Type Safe Abstractions Using Program Generators

ogi tech report 95-013

Tim Sheard and Neal Nelson

Oregon Graduate Institute of Science & Technology
P.O. Box 91000, Portland, OR 97291-1000 USA
{sheard,nealn}@cse.ogi.edu

Abstract

Writing program generators involves the development of programs that manipulate *representations* of programs, thus offering unlimited possibliities for abstraction. Abstractions not expressable in typed languages can always be expressed as generators. For example generator mechanisms are implicit in the eq-types of Standard ML, and the deriving clauses of Haskel type classes, neither of which can be encoded within the language. While these mechanisms always generate well typed code, they are "hard coded" into the compiler.

Our goal is to incorporate generator like abstraction mechanisms into programming languages, while ensuring that only well typed programs are executed. This can be accomplished by a generate and then type-check approach or an inference mechanism that guarantees that only well typed programs are generated. In this paper we investigate both possibilites.

The problems associated with program generators include problems of "hygiene" and the type compatibility of the programs being generated. Naive approaches to typing generators have either been inflexible (giving rigid, invariant types to the meta-level expressions representing programs requiring every object term to have the same type) or undecidable (requiring dependent types with arbitrary equality theories on expression equality).

We solve the hygiene problems by the use of *syntactic closures* and have approached the type problem in two ways. First by using a two level system (à la Nielson and Nielson [9]) we are able to embed a meta-computation phase which associates invariant `code` types to object expressions, into a later phase which then indexes these `code` types with the object expression's type. This ensures that only well-typed programs reach the run-time phase. We guarantee that the meta-computation phase terminates by restricting its expressiveness by the use of catamorphisms as the exclusive mechanism to encode recursion.

Second we introduce a theory of dependent types for two-level languages that has a useful, decidable theory due to the use of catamorphisms, rather than arbitrary recursion, in the expressions that may index dependent types. We show that this can give useful types to program generators which detect type problems in generated code at the compile-time of the generator.

Third we show that by embedding second level type declarations as values in first level computations we are able to construct polytypic program generators, such as polymorphic equality and generic map using our theory.

1 Introduction

We propose a language with multiple levels that distinguishes expressions meant to run at compile-time, link-time, run-time etc. For each level, n, there is a phase that type-checks, elaborates, and translates meta-code for that level. On successful completion of that phase, the next phase involves the execution of the level n meta-code along with the type-checking, elaboration, and translation of the code at level $n+1$.

The overriding reason to resort to a multi-level system is the ability to express abstractions not typable by traditional type systems using program generators, yet still maintain the invariant that well-typed programs do not go wrong. In a multi level system this property can be guaranteed in a very simple manner. First limit the expressiveness of all but the last level (the meta-computation levels) in order to guarantee their termination, and second use ordinary Hindley-Milner type inference to type the generated code in the context where it is incorporated into higher levels.

In this paper we concentrate on two level programs. Level 1 being what we would normally associate as compile-time and level 2 being run-time. Thus we observe that compile-time can be expanded to accomplish more than just the transformation of source code to target code. It can also be used as a meta-programming phase in which user supplied code directs computation that constructs object-programs, which are incorporated and compiled along with the programs explicitly supplied by the programmer. At run-time the sum of the generated code and the explicitly supplied code is executed to supply results.

A two level language distinguishes three phases. In phase i, meta-code is typed, elaborated, and translated. In phase ii, the meta-code is executed to produce object-code, and the sum of the generated object code and the explicitly supplied second level code is compiled. In phase iii, (what we normally associate as run-time) the compiled code is executed to produce results.

In this two level, three phase scheme, type errors are all caught in either phase i or phase ii. Because of the termination properties of the meta language all compilations are guaranteed to terminate. While this maintains the desired property that well typed programs do not go wrong, type errors in generated code will not be found until they are

Programs

prog	::=	`level 1` d^* `level 2` d^* e	programs
d	::=	`type` T = (`Fn` α_i `=>`)* t	declarations
	$\Vert$	`type` T = (`Fn` α_i `=>`)* `Fix` β `=>` t	
	$\Vert$	`val` v = e	
t	::=	`unit` $\Vert$ `String` $\Vert$ `Int` $\Vert$ t`*`t' $\Vert$ t `->` t' $\Vert$ t`|`t'	types
	$\Vert$	α $\Vert$ T t^*	
e	::=	() $\Vert$ v $\Vert$ $e\ e$ $\Vert$ `fn` v `=>` e	
	$\Vert$	c	constants like: 5
	$\Vert$	(e , e) $\Vert$ `fn` (v_1, v_2) `=>` e	products
	$\Vert$	`inl` e $\Vert$ `inr` e $\Vert$ `fn` v_l `=>` e `|` v_r `=>` e	sums
	$\Vert$	`in`T e $\Vert$ `fn in`T v `=>` e $\Vert$ `cata`T e $\Vert$ `fix` v `=>` e	recursive types
	$\Vert$	`<`e`>` $\Vert$ $\sim e$	meta expressions

Figure 1: **Syntax of the two level language**

compiled in phase ii.

To solve this problem we have developed a type system incorporating value-dependent types in the first phase. This allows us to type interesting meta-programs in phase i, and to determine that all programs generated by such meta-programs are well-typed.

2 The Two Phase Language

The syntax of our two level language is presented in Figure 1. Programs consist of two levels. Each level is a sequence of declarations. The second level declarations are followed by an expression, that computes the result of the program. A declaration is either a type declaration or a value declaration. To ensure the safety of such a system, we place limits on what may appear in each level. In the sequel, we describe a type systems for each level that admits only type-safe programs.

2.1 Type declarations

Types include the primitive types `unit`, `Int`, and `String`, binary pairs, binary sums and function types. Type abstractions are specified using `Fn` and are limited to the outermost scope of type declarations. Recursive types are specified explicitly using, `Fix`, the fix point operator on types. Like type abstractions the fix point operator must appear at top level as well, except it must appear immediately after type abstractions (if any). We use the syntax `Fix x => T x` for μ `T`, but make explicit that `T` is a type constructor that takes a a type parameter `x`.

2.2 The meta-language

The meta-extensions, `<`e`>` and $\sim e$ make our language a reflective *meta-programming* system. In meta-programming, programmers write *meta-programs* which manipulate and construct *object-programs*. In a reflective system the meta-language and the object language are the same, and there is a mechanism for executing the constructed object language programs.

In the two level language object language terms are constructed using the object bracket (`< >`), and the escape notation ($\sim$). In this notation, object brackets surrounding a lambda term denote the data structure that represents that term.

An escaped term inside object brackets allows the term to "escape" the effects of the object brackets. Thus inside object brackets wherever a term is expected an escaped term may occur. The escaped term is a meta-computation which "computes" a value to "fill in" the hole in the object term at that point. This object bracket, escape notation is similar to the back-quote-comma, quasi-quotation mechanism of Lisp and Scheme, where object brackets play the role of back-quote and escape plays the role of comma. For example:

```
val double x = <~x + ~x>
val f = <fn x => ~(double <x>) * 3>
```

The function `double` takes an argument, `x:`*object-term*, and produces a result of type *object-term*. `double <y-1>` produces `<(y-1) + (y-1)>`. The object-term, `f` represents the abstraction `<fn x => (x + x) * 3>`. Note the use of escape (`~(double <x>)`) to allow allow the application of the `double` function to escape the object brackets, and the use of the object brackets around `x` because the application of `double` is escaped but its argument, the object bound variable `x`, is not.

2.3 The expression language

In our language we choose untraditional ways of representing products, sums, and recursive types because representing tuples as nested binary-products, and sums as nested binary sums supports a superior notation for meta-programming. This is because each syntactic construct has a fixed "size" and "shape". There are no syntactic constructs with unbounded size. For convenience, though, we wish to use a display notation that is more familiar. In Section 2.5 we provide rules which map more familiar notation to the expression language of Figure 1 we are about to describe.

The unusual features of the expression language include our treatment of all products and sums as binary products and sums. Products are constructed using the usual parenthesis-comma notation, `(x,y)`. The operators `inl` and `inr` are the left and right injectors of the sum type. *Product-abstraction*: `(fn (x,y) => e)` is defined by the rule `(fn (x,y) => f(x,y)) (`e_1`,`e_2`) = f(`e_1`,`e_2`)`. *Sum-abstraction*: `(fn x => m | y => n)`, when applied to `(inl u)`, computes `(fn x => m) u` and, when applied to `(inr v)`, computes `(fn y => n) v`.

Unlike many functional languages, our language does not use explicit value constructors to construct instances of recursive types. Instead, it uses the explicit product, sum, and

```
                                 (x,y,z,w)  ⇒  (x,(y,(z,w)))                              (1)
                          fn (x,y,z) => e  ⇒  fn (x,m) => (fn (y,z) => e) m              (2)
                fn x => a | y => b | z => c  ⇒  fn x => a | m => (fn y => b | z => c) m  (3)
fn x => (fn (a,b) => e) x | y => (fn (m,n) => w) y  ⇒  fn (a,b) => e | (m,n) => w       (4)
                       if t then e1 else e2  ⇒  (fn () => e2 | () => e1) t              (5)
```

Figure 2: **Syntactic Shorthands**

`in T` operators. The operator `in T` is the mediating morphisms between values of type `T` and its fixpoint, specifically between `T(Fix` x `=> T` x `)` and `Fix` x `=> T` x .

The ordinary value constructors of a recursive type can be defined in terms of these operators. For example, for ML style lists:

```
datatype 'a list = Nil | Cons of 'a * 'a list
```

we have instead

```
    type L = Fn a => Fn x => unit | a * x
    type List = Fn a => Fix x => L a x
```

Thus `in L` has type:

```
     (unit | a * (Fix x => L a x)) -> (Fix x => L a x)
```

and the constructors `Nil` and `Cons` are defined by:

```
   val Nil = in L (inl ())
   val Cons = fn (a,r) => in L (inr (a,r))
```

Traditional languages use case statements to decompose constructed values. Our term language can capture any case analysis over a value construction by using sum-, product-, and in-abstractions. For example,

```
    case e of Nil => e1 | Cons(a,r) => e2
```

can be expressed by the following composition of abstraction operators applied to `e`:

```
(fn in L x =>
    (fn z => e1 | y => (fn (a,r) => e2) y) x) e
```

2.4 Recursion and catamorphisms

In the expression language recursion in functions is expressed explicitly using the `fix` x `=>` e operator[1]. For example the recursive factorial function could be expressed by:

```
val fact =
    fix f => fn x => if x=0 then 1 else x * f(x-1)
```

The catamorphism operator, `cata T` ϕ , expresses uniform recursion[6] over the type: `Fix x => T x`. The type of `cata` T is $(\mathtt{T}\alpha \to \alpha) \to$ `(Fix x => T x)` $\to \alpha$. The `cata` operator obeys the following equation: `cata T` ϕ `(in T` x `)` $=$ ϕ`(T(cata T` ϕ `)` x`)` where the use of the type operator `T` in the value space is its natural lifting as a functor.

Note that `cata`, `in`, and `in`-abstraction are explicitly annotated with names, T, which are introduced in type declarations. To illustrate the use of catamorphisms, we define a function to sum the values in a list as a catamorphism over `List` as defined above.

```
   val total = cata L (fn () => 0 | (x,y) => x+y)
```

[1] We use lowercase `fix` for the operator on values and capitalized `Fix` for for the operator on types. We use a similar notation for the abstraction operators `Fn` and `fn`.

For reader unfamiliar with catamorphisms as control structures we have provided Appendix A.

Both the `fix` combinator and `cata` operators can express recursive computations, though the recursion in catamorphism is implicit and is "hidden" inside. It is important to note that if ϕ is a terminating function then `cata T` ϕ is also terminating. We will exploit this important property in the sequel to guarantee that all programs in the first level of our two level language terminate.

2.5 Syntactic shorthands

As illustrated in Figure 2 in Eqn. 1, we may sometimes write n-ary products, by which we mean right associative binary products. In Eqn. 2, a similar rule holds for n-ary product abstractions, and in Eqn. 3 for n-ary sum-abstractions. In Eqn. 4, a sum-abstraction, each of whose arms is the application of a product-abstraction to the sum-abstraction variables, may sometimes be written using "patterns", In Eqn. 5, the if-expression is a syntactic shorthand for a sum-abstraction over `Bool`, where `Bool` is defined by: `type Bool = unit | unit` and the constants `true` and `false` are defined by: `val true = inr ()` and `val false = inl ()`.

We sometimes use the integer constants `0`, `1`, `2`, etc. as if they were ordinal natural numbers, i.e. elements of the type `Nat` defined by:

```
type N = Fn a => unit | a
type Nat = Fix a => N a
val zero = in T (inl ())
val succ = fn x => in N (inr x)
```

where `0` = `zero`, `1` = `succ(zero)`, etc.

Finally, we write `Fix T` for the fixpoint of the type constructor `T` instead of the more verbose `Fix x => T x`.

2.6 Meta-language semantics

Meta-programming systems are notorious for insufficiently specifying the details about handling environments in which object terms may be "evaluated". These environments specify bindings for the object term's free variables. For example if the object term `<f 3>` is constructed in a context where `f` has one meaning, and evaluated in another context where `f` has another meaning, in the evaluation which meaning does `f` refer to? Object-terms which contain abstractions also may exhibit problems with inadvertent variable capture. The absence of these problems has been called hygiene[4]. Any meta programming system must supply semantics which describe precisely how these problems are handled.

In Figure 3 we give semantics for the *meta-lambda calculus*, a subset of our two level language. The syntactic category e describes the formation rules for *terms*. The semantic *value* domain is the separated sum: *int* | *value* $\to$ *value* | *term*, where terms have local environments ρ. The

$$
\begin{array}{lcl}
e : \mathit{term} & ::= & \overline{n} \;\;||\;\; v \;\;||\;\; e\;e \;\;||\;\; \texttt{fn}\; v \;\texttt{=>}\; e \;\;||\;\; \texttt{<}e\texttt{>} \;\;||\;\; \sim e \quad \textbf{where}\;\; n : \texttt{Int} \quad v : \mathit{variable} \\
S : \mathit{value} & ::= & [\,n\,]^i \;\;||\;\; [\,g\,]^f \;\;||\;\; [\,e\,]^t_\rho \quad \textbf{where}\;\; \rho : \mathit{variable} \to \mathit{value}, \quad g : \mathit{value} \to \mathit{value} \\
\mathit{ext}\;\sigma\;v\;s & = & \lambda\,u\,.\,\mathit{if}\;\; u = v \;\;\mathit{then}\;\; s \;\;\mathit{else}\;\; \sigma\;u
\end{array}
$$

$$(\llbracket\, x : \mathit{term} \,\rrbracket^{\sigma : \mathit{variable} \to \mathit{value}}) : \mathit{value}$$

$$
\begin{array}{lcll}
\llbracket\, \overline{n} \,\rrbracket^\sigma & = & [\,n\,]^i & \\
\llbracket\, v \,\rrbracket^\sigma & = & \sigma\;v & \\
\llbracket\, e_1\;e_2 \,\rrbracket^\sigma & = & g\;\llbracket\, e_2 \,\rrbracket^\sigma & \text{when } \llbracket\, e_1 \,\rrbracket^\sigma \;=\; [\,g\,]^f \\
\llbracket\, \texttt{fn}\;\; v \;\texttt{=>}\; e \,\rrbracket^\sigma & = & [\,\lambda s.\llbracket\, e \,\rrbracket^{\mathit{ext}\;\sigma\;v\;s}\,]^f & \\
\llbracket\, \texttt{<}\overline{n}\texttt{>} \,\rrbracket^\sigma & = & [\,\overline{n}\,]^t_\epsilon & \\
\llbracket\, \texttt{<}v\texttt{>} \,\rrbracket^\sigma & = & \mathcal{C}\;(\sigma\;v) & \\
\llbracket\, \texttt{<}e_1\;e_2\texttt{>} \,\rrbracket^\sigma & = & [\,\{\!\!\llbracket\, \texttt{<}e_1\texttt{>} \,\rrbracket\!\!\}^\sigma_{\rho_1}\;\; \{\!\!\llbracket\, \texttt{<}e_2\texttt{>} \,\rrbracket\!\!\}^\sigma_{\rho_2}\,]^t_{\rho_1 \cup \rho_2} & \\
\llbracket\, \texttt{<fn}\;\; v \;\texttt{=>}\; e\texttt{>} \,\rrbracket^\sigma & = & [\,\texttt{fn}\;\; v' \;\texttt{=>}\; \{\!\!\llbracket\, \texttt{<}e\texttt{>} \,\rrbracket\!\!\}^{\mathit{ext}\;\sigma\;v\;[\,v'\,]^t_\epsilon}_\rho\,]^t_\rho & \text{where } v' \text{ is a new variable} \\
\llbracket\, \texttt{<<}e\texttt{>>} \,\rrbracket^\sigma & = & [\,\texttt{<}\{\!\!\llbracket\, \texttt{<}e\texttt{>} \,\rrbracket\!\!\}^\sigma_\rho\texttt{>}\,]^t_\rho & \\
\llbracket\, \texttt{<}\sim e\texttt{>} \,\rrbracket^\sigma & = & \llbracket\, e \,\rrbracket^\sigma & \\
\llbracket\, \sim e \,\rrbracket^\sigma & = & \llbracket\, \{\!\!\llbracket\, e \,\rrbracket\!\!\}^\sigma_\rho \,\rrbracket^\rho & \\
 & & & \\
\{\!\!\llbracket\, e \,\rrbracket\!\!\}^\sigma_\rho & = & y & \text{when } \llbracket\, e \,\rrbracket^\sigma \;=\; [\,y\,]^t_\rho \\
 & & & \\
\mathcal{C}\;\;[\,e\,]^t_\rho & = & [\,e\,]^t_\rho & \\
\mathcal{C}\;\;s & = & [\,v'\,]^t_{\{v' \mapsto s\}} &
\end{array}
$$

Figure 3: **The Semantics of the Meta Lambda Calculus**

syntactic formation rules for the semantic domain of values is given by S. Because we need to discriminate between the three summands of S we use the [ ] notation with superscripts i,f, and t that denote discriminants. Thus $[e]^t_\rho$ denotes a value, where e is an object term and ρ is its local environment. We use the notation: $\llbracket\, \_\_ \,\rrbracket^\sigma : \mathit{term} \to \mathit{value}$, for the meaning function for terms under the mapping $\sigma : \mathit{variable} \to \mathit{value}$. If a term e under σ has as its meaning some term value $[\,y\,]^t_\rho$ with local environment mapping ρ we write $\{\!\!\llbracket\, e \,\rrbracket\!\!\}^\sigma_\rho \;=\; y$.

Excluding the meta-programming capabilitys (< > and $\sim$) the meta-lambda calculus has a completely standard semantics. In function application if e_1 does not evaluate to a function value then the semantics returns bottom. Unlike the the quasi quotation mechanism of lisp-like languages, a bracketed term never evalutes to an object term containing free variables. The rule for object variables ($\llbracket\, \texttt{<}v\texttt{>} \,\rrbracket^\sigma$), looks up the variable in the static environment, σ, where the bracket term appears. This value is handled by the function $\mathcal{C}$. Object bound variables (variables bound by bracketed abstractions), are just mapped to themselves, $[\,e\,]^t_\rho$. If the variable is not an object bound variable, then it is a value s. A term value is constructed, the term inside the value is a new variable, and the local environment of the term value, maps this new variable to the value s, $[\,v'\,]^t_{\{v' \mapsto s\}}$. Thus the free variables in object brackets are embedded in the local environment ρ as a "syntactic closure". Thus all free variables in bracketed terms are captured in the static environment where the object bracketed expression occurs.

The possibility of inadvertent variable capture is not possible because the meaning of an object abstraction, constructs a new abstraction with a new variable and consistently renames all occurnces of the old variable. The semantics ensures that no bound variable in a constructed piece of code can be known by the programmer, thus ensuring good hygiene.

Reflection, the execution of object language terms, is indicated by an escaped term, e, which is not embedded in object brackets. The term e is evaluated to obtain a term value with local environment ρ. This term is then re-evaluated in this local environment. These semantics generalize to the two level language with products, sums, and recursive types quite easily.

2.6.1 A complete example

Consider the meta-computation specified by the function `ntuple` that given a natural number `n`, constructs an object abstraction which builds an $(n-1)$-tuple from its argument.

```
val ntuple =
    fn n =>
       <fn x =>
           ~(cata ℕ (fn () => <x>
                      | s => <(x,~s)>) n)>
```

Given an integer `n` the function `ntuple` builds an object abstraction, whose bound variable is `x` and whose body is some term computed by the escaped catamorphism over `n`. If `n` is 0, it returns just the variable `x`, otherwise it returns a product whose first component is `x` and whose second component is the result of applying the catamorphism to `n-1`.

Thus the result of applying `ntuple` to the natural number 0 is: `<fn x => x>`, to 1 is: `<fn x => (x,x)>`, and to 2 is: `<fn x => (x,(x,x))>`, etc.

The purpose of meta computation is to "compute" code in level 1, which is to be "spliced" into the level 2 program. In Figure 4 a small but complete two level program is illustrated. Here escaped expressions involving values defined in level 1 direct meta-computation in the compilation phase of level 2.

The level 1 `ntuple` function is used during the compilation of the level 2 program to construct the abstraction `fn x => (x,(x,(x,x)))`, which is used as the meaning of the level 2 variable `quadruple`. Note that `(x,(x,(x,x)))`, really is a quadruple since by the suntactic shorthand described in Eqn. 1, it is equivalent to `(x,x,x,x)`. This program computes: `(33,33,33,33)`.

```
level 1

type N = Fn a => unit | a
type Nat = Fix a => N a
val ntuple = fn n => <fn x => ~(cata N (fn () => <x> | s => <(x,~s)>) n)>

level 2

val quadruple = ~(ntuple 3)

quadruple 33
```

Figure 4: **A small but complete two level program**

3 Typing the two level language

Figure 1 gives rules for syntacticly valid term constructions. But not all such terms have meaning. Traditionally, a type system is used to report invalid terms by assigning types to terms. In Figure 5 we give type rules for the two level language. The rules are organized into three sections. The core rules are valid for typing both level 1 and level 2 terms. We give additional rules that should be unioned with the core rules to complete the rules for each level. We introduce the type `code` as the rigid type of all object terms.

Note that the type rules ensure that bracketed terms may appear only in level 1 terms, and that the fix point operator (and hence recursion and thus the possibility of non-termination) may appear in only level 2 terms. In level 1 terms, escapes may occur only inside brackets, and in level 2 terms, escapes may occur at any point. The escaped terms in level 2 may reference only constants and level 1 values.

Note that the rule **2-esc** implies that, in order to type level 2 terms, the type system must evaluate level 1 terms to obtain object language terms which are then typed.

It is important to note that the additional rules for level 1 rigidly type every object language term with the a single type, `code`. We will return and reconsider this design choice later.

4 Why Two Levels?

The reason for using typing at two levels is that it is possible to define abstractions using generators which cannot be well typed under the usual Hindley-Milner polymorphic type system.

To illustrate the problem, consider the following function *last*. It takes a right associative nested family of tuples and produces the last element of an $(n+1)$-tuple.

$$\begin{array}{rcl} last\,0\,x & = & x \\ last\,n\,(x,y) & = & (last\,(n-1))\,y \end{array}$$

Where $last\ 1\ (x,y) \ \Rightarrow\ y$ and $last\ 2\ (x,y,z)\ \Rightarrow\ z$. This function is not typable in traditional type systems, since the two clauses of the definition of *last* have two different types. By using two levels, both clauses of *last* compute something with type `code`, because both compute object language terms.

```
last 0 = <fn x => x>
last n = <fn (x,y) => ~(last (n-1)) y>
```

Because type checking level 2 code involves executing level 1 code, and we want our type checker to always terminate, we limit the expression of recursive computations in level 1 to those using `cata`. Thus we write the equivalent and provably terminating:

```
val last =
    fn n => cata N (fn () => <fn x => x>
                     | f => <fn (x,y) => ~f y>) n
```

It is only after applying `last` to a constant in some escaped level 2 term do we attempt to type the object code produced.

Type problems with object language terms generated by level 1 programs are only detected in phase ii. Can we do better? Can we guarantee the generator generates only well-typed code? Can we do it when the generator is type-checked once and for all, rather than wait each time until the code is generated? For some programs the answer is yes! To understand why, consider that if `last` is applied to a constant, say 2, the resulting object language term does have a fixed type, but now the type of `last` *depends* upon the value of its first argument. A function whose type depends upon the value of its argument is said to have a dependent type. We will write the type of `last` as: $\Pi\, \mathtt{n} : \mathtt{Nat}\ .\, \mathcal{E}(\mathtt{n})$ to denote that its type depends upon the value of `n`. By introducing a richer type system involving dependent types we can type many programs such as `last`. We will see that dependent types will also allow us to relax the design decision of allowing only rigid `code` types.

5 Catamorphisms and Dependant Types

Many dependent type systems are undecidable (requiring dependent types with arbitrary equality theories on expression equality). In this paper we restrict dependent types to the use of catamorphisms (rather than arbitrary recursion) in the expressions that index dependent types. Our theory of dependent types for two-level languages has a useful, decidable theory.

$$\begin{array}{rcl}
\sigma & ::= & S(t) \ \ \|\ \ \Pi\, v : \sigma\,.\,\sigma' \ \ \|\ \ \mathtt{Cata}\ T\ (\,\Lambda\,\omega\,)\ v \\
\omega & ::= & \nu \ \ \|\ \ \omega|\omega \\
\nu & ::= & (\,\rho_1\ \mathtt{=>}\ \sigma_1) \\
\rho & ::= & \alpha \ \ \|\ \ \mathtt{unit} \ \ \|\ \ \mathtt{Int} \ \ \|\ \ \mathtt{String} \ \ \|\ \ \rho \times \rho' \\
 & \| & \rho \to \rho' \ \ \|\ \ \rho + \rho'
\end{array}$$

Figure 6: **Syntax of Types for the Dependent Type System**

Core Language Rules

$$\text{unit:}\quad \frac{\Gamma \vdash}{\Gamma \vdash () : 1} \qquad\qquad \text{var:}\quad \frac{\{v : \tau\} \in \Gamma}{\Gamma \vdash v : \tau}$$

$$\text{abs:}\quad \frac{\Gamma \cup \{v : \sigma\} \vdash e : \tau}{\Gamma \vdash \texttt{fn}\ v\ \texttt{=>}\ e : \sigma \to \tau} \qquad\qquad \text{app:}\quad \frac{\Gamma \vdash f : \sigma \to \tau, \quad x : \sigma}{\Gamma \vdash f\ x : \tau}$$

$$\text{prod-abs:}\quad \frac{\Gamma \cup \{v_1 : \sigma, v_2 : \tau\} \vdash e : \gamma}{\Gamma \vdash \texttt{fn}\ (v_1, v_2)\ \texttt{=>}\ e : (\sigma \times \tau) \to \gamma} \qquad\qquad \text{prod:}\quad \frac{\Gamma \vdash e_1 : \sigma, \quad e_2 : \tau}{\Gamma \vdash (e_1\ ,\ e_2) : \sigma \times \tau}$$

$$\text{suml:}\quad \frac{\Gamma \vdash x : \sigma}{\Gamma \vdash \texttt{inl}\ x : \sigma + \tau} \qquad\qquad \text{sumr:}\quad \frac{\Gamma \vdash x : \tau}{\Gamma \vdash \texttt{inr}\ x : \sigma + \tau}$$

$$\text{sum-abs:}\quad \frac{\Gamma \cup \{v_l : \sigma_l\} \vdash e_l : \tau, \quad \Gamma \cup \{v_r : \sigma_r\} \vdash e_r : \tau}{\Gamma \vdash \texttt{fn}\ v_l\ \texttt{=>}\ e_l\ \texttt{|}\ v_r\ \texttt{=>}\ e_r : (\sigma_l | \sigma_r) \to \tau}$$

$$\text{in:}\quad \frac{\Gamma \vdash x : T(\texttt{Fix}\,T)}{\Gamma \vdash \texttt{in}\,T\ \ x\ : \texttt{Fix}\,T} \qquad\qquad \text{out:}\quad \frac{\Gamma \cup \{v : T(\texttt{Fix}\,T)\} \vdash x : \alpha}{\Gamma \vdash \texttt{fn}\ \texttt{in}\,T\ \ v\ \texttt{=>}\ x : (\texttt{Fix}\,T) \to \alpha}$$

$$\text{cata:}\quad \frac{\begin{array}{c} T\ = \Lambda\,\alpha_1\,.\cdots.\Lambda\,\alpha_k\,.\tau \\ \Gamma \vdash f : T\ \beta_1\ \cdots\ \beta_{k-1}\ \gamma \to \gamma \end{array}}{\Gamma \vdash \texttt{cata}\,T\ f\ : (\texttt{Fix}\,x\ \texttt{=>}\ T\ \beta_1\ \cdots\ \beta_{k-1}\ x) \to \gamma}$$

Addtional Rules for Level 1

$$\text{br-unit:}\quad \Gamma \vdash \texttt{<()>} : \texttt{code}$$

$$\text{br-esc:}\quad \frac{\Gamma \vdash e : \alpha}{\Gamma \vdash \texttt{<\sim}\ e\texttt{>} : \alpha} \qquad\qquad \text{br-var:}\quad \frac{\Gamma \vdash v : \alpha}{\Gamma \vdash \texttt{<}v\texttt{>} : \texttt{code}}$$

$$\text{br-app:}\quad \frac{\Gamma \vdash \texttt{<}f\texttt{>} : \texttt{code}, \texttt{<}x\texttt{>} : \texttt{code}}{\Gamma \vdash \texttt{<}f\ x\texttt{>} : \texttt{code}} \qquad\qquad \text{br-abs:}\quad \frac{\Gamma \cup \{v : \alpha\} \vdash \texttt{<}e\texttt{>} : \texttt{code}}{\Gamma \vdash \texttt{<fn}\ v\ \texttt{=>}\ e\texttt{>} : \texttt{code}}$$

$$\text{br-prod-abs:}\quad \frac{\Gamma \cup \{v_1 : \sigma, v_2 : \tau\} \vdash \texttt{<}e\texttt{>} : \texttt{code}}{\Gamma \vdash \texttt{<fn}\ (v_1, v_2)\ \texttt{=>}\ e\texttt{>} : \texttt{code}} \qquad\qquad \text{br-prod:}\quad \frac{\Gamma \vdash \texttt{<}e_1\texttt{>} : \texttt{code}, \quad \texttt{<}e_2\texttt{>} : \texttt{code}}{\Gamma \vdash \texttt{<}(e_1\ ,\ e_2)\texttt{>} : \texttt{code}}$$

$$\text{br-suml:}\quad \frac{\Gamma \vdash \texttt{<}x\texttt{>} : \texttt{code}}{\Gamma \vdash \texttt{<inl}\ x\texttt{>} : \texttt{code}} \qquad\qquad \text{br-sumr:}\quad \frac{\Gamma \vdash \texttt{<}x\texttt{>} : \texttt{code}}{\Gamma \vdash \texttt{<inr}\ x\texttt{>} : \texttt{code}}$$

$$\text{br-sum-abs:}\quad \frac{\Gamma \cup \{v_l : \sigma_l\} \vdash \texttt{<}e_l\texttt{>} : \texttt{code}, \quad \Gamma \cup \{v_r : \sigma_r\} \vdash \texttt{<}e_r\texttt{>} : \texttt{code}}{\Gamma \vdash \texttt{<fn}\ v_l\ \texttt{=>}\ e_l\ \texttt{|}\ v_r\ \texttt{=>}\ e_r\texttt{>} : \texttt{code}}$$

$$\text{br-in:}\quad \frac{\Gamma \vdash \texttt{<}w\texttt{>} : \texttt{code}}{\Gamma \vdash \texttt{<in}\,T\ \ w\ \texttt{>} : \texttt{code}} \qquad\qquad \text{br-out:}\quad \frac{\Gamma \cup \{v : T(\texttt{Fix}\,T)\} \vdash \texttt{<}e\texttt{>} : \texttt{code}}{\Gamma \vdash \texttt{<fn in}\,T\ \ v\ \texttt{=>}\ e\texttt{>} : \texttt{code}}$$

Addtional Rules for Level 2

$$\text{2-esc:}\quad \frac{\Gamma \vdash x : \texttt{code}, \ [\![\, x \,]\!]^{\sigma}_{\rho}\ =\ y,\ y : \beta}{\Gamma \vdash \sim x :\ \beta} \qquad\qquad \text{2-fix:}\quad \frac{\Gamma \cup \{v : \alpha \to \beta\} \vdash e : \alpha \to \beta}{\Gamma \vdash \texttt{fix}\,v\ \texttt{=>}\ e : \alpha \to \beta}$$

Figure 5: **Type rules for the two level language**

Addtional Rules for Level 1

br-unit: $\Gamma \vdash \texttt{<()>} : \texttt{code(unit)}$

br-esc: $$\frac{\Gamma \vdash e : \alpha}{\Gamma \vdash \texttt{<}\sim e\texttt{>} : \alpha}$$

br-var: $$\frac{\Gamma \vdash v : \alpha}{\Gamma \vdash \texttt{<}v\texttt{>} : \texttt{code}(\alpha)}$$

br-app: $$\frac{\Gamma \vdash \texttt{<}f\texttt{>} : \texttt{code}(\alpha \to \beta), \texttt{<}x\texttt{>} : \texttt{code}(\alpha)}{\Gamma \vdash \texttt{<}f\ x\texttt{>} : \texttt{code}(\beta)}$$

br-abs: $$\frac{\Gamma \cup \{v : \alpha\} \vdash \texttt{<}e\texttt{>} : \texttt{code}(\beta)}{\Gamma \vdash \texttt{<fn}\ v\ \texttt{=>}\ e\texttt{>} : \texttt{code}(\alpha \to \beta)}$$

br-prod-abs: $$\frac{\Gamma \cup \{v_1 : \sigma, v_2 : \tau\} \vdash \texttt{<}e\texttt{>} : \texttt{code}(\gamma)}{\Gamma \vdash \texttt{<fn}\ (v_1, v_2)\ \texttt{=>}\ e\texttt{>} : \texttt{code}((\sigma \times \tau) \to \gamma)}$$

br-prod: $$\frac{\Gamma \vdash \texttt{<}e_1\texttt{>} : \texttt{code}(\sigma), \quad \texttt{<}e_2\texttt{>} : \texttt{code}(\tau)}{\Gamma \vdash \texttt{<(}e_1\ \texttt{,}\ e_2\texttt{)>} : \texttt{code}(\sigma \times \tau)}$$

br-suml: $$\frac{\Gamma \vdash \texttt{<}x\texttt{>} : \texttt{code}(\sigma)}{\Gamma \vdash \texttt{<inl}\ x\texttt{>} : \texttt{code}(\sigma + \tau)}$$

br-sumr: $$\frac{\Gamma \vdash \texttt{<}x\texttt{>} : \texttt{code}(\tau)}{\Gamma \vdash \texttt{<inr}\ x\texttt{>} : \texttt{code}(\sigma + \tau)}$$

br-sum-abs: $$\frac{\Gamma \cup \{v_l : \sigma_l\} \vdash \texttt{<}e_l\texttt{>} : \texttt{code}(\tau), \quad \Gamma \cup \{v_r : \sigma_r\} \vdash \texttt{<}e_r\texttt{>} : \texttt{code}(\tau)}{\Gamma \vdash \texttt{<fn}\ v_l\ \texttt{=>}\ e_l\ \texttt{|}\ v_r\ \texttt{=>}\ e_r\texttt{>} : \texttt{code}((\sigma_l | \sigma_r) \to \tau)}$$

br-in: $$\frac{\Gamma \vdash \texttt{<}w\texttt{>} : \texttt{code}(T(\texttt{Fix}\, T))}{\Gamma \vdash \texttt{<in}\,T\ \ w\ \texttt{>} : \texttt{code(Fix}\, T\texttt{)}}$$

br-out: $$\frac{\Gamma \cup \{v : T(\texttt{Fix}\, T)\} \vdash \texttt{<}e\texttt{>} : \texttt{code}(\alpha)}{\Gamma \vdash \texttt{<fn in}\,T\ \ v\ \texttt{=>}\ e\texttt{>} : \texttt{code}((\texttt{Fix}\, T) \to \alpha)}$$

Figure 7: **Indexed `code` typing of level 1 programs.**

We give the syntax of our dependent type system in Figure 6. We will discuss its meaning by explaining how to type the `last` example.

Types include the types, t of Figure 1 as well as the two new types Π and *Cata*. Note that the dependent variable v introduced by Π types may only appear in *Cata* types. A *Cata* type encodes a computation, which when given a value, produces a type. Patterns, ρ, in type abstractions allow a limited form of case analysis of types inside fold types.

In the dependent type system we replace the abstraction type rules with rules which give every abstraction a Π type. For example:

dep-abs: $$\frac{\Gamma \cup \{v : \tau_1\} \vdash e : \tau_2}{\Gamma \vdash \texttt{fn}\ v\ \texttt{=>}\ e : \Pi v : \tau_1 \,.\, \tau_2}$$

Thus if the non-generator version of *last* (the version in italics) is written as a catamorphism:

```
last = fn n =>
          cata N (fn () => (fn x => x)
                   | f  => (fn (x,y) => f y)) n
```

It can be given a Π type. Since the body of the abstraction is a catamorphism, and the *cata* rule of the core rules fail, we give the catamorphism in the definition of `last` a *Cata* type. The type of `last` is:

$$\Pi\, n : \texttt{Nat}\ \ .\texttt{Cata N} \left(\begin{array}{ll} \Lambda & \alpha\ \texttt{=>}\ \beta \to \beta \\ | & \gamma \to \delta\ \texttt{=>}\ (\epsilon \times \gamma) \to \delta \end{array} \right) n$$

One interprets this as follows: given a natural number n we can compute the type of `last` n. The first clause in the fold type, $\Lambda\, \alpha\ \texttt{=>}\ \beta \to \beta$, gives the type of `last` $0\ :\ \beta \to \beta$. This is accomplished by applying the type abstraction to the type unit, (the domain of zero). The second summand in the fold type, $\Lambda\ (\gamma \to \delta)\ \texttt{=>}\ (\epsilon \times \gamma) \to \delta$ tells how to compute the type of (`last` n) : $(\epsilon \times \gamma) \to \delta$ given the type of $(\texttt{last}(n-1)) : \gamma \to \delta$.

Intuitively *Cata* types allows us to type catamorphisms such as: `cata T phi`, where each clause of the sum abstraction, `phi`, has a different co-domain. Given a catamorphism, if the original core rule **cata** fails, we may use a the richer rule, **dep-cata**, below that gives the catamorphism a *Cata* type.

$$\frac{\begin{array}{c}\Gamma \vdash T\ =\ \texttt{Fn}\, v_t\ \texttt{=>}\ E_i(v_t) + \cdots + E_n(v_t)\ , \\ v : \texttt{Fix}\, T, \quad \nu_i\ :\ E_i(\alpha_i) \to \beta_i, \quad z_i = (\hat{E}_i(\alpha_i))\ \texttt{=>}\ \beta_i\end{array}}{\Gamma \vdash \texttt{cata}\, T\ (\texttt{fn}\, \nu_1 | \ldots | \nu_n)\ v\ :\ \texttt{cata}\ T\ (\Lambda\, z_1 | \ldots | z_n)\ v}$$

$$\text{when } \forall_{i,j \in 1..n}\, .\, E_j(\alpha_j) \cong E_j(\beta_i)$$

To type a catamorphism, the annotation T must be a type abstraction over v_t whose body is a sum. The summands $E_i(v_t)$ are type expressions which depend upon variable v_t. The cata variable v must have as its type the fixpoint of T .

The α's and β's range over types σ. The notation $E_i(\alpha)$ denotes the type (in the syntactic category σ of types) given by $S(E_i(v_t))$ with α substituted for v_t. In a similar fashion $\hat{E}_i$ works in the syntactic category of patterns ρ.

The ν_i are the cata actions to take, depending upon which summand the value of v is in. The domains of the ν_i functions must be appropriately shaped according to E_i for some α_i. Furthermore, in the cata type expression, the $\hat{E}_i(\alpha)$ patterns of the z_i have to match the $E_i(\alpha)$ domains of the ν_i's.

Finally, the collection of $E_i(\alpha_i)$ types must satisfy the "cata-coherence condition" underneath the rule that guarantees that both the types of the ν_i's and the z_i's patterns will properly match when the catamorphism unwinds itself. This condition states that the co-domain type of each clause, ν_i, must be compatible with the recursive parts of the domain[2] of every ν. For the type of `last` $\Pi\ n$: `Nat` `.Cata N` ($\Lambda\, \alpha\ \texttt{=>}\ \beta \to \beta\ |\ \gamma \to \delta\ \texttt{=>}\ (\epsilon \times \gamma) \to \delta$) n this manifests itself in the two conditions $(\beta \to \beta) \cong (\epsilon \times \gamma) \to \delta$ and $(\gamma' \to \delta') \cong (\epsilon \times \gamma) \to \delta$.

[2]If there are any. For example the domian for ν_{zero} is () and has no recursive parts.

```
level 1
val ext = fn env => fn s => fn x => cons((s,x),env)

val lookup = fn env => cata L (fn () => fn s => error "not found"
                               | ((t,x),g) => fn s => if s=t then x else g s) env
val mapgen =
fn x =>
  (cata Ty
    (fn  s => (fn env => <fn x => x>)
      |  s => (fn env => lookup s env)
      |  (f,g) => (fn env => <fn (p,q) => (~(f env) p,~(g env) q)>)
      |  (f,g) => (fn env => error "arrow type in map")
      |  (f,g) => (fn env => <fn lf => inL(~(f env) lf) | rt => inR(~(g env) rt)>)
      |  (s,f) => (fn env => <fn a => ~(f (ext env s <a>))>)
      |  (T,s,h) => (fn env => <fix m =>
                                   fn in ~T x => in ~T ~(h (ext env s <m>)) x >))
    x) Nil

level 2
type L = Fn a => (Fix x => unit | (a * x))
type List = Fn a => Fix l => unit | (a * l)
val nil = in [L] (inL ())
fun cons x = in [L] (inR x)

val map = ~(mapgen List)

map (fn x => x+1) (cons(2,cons(4,nil)))
```

Figure 8: **A complete two level program illustrating poly-typic map**

6 Indexed `code` Types

It is possible to give object language terms a much richer type indicating the type of the object term as well as the fact that the object terms are of type `code`. We indicate this by indexing the the type object language terms `code` with a type: `code`(α). Because a single meta program may produce many object language terms, each with a different type, many generators become impossible to type under this indexed code type scheme. This is because the type of the generator cannot be described by a single parametrically polymorphic type scheme.

By incorporating dependent types as described earlier we can solve this problem. The dependent types describe how to calculate the type of a generated program from the argument to the generator. The types of the generated programs are all members of a single family of types which are generated by a *Cata* type when applied to the generators input.

To infer indexed code types we need to enrich the rules described for level 1 programs. In Figure 7 we give the richer rules for inferring indexed `code` types from level 1 programs.

So finally using these rules, and the rule **dep-cata** the generator version of the function `last`:

```
val last =
    fn n => cata N (fn () => <fn x => x>
                  | f => <fn (x,y) => ~f y>) n
```

is given the type:

$$\Pi\, n : \texttt{Nat}\ .\quad \texttt{Cata N} \left(\begin{array}{ll} \Lambda & \alpha \ \texttt{=>}\ \texttt{code}(\beta \to \beta) \\ | & \texttt{code}(\gamma \to \delta)\ \texttt{=>}\ \texttt{code}((\epsilon \times \gamma) \to \delta) \end{array} \right) n$$

7 Polytypic Generators

Poly-typic programs are ad-hoc polymorphic programs which execute different code at each instance. A polytypic algorithm abstracts over the structure of data-types so that a single algorithm specifies many different code sequences once a particular data-type is fixed. Generic equality and map are well known examples. A program generator whose input is a data structure similar in shape and structure to type declarations can be used as a polytypic program generator. One such data structure is the type `Intension`:

```
type Ty = Fn t =>
  string                        (* int, string    *)
| string                        (* var            *)
| (t * t)                       (* product        *)
| (t * t)                       (* arrow          *)
| (t * t)                       (* sum            *)
| (string * t)                  (* Fn a => a * b  *)
| (annotation * string * t)     (* Fix x => t     *)

type Intension  = Fix x => Ty x
```

All types declarations in the two level language have an embedding in the type `Intension` (but not all `Intensions` represent valid type declarations since `Intension` does not require type abstraction and fix point to be the outermost constructors). The type `annotation` is a primitive type that will be explained later.

Generators over `Intension` specify polytypic functions. For example the map function for any type can be generated by the generator `mapgen` in figure 8

In level 1 the functions `ext` and `lookup` are used to manage a list of `string` $\times$ `code` pairs, that represents a mapping of the type variables to pieces of code. The function `lookup` is written as a catamorphism to guarantee its termination. The catamorphism inside the function `mapgen` when applied to `x:Intension`, produces a function from type variable mappings to pieces of code. This function is applied to the empty mapping (`Nil`) to produce a piece of code. The variable `T` of type `annotation` is used to annotate the `in` and the `in` abstraction operators in the last clause of the `cata`. When `mapgen` is applied to the `Intension` associated with the type declaration for `List` it produces the piece of code:

```
fn a =>
 fix m =>
     fn in L x =>
         in L (fn lf => (fn x => x) lf
                 | rt => (fn (p,q) => (a p,m q)) rt)
             x
```

which is the recursive definition of the map for lists.

7.1 Level 2 types as level 1 values

Of course users must encode the `Intensional` data structure which the `mapgen` function uses as input. Or do they? If all type declarations can be embedded into type `Intension` the compiler could automate this encoding. By predefining the types `Ty` and `Intension` in the level 1 compiler, and by arranging to bind the names of level 2 types to values of type `Intension` in escaped level 2 code, level 2 types can be treated as level 1 values of type `Intension`. This is where the value of `List` comes from in the escaped level 2 term: `(mapgen List)`. Since every valid type declaration has an embedding in type `Intension` it is easy for the compiler of the two level language to do this.

Using this feature it is possible to use our two level language as a general purpose polytypic generator system.

8 Relationship to other work

The use of two level languages with two level type systems comes directly from Nielson and Neilson [9] where it is used to specify compilers. Catamorphisms and some of their properties are discussed in [6, 11]. Harper and Morrisett [1] first use catamorphisms to express types, but their catamorphisms induct only over the structure of primitive types, rather than arbitrary values. Poly-typic programming, the specification of algorithms abstracted over the data structures they operate on has roots in some of our earlier work [10], but has been most succesfully promoted by Johan Jeuring [2, 3].

Martin-Löf pioneered the use of dependent type systems which have now become standard in many types systems for programming logics [5] and we incorporate them into the terminating parts of our language. Nelson investigates the issue of type inference for dependent types [7, 8].

9 Conclusion

Programming languages have always embodied a conceptual distinction between their compile and link-time aspects and their run-time behavior, distinctions that programmers have always been aware of to some extent. Recently more and more programs involve programming *staticly* as programming languages have greatly expanded their static facilities to provide richer methods of organizing and structuring programs such as polymorphism, overloading, modules, type classes, etc. An overall goal of these static facilities is increased ability to express abstraction along with its associated benefits of decreased program maintenance and increased software component reuse. Since programmers must already be aware of these static abstractions to program effectively in modern languages they should be able to create their own rather than be content with the static features supplied by any given language.

To provide such mechanisms in a type safe manner requires extending type systems to handle these abstractions. We have illustrated two extending mechanisms, multi-level type systems and dependent types.

References

[1] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd ACM Symposium on Principles of Programming Languages*, January 1995. to Appear.

[2] Johan Jeuring. Polytypic combinatorial functions. unpublished manuscript, email johan@cs.chalmers.se, 1994.

[3] Johan Jeuring. Polytypic pattern matching. In *Proceedings of the conference on Functional Programming and Computer Architecture*, La Jolla, California, June 1995.

[4] Eugene Kohlbecker, Daniel Friedman, Mathias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the SIGPLAN '86 ACM Conference on Lisp and Functional Programming*, pages 151–161, New York, August 1986. ACM Press. Cambridge, Ma.

[5] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118. North Holland, 1975.

[6] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts*, pages 124–144, August 1991.

[7] Neal Nelson. Primitive recursive functionals with dependant types. In *Mathematical Foundations of Programming Semantics*. Springer Verlag, march 1991. Lecture Notes in Computer Science 598.

[8] Neal Nelson. *Type Inference and Reconstruction for First Order Dependent Types*. PhD thesis, Department of Computer Science and Engineering, Oregon Graduate Institute, 1995.

[9] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1992.

[10] T. Sheard. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 19(4):531–557, October 1991.

[11] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings of the conference on Functional Programming and Computer Architecture*, Copenhagen, June 1993.

Standard ML Version

```
datatype Nat = Zero | Succ of Nat;


fun cata_Nat (Z,S) Zero = Z
  | cata_Nat (Z,S) (Succ x) = S(cata_Nat (Z,S) x);



datatype 'a list = Nil | Cons of 'a * 'a list;


fun cata_list (N,C) Nil = N
  | cata_list (N,C) (Cons(x,xs)) =
        C(x,cata_list (N,C) xs);



datatype 'a Tree = Tip of 'a |
                   Node of 'a Tree * 'a Tree

fun cata_Tree (T,N) (Tip x) = T x
  | cata_Tree (T,N) (Node(x,y)) =
        N(cata_Tree (T,N) x,cata_Tree (T,N) y);
```

Two Level Language Version

```
type N = Fn x => unit | x
type Nat = Fix x => N x

val mapN = fn f => (fn () => inl ()
                     | x  => inr (f x))
val cataN =
    fn phi => fn in N x = phi(mapN (cataN phi) x)

type L = Fn a => Fn x => unit | a * x
type List = Fn a => Fix x => L a x

val mapL = fn f => (fn ()    => inl ()
                     | (x,y) => inr(x,f y))

val cataL =
    fn phi => fn in L x = phi(mapL (cataN phi) x)

type T = Fn a => Fn x => a | x * x
type Tree = Fn a => Fn x => T a x

val mapT = fn f => (fn a     => inl a
                     | (x,y) => inr(f x,f y))

val cataL =
    fn phi => fn in T x = phi(mapT (cataN phi) x)
```

Figure 9: **Side by side comparison**

A Catamorphisms Explained

In a traditional functional language with constructors for recursive types, a catamorphism, cata T $(f_1, \ldots, f_n)$, can be viewed as a function that replaces every constructor $\mathbf{C}_i$ in a recursive value with a corresponding function, f_i. For example by repeatedly replacing **Cons** with c and **Nil** with n we proceed:

$$\begin{aligned} & \texttt{cata}\ \mathit{List}\ (n,c)\ (\mathbf{Cons}(7,\mathbf{Cons}(3,\mathbf{Nil}()))) \\ = \ & c\ (7, \texttt{cata}\ \mathit{List}\ (n,c)\ (\mathbf{Cons}(3,\mathbf{Nil}()))) \\ = \ & c\ (7, c\ (3, \texttt{cata}\ \mathit{List}\ (n,c)\ (\mathbf{Nil}()))) \\ = \ & c\ (7, c\ (3, n)) \end{aligned}$$

This is accomplished by a recursive function which takes a vector of functions f_i as an argument and which places a recursive call to the catamorphism on every "recursive" component. This is illustrated in the **Standard ML Version** side of Figure 9.

In the two level language the pattern of recursion in a type definition is first captured by a non-recursive type definition (or functor) such as N, L and T. The recursion in the type is introduced later using the explicit fix point operator in a separate type declaration. The constructors (Zero, Nil, Cons, etc.) of the ML version are captured by composition of the in operator and the sum injection functions inl and inr as was illustrated in section 2.3. Thus in the two level language instead of replacing each constructor with an associated function, we replace the in operator with an associated sum-abstraction, phi, after recursively reducing all the recursive components by using the appropriate map applied to (cata phi). It is interesting to note that the map functions capture exactly where the recursive calls are to be placed, and follow directly from the non-recursive type declarations of N, L and T.

The cata T operator is built directly from the type definition by the compiler in this manner.

To use a catamorphism the programmer must only supply the name of the non recursive type and the sum-abstraction phi. For example:

```
val total = cata L (fn () => 0 | (x,y) => x+y)

val length = cata L (fn () => 0 | (x,y) => y+1)

val app =
    fn x => fn y =>
         cata L (fn () => y | (x,y) => Cons(x,y)) x

val addition =
     fn x => fn y =>
          cata N (fn () => y | m => m+1) x

val flatten =
    cata T (fn x => Cons(x,Nil) | (m,n) => app m n)
```