Sexy types in action

Editor: *Philip Wadler, University of Edinburgh*; *wadler@inf.ed.ac.uk*

Chung-chieh Shan
Harvard University
Cambridge MA 02138 USA

The Hindley-Milner type system (Hindley 1969; Milner 1978) and its Damas-Milner inference algorithm (Damas and Milner 1982) for the λ-calculus (Church 1932, 1940) are widely adopted in modern functional programming languages like Haskell and ML. The system delicately balances power and tractability: many polymorphic functions can be built, and many safety constraints checked, without bogging down compilers in undecidability or programmers in verbosity.

As programmers became familiar with the system's power, however, they also became frustrated by its limitations. Much recent research thus explores the design space beyond the Hindley-Milner-Damas system, termed "sexy types" by Peyton Jones (2003). Two features often requested and implemented are *higher-rank polymorphism* and *existential types*. This annotated bibliography summarizes these features (§1) and motivates them by enumerating their present-day applications (§2–3).

1 Polymorphic typed λ-calculus

A polymorphic value is one that can take multiple types. We are concerned here with *parametric* polymorphism (Strachey 1967), which corresponds to universal quantification over types. To start, we review the typing rules for the polymorphic (or second-order) typed λ-calculus, also known as System F, invented independently by Girard (1972; Girard et al. 1989) and Reynolds (1974, 1990).

A function is created by discharging an assumption in the typing environment, and invoked by applying it to a value of the appropriate argument type.

$$\frac{\begin{array}{c}[x:\tau]\\ \vdots \\ E:\tau'\end{array}}{\lambda x\!:\!\tau.\, E : \tau \to \tau'} \to\!\mathrm{I}^x \qquad \frac{F:\tau\to\tau' \quad E:\tau}{FE:\tau'} \to\!\mathrm{E} \qquad (1)$$

Universal quantification is introduced by *generalizing* the type of a value, and eliminated by *specializing* or *instantiating* a polymorphic type.

$$\frac{E:\tau}{\Lambda a.\, E : \forall a.\, \tau} \forall\mathrm{I}\ (a \text{ does not escape}) \qquad (2a)$$

$$\frac{E:\forall a.\,\tau'}{E\tau : \tau'[\tau/a]} \forall\mathrm{E} \qquad (2b)$$

In the $\forall$I rule above, the type variable a that is quantified over must not *escape*—that is, it must not appear free in an undischarged typing assumption. When a type checker uses this rule bottom-up to check a universal type, it ensures that the side condition on the type variable a holds by generating a fresh dummy type (termed an *eigenvariable*) and substituting it for a. That is, it α-converts the bound variable a in $\forall a.\, \tau$.

In the $\forall$E rule above, $\tau'[\tau/a]$ denotes the capture-avoiding substitution of τ for a in τ'. The result of the substitution is said to *subsume* the universally quantified type $\forall a.\, \tau'$.

The term syntax of the polymorphic typed λ-calculus makes both of these rules explicit. For example, the polymorphic identity function is written as the term $\Lambda a.\, \lambda x\!:\!a.\, x$. It is given the type $\forall a.\, a \to a$ by the following derivation.

$$\cfrac{\cfrac{[x:a]}{\lambda x\!:\!a.\, x : a\to a} \to\!\mathrm{I}^x}{\Lambda a.\, \lambda x\!:\!a.\, x : \forall a.\, a \to a} \forall\mathrm{I}$$

Because it is too verbose to mark every introduction and elimination of a universal quantifier, we would like to make the rules in (2) implicit in terms, as follows.

$$\frac{E:\tau}{E:\forall a.\,\tau} \forall\mathrm{I}\ (a \text{ does not escape}) \qquad \frac{E:\forall a.\,\tau'}{E:\tau'[\tau/a]} \forall\mathrm{E} \qquad (3)$$

We would also like to omit the argument type τ in the function term $\lambda x\!:\!\tau.\, E$. Unfortunately, this move makes type checking and type inference undecidable (Wells 1999). To recover decidability while preserving implicit polymorphism, the Hindley-Milner-Damas type system trades off expressive power by restricting the *rank* of types to 1. The rank of a type is the maximum number of function arrows to the left of which a universal quantifier appears. Formally, the following syntax defines a stratified language of rank-n types τ^n, where n ranges over the non-negative integers.

$$\begin{aligned} \tau^0 &::= a \mid \tau^0 \to \tau^0 \\ \tau^{n+1} &::= \forall a.\, \tau^{n+1} \mid \tau^n \to \tau^{n+1} \end{aligned}$$

A rank-0 type τ^0 has no universal quantifier; it is called a *monotype*. Hindley, Milner, and Damas's innovation is to require monotypes in (1), allow rank-1 types in (3), and add the "let" construct below for polymorphic assumptions (Clément et al. 1986).

$$\frac{E : \tau^1 \qquad \begin{array}{c}[x : \tau^1] \\ \vdots \\ E' : \tau^0\end{array}}{\text{let } x = E \text{ in } E' : \tau^0}\,\text{Let}^x \tag{4}$$

In the polymorphic typed λ-calculus, the only (closed) term of the type $\forall a.\, a \to a$ is the identity function. In general, terms in the polymorphic typed λ-calculus are always *parametric*, which intuitively means that a value acts "uniformly" at all types it can take, regardless of how universally quantified type variables are instantiated. A function that negates booleans while leaving non-booleans unchanged would not be parametric. This property of uniformity, or *type abstraction* (Reynolds 1983), is usually formalized by introducing *logical relations* between types and guaranteeing that, for example, a function always maps related arguments to related results.

A semantic model of the polymorphic typed λ-calculus is said to be parametric if all of its values are parametric, so the only value in the model that has the type $\forall a.\, a \to a$, for example, is the identity function. In other words, if f is a function from a to b, and g is a value of type $\forall a.\, a \to a$, then a parametric model guarantees that $f \circ g = g \circ f$. More generally, each polymorphic type gives rise to a *parametricity law*, an equation satisfied by any value expressible as a λ-term and any value in a parametric model (Reynolds 1983; Wadler 1989, 2004). As we will see below, useful parametricity laws tend to arise from higher-rank polymorphism.

2 Higher-rank polymorphism

Extensions to Hindley, Milner, and Damas's system allow *higher-rank polymorphism*: types with rank higher than 1. Type inference for rank-2 polymorphism is decidable without any help in the form of additional type annotations by the programmer, but is too complicated to be implemented so far (Kfoury and Tiuryn 1992; Kfoury and Wells 1994). Type inference for arbitrarily higher-rank polymorphism requires programmer annotations in order to be decidable. Several such systems have been proposed and implemented in production compilers such as the Glasgow Haskell Compiler (Jones 1997; Odersky and Läufer 1996; Peyton Jones and Shields 2004; Le Botlan and Rémy 2003).

2.1 Algebraic data types

Polymorphic functions can encode recursive data types (Reynolds 1983; Reynolds and Plotkin 1993; Böhm and Berarducci 1985). For example, a singly-linked list of integers is defined in Haskell as

$$\text{data } \mathit{IntegerList} = \mathit{Nil} \mid \mathit{Cons\ Integer\ IntegerList},$$

but we can encode this recursive data type as the polymorphic function type

$$\mathit{IntegerList} \equiv \forall l.\, l \to (\mathit{Integer} \to l \to l) \to l.$$

Informally speaking, a list of integers is equivalent to a parametrically polymorphic function that maps a pair of list constructors (one for the empty list and one for non-empty lists) to a list, for any (abstract) list data type.

2.2 Lazy functional state threads

The *runST* function, for running a stateful computation in a purely functional language like Haskell (Launchbury and Peyton Jones 1994, 1995; Moggi and Sabry 2001), has the rank-2 type

$$\forall a.\, (\forall s.\, ST\ s\ a) \to a.$$

The type variable s is a dummy that identifies the state thread being run: two operations that are performed within the same state thread automatically have their s variables unified by the type checker. The type for *runST* thus ensures that state does not leak or interfere with other stateful computations. The proof of this safety property uses parametricity at the type $\forall s.\, ST\ s\ a$.

2.3 Generic (polytypic) programming

A type a is a "traversable term" type if any traversal on the subterms of a can be "lifted" to a traversal on a itself.

For instance, if we have a traversal on list elements, then we have a traversal on lists:

$$map :: \forall a.\, (a \to a) \to ([a] \to [a])$$

This idea of lifting traversals from subterms to terms can be combined with a moderate amount of run-time type-safe casting to achieve a form of generic programming (Lämmel and Peyton Jones 2003). The traversable types *a* belong to a type class *Term*, which supports operations such as

$$\begin{aligned} gmapT :: (\forall b.\, Term\ b \Rightarrow b \to b) \to& \\ (\forall a.\, Term\ a \Rightarrow a \to a).& \end{aligned}$$

Note that this operation has a rank-2 type: the type variable *b* is universally quantified over (with a type-class constraint) to the left of an arrow.

If one wishes to avoid run-time type-casting, then one would need to manually write—or mechanically generate—a slightly different traversal function for every type or type constructor over which generic processing is desired. For example, just as one would write the *map* function above for the list type constructor, one would write a function of the rank-2 type

$$\forall f.\, (\forall a.\, (a \to a) \to (f\ a \to f\ a)) \to (Fix\ f \to Fix\ f)$$

for the type constructor *Fix*, defined by

$$\text{data } Fix\ f = Fix\ (f\ (Fix\ f)),$$

which has the second-order kind $(* \to *) \to *$. In short, it takes terms of higher-rank types to (generically) operate on types of higher-order kinds (Hinze 2000b).

2.4 Converting between monads

Yakeley (2003) mentions an application of higher-rank functions that is similar in shape to the above approaches to generic programming: converting between data types that are parameterized over a monad (and thus have higher-order kind). Suppose that we are writing an interpreter for an impure programming language, and have defined a type of values as follows.

$$\begin{aligned} \text{data } Value\ m =\ & Str\ String \\ |\ & Num\ Double \\ |\ & Fun\ ([Value\ m] \to m\ (Value\ m)) \\ |\ & Cmd\ ([Value\ m] \to m\ ()). \end{aligned}$$

Here *m* is the monad in which impure computations take place. If we need to move from one monad m_1 to another monad m_2, then we need a function to map *Value* m_1 to *Value* m_2. Such a function would have the following rank-2 (and higher-order polymorphic) type:

$$\forall m_1.\forall m_2.(\forall a.m_1\ a \to m_2\ a) \to (Value\ m_1 \to Value\ m_2).$$

The coproducts technique of monad combination (Lüth and Ghani 2002) can also be viewed as a type of second-order kind whose values are manipulated by functions of higher rank. Given two well-behaved monads m_1 and m_2, their coproduct *Plus* m_1 m_2 is a new monad, where the type constructor *Plus* has the second-order kind $(* \to *) \to (* \to *) \to (* \to *)$. Accompanying the type constructor *Plus* is a term combinator *coprod*, which has the rank-2 type

$$\begin{aligned} \forall m_1.\, \forall m_2.\, \forall n.\, (Monad\ m_1, Monad\ m_2, Monad\ n) \Rightarrow& \\ (\forall a.\, m_1\ a \to n\ a) \to (\forall a.\, m_2\ a \to n\ a) \to& \\ (\forall a.\, Plus\ m_1\ m_2\ a \to n\ a).& \end{aligned}$$

2.5 Deforestation

Define the list-producing function *build* by

$$build\ g = g\ (:)\ [],$$

and give it the rank-2 type

$$\forall a.\, (\forall b.\, (a \to b \to b) \to b \to b) \to [a].$$

Also, define the list-consuming function *reduce* by

$$\begin{aligned} reduce\ []\ k\ z &= z, \\ reduce\ (a : as)\ k\ z &= k\ a\ (reduce\ as\ k\ z). \end{aligned}$$

(This is a variant of the standard *foldr* function.) Then $reduce \circ build$ is equal to the identity function in a parametric model. This law can be used as a rewriting rule in the left-to-right direction to eliminate intermediate trees in an optimizing functional compiler, in other words, to deforest (Gill et al. 1993).

One way to understand the type of *build* above is to see *b* as an abstract data type that, in the body of each "builder" argument *g* passed to *build*, supports only the two operations $cons :: a \to b \to b$ and $nil :: b$. This encapsulation, enforced by *build*'s rank-2 type, then guarantees by parametricity that $reduce \circ build$ can be deforested away. This idea of locally providing data constructors and enforcing their abstractness using higher-rank types can be extended to eliminate intermediate data structures like repeated list concatenations (Voigtländer 2002).

2.6 Data type invariants

When the same type constructor is applied to multiple, distinct types in the same data type definition, as the monad argument m is applied to both $\mathit{Value}\ m$ and $()$ in the definition of *Value* above, processing the resulting data type often calls for rank-2 polymorphism.

The Haskell type system's flexibility in allowing the same type constructor to be applied to multiple types can be used to enforce an impressive variety of invariants on data types (Hinze 2001). For example (Okasaki 1999), if v is a functor (of kind $* \to *$) that maps a scalar type to a (fixed-length) vector type, then the type $\mathit{Square}\ v\ a$, defined by

$$\text{data } \mathit{Square}\ v\ a = \mathit{Square}\ (v\ (v\ a)),$$

is a matrix comprised of scalars of type a—and a *square* one at that. In order to look up an entry in such a square matrix, we can implement a function of the type

$$(\forall b.\ v\ b \to \mathit{Int} \to b) \to \mathit{Square}\ v\ a \to \mathit{Int} \to \mathit{Int} \to a,$$

which has rank 2 because the first argument to the function is a vector-indexing function that is applied at the type $b = a$ as well as the type $b = v\ a$.

For another example (Bird and Paterson 1999), λ-terms can be represented, and required in the static type system to be closed (that is, to have no free variable), by the type

$$\begin{aligned} \text{data } \mathit{Term}\ v = {} & \mathit{Var}\ v \\ \mid\ & \mathit{App}\ (\mathit{Term}\ v)\ (\mathit{Term}\ v) \\ \mid\ & \mathit{Lam}\ (\mathit{Term}\ (\mathit{Maybe}\ v)). \end{aligned}$$

This type is said to be *nested*, or *non-regular*, because the constructor *Term* being defined is applied to not (just) v but (also) $\mathit{Maybe}\ v$. To process λ-terms represented thus, it is useful to define a generalized fold function *gfold*, analogous to the *reduce* function defined above for lists. In order to be defined, the *gfold* function must have a rank-2 type like

$$\begin{aligned} &\forall n.\ \forall b.\ (\forall a.\ a \to n\ a) \to (\forall a.\ n\ a \to n\ a \to n\ a) \to \\ &\qquad (\forall a.\ n\ (\mathit{Maybe}\ a) \to n\ a) \to \mathit{Term}\ b \to n\ b. \end{aligned}$$

Here the type variable n quantifies universally over any type constructor of kind $* \to *$.

3 Existential types

Existential types are type variables quantified over by an existential quantifier. For example, the type

$$\exists a.\ (a \to \mathit{Int}, \mathit{Int} \to a)$$

is inhabited by values such as

$$\begin{array}{cl} (\mathit{id}, \mathit{id}) & \text{for } a = \mathit{Int}, \\ (\mathit{succ}, \mathit{succ}) & \text{for } a = \mathit{Int}, \\ (\mathit{ord}, \mathit{chr}) & \text{for } a = \mathit{Char}, \\ (\mathit{const}\ 5, \mathit{const}\ \mathit{True}) & \text{for } a = \mathit{Bool}, \end{array}$$

and so on. Because an existentially quantified type variable can be instantiated by any type when an existentially typed value is constructed, the consumer of such a value has no information about the type except what is specified under the scope of the existential quantification. For example, given a value of the above type, the only valid operations on the opaque type a are to map *Int*s to and from a. Existential types are thus a natural way to encode and enforce abstraction boundaries in the type system.

To be more concrete, an existentially typed value like those above are produced using a rule like

$$\frac{E : [\tau'/a]\tau}{(\langle \tau', E\rangle : \exists a.\ \tau) : \exists a.\ \tau}\ \exists\text{I}, \tag{5}$$

known as packing or $\exists$-introduction. An existentially typed value is consumed using a rule like

$$\frac{E : \exists a.\ \tau \qquad \begin{array}{c}[x : \tau] \\ \vdots \\ E' : \tau'\end{array}}{\text{let } \langle a, x\rangle = E \text{ in } E' : \tau'}\ \exists\text{E}^x\ (a \text{ does not escape}), \tag{6}$$

known as unpacking or $\exists$-elimination. For a to not escape in the above rule is for it to not appear free in τ' or an undischarged typing assumption.

Existential types can be expressed in terms of higher-rank polymorphism. This reduction proceeds in two steps. First, a function that consumes an existentially typed value, say of type $(\exists a.\ \tau) \to \tau'$ where τ may mention a, is equivalent to a function that is polymorphic in its input type, in other words of type $\forall a.\ (\tau \to \tau')$. This equivalence can be understood computationally: if a function can operate on the type τ without knowing what type a is, then the function must be able to operate on any type a, and vice versa. The same equivalence can also be understood logically, as one between the formulas $(\exists x.\ P(x)) \to Q$ and $\forall x.\ (P(x) \to Q)$.

The second step of the reduction is a continuation-passing-style transform: in a parametric model, any type σ is equivalent to the type $\forall b.\ (\sigma \to b) \to b$. (To convert from σ, map v to $\Lambda b.\ \lambda k{:}\sigma \to b.\ kv$; to convert to σ, map m to $m\ \sigma\ \mathit{id}$.) If we let σ be $\exists a.\ \tau$ and τ' be b, then we get the equivalence

$$\exists a.\tau\ \equiv\ \forall b.((\exists a.\tau) \to b) \to b\ \equiv\ \forall b.(\forall a.(\tau \to b)) \to b.$$

Via this translation, programs that use existential types can be transformed to use higher-rank polymorphism instead, and compilers that handle higher-rank polymorphism can be extended to handle existential types as well.

Given this translation, we will not distinguish in the sequel between applications of existential types and "existential applications" of higher-rank polymorphism. Indeed, lazy functional state threads, an application of higher-rank types from §2.2, can be achieved with existential types too.

3.1 Abstract data types

Existential types can be used as a poor man's module facility, encapsulating an abstract data type along with an access interface comprised of operations on the type (Mitchell and Plotkin 1988). For example, an abstract data type for queues, say of the element type *Task*, might be expressed as the type

$$\exists a.\,(a, a \to \mathit{Task} \to a, a \to \mathit{Maybe}\ (\mathit{Task}, a)).$$

The three elements of the tuple are, respectively: the empty queue; a function to append a *Task* to a queue; and a function to remove a *Task* from the front of a queue, if any. Because the queue type a is existentially quantified over, implementation details such as the internal representation of queues are hidden from clients. Multiple implementations of the same abstract data type—in other words, multiple values of the same existential type above—can be introduced and used in the same program; the type system prevents queues from one implementation from being operated on by another.

3.2 Object orientation

In the terminology of object-oriented programming, to encapsulate an object is to combine its state and behavior into one unit whose internal composition is protected from external prying. For example, a one-dimensional point might be encapsulated as the type

$$\exists a.\,(a, a \to \mathit{Double}, a \to \mathit{Double} \to a).$$

Here the type a represents the state of the point, and *Double* is the type of the point's coordinate. The three elements of the tuple are, respectively: the current state of the point, a function to query the point for its coordinate, and a function to move the point. The latter two elements of the tuple are methods operating on the point; as such, they are defined once for each point class rather than each point object.

Heterogeneous values of existential type can be stored in lists and other data structures (Läufer 1996). Furthermore, given a type system that supports subtyping—as with type class constraints in Haskell (Wadler and Blott 1989; Hall et al. 1996), or extensible records in some Haskell extensions (Gaster and Jones 1996; Jones and Peyton Jones 1999)—we can create functions that are polymorphic in the object-oriented sense, that is, functions that can act on objects of different concrete implementation types as long as they support certain required interfaces. Finally, we can implement a variety of inheritance mechanisms in terms of higher-order kinds and higher-rank types, hence displaying the essential differences among these mechanisms (Pierce and Turner 1994).

When an existential type is used to encapsulate an abstract data type as described in §3.1, a value of the type is deconstructed as soon as the consumer gets its hands on the value. This deconstruction pattern—in which an existentially typed value is opened up only once, at the top level of a program unit—is an idiom that corresponds to importing a module. By contrast, the encoding of object orientation in terms of existential types described in this section uses a dual deconstruction pattern in which an existentially typed value is opened up every time it is accessed, at the leaves of program expressions.

3.3 Programs from algebraic specifications

Programs written in a functional style are often derived from specifications of abstract data types. A specification explains how values are constructed, mutated, and observed. Because (purely) functional programming languages support equational reasoning, a functional implementation of an abstract data type can often be derived from, and optimized based on, a set of algebraic laws that together determine the observable behavior of the type (Hughes 1995).

A monad m, for example, supports two ways to construct values of type $m\ a$: either from a value of type a, or from a value of type $m\ b$ and a value of type $b \to m\ a$.

$$\begin{aligned} \mathit{unit} &:: a \to m\ a \\ \mathit{bind} &:: m\ b \to (b \to m\ a) \to m\ a \end{aligned}$$

A simple implementation of a monad, then, is an algebraic data type in which *unit* and *bind* are simply data constructors.

$$\begin{aligned} &\text{data } M\ a = \mathit{Unit}\ a \mid \exists b.\ \mathit{Bind}\ (M\ b)\ (b \to M\ a) \\ &\quad \mathit{unit} = \mathit{Unit} \\ &\quad \mathit{bind} = \mathit{Bind} \end{aligned}$$

The data type M defined above naturally involves existential quantification over the type variable b. This implementation of a monad is of course not an optimized one, but similar principles (and algebraic calculations) lead to optimized implementations of monads for nondeterminism and backtracking (Hinze 2000a; Claessen 2002).

3.4 Data type invariants

To check code that consumes an existential type $\exists a.\ \tau$ using the $\exists$E rule in (6), the type checker treats a as an eigenvariable, just as when checking code that produces a universal type. That is, it generates a fresh dummy type and substitutes it for a. The dummy type is not allowed to leak out beyond the scope of the existential elimination construct, so the consumer can only use a in ways provided for in τ.

Fresh dummy type symbols can be used to enforce data type invariants at compile time. When these dummy types are used only at compile time and correspond to no run-time data, they are called *phantom types*. One use of phantom types that we have already seen is non-interference checking on lazy functional state threads, described in the first section. Another use is to make sure that code operating on red-black trees preserves invariants on sub-tree depth (Kahrs 2001). There, the dummy symbol is generated by existential quantification rather than rank-2 polymorphism, and represents not a stateful computation thread but the root of a binary tree.

3.5 Containing static type-checking

As shown by the applications above, skillful use of sexy types can often turn what is usually regarded as a run-time invariant into a compile-time check. To implement such checks is to reify dynamic properties of values as refined distinctions between types. These distinctions in turn increase the degree of heterogeneity among types in the program.

For example, to ensure that a database front-end program generates only well-typed queries for the back-end, Yakeley (2003) uses static types to distinguish between relational table columns containing integers, strings, and dates. With such a distinction in place, a homogeneous list of type [*Column*] can no longer store a list of table columns, because each column may contain a different type of data, and *Column Integer* cannot be placed in the same list as *Column String*. What can be placed in the same list are values of the type $\exists a.\ \mathit{Column}\ a$. Thus the Haskell code that generates SQL code can take as input a list of type [*Any Column*], where *Any* is defined by

$$\text{data } \mathit{Any}\ f = \exists a.\ \mathit{Any}\ (f\ a),$$

and is a type constructor of kind $(* \to *) \to *$. Similarly, Yakeley's Scheme interpreter in Haskell uses existential quantification to abstract over the number of mutually recursive values defined in a letrec binding form.

A special case of using existential quantification to contain the heterogeneity of static types is to dynamically type and cast arbitrary Haskell values. Haskell's type system turns out to be sexy enough to express dynamic typing without additional language support (Baars and Swierstra 2002).

In programs that interact with less predictable aspects of the real world, it is impossible to guarantee that the human user will never enter a letter when asked to input a number, that the database client will always send a well-formed query, that the comma-delimited data file will always contain the same number of fields on every line, and so on. In such situations, existential types let us take advantage of static checking when possible, but defer to run time if necessary (Kiselyov 2003).

3.6 Dynamic type-class instances

It is often useful to create type class instances dynamically in Haskell, but it is not obvious how. For example, in order to conduct integer arithmetic modulo 683 and 359, we might define instances like

$$\begin{aligned} &\text{instance } \mathit{Integral}\ a \Rightarrow \mathit{Num}\ (\mathit{Mod683}\ a) \text{ where} \ldots \\ &\text{instance } \mathit{Integral}\ a \Rightarrow \mathit{Num}\ (\mathit{Mod359}\ a) \text{ where} \ldots \end{aligned}$$

so that the Haskell expression

$$100 \times 100 + 200 \times 200 :: \mathit{Mod683\ Int}$$

computes

$$\mathit{mod}\ (\mathit{mod}\ (100 \times 100)\ 683 + \mathit{mod}\ (200 \times 200)\ 683)\ 683.$$

However, if the moduli to use only become known at run time (for example in a cryptography server), then we need to dynamically manufacture values of the existential type

$$\exists a.\ \mathit{Num}\ a \Rightarrow a \to \mathit{Int}$$

from these moduli. Haskell provides no special language support for such dynamic (or local-scope) creation of type-class instances, but it can be achieved by encoding values as types (Thurston 2001; Kiselyov and Shan 2004).

4 Acknowledgements

This article was originally prepared for Norman Ramsey's seminar on advanced functional programming at Harvard University in the fall of 2003. Thanks to Stefan Kahrs, Oleg Kiselyov, Gregory Morrisett, Simon L. Peyton Jones, Josef Svenningsson, Janis Voigtländer, Philip L. Wadler, Carl Witty, and Ashley Yakeley for responding to our original query and helpful comments on draft versions. This work is supported by the United States National Science Foundation under Grant BCS-0236592.

References

Baars, Arthur I., and S. Doaitse Swierstra. 2002. Typing dynamic typing. In ICFP (2002), 157–166.

Bird, Richard, and Ross Paterson. 1999. de Bruijn notation as a nested datatype. *Journal of Functional Programming* 9(1): 77–91.

Böhm, Corrado, and Alessandro Berarducci. 1985. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science* 39:135–154.

Church, Alonzo. 1932. A set of postulates for the foundation of logic. *Annals of Mathematics* II.33(2):346–366.

———. 1940. A formulation of the simple theory of types. *Journal of Symbolic Logic* 5(2):56–68.

Claessen, Koen. 2002. Parallel parsing processes. Submitted to the *Journal of Functional Programming*. http://www.math.chalmers.se/~koen/Papers/parsing-pearl.ps.

Clément, Dominique, Thierry Despeyroux, Gilles Kahn, and Joëlle Despeyroux. 1986. A simple applicative language: Mini-ML. In *Proceedings of the 1986 ACM conference on Lisp and functional programming*, 13–27. New York: ACM Press.

Damas, Luis, and Robin Milner. 1982. Principal type-schemes for functional programs. In *POPL '82: Conference record of the annual ACM symposium on principles of programming languages*, 207–212. New York: ACM Press.

Gaster, Benedict R., and Mark P. Jones. 1996. A polymorphic type system for extensible records and variants. Tech. Rep. NOTTCS-TR-96-3, School of Computer Science and Information Technology, University of Nottingham.

Gill, Andrew, John Launchbury, and Simon L. Peyton Jones. 1993. A short cut to deforestation. In *Functional programming languages and computer architecture: 6th conference*, 223–232. New York: ACM Press.

Girard, Jean-Yves. 1972. Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur. Thèse de doctorat d'état, Université Paris VII.

Girard, Jean-Yves, Paul Taylor, and Yves Lafont. 1989. *Proofs and types*. Cambridge: Cambridge University Press.

Hall, Cordelia V., Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems* 18(2):109–138.

Hindley, J. Roger. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society* 146:29–60.

Hinze, Ralf. 2000a. Deriving backtracking monad transformers. In *ICFP '00: Proceedings of the ACM international conference on functional programming*, vol. 35(9) of *ACM SIGPLAN Notices*, 186–197. New York: ACM Press.

———. 2000b. A new approach to generic functional programming. In *POPL '00: Conference record of the annual ACM symposium on principles of programming languages*, 119–132. New York: ACM Press.

———. 2001. Manufacturing datatypes. *Journal of Functional Programming* 11(5):493–524.

Hughes, John. 1995. The design of a pretty-printing library. In *Advanced functional programming: 1st international spring school on advanced functional programming techniques*, ed. Johan Jeuring and Erik Meijer, 53–96. Lecture Notes in Computer Science 925, Berlin: Springer-Verlag.

ICFP. 2002. *ICFP '02: Proceedings of the ACM international conference on functional programming*. New York: ACM Press.

Jones, Mark P. 1997. First-class polymorphism with type inference. In *POPL '97: Conference record of the annual ACM symposium on principles of programming languages*, 483–496. New York: ACM Press.

Jones, Mark P., and Simon L. Peyton Jones. 1999. Lightweight extensible records for Haskell. In *Proceedings of the 1999 Haskell workshop*, ed. Erik Meijer. Tech. Rep. UU-CS-1999-28, Department of Computer Science, Utrecht University.

Kahrs, Stefan. 2001. Red-black trees with types. *Journal of Functional Programming* 11(4):425–432.

Kfoury, Assaf J., and Jerzy Tiuryn. 1992. Type reconstruction in finite rank fragments of the second-order λ-calculus. *Information and Computation* 98(2):228–257.

Kfoury, Assaf J., and Joe B. Wells. 1994. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *Proceedings of the 1994 ACM conference* *on Lisp and functional programming*, 196–207. New York: ACM Press.

Kiselyov, Oleg. 2003. Polymorphic stanamically balanced AVL trees. `http://okmij.org/ftp/Haskell/types.html`.

Kiselyov, Oleg, and Chung-chieh Shan. 2004. Implicit configuration—or, type classes reflect the value of types. `http://www.eecs.harvard.edu/~ccshan/prepose/`.

Lämmel, Ralf, and Simon L. Peyton Jones. 2003. Scrap your boilerplate: A practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on types in languages design and implementation*, 26–37. New York: ACM Press.

Läufer, Konstantin. 1996. Type classes with existential types. *Journal of Functional Programming* 6(3):485–517.

Launchbury, John, and Simon L. Peyton Jones. 1994. Lazy functional state threads. In *PLDI '94: Proceedings of the ACM conference on programming language design and implementation*, vol. 29(6) of *ACM SIGPLAN Notices*, 24–35. New York: ACM Press.

———. 1995. State in Haskell. *Lisp and Symbolic Computation* 8(4):293–341.

Le Botlan, Didier, and Didier Rémy. 2003. ML^F: Raising ML to the power of System F. In *ICFP '03: Proceedings of the ACM international conference on functional programming*, 27–38. New York: ACM Press.

Lüth, Christoph, and Neil Ghani. 2002. Composing monads using coproducts. In ICFP (2002), 133–144.

Milner, Robin. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* 17: 348–375.

Mitchell, John C., and Gordon D. Plotkin. 1988. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems* 10(3):470–502.

Moggi, Eugenio, and Amr Sabry. 2001. Monadic encapsulation of effects: A revised approach (extended version). *Journal of Functional Programming* 11(6):591–627.

Odersky, Martin, and Konstantin Läufer. 1996. Putting type annotations to work. In *POPL '96: Conference record of the annual ACM symposium on principles of programming languages*, 54–67. New York: ACM Press.

Okasaki, Chris. 1999. From fast exponentiation to square matrices: An adventure in types. In *ICFP '99: Proceedings of the ACM international conference on functional programming*, vol. 34(9) of *ACM SIGPLAN Notices*, 28–35. New York: ACM Press.

Peyton Jones, Simon L. 2003. Wearing the hair shirt: A retrospective on Haskell. Invited talk at POPL 2003.

Peyton Jones, Simon L., and Mark B. Shields. 2004. Practical type inference for arbitrary-rank types. Submitted to the *Journal of Functional Programming*. `http://research.microsoft.com/~simonpj/papers/putting/`.

Pierce, Benjamin C., and David N. Turner. 1994. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming* 4(2):207–247.

Reynolds, John C. 1974. Towards a theory of type structure. In *Programming symposium: Proceedings, colloque sur la programmation*, ed. Bernard Robinet, 408–425. Lecture Notes in Computer Science 19, Berlin: Springer-Verlag.

———. 1983. Types, abstraction and parametric polymorphism. In *Information processing 83: Proceedings of the IFIP 9th world computer congress*, ed. R. E. A. Mason, 513–523. Amsterdam: Elsevier Science.

———. 1990. Introduction to part II, polymorphic lambda calculus. In *Logical foundations of functional programming: Proceedings of the Year of Programming Institute*, ed. Gérard Huet, 77–86. Boston: Addison-Wesley.

Reynolds, John C., and Gordon D. Plotkin. 1993. On functors expressible in the polymorphic typed lambda calculus. *Information and Computation* 105(1):1–29.

Strachey, Christopher. 1967. Fundamental concepts in programming languages. Lecture notes for the International Summer School in Computer Programming. Also as *Higher-Order and Symbolic Computation* 13(1–2):11–49, 2000.

Thurston, Dylan. 2001. Modular arithmetic. Messages to the Haskell mailing list; `http://www.haskell.org/pipermail/haskell-cafe/2001-August/002132.html`; `http://www.haskell.org/pipermail/haskell-cafe/2001-August/002133.html`.

Voigtländer, Janis. 2002. Concatenate, reverse and map vanish for free. In ICFP (2002), 14–25.

Wadler, Philip L. 1989. Theorems for free! In *FPCA '89: 4th international conference on functional programming languages and computer architecture*, 347–359. New York: ACM Press.

———. 2004. The Girard-Reynolds isomorphism (second edition). `http://homepages.inf.ed.ac.uk/wadler/papers/gr2/gr2.pdf`.

Wadler, Philip L., and Stephen Blott. 1989. How to make *ad-hoc* polymorphism less *ad hoc*. In *POPL '89: Conference record of the annual ACM symposium on principles of programming languages*, 60–76. New York: ACM Press.

Wells, Joe B. 1999. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic* 98(1–3):111–156.

Yakeley, Ashley. 2003. Where are higher-rank and existential types used? Message to the Haskell mailing list; `http://haskell.org/pipermail/haskell/2003-September/012670.html`.

Chung-chieh Shan is a computer science PhD candidate studying computational linguistics at Harvard University.