1 and (*f* 'a 'b) gives 2. The semantics of **case-lambda** tries the clauses in order, and the first clause to accept the given number of arguments is used. Through the **case-lambda** type, which was introduced in section 3.2, Typed Scheme supports functions defined in this paper.

In order to support such types in the local inference algorithm, we must consider each conjunct of the type in turn. For example, if we are interested in an instantiation of $(\alpha \ldots_\alpha \to \beta)$ that makes it a supertype of

(**case-lambda** (**Integer** → **Integer**)
(**Integer Boolean** → **Integer**))

there are multiple possibilities. The two possible constraint sets, via the algorithm described above, are

$$\{\alpha_1 <: \textbf{Integer}, \textbf{Integer} <: \beta\}$$

and

$$\{\alpha_1 <: \textbf{Integer}, \alpha_2 <: \textbf{Boolean}, \textbf{Integer} <: \beta\}$$

Therefore, a call to the constraint generator must return multiple possibilities. When two such lists of constraint sets are combined, we take the cartesian product of the combinations, excluding those that cannot be combined because they have incompatible constraints. Although such cross-combinations could generate extremely large sets, in practice it has not been noticeable. After generating the final list of constraint sets, if a substitution is needed, it can be generated from any constraint set.

The other change needed to accommodate **case-lambda** is that type-checking a function application must check each conjunct in turn. While this complicates the implementation of the type-checker, it introduces no problems for the novel aspects of variable-arity polymorphism.

Multiple Values Scheme, and therefore Typed Scheme, allows expressions to return multiple values simultaneously. These multiple values are not packaged in a data structure, and only certain contexts accept multiple values. Just as argument sequences are a natural place to allow variable-length sequences, however, multiple values fit naturally with variable arity as well.

The basic extension to Typed Scheme in support of variable-arity multiple values is to allow dotted pre-types such as (**Values Boolean** $\alpha \ldots_\alpha$). While this, as with other dotted pre-types, cannot be the type of a closed expression, it can appear as the result type of a polymorphic function.

The combination of multiple values and variable-arity polymorphism allows us to give types to Scheme primitives such as *values* and *call-with-values*. The *values* function takes any number of arguments, and returns all of them as multiple values. Its type is

$$(\forall\ (\alpha \ldots)\ (\alpha \ldots_\alpha \to (\textbf{Values}\ \alpha \ldots_\alpha)))$$

The *call-with-values* function is one of the fundamental mechanisms for consuming multiple values. It takes a producer thunk, which produces some number of values, and a consumer function, which accepts that number of arguments, and wires them together. Thus, its type is

$$(\forall\ (\beta\ \alpha \ldots)\ ((\to (\textbf{Values}\ \alpha \ldots_\alpha))\ (\alpha \ldots_\alpha \to \beta) \to \beta))$$

Previous work on type systems for Scheme has not been able to statically check such uses of multiple values.

Operations on Rest Parameters The standard *map*, *ormap*, and *andmap* functions are treated specially in Typed Scheme when applied to rest parameters and expressions classified by dotted pre-types. These uses correspond to the *lift*, *any*, and *all* special forms used in the calculus of section 4.

Finally, since rest parameters, even those classified by dotted pre-types, are represented by lists at runtime in Typed Scheme, we allow these rest parameters to be used in contexts that expect lists, dropping the additional information in the dotted pre-type. This supports many common Scheme idioms, such as printing the list of arguments in a debugging mode.

6. Evaluation and Limitations

PLT Scheme consists of two bodies of code: a core distribution and a web-based repository of user-contributed libraries (and some applications). Together, the base comprises some 800,000 lines of code. Mining this 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 should be able to cope with a good portion of these definitions and uses.

6.1 Measurements of Existing Code

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

- There are at least 2038 definitions of variable-arity functions. Of these, 1761 actually use the rest parameter in the body of the function; the rest simply ignore their arguments.
- There are 488 uses of variable-arity functions from the core Scheme library (*map*, *for-each*, *foldl*, *foldr*, *andmap*, and *ormap*) with more than the minimum number of arguments; the largest number of additional arguments is six.
- There are 359 uses of the *values* and *call-with-values* functions.

  Note: PLT Scheme, like all full-fledged Scheme implementations, provides additional linguistic constructs for dealing with multiple values. Our search pattern doesn't cover those.

These results definitely demonstrate the need for a type system that deals with variable-arity functions. Programmers use the ones from the core library and use them at multiple arities. Furthermore, programmers define such functions, though it remains to be seen how much power these definitions assume. Last but not least, expressions that produce multiple values are ubiquitous, and multiple values flow into functions whose types exploit variable-arity polymorphism.

It is this kind of inspection of our code base that inspires a careful investigation of the issue of variable-arity functions. We simply cannot ask our programmers to duplicate their code or to duplicate type cases just because our type system doesn't accommodate their acceptance of the expressive power of plain scheme.

6.2 Evaluation of Examples

Simply counting definitions and uses of variable-arity functions doesn't suffice. Each definition and use demands a separate inspec-

tion 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 40 cases, i.e., 10 randomly-chosen example function applications using each of *map*, *for-each*, *andmap* and *call-with-values*. 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. For *call-with-values*, we are able to type 6 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. Applying *map* to such a list simply does not preserve the length information for the list or anything else about its structure.

Concerning *call-with-values*, our type system is less accommodating than for list-processing functions. An examination of the failures suggests that *call-with-values* is mostly used to call unknown functions while remaining polymorphic in the number of returned values. Further investigation is needed into how best to support such programs.

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 four categories:

- A majority of the functions can be typed with uniform rest arguments.
- Twelve of the functions are non-uniform and require variable-arity polymorphism. Our type checker can assign types to all of them.

  Here are two random examples:

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

```
(: test (∀ (β α ...) (β (α ...α → β) α ...α → Boolean)))
;; applies p to args and
;; compares the result to desired.
(define (test desired p . args)
  (let* ((result (apply p args)))
    (unless (equal? desired result)
            (fprintf (current-error-port)
                     "desired: ~v\ngot: ~v\ntest: ~v\n"
                     desired result (cons p args)))))
```

- A significant number use variable arity to simulate optional arguments. We recommend that programmers rewrite such functions using **case-lambda**, which Typed Scheme already incorporates and which signals the intention of the programmer more clearly than a type encoding.
- The remainder cannot be typed using our system.

We claim that these inspections demonstrate two important points. First, all of the various ways in which Typed Scheme handles varying numbers of arguments (**case-lambda**, uniform variable-arity, and non-uniform variable-arity) 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.

6.3 Limitations and Future Work

Typed Scheme currently suffers from two major limitations for the definition of non-uniform variable-arity functions. First, some functions perform calculations that involve the length of the rest argument. Our type system cannot verify that such operations are type correct.

Second, there are several type constructors where use of ... would make sense, but which we have not yet investigated, e.g., ($\bigcup \alpha \ldots_\alpha$) or (**List** $\alpha \ldots_\alpha$), where the **List** type constructor represents fixed length, heterogeneous lists. Doing so would help with several Scheme functions that deal with fixed-length lists.

Finally, our investigation into the code base reveals an open problem concerning the *compose* function. Naturally, in Scheme *compose* computes the composition of an arbitrary number of functions and deals with multiple values. Our type system easily deals with the full-fledged two-argument version:

```
(: compose
 (∀ (α ...)
  (∀ (γ β ...)
   ((α ...α → (Values β ...β))
    (β ...β → γ)
    →
    (α ...α → γ)))))
```

Even without considering multiple-value return, however, we know of no type system that can type-check Scheme's variable-arity *compose*.

7. 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 some 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. To allow for non-uniform variable-arity functions like `printf`, the programmer must provide the type of each extra argument manually and dynamically cast each argument to the appropriate type while retrieving it. Java and C# are two statically typed languages that have variable-arity functions, but access occurs via a homogeneous array and thus the types of the extra arguments must be uniform.

Dzeng and Haynes (1994) 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 (Haynes 1995), 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, their system has several limitations. Most importantly, they are unable to type many of the definitions of variable-arity functions, such as *map*. Additionally, their system requires full type inference to avoid exposing users to the underlying details of row types, and is also designed around a Hindley-Milner style type inference algorithm, and is thus incompatible with the remainder of the design of Typed Scheme. Their system does not support first-class polymorphic functions, and significantly restricts the contexts in which the rest argument of a variadic function can appear. Finally, they do not consider multiple values.

Gregor and Järvi (2007) propose an extention 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, especially those without template systems. The template approach at once greatly simplifies the problem, since template processing is a pre-processing step. It also significantly complicates the language, since arbitrary compuation can be performed. Further, the template approach prevents checking of variadic functions at the definition site, meaning that errors are reported later. Sheard and Peyton Jones (2002) also propose a solution for `zipWith` based on preprocessing, using the Template Haskell preprocessor, with similar limitations.

Tullsen (2000) attempts to bring non-uniform variable-arity functions to Haskell via the Zip Calculus. The Zip Calculus is a type system with restricted dependent types which avoid undecidability as well as special kinds that serve as tuple dimensions. This work is purely theoretical and does not come with practical evaluation. Additionally, the presented limitations of the Zip Calculus seem to imply that it cannot assign a type to `zipWith` without further extension, whereas the typing of *map*, which corresponds directly to `zipWith`, is one of the primary features of our system.

Similarly, McBride (2002) and Moggi (2000) present restricted forms of dependent typing in which the number of arguments is passed as a parameter to variadic functions. Our system, while obviously not allowing the expression of every dependently-typable program, suffices for all examples we have encountered without requiring the significant increase in the complexity of the type system that dependent types imply.

8. Conclusion

In this paper, we have presented a design for polymorphic functions with variable arity. Our system accomodates uniform and non-uniform variadic functions. We have also described how to integrate the system with the rest of the Typed Scheme language, as well as how to extend a local type inference algorithm to handle variable-arity polymorphism. Finally, we validated our design against existing Scheme and Typed Scheme code. Our system is part of the latest development release of PLT Scheme, available from

```
http://pre.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 $\tau^* \to \tau$. To accomodate variable-arity polymorphism, however, we must lift this abstraction one level up. For example, given the type $(\forall\,(\alpha\ldots)\,(\alpha\ldots_\alpha \to \textbf{Boolean}))$, the *kind* of this type is simply $\star^* \to \star$. So we see that *non-uniform* variable arity at the type level is reflected in *uniform* variable arity at the kind level.

References

American National Standard Programming Language C, ANSI X3.159-1989. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, December 14 1989.

Hsianlin Dzeng and Christopher T. Haynes. Type Reconstruction for Variable-Arity Procedures. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and Functional Programming*, pages 239–249, New York, NY, USA, 1994. ACM Press.

Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, 2002.

Matthew Flatt. PLT MzScheme: Language Manual. Technical Report PLT-TR2008-1-v4.0.2, PLT Scheme Inc., 2008. `http://www.plt-scheme.org/techreports/`.

Emden R. Gansner and John H. Reppy. *The Standard ML Basis Library*. Cambridge University Press, New York, NY, USA, 2002. ISBN 0-52179-478-1.

J.-Y. Girard. 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 J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

J. Gosling, B. Joy, G. L.Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Welsley, 2005.

Douglas Gregor and Jaakko Järvi. Variadic templates for C++. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1101–1108, New York, NY, USA, 2007. ACM Press. ISBN 1-59593-480-4. doi: http://doi.acm.org/10.1145/1244002.1244243.

Christopher T. Haynes. Infer: A Statically-typed Dialect of Scheme. Technical Report 367, Indiana University, 1995.

Jacob Matthews. Component Deployment with PLaneT: You Want it *Where?* In *Proceedings of the Seventh Workshop on Scheme and Functional Programming, University of Chicago, Technical Report TR-2006-06*, 2006.

Conor McBride. Faking it: Simulating Dependent Types in Haskell. *J. Funct. Program.*, 12(5):375–392, 2002. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796802004355.

Eugenio Moggi. Arity polymorphism and dependent types. Invited talk, APPSEM Workshop on Subtyping and Dependent Types in Programming, July 7 2000.

Benjamin C. Pierce and David N. Turner. Local Type Inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, 1974. Springer-Verlag. ISBN 3-540-06859-7.

Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

The GHC Team. *The Glasgow Haskell Compiler User's Guide*, 2008. URL `http://www.haskell.org/ghc/`.

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage Migration: From Scripts to Programs. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 964–974. Companion (Dynamic Languages Symposium), 2006.

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 395–406, 2008.

Mark Tullsen. The Zip Calculus. In Roland Backhouse and Jose Nuno Oliveira, editors, *Mathematics of Program Construction, 5th International Conference, MPC 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 28–44. Springer-Verlag, July 2000.

Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, pages 38–94, 1994. First appeared as Technical Report TR160, Rice University, 1991.