Programming with Static Invariants in Ωmega

Nathan Linger & Tim Sheard

September 17, 2004

Abstract

We present an implementation of binomial heaps that enforces the balancing invariants at compile time. We present an implementation of a *typed* CPS transform that enforces the typing invariants of both the source and target terms.

1 Introduction

There has been a lot of interest recently in generalizing algebraic datatypes as found in most functional programming languages. These generalizations have been called *First Class Phantom Types* [1] *Guarded ADTs* [13, 12] *Wobbly types* [9], and in our own work *Equality Qualified Types* [**?**, 10, 11]. Such types allow users to parameterize types with descriptions of static properties of their values. A simple example is the type constructor *Seq* which describes sequences of elements whose lengths are statically known. For example $[1, 45, 2]::\mathit{Seq}\ 3\ \mathit{Int}$. This says that $[1, 45, 2]$ is a sequence of *Int* whose length is statically known to be 3. A type system employing such a generalization can enforce strong static invariants. This leads to programs that are less likely to contain errors, and are easier to write, since the types often guide the construction of the function definitions.

In order to explore these possibilites we implemented a new language Ωmega. Ωmega is descended from Haskell – Its syntax and type system are similar to Haskell's, but it introduces a new kind of type constraint (similar to Haskell's class constraints) which constrains two types to be equal. A function with an equality constrained type, can only be applied to values whose types force the constrained types to be the same. This has been described elsewhere [**?**, 10, 11], and is introduced again for the convenience of the reader in Section 2.

This paper introduces several other features of Ωmega that work in combination with equality qualified types (and could also work in combination with any of the other variants of generalized datatypes) to allow even more expressive static invariants. These features are user-defined kinds and support for functions on types. The contributions of this paper are:

- Demonstrating the usefulness of an extensible kind system. Introducing a new kind, defines a kind and new types that are classified by that kind.

The new kind is different from the kind that classifies types that classify values (often called *star* or *type*). Types introduced by the kind mechanism make ideal *parameters* to equality qualified data types, and can be used to express a wide variety of static constraints.

- The ability to define type functions, and the ability to use such type functions in the types of functions manipulating values with equality constrained types. Type functions (as opposed to type constructors such as *List*, and *Tree*) describe computations on types. There use is reminiscent of the use of dependent types (see Section **??**) but we claim that Ωmega's type system implements a simple subset of the dependent types that is simple to use, understand, and implement. Type functions require a new approach to unification when integrated into a system with type inference. The system of equality constrained types provides just the hooks necessary to do this in a safe and sound manner.

what is meant by "safe and sound"?

- We also present two examples in Ωmega which its power to soundly define functions with complex static invariants. The two examples are (1) an implementation of binomial heaps and (2) a typed continuation-passing-style transform.

2 Equality Qualified Types and Kinds in Ωmega

We introduce equality qualified types by defining the type constructor *Seq* from above. In order to do this we must also be able to describe natural numbers at the type level. To do this we use Ωmega's kind system. The code below introduces a new kind *Nat*, new types *Seq*, *Zero*, and *Succ*, and a new function *map*.

$$\mathbf{kind}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}$$

$$\begin{array}{ll}
\mathbf{data}\ \mathit{Seq}\ a\ n & \\
\quad = \quad \mathit{Nil} & \mathbf{where}\ n = \mathit{Zero} \\
\quad \mid \exists\, m.\mathit{Cons}\ a\ (\mathit{Seq}\ a\ m) & \mathbf{where}\ n = \mathit{Succ}\ m
\end{array}$$

$$\begin{array}{l}
\mathit{map} :: (a \to b) \to \mathit{Seq}\ a\ n \to \mathit{Seq}\ b\ n \\
\mathit{map}\ f\ (\mathit{Nil}) \qquad\quad\ = \mathit{Nil} \\
\mathit{map}\ f\ (\mathit{Cons}\ x\ \mathit{xs}) = \mathit{Cons}\ (f\ x)\ (\mathit{map}\ f\ \mathit{xs})
\end{array}$$

The kind declaration for *Nat* introduces two new type constructors *Zero* and *Succ* which encode the natural numbers at the type level. Kinds are similar to types in that, while types classify values, kinds classify types. We indicate this by the *classifies* relation (::). For example: $5 :: \mathit{Int} :: *$. We say 5 is classified by *Int*, and *Int* is classified by $*$ (star-zero). Figure 1 illustrates the relationship between the values, types, and kinds introduced above.

Constructor functions *Nil* and *Cons* construct elements of the *Seq* datatype. The type of a constructor function is described in the datatype declaration. For example, the following clause in the *Seq* declaration:

value		type		kind
5	::	Int	::	*
		Zero	::	Nat
		Succ	::	Nat $\rightsquigarrow$ Nat
		Seq	::	* $\rightsquigarrow$ Nat $\rightsquigarrow$ *
Nil	::	Seq α Zero	::	*
Cons	::	$\alpha \rightarrow$ Seq α $n \rightarrow$ Seq α (Succ n)	::	*

Figure 1: Classification of values (Nil,Cons), and types (Zero,Succ, and Seq).

$\exists\ m.Cons\ a\ (Seq\ a\ m)$ **where** $n = Succ\ m$

introduces the *Cons* constructor function. Without the **where** qualification, the constructor function *Cons* would have type $Cons{::}a \rightarrow Seq\ a\ m \rightarrow Seq\ a\ n$. The **where** *qualifies* *Cons*'s type, in effect saying ($Cons :: a \rightarrow Seq\ a\ m \rightarrow Seq\ a\ n$) provided $n = Succ\ m$. We capture this formally by writing the equality qualified type $Cons{::}(\forall\ a\ n\ m.(n = Succ\ m) \Rightarrow a \rightarrow Seq\ a\ m \rightarrow Seq\ a\ n)$. The equations behind the *fat arrow* ($\Rightarrow$) are equality qualifications. Since n is a universally quantified type variable, there is only one way to *solve* the qualification $n = Succ\ m$ (by making n equal to $Succ\ m$). Because of this unique solution, *Cons* also has the (less general) type $(\forall\ a\ m.a \rightarrow Seq\ a\ m \rightarrow Seq\ a\ (S\ m))$. This type guarantees that *Cons* can only be applied in contexts where $n = Succ\ m$.

Each data-constructor declaration in Ωmega may contain a **where** clause containing a list of type equations. The mechanism Ωmega uses to keep track of type equalities is similar to type class system of Haskell. A special qualified [6] type is used to assert equality between types, and a constraint solving system is used to track, propogate, simplify and discharge these assertions. Equality constraints are tracked in much the same manner as class constraints in Haskell. For example, when assigning a type to a type constructor, the equations specified in the **where** clause just become predicates in a qualified type. Thus, the constructor *Cons* is given the type $(\forall\ a\ m\ n.(n = Succ\ m) \Rightarrow a \rightarrow Seq\ a\ m \rightarrow Seq\ a\ n)$ The equation $n = Succ\ m$ is just another form of predicate, similar to the class membership predicate ($Ord\ a$) in the Haskell type $(\equiv) :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Bool$).

Tracking equality constraints. When type-checking an expression, the Ωmega type checker keeps two sets of equality constraints: *obligations* and *assumptions*.

Obligations. The first set of constraints is a set of *obligations*. Obligations are generated by the type-checker when (a) the program constructs data-values with constructors that contain equality constraints; or (b) an explicit type signature in a definition is encountered; or (c) when type functions are used (see Section 3).

For example, consider type-checking the expression *Nil*. The constructor *Nil* is assigned the type $\forall\ a\ n.n = Zero \Rightarrow Seq\ a\ n$, thus $Nil :: Seq\ a\ n$ but also generates the obligation $n = Zero$. Since *Nil* is polymorphic in a and n, the type variable n can be instantiated to $Zero$. Instantiating n to $Zero$ makes the equality constraint obligation $Zero = Zero$, which can be trivially discharged by the type checker. Other constraints may require more work to discharge.

Assumptions. The second set of constraints is a set of *assumptions* or *facts*. Whenever, a constructor based pattern appears in a binding position, and the constructor was defined using a **where** clause, the type equalities in the **where** clause are added to the current set of assumptions in the scope of the pattern. These assumptions can be used to discharge obligations. For example, consider type checking the definition of map. Recall $map :: (a \to b) \to Seq\ a\ n \to Seq\ b\ n$.

In the first equation, the pattern Nil of type $Seq\ a\ n$ introduces the assumption $n = Zero$ because of Nil's qualified type $\forall\ a\ n.n = Zero \Rightarrow Seq\ a\ n$ This assumption can be used to discharge obligations incurred on the right hand side of the equation. Now, consider the second equation $map\ f\ (Cons\ x\ s) = Cons\ (f\ x)\ (map\ f\ xs)$. We know map's type, so the second pattern $(Cons\ x\ xs)$ must have type $(Seq\ a\ n)$. This implies $x :: a$ and $xs :: Seq\ a\ m$ provided $n = Succ\ m$. This equation is added to the list of assumptions when typing the right-hand-side of the second equation: $Cons\ (f\ x)\ (map\ f\ xs)$. This right-hand-side should have type $(Seq\ b\ n)$ (the range of map's type). Since $xs :: Seq\ a\ m$ we can compute $(map\ f\ xs) :: Seq\ b\ m$, Since $x :: a$ we can compute $(f\ x) :: b$, thus it appears $(Cons\ (f\ x)\ (map\ f\ xs)) :: Seq\ b\ (Succ\ m)$. But given the equality constraint $n = Succ\ m$ the right hand side has type $Seq\ b\ n$ as expected. Equality qualified types can encode powerful static invariants about data. For example the type of map provides a proof that map returns a list of the same length as its input list.

3 Type functions in Ωmega

We now introduce type-functions. To see why this is necessary we ask the rhetorical question: *What is the type of the append function on static-length lists*? The append function should have type $Seq\ a\ n \to Seq\ a\ m \to Seq\ a\ (n + m)$. In order to type such functions it is necessary to do arithmetic at the type level. This means defining functions over types. In Ωmega we do this by writing equations over types. We surround type-function application by braces (e.g. $\{sum\ x\ y\}$) to distinguish it from type-constructor application (e.g. $List\ Int$). We also define the type-function sum and the value-function $append$ of static sequences.

$$
\begin{array}{l}
sum :: Nat \leadsto Nat \leadsto Nat \\
\{sum\ Zero\ \quad\ m\} = m \\
\{sum\ (Succ\ n)\ m\} = Succ\{sum\ n\ m\} \\
append :: Seq\ a\ n \to Seq\ a\ m \to Seq\ a\{sum\ n\ m\} \\
append\ (Nil)\ \qquad\quad ys = ys \\
append\ (Cons\ x\ xs)\ ys = Cons\ x\ (append\ xs\ ys)
\end{array}
$$

How is a function, like append, whose type includes a type-function application, type checked? And, what role do the type function equations play in the process? Consider the second equation: $append\ (Cons\ x\ xs)\ ys =$

Cons x (*append* *xs* *ys*). From the type signature of *append*, we know (*Cons* x *xs*):: *Seq* a n, and *ys* :: *Seq* a m. We need to check that the right-hand-side *Cons* x (*append* *xs* *ys*) has type *Seq* a\{ *sum* n m\}. Because *Cons* has an equality qualified type, from the pattern (*Cons* x *xs*) we know (x :: a) and (*xs* :: *Seq* a k) plus we get to add the equality (n = *Succ* k) to the set of assumptions. From this we can compute that (*append* *xs* *ys* :: *Seq* a\{ *sum* k m\}). Thus it appears that the right- hand-side *Cons* x (*append* *xs* *ys*) actually has type *Seq* a (*Succ*\{ *sum* k m\}). To reconcile this apparent difference, the type checking algorithm unifies the type *Seq* a\{ *sum* n m\} with the type *Seq* a (*Succ*\{ *sum* k m\}). This unification succeeds only if the equation \{ *sum* n m\} = *Succ*\{ *sum* k m\} holds. Since this cannot always be immediately determined, the unification algorithm delays unification, and generates and propagates this equation as a new obligation. After checking the rest of the right-hand-side (which propagates no additional obligations) we must discharge the obligation \{ *sum* n m\} = *Succ*\{ *sum* k m\} under the assumption (n = *Succ* k). We use the assumption by substituting *Succ* k for n, and then discharge \{ *sum* (*Succ* k) m\} = *Succ*\{ *sum* k m\}. By applying the second clause in the definition of *sum* we get *Succ*\{ *sum* k m\} = *Succ*\{ *sum* k m\} which is finally discharged.

By delaying the use of both assumptions and type-function equations until after all other unification is performed, we avoid the problem that sometimes occurs when type variables are not instantiated in time. For example without knowing that n = *Succ* k we could not apply the type-function rule reducing *sum* n m to *Succ*\{ *sum* k m\}.

4 Binomial Heaps

Binomial Heaps are a collection type that support the following operations, each listed with its time complexity.

```
empty      :: Heap                    -- O(1)
singleton  :: Elem → Heap             -- O(1)
insert     :: Elem → Heap → Heap      -- O(lg n)
merge      :: Heap → Heap → Heap      -- O(lg n)
minElem    :: Heap → Maybe Elem       -- O(lg n)
extractMin :: Heap → Maybe Heap       -- O(lg n)
```

That every operation takes logortihmic (or better) time is due to a sophisticated set of invariants that structure the binomial heap. Correctly implementing heap operations that maintain these invariants is a non-trivial task. Any help in this endeavor is greatly appreciated. In Ωmega we can express almost all the invariants and let the type checker enforce them.

The *Heap* type should really be parameterized by the type of its elements. For simplicity of exposition, we just treat *Elem* as an abstract type that supports comparison operations. The desired parameterization poses no problems but clutters the presentation of the static properties.

4.1 Natural Numbers as Types

The datatypes we use to encode binomial heaps are indexed by natural numbers. These parameters encode structural invariants for each type. Since we do not use dependent types, we must develop *types* whose structure mirrors that of the naturals. For our purposes here, two representations of numbers are useful. The natural numbers (*N1*) in unary, and the positive natural numbers (*N2*) in binary.

$$\begin{array}{ll} \textbf{kind}\ \mathbb{N}_1 = \mathit{Zero} \mid \mathit{Succ}\ \mathbb{N}_1 & \text{-- } 0 \text{ or } n+1 \\ \textbf{kind}\ \mathbb{N}_2 = \mathit{One} \mid \mathit{Even}\ \mathbb{N}_2 \mid \mathit{Odd}\ \mathbb{N}_2 & \text{-- } 1 \text{ or } 2n \text{ or } 2n+1 \end{array}$$

The binary representation is least-significant-bit first. For example, the type $\mathit{Odd}\ (\mathit{Even}\ (\mathit{Odd}\ \mathit{One}))$ stands for the number $1101_2 = 13_{10}$.

We encode only positive naturals in the binary representation. Instead we could include zero by defining $\textbf{kind}\ \mathbb{N}_2 = \mathit{Zero2} \mid \mathit{Even}\ \mathbb{N}_2 \mid \mathit{Odd}\ \mathbb{N}_2$. The problem with this definition is that it does not rule out trailing zeroes. For example *Even Zero2* and *Zero2* both stand for the number 0.

We define increment and addition type-functions on the binary representation of the naturals.

$$\begin{array}{l} \mathit{inc} :: \mathbb{N}_2 \rightsquigarrow \mathbb{N}_2 \\ \{\mathit{inc}\ \mathit{One}\} \qquad\quad = \mathit{Even}\ \mathit{One} \\ \{\mathit{inc}\ (\mathit{Even}\ n)\} = \mathit{Odd}\ n \\ \{\mathit{inc}\ (\mathit{Odd}\ n)\} \;\; = \mathit{Even}\{\mathit{inc}\ n\} \end{array}$$

Addition is defined as a pair of mutually recursive functions: *add* (without carry) and *addc* (with carry).

$$\begin{array}{llll} \multicolumn{4}{l}{\mathit{add} :: \mathbb{N}_2 \rightsquigarrow \mathbb{N}_2 \rightsquigarrow \mathbb{N}_2} \\ \{\mathit{add}\ \mathit{One} & m\} & = & \{\mathit{inc}\ m\} \\ \{\mathit{add}\ n & \mathit{One}\} & = & \{\mathit{inc}\ n\} \\ \{\mathit{add}\ (\mathit{Even}\ n) & (\mathit{Even}\ m)\} & = & \mathit{Even}\{\mathit{add}\ \ n\ m\} \\ \{\mathit{add}\ (\mathit{Even}\ n) & (\mathit{Odd}\ \ m)\} & = & \mathit{Odd}\ \{\mathit{add}\ \ n\ m\} \\ \{\mathit{add}\ (\mathit{Odd}\ \ n) & (\mathit{Even}\ m)\} & = & \mathit{Odd}\ \{\mathit{add}\ \ n\ m\} \\ \{\mathit{add}\ (\mathit{Odd}\ \ n) & (\mathit{Odd}\ \ m)\} & = & \mathit{Even}\{\mathit{addc}\ n\ m\} \end{array}$$

$$\begin{array}{llll} \multicolumn{4}{l}{\mathit{addc} :: \mathbb{N}_2 \rightsquigarrow \mathbb{N}_2 \rightsquigarrow \mathbb{N}_2} \\ \{\mathit{addc}\ \mathit{One} & m\} & = & \{\mathit{inc}\{\mathit{inc}\ m\}\} \\ \{\mathit{addc}\ n & \mathit{One}\} & = & \{\mathit{inc}\{\mathit{inc}\ n\}\} \\ \{\mathit{addc}\ (\mathit{Even}\ n) & (\mathit{Even}\ m)\} & = & \mathit{Odd}\ \{\mathit{add}\ \ n\ m\} \\ \{\mathit{addc}\ (\mathit{Even}\ n) & (\mathit{Odd}\ \ m)\} & = & \mathit{Even}\{\mathit{addc}\ n\ m\} \\ \{\mathit{addc}\ (\mathit{Odd}\ \ n) & (\mathit{Even}\ m)\} & = & \mathit{Even}\{\mathit{addc}\ n\ m\} \\ \{\mathit{addc}\ (\mathit{Odd}\ \ n) & (\mathit{Odd}\ \ m)\} & = & \mathit{Odd}\ \{\mathit{addc}\ n\ m\} \end{array}$$

4.2 Binomial Trees

Binomial heaps are composed of simpler structures called binomial trees, which we explain first. The datatype T_k of binomial trees is based on the arithmetic identity

$$2^k \quad = \quad 1 + (2^{k-1} + 2^{k-2} + \cdots + 2^0) \quad \text{where } k \geqslant 0$$

We also make use of an auxillary datatype F_k of forests (of tress) stored in descending order: $T_{k-1}, T_{k-2}, \ldots, T_0$.

$$
\begin{array}{lll}
\textbf{data}\ T_k = & \mathit{Tree}\ \mathit{Elem}\ F_k & \\
\textbf{data}\ F_k = & \mathit{FNil} & \textbf{where}\ k = \mathit{Zero} \\
 & \mid \exists\, j.\mathit{FCons}\ T_j\ F_j & \textbf{where}\ k = \mathit{Succ}\ j
\end{array}
$$

This definition defines type constructors T and F of kind $\mathbb{N}_1 \rightsquigarrow *$. The **where** clauses in this definition effectively give the data constructors the following types:

$$
\begin{array}{ll}
\mathit{Tree} & :: \mathit{Elem} \rightarrow F_k \rightarrow T_k \\
\mathit{FNil} & :: F_{\mathit{Zero}} \\
\mathit{FCons} & :: T_k \rightarrow F_k \rightarrow F_{\mathit{Succ}\ k}
\end{array}
$$

Trees of type T_k contain exactly 2^k elements. Forests of type F_k contain exactly $2^k - 1$ elements. Convince yourself of this by an inductive argument based on the types of the data constructors. Note that the boundary condition $2^0 = 1$ holds.

$$
\begin{array}{l}
\mathit{singletonT} :: \mathit{Elem} \rightarrow T_{\mathit{Zero}} \\
\mathit{singletonT}\ a = \mathit{Tree}\ a\ \mathit{FNil}
\end{array}
$$

Before we move to the inductive step we need to discuss the heap invariant. Binomial heaps satisfy two kinds of invariants. The first kind of invariant is static and structural, having to do with the relative shapes and sizes of the trees in the heap, and is captured in the type indexs of the data structures. This is what the inductive proof is about. The second is the *heap invariant*. Binomial heaps are composed of Binomial trees. The heap invariant states the every node in a binomial tree in a binomial heap has a value no greater than that of any its descendents. The heap invariant is captured by making the construction of binomial trees abstract. Singleton trees vacuously meet the heap invariant (they have only one element) and are made by using the abstract constructor ($\mathit{singletonT}$). Larger trees are constructed by an abstract constant-time merge function (mergeT) corresponding to the identity $2^k + 2^k = 2^{k+1}$. We construct this function in two steps. First we define mergeT':

$$
\begin{array}{l}
\mathit{mergeT}' :: T_k \rightarrow T_k \rightarrow T_{\mathit{Succ}\ k} \\
\mathit{mergeT}'\ (\mathit{Tree}\ e\ f)\ t = \mathit{Tree}\ e\ (\mathit{FCons}\ t\ f)
\end{array}
$$

The top element of $\mathit{mergeT}'\ t_1\ t_2$ is the top element of t_1. A simple wrapper function ensures that merging preserves the heap property by swapping t_1 with t_2 when necessary.

$$
\begin{array}{l}
\mathit{top} :: T_k \rightarrow \mathit{Elem} \\
\mathit{top}\ (\mathit{Tree}\ i\ \_) = i
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{mergeT} :: T_k \rightarrow T_k \rightarrow T_{\mathit{Succ}\ k}} \\
\mathit{mergeT}\ t_1\ t_2 & \mid \mathit{top}\ t_1 \leqslant \mathit{top}\ t_2 & = \mathit{mergeT}'\ t_1\ t_2 \\
 & \mid \mathit{otherwise} & = \mathit{mergeT}'\ t_2\ t_1
\end{array}
$$

So long as we only use $\mathit{singletonT}$ and mergeT to build trees, we will preserve the heap invariant. The function mergeT also preserves the structural invariant, which is manifest in its type. This completes the inductive step of the structural proof. You can't construct well typed binomial trees, that don't meet the structural invariant. There is one other function that needs to break the binomial tree abstraction (used to implement *extractMin*) but it also preserves the heap invariant.

4.3 Binomial Heaps

As we develop the arithmetic analogy, the binary representation of natural numbers is the basis for *binomial heaps*. Since there is a unique binary representation for any natural number n, there is a unique set of (distinct) binomial tree sizes that together contain n elements. For example, we can contain 13 elements in a collection of binomial heaps of types T_0, T_2, and T_3 because $13 = 2^0 + 2^2 + 2^3$. In this way, the structure of binomial heaps mirrors that of binary numbers.

Here is a first attempt at defining the *Heap* datatype.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Heap}\ n \\
\quad = \quad \mathit{Last}\ \ T_? \qquad\qquad\quad \textbf{where}\ n = \mathit{One} \\
\quad \mid \exists\, m.\mathit{Skip} \qquad (\mathit{Heap}\ n)\ \textbf{where}\ n = \mathit{Even}\ m \\
\quad \mid \exists\, m.\mathit{Cons}\ T_?\ (\mathit{Heap}\ n)\ \textbf{where}\ n = \mathit{Odd}\ m
\end{array}
$$

What's missing is an accumulating parameter to specify the orders of binomial trees. We arrive at the following definition.

$$
\begin{array}{l}
\textbf{data}\ \mathit{SubHeap}\ n\ k \\
\quad = \quad \mathit{Last}\ T_k \qquad\qquad\qquad\qquad\quad\ \textbf{where}\ n = \mathit{One} \\
\quad \mid \exists\, m.\mathit{Skip} \qquad (\mathit{SubHeap}\ m\ (\mathit{Succ}\ k))\ \textbf{where}\ n = \mathit{Even}\ m \\
\quad \mid \exists\, m.\mathit{Cons}\ T_k\ (\mathit{SubHeap}\ m\ (\mathit{Succ}\ k))\ \textbf{where}\ n = \mathit{Odd}\ m
\end{array}
$$

$$\textbf{data}\ \mathit{Heap} = \exists\, n.\mathit{Heap}\ (\mathit{SubHeap}\ n\ \mathit{Zero}) \mid \mathit{Empty}$$

A wrapper datatype *Heap* initializes the accumulating parameter at *Zero*. The wrapper type also hides the size parameter by binding it existentially. This shields users of the heap library from the details of its implementation.

The type constructor *SubHeap* has kind $\mathbb{N}_2 \leadsto \mathbb{N}_1 \leadsto *$. We use the unary representation to count off the orders of the binomial trees in a heap, and the binary representation to determine the shape of the subheap. Any $\mathit{SubHeap}\ n\ k$ value contains $n*2^k$ elements. By corollary, a *Heap* value with hidden existential parameter n contains n elements. Figure 2 shows an example heap.

4.3.1 Primitive Heaps

Here are primitive heaps containing 0 and 1 elements.

$$
\begin{array}{l}
\mathit{empty} :: \mathit{Heap} \\
\mathit{empty} = \mathit{Empty} \\[4pt]
\mathit{singleton} :: \mathit{Elem} \to \mathit{Heap} \\
\mathit{singleton}\ a = \mathit{Heap}\ (\mathit{Last}\ (\mathit{singletonT}\ \ a))
\end{array}
$$

4.3.2 Insertion

Inserting a binomial tree into a binomial (sub-) heap corresponds to incrementing a binary number. The tree to be inserted corresponds to the least significant "carry" bit that is being added on. This is why the k parameters of the sub-heap

```
SubHeap (
  Cons  (Tree 1 FNil) (
  Skip  (
  Last  (Tree 2
        (FCons (Tree 4
                   (FCons (Tree 5 FNil)
                   FNil))
        (FCons (Tree 3 FNil) FNil)))
    )))
```

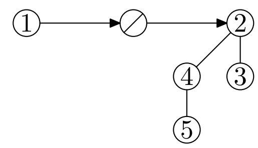

Figure 2: A binomial heap of size $5_{10} = 101_2$. It contains several sub-heaps. The *Cons* heap has type *SubHeap* (*Odd* (*Even One*)) *Zero*, the *Skip* heap has type *SubHeap* (*Even One*) (*Succ Zero*), and the *Last* heap has type *SubHeap One* (*Succ* (*Succ Zero*)). Translating the binary and unary representations in the types to decimal, we see *Cons* with type *Subheap* 5 0, *Skip* with type *Subheap* 2 1, and *Last* with type *Subheap* 1 2. Check for yourself that a heap with type *Subheap* n k has $n * 2^k$ elements.

and the tree have to match. We encourage the reader to compare this definition with the definition of *inc*.

$$
\begin{aligned}
&ins :: T_k \rightarrow SubHeap\ n\ k \rightarrow SubHeap\{inc\ n\}k\\
&ins\ t\ (Last\ x) \quad\ \ = Skip\ (Last\ (mergeT\ x\ t))\\
&ins\ t\ (Skip \quad\ h) = Cons\ t\ h\\
&ins\ t\ (Cons\ x\ h) = Skip\ (ins\ (mergeT\ x\ t)\ h)
\end{aligned}
$$

At top level, we call a wrapper function that inserts a single element into a *Heap* as a tree of size 2^0.

$$
\begin{aligned}
&insert :: Elem \rightarrow Heap \rightarrow Heap\\
&insert\ e\ Empty \quad = singleton\ e\\
&insert\ e\ (Heap\ h) = Heap\ (ins\ (singletonT\ e)\ h)
\end{aligned}
$$

We will also need an insertion function corresponding to an increment-by-two operation.

$$
\begin{aligned}
&ins2 :: T_{Succ\ k} \rightarrow SubHeap\ n\ k \rightarrow SubHeap\{inc\{inc\ n\}\}k\\
&ins2\ t\ (Last\ x) \quad\ \ = Cons\ x\ (Last\ t)\\
&ins2\ t\ (Skip \quad\ h) = Skip\ (ins\ t\ h)\\
&ins2\ t\ (Cons\ x\ h) = Cons\ x\ (ins\ t\ h)
\end{aligned}
$$

4.3.3 Merging

Adding two binary numbers corresponds to merging two binomial heaps. Since addition over binary numbers, is defined by mutually recursive type functions, we define two mutually recursive merge functions, one with an extra binomial tree argument (corresponding to a carry bit) and one without. We encourage the reader to compare these definitions with the definitions of *add* and *addc*.

$$
\begin{aligned}
&mrg :: SubHeap\ n\ k \rightarrow SubHeap\ m\ k \rightarrow SubHeap\{add\ n\ m\}k\\
&mrg\ (Last \quad x) \quad m \qquad\quad\ \ = ins\ x\ m\\
&mrg\ n \qquad\qquad\quad\ (Last\ \ y) \quad = ins\ y\ n
\end{aligned}
$$

$$
\begin{array}{l}
mrg\ (Skip\ \ n)\ (Skip\ \ m) = Skip\ \ (mrg\ \ n\ m) \\
mrg\ (Skip\ \ n)\ (Cons\ y\ m) = Cons\ y\ (mrg\ \ n\ m) \\
mrg\ (Cons\ \ x\ n)\ (Skip\ \ m) = Cons\ x\ (mrg\ \ n\ m) \\
mrg\ (Cons\ \ x\ n)\ (Cons\ y\ m) = Skip\ \ (mrgc\ (mergeT\ x\ y)\ n\ m)
\end{array}
$$

$$
\begin{array}{l}
mrgc :: T_k \to SubHeap\ n\ k \to SubHeap\ m\ k \to SubHeap\{addc\ n\ m\}k \\
mrgc\ x\ (Last\ \ y)\ \ m\ \ = ins2\ (mergeT\ x\ y)\ m \\
mrgc\ x\ n\ \ (Last\ \ z)\ \ = ins2\ (mergeT\ x\ z)\ n \\
mrgc\ x\ (Skip\ \ n)\ (Skip\ \ m) = Cons\ x\ (mrg\ \ n\ m) \\
mrgc\ x\ (Skip\ \ n)\ (Cons\ z\ m) = Skip\ \ (mrgc\ (mergeT\ x\ z)\ n\ m) \\
mrgc\ x\ (Cons\ y\ n)\ (Skip\ \ m) = Skip\ \ (mrgc\ (mergeT\ x\ y)\ n\ m) \\
mrgc\ x\ (Cons\ y\ n)\ (Cons\ z\ m) = Cons\ x\ (mrgc\ (mergeT\ y\ z)\ n\ m)
\end{array}
$$

Again, we define a wrapper function for merging top-level *Heap*s.

$$
\begin{array}{l}
merge :: Heap \to Heap \to Heap \\
merge\ Empty\ \ h2\ \ = h2 \\
merge\ h1\ \ Empty\ \ = h1 \\
merge\ (Heap\ h1)\ (Heap\ h2) = Heap\ (mrg\ h1\ h2)
\end{array}
$$

4.3.4 Retrieving and Removing the Minimum Element

Due to space considerations, we omit a full exposition of the *minElem* and *extractMin* operations. The most interesting part of those definitions involves a type-indexed "zipper" type [5] that represents *SubHeap* contexts. This gives evidence that Huet's zipper technique scales well to indexed types. The interested reader may refer to the Appendix for details.

5 Typed CPS Transform

We now give an Ωmega implementation of the call-by-value CPS transform as presented by Hatcliff and Danvy [3]. They use the following syntax:

$$
\begin{array}{llcl@{\qquad}llcl}
(comp.) & m, n & ::= & v \mid m\ n & (type) & \tau & ::= & Int \mid \tau \to \tau \mid R \\
(value) & v & ::= & i \mid x \mid \lambda x.\ n & (env.) & \Gamma & ::= & \cdot \mid \Gamma, x : \tau
\end{array}
$$

Lambda terms are divided into values and computations. We commit here to a call-by-value source language by categorizing variables as values. The type R is a special type introduced by the CPS transform as the final result type of any computation.

Then they define a call-by-value CPS transform on terms.

$$
\begin{array}{lcl}
\langle\!| v |\!\rangle & = & \lambda k.\ k\ \langle v \rangle \\
\langle\!| m\ n |\!\rangle & = & \lambda k.\ \langle\!| m |\!\rangle\ (\lambda y_0.\ \langle\!| n |\!\rangle\ (\lambda y_1.\ y_0\ y_1\ k))
\end{array}
\qquad
\begin{array}{lcl}
\langle i \rangle & = & i \\
\langle x \rangle & = & x \\
\langle \lambda x.\ n \rangle & = & \lambda x.\ \langle\!| n |\!\rangle
\end{array}
$$

The CPS transform has the following effect on types (and type environments):

$$
\begin{array}{lcl}
\langle\!| \tau |\!\rangle & = & (\langle \tau \rangle \to R) \to R \\
\langle Int \rangle & = & Int \\
\langle \tau \to \sigma \rangle & = & \langle \tau \rangle \to \langle\!| \sigma |\!\rangle
\end{array}
\qquad
\begin{array}{lcl}
\{\!| \cdot |\!\} & = & \cdot \\
\{\!| \Gamma, x : \tau |\!\} & = & \{\!| \Gamma |\!\}, x : \langle \tau \rangle
\end{array}
$$

Following two (impressive looking) theorems express that the CPS translation is type preserving:

If $\Gamma \vdash n : \tau$ then $\{\!|\Gamma|\!\} \vdash \langle\!|n|\!\rangle : \langle\!|\tau|\!\rangle$ and

if $\Gamma \vdash v : \tau$ then $\{\!|\Gamma|\!\} \vdash \langle v \rangle : \langle \tau \rangle$.

Our implementation of CPS in Ωmega will prove these theorems using type checking alone.

5.1 Type Representations and Singleton Types

The types of the object language can be modelled by Ωmega types. Since Ωmega already has *Int* types and function types ($\_ \rightarrow \_$), we only need introduce a new type R.

$$\textbf{data}\ R = \mathit{FinalAnswer}$$

We can define a *representation type* for the subset of Ωmega's type language that we will employ in the CPS example. A representation (a.k.a. singleton) type T is a parameterized type. A (run-time) value of type $T\ a$ represents (or mirrors the shape of) the the static type a. For example consider *Type* t below.

$$\begin{array}{lll} \multicolumn{3}{l}{\textbf{data}\ \mathit{Type}\ t} \\ = & \mathit{Int} & \textbf{where}\ t = \mathit{Int} \\ \mid & \exists\, a\ b.\mathit{Arr}\ (\mathit{Type}\ a)\ (\mathit{Type}\ b) & \textbf{where}\ t = a \rightarrow b \\ \mid & R & \textbf{where}\ t = R \end{array}$$

Note how every value of type *Type* a has the same shape as a. For example: $\mathit{Int} :: \mathit{Type}\ \mathit{Int}$, and $R :: \mathit{Type}\ R$, and if $a :: \mathit{Type}\ x$ and $b :: \mathit{Type}\ y$, then $\mathit{Arr}\ a\ b :: \mathit{Type}\ (x \rightarrow y)$. For example, the value $\mathit{Arr}\ \mathit{Int}\ R$ has type $\mathit{Type}\ (\mathit{Int} \rightarrow R)$. The useful property of *Type* is that there is a one-to-one correspondence between values of type *Type* t and types t. Values of type *Type* t are ordinary run-time values and can be computed over like any other values. Knowing the shape (which constructors are used to build it) of a value of type $T\ t$ determines its type, and knowing the type of such a value determines its shape.This is why they are some times called singleton types.

5.2 Object Terms

As we describe the CPS transform we make use of two languages. The object-language represents the programs we are transforming (CPS terms) and the meta-language (Ωmega) describes the transformation. Object-programs are data in the meta-language. The Ωmega data-structures representing CPS object-programs are defined using equality qualified types parameterized by two type indexes: an environment e, and an object-level type t. This means that Ωmega's type system guarantees that all well-typed meta-level terms representing object-level terms, represent only well-typed object-terms. The technique relies on de Bruijn indices to represent variables. [8]

$$\begin{array}{llllll} \textbf{data}\ \mathit{Cmp}\ e\ t = & & \mathit{Val} & (\mathit{Val}\ e\ t) & \\ & \mid\ \exists\, s. & \mathit{App} & (\mathit{Cmp}\ e\ (s \rightarrow t))\ (\mathit{Cmp}\ e\ s) & \\ \textbf{data}\ \mathit{Val}\ e\ t\ = & & \mathit{Lit} & \mathit{Int} & \textbf{where}\ t = \mathit{Int} \\ & \mid & \mathit{Var} & (\mathit{Var}\ e\ t)\ (\mathit{Type}\ t) & \end{array}$$

$$\begin{array}{lll} & \mid \exists\, a\ b. \mathit{Lam}\ (\mathit{Type}\ a)\ (\mathit{Cmp}\ (e, a)\ b) & \textbf{where}\ t = a \rightarrow b \\ \textbf{data}\ \mathit{Var}\ e\ t & = \exists\, f.\quad \mathit{Zero} & \textbf{where}\ e = (f, t) \\ & \mid \exists\, f\ s. \mathit{Succ}\ (\mathit{Var}\ f\ t) & \textbf{where}\ e = (f, s) \end{array}$$

These datatypes correspond to typing judgments and their values correspond to proofs of those judgments. We are using nested pairs (growing to the right) to model typing environments and a fragment of the Ωmega types (as discussed in Section 5.1 above) to model object-types. Some examples are germane here. Shorthand for some common variable indices will prove useful.

$$\begin{array}{l} \mathit{zero}\ :: \mathit{Type}\ a \rightarrow \mathit{Cmp}\ (e, a)\ a \\ \mathit{zero}\ t = \mathit{Val}\ (\mathit{Var}\ \mathit{Zero}\ t) \\[4pt] \mathit{one}\ \ :: \mathit{Type}\ a \rightarrow \mathit{Cmp}\ ((e, a), b)\ a \\ \mathit{one}\ \ t = \mathit{Val}\ (\mathit{Var}\ (\mathit{Succ}\ \mathit{Zero})\ t) \\[4pt] \mathit{two}\ \ :: \mathit{Type}\ a \rightarrow \mathit{Cmp}\ (((e, a), b), c)\ a \\ \mathit{two}\ \ t = \mathit{Val}\ (\mathit{Var}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}))\ t) \end{array}$$

Consider representing the object-level term $\lambda x : Int \rightarrow Int.\ \lambda y : Int.\ x\ y$ The meta-level term: $m = \mathit{App}\ (\mathit{one}\ (\mathit{Arr}\ \mathit{Int}\ \mathit{Int}))\ (\mathit{zero}\ \mathit{Int})$ has type $\mathit{Cmp}\ ((e, \mathit{Int} \rightarrow \mathit{Int}), \mathit{Int})\ \mathit{Int}$. Notice how the environment is a nested pair. By wrapping m in two lambdas $\mathit{Lam}\ (\mathit{Arr}\ \mathit{Int}\ \mathit{Int})\ (\mathit{Val}\ (\mathit{Lam}\ \mathit{Int}\ m))$, we obtain a term with type $\mathit{Val}\ e\ ((\mathit{Int} \rightarrow \mathit{Int}) \rightarrow \mathit{Int} \rightarrow \mathit{Int}))$

5.3 Reading Off Types

The abstract syntax includes enough type information to recover at run-time the (representation of the) type of any lambda term.

$$\begin{array}{ll} \mathit{expType}\ :: \mathit{Cmp}\ e\ t \rightarrow \mathit{Type}\ t & \\ \mathit{expType}\ (\mathit{Val}\ v) & = \mathit{valType}\ v \\ \mathit{expType}\ (\mathit{App}\ m\ n) & = \mathit{codomain}\ (\mathit{expType}\ m) \\[4pt] \mathit{valType}\ \ :: \mathit{Val}\ e\ t \rightarrow \mathit{Type}\ t & \\ \mathit{valType}\ \ (\mathit{Lit}\ i) & = \mathit{Int} \\ \mathit{valType}\ \ (\mathit{Var}\ x\ t) & = t \\ \mathit{valType}\ \ (\mathit{Lam}\ t\ n) & = \mathit{Arr}\ t\ (\mathit{expType}\ n) \\[4pt] \mathit{codomain} :: \mathit{Type}\ (a \rightarrow b) \rightarrow \mathit{Type}\ b & \\ \mathit{codomain}\ (\mathit{Arr}\ a\ b) = b & \end{array}$$

5.4 Shifting de Bruijn Indices

The CPS transform introduces new lambda binders for additional continuation parameters. This necessitates incrementing some variable indices to point past any newly introduced binder, an operation known as a *shift*.

$$\begin{array}{ll} \mathit{shift} :: \mathit{Cmp}\ e\ t \rightarrow \mathit{Cmp}\ (e, s)\ t & \\ \mathit{shift}\ e & = \mathit{shiftE}\ \mathit{Succ}\ e \\[4pt] \mathit{shiftE} :: (\forall\, t. \mathit{Var}\ e\ t \rightarrow \mathit{Var}\ f\ t) \rightarrow \mathit{Cmp}\ e\ t \rightarrow \mathit{Cmp}\ f\ t & \\ \mathit{shiftE}\ f\ (\mathit{App}\ n\ m) & = \mathit{App}\ (\mathit{shiftE}\ f\ n)\ (\mathit{shiftE}\ f\ m) \\ \mathit{shiftE}\ f\ (\mathit{Val}\ v) & = \mathit{Val}\ (\mathit{shiftV}\ f\ v) \end{array}$$

$$
\begin{aligned}
&\mathit{shiftV} :: (\forall\, t. \mathit{Var}\ e\ t \to \mathit{Var}\ f\ t) \to \mathit{Val}\ e\ t \to \mathit{Val}\ f\ t\\
&\mathit{shiftV}\ f\ (\mathit{Lit}\ i) \quad\ \ = \mathit{Lit}\ i\\
&\mathit{shiftV}\ f\ (\mathit{Var}\ x\ s)\ \ = \mathit{Var}\ (f\ x)\ s\\
&\mathit{shiftV}\ f\ (\mathit{Lam}\ s\ n) = \mathit{Lam}\ s\ (\mathit{shiftE}\ (\mathit{lift}\ f)\ n)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{lift} :: (\forall\, t. \mathit{Var}\ e\ t \to \mathit{Var}\ f\ t) \to \mathit{Var}\ (e, s)\ t \to \mathit{Var}\ (f, s)\ t\\
&\mathit{lift}\ s \quad (\mathit{Zero}) \quad\ \ = \mathit{Zero}\\
&\mathit{lift}\ s \quad (\mathit{Succ}\ x) \ \ = \mathit{Succ}\ (s\ x)
\end{aligned}
$$

Since *shift* is really manipulating typing judgments, we can think of its type as a kind of weakening lemma for de Bruijn terms. In this view, the *shift* function itself is a *proof* of the lemma (provided it terminates on all inputs).

5.5 CPS Transform on Types

Now we define the effect of the call-by-value CPS transform on the types of object terms. This is conveniently expressed with Ωmega's type functions.

$$
\begin{aligned}
&\mathit{cps} \ :: * \leadsto * \leadsto *\\
&\{\mathit{cps}\ \ r\ t\} \qquad\quad = (\{\mathit{cpsv}\ r\ t\} \to r) \to r
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{cpsv} :: * \leadsto * \leadsto *\\
&\{\mathit{cpsv}\ r\ \mathit{Int}\} \qquad = \mathit{Int}\\
&\{\mathit{cpsv}\ r\ (a \to b)\} = \{\mathit{cpsv}\ r\ a\} \to \{\mathit{cps}\ r\ b\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{cpse} :: * \leadsto * \leadsto *\\
&\{\mathit{cpse}\ r\ ()\} \qquad\ \ = ()\\
&\{\mathit{cpse}\ r\ (a, b)\} \quad = (\{\mathit{cpse}\ r\ a\}, \{\mathit{cpsv}\ r\ b\})
\end{aligned}
$$

We then reflect the type functions *cps* and *cpsv* at the value level.

$$
\begin{aligned}
&\mathit{cpsT} \ :: \mathit{Type}\ r \to \mathit{Type}\ t \to \mathit{Type}\{\mathit{cps}\ r\ t\}\\
&\mathit{cpsT} \ \ r\ t \qquad\quad\ = \mathit{Arr}\ (\mathit{Arr}\ (\mathit{cpsTv}\ r\ t)\ r)\ r
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{cpsTv} :: \mathit{Type}\ r \to \mathit{Type}\ t \to \mathit{Type}\{\mathit{cpsv}\ r\ t\}\\
&\mathit{cpsTv}\ r\ \mathit{Int} \qquad\ = \mathit{Int}\\
&\mathit{cpsTv}\ r\ (\mathit{Arr}\ a\ b) = \mathit{Arr}\ (\mathit{cpsTv}\ r\ a)\ (\mathit{cpsT}\ r\ b)
\end{aligned}
$$

5.6 CPS Transform on Terms

Finally we define the CPS transform on terms. The transformation is given by three functions, one for each syntactic category.

$$
\begin{aligned}
&\mathit{cpsC} :: \mathit{Type}\ r \to \mathit{Cmp}\ e\ a \to \mathit{Cmp}\{\mathit{cpse}\ r\ e\}\{\mathit{cps}\ \ r\ a\}\\
&\mathit{cpsV} :: \mathit{Type}\ r \to \mathit{Val}\ \ e\ a \to \mathit{Val}\ \ \{\mathit{cpse}\ r\ e\}\{\mathit{cpsv}\ r\ a\}\\
&\mathit{cpsX} :: \mathit{Type}\ r \to \mathit{Var}\ \ e\ a \to \mathit{Var}\ \ \{\mathit{cpse}\ r\ e\}\{\mathit{cpsv}\ r\ a\}
\end{aligned}
$$

Each function has a *Type* r parameter because the CPS transform is parameterized by the final result type r. We are transforming object types as well as object terms: the essence of a type-respecting transformation.

The *Var* and *Val* cases are straightforward. In fact *cpsX* is operationally the identity function, but it has a more complicated type. Another curious fact about *cpsX* is that the *Type* r parameter is never used, but is necessary for type-checking. Otherwise, in the *Succ* case, we can't unify the r of the calling *cpsX* with the fresh r' resulting from the recursive call.

$$
\begin{array}{l}
\mathit{cpsX} :: \mathit{Type}\ r \rightarrow \mathit{Var}\ e\ a \rightarrow \mathit{Var}\{\mathit{cpse}\ r\ e\}\{\mathit{cpsv}\ r\ a\} \\
\mathit{cpsX}\ r\ (\mathit{Zero}) \quad\ \ = \mathit{Zero} \\
\mathit{cpsX}\ r\ (\mathit{Succ}\ x) \ \ = \mathit{Succ}\ (\mathit{cpsX}\ r\ x)
\end{array}
$$

$$
\begin{array}{l}
\mathit{cpsV} :: \mathit{Type}\ r \rightarrow \mathit{Val}\ \ e\ a \rightarrow \mathit{Val}\{\mathit{cpse}\ r\ e\}\{\mathit{cpsv}\ r\ a\} \\
\mathit{cpsV}\ r\ (\mathit{Lit}\ i) \quad\ \ = \mathit{Lit}\ i \\
\mathit{cpsV}\ r\ (\mathit{Var}\ x\ t)\ = \mathit{Var}\ (\mathit{cpsX}\ r\ x)\ (\mathit{cpsTv}\ r\ t) \\
\mathit{cpsV}\ r\ (\mathit{Lam}\ t\ n) = \mathit{Lam}\ (\mathit{cpsTv}\ r\ t)\ (\mathit{cpsC}\ r\ n)
\end{array}
$$

The *Cmp* cases are the most involved, as they introduce new binders for continuation parameters. This is where we need the *shift* operation.

$$
\begin{array}{l}
\mathit{cpsC} :: \mathit{Type}\ r \rightarrow \mathit{Cmp}\ e\ a \rightarrow \mathit{Cmp}\{\mathit{cpse}\ r\ e\}\{\mathit{cps}\ r\ a\} \\
\mathit{cpsC}\ r\ (\mathit{Val}\ v) \quad\ \ = \mathit{Val}\ (\mathit{Lam}\ \mathit{kt}\ (\mathit{App}\ (\mathit{zero}\ \mathit{kt})\ (\mathit{shift}\ (\mathit{Val}\ v')))) \\
\quad \textbf{where}\ v' \qquad\ = \mathit{cpsV}\ r\ v \\
\qquad\qquad \mathit{kt} \qquad = \mathit{Arr}\ (\mathit{valType}\ v')\ r
\end{array}
$$

$$
\begin{array}{l}
\mathit{cpsC}\ r\ (\mathit{App}\ m\ n) = \textbf{case}\ \mathit{expType}\ m'\ \textbf{of} \\
\qquad\qquad\qquad \mathit{Arr}\ (\mathit{Arr}\ (\mathit{yt}_0@(\mathit{Arr}\ \mathit{yt}_1\ (\mathit{Arr}\ \mathit{kt}\ \_)))\ \_)\ \_ \rightarrow \\
\qquad\qquad\qquad\quad \mathit{Val}\ (\mathit{Lam}\ \mathit{kt} \\
\qquad\qquad\qquad\quad (\mathit{App}\ (\mathit{shift}\ m') \qquad\quad (\mathit{Val}\ (\mathit{Lam}\ \mathit{yt}_0 \\
\qquad\qquad\qquad\quad (\mathit{App}\ (\mathit{shift}\ (\mathit{shift}\ n'))\ (\mathit{Val}\ (\mathit{Lam}\ \mathit{yt}_1 \\
\qquad\qquad\qquad\quad (\mathit{App}\ (\mathit{App}\ (\mathit{one}\ \mathit{yt}_0)\ (\mathit{zero}\ \mathit{yt}_1))\ (\mathit{two}\ \mathit{kt})) \\
\qquad\qquad\qquad\quad))))))) \\
\quad \textbf{where}\ m' \qquad = \mathit{cpsC}\ r\ m \\
\qquad\qquad n' \qquad\ \ = \mathit{cpsC}\ r\ n
\end{array}
$$

6 Alternatives to Type-Functions

There are a few alternatives to the use of type functions of Ωmega. We explore them here because they can be used to give a sound basis to a semantics for type functions. But while they are sound and do not require the additional mechanism required by type functions, they are often clumsy and hard to use. Our addition of type functions to Ωmegawas inspired by programming these more tedious examples, and wondering how we could make our lives easier. Throughout this section we illustrate the techniques using the *sum*/*append* example.

$$
\begin{array}{l}
\mathit{sum} :: \mathit{Nat} \leadsto \mathit{Nat} \leadsto \mathit{Nat} \\
\{\mathit{sum}\ \mathit{Zero} \qquad m\} = m \\
\{\mathit{sum}\ (\mathit{Succ}\ n)\ m\} = \mathit{Succ}\{\mathit{sum}\ n\ m\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{append} :: \mathit{Seq}\ a\ n \rightarrow \mathit{Seq}\ a\ m \rightarrow \mathit{Seq}\ a\{\mathit{sum}\ n\ m\} \\
\mathit{append}\ (\mathit{Nil}) \qquad\quad \mathit{ys} = \mathit{ys} \\
\mathit{append}\ (\mathit{Cons}\ x\ \mathit{xs})\ \mathit{ys} = \mathit{Cons}\ x\ (\mathit{append}\ \mathit{xs}\ \mathit{ys})
\end{array}
$$

One alternative is to use a datatype of explicit “proofs” of the addition relation. This datatype is the relational incarnation of the *sum* type function.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Sum}\ n\ m\ s \\
\quad = \qquad \mathit{Base} \qquad\qquad\qquad\ \textbf{where}\ n = \mathit{Zero}, \quad\ s = m \\
\quad \mid \exists\ n'\ s'.\mathit{Step}\ (\mathit{Sum}\ n'\ m\ s')\ \textbf{where}\ n = \mathit{Succ}\ n', s = \mathit{Succ}\ s'
\end{array}
$$

$append :: Sum\ n\ m\ s \rightarrow Seq\ a\ n \rightarrow Seq\ a\ m \rightarrow Seq\ a\ s$
$append\ (Base)\quad (Nil)\quad ys = ys$
$append\ (Step\ sum)\ (Cons\ x\ xs)\ ys = Cons\ x\ (append\ sum\ xs\ ys)$

This alternative has the disadvantage of requiring an extra *Sum* input. We can overcome this limitation by giving *append* the type $Seq\ a\ n \rightarrow Seq\ a\ m \rightarrow \exists\ s.(Sum\ n\ m\ s, Seq\ a\ s)$. Since Ωmega doesn't directly support existential types in this way, we simulate them using a combination of rank-2 polymorphism and continuation-passing style.

$append :: Seq\ a\ n \rightarrow Seq\ a\ m \rightarrow (\forall\ s.Sum\ n\ m\ s \rightarrow Seq\ a\ s \rightarrow r) \rightarrow r$
$append\ (Nil)\quad ys\ k = k\ Base\ ys$
$append\ (Cons\ x\ xs)\ ys\ k = append\ xs\ ys\ (\lambda sum\ zs \rightarrow$
$\qquad k\ (Step\ sum)\ (Cons\ x\ zs))$

In a language supporting multi-parameter type classes with functional dependencies [7] (as do common extensions to Haskell), we can forego explicitly constructing *Sum* proofs. We instead define a type class for the *Sum* relation, following Hallgren [2]

Does this actually work? Try it out in GHC!

class $Sum\ n\ m\ s \mid n\ m \rightarrow s$ **where** { }
instance $Sum\ Zero\quad m\ m$ **where** { }
instance $Sum\ n\ m\ s \Rightarrow Sum\ (Succ\ n)\ m\ (Succ\ s)$ **where** { }

Then we use the same CPS trick, but omitting the proof objects.

$append :: Seq\ a\ n \rightarrow Seq\ a\ m \rightarrow (\forall\ s.Sum\ n\ m\ s \Rightarrow Seq\ a\ s \rightarrow r) \rightarrow r$
$append\ (Nil)\quad ys\ k = k\ ys$
$append\ (Cons\ x\ xs)\ ys\ k = append\ xs\ ys\ (\lambda zs \rightarrow k\ (Cons\ x\ zs))$

Type inference introduces all the proof objects as implicit dictionary arguments. In this case, the dictionaries are empty and should be optimized away.

We can also wrap up the guarded existential in a datatype, but the end result is essentially a verbose version of continuation-passing-style. Which one is preferable is a matter of taste.

data $SumSeq\ n\ m = \exists\ s.Sum\ n\ m\ s \Rightarrow SumSeq\ (Seq\ a\ s)$
$append :: Seq\ a\ n \rightarrow Seq\ a\ m \rightarrow Seq\ a\{sum\ n\ m\}$
$append\ (Nil)\quad ys = SumSeq\ ys$
$append\ (Cons\ x\ xs)\ ys =$ **case** $append\ xs\ ys$ **of**
$\qquad SumSeq\ zs \rightarrow SumSeq\ (Cons\ x\ zs)$

It remains to be seen whether most interesting uses of type functions fall into the set that can be implemented using type classes. If so, we would like to use the syntax of type functions as an interface to the underlying machinery of type classes with functional dependencies. Neubauer et al make a similar proposal [4].

7 Conclusion

Type functions make for elegant code, but need a more solid foundation.

8 Related Work

8.1 Non-regular types

also encode invariants.
sometimes less natural.
cite Okasaki citeokasaki-adventure, Hinze [?].

8.2 Type Functions

Fun with Functional Dependencies [2]
Functional Notation for Functional Dependencies [4]

References

[1] James Cheney and Ralf Hinze. First class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cis/TR2003-1901, 2003.

[2] Thomas Hallgren. Fun with functional dependencies, 2001.

[3] John Hatcliff and Olivier Danvy. Thunks and the lambda-calculus. *J. Functional Programming*, 7(3):303–319, May 1997.

[4] Ralf Hinze. Constructing red-black trees. pages 89–99, September 1999. The proceedings appear as a technical report of Columbia University, CUCS-023-99.

[5] Gérard Huet. Functional pearl: The zipper. *J. Functional Programming*, 7(5):549–554, 1997.

[6] Mark P. Jones. *Qualified types :–theory and practice.* PhD thesis, Keble College, Oxford University, 1992.

[7] Mark P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244. Springer-Verlag, 2000.

[8] Emir Pasalic. *The Role of Type Equality in Meta-Programming.* PhD thesis, OGI School of Science & Engineering, Oregon Health & Science University, 2004.

[9] Simon Peyton-Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: Type inference for generalised algebraic data types. July 2004.

[10] Tim Sheard. Languages of the future. October 2004. OOPSLA Companion Volume.

[11] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM'04).* `http://cs-www.cs.yale.edu/homes/carsten/lfm04/`, pages 106–124, Cork, Ireland, July 2004.

[12] Vincent Simonet and Franois Pottier. Constraint-based type inference for guarded algebraic data types. July 2003.

[13] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.

9 Appendix: Code for *minElem* and *extractMin*

9.1 Retrieving the Minimum Element

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{minElem} :: \mathit{Heap} \to \mathit{Maybe}\ \mathit{Elem}} \\
\mathit{minElem}\ \mathit{Empty} & = \mathit{Nothing} \\
\mathit{minElem}\ (\mathit{Heap}\ h) & = \mathit{Just}\ (\mathit{minElem}_1\ h)
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{minElem}_1 :: \mathit{SubHeap}\ n\ k \to \mathit{Elem}} \\
\mathit{minElem}_1\ (\mathit{Last}\ t) & = \mathit{top}\ t \\
\mathit{minElem}_1\ (\mathit{Skip}\ \ h) & = \mathit{minElem}_1\ h \\
\mathit{minElem}_1\ (\mathit{Cons}\ t\ h) & = \mathit{minElem}_2\ h\ (\mathit{top}\ t)
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{minElem}_2 :: \mathit{SubHeap}\ n\ k \to \mathit{Elem} \to \mathit{Elem}} \\
\mathit{minElem}_2\ (\mathit{Last}\ t)\ \ x & = \mathit{min}\ x\ (\mathit{top}\ t) \\
\mathit{minElem}_2\ (\mathit{Skip}\ \ h)\ x & = \mathit{minElem}_2\ h\ x \\
\mathit{minElem}_2\ (\mathit{Cons}\ t\ h)\ x & = \mathit{minElem}_2\ h\ (\mathit{min}\ x\ (\mathit{top}\ t))
\end{array}
$$

9.2 Removing the Minimum Element

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{extractMin} :: \mathit{Heap} \to \mathit{Maybe}\ \mathit{Heap}} \\
\mathit{extractMin}\ \mathit{Empty} & = \mathit{Nothing} \\
\mathit{extractMin}\ (\mathit{Heap}\ h) & = \mathit{Just}\ (\mathit{unfocus}\ (\mathit{minFocus1}\ \mathit{Top}\ h))
\end{array}
$$

9.2.1 "The Zipper".

$$
\begin{array}{lllll}
\mathbf{data}\ \mathit{Path}\ k = & \mathit{Top} & & & \mathbf{where}\ k = \mathit{Zero} \\
 & \mid \exists\, j.\mathit{Skip}' & & (\mathit{Path}\ j) & \mathbf{where}\ k = \mathit{Succ}\ j \\
 & \mid \exists\, j.\mathit{Cons}' & (T\ j) & (\mathit{Path}\ j) & \mathbf{where}\ k = \mathit{Succ}\ j
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{zipHeap} :: \mathit{Path}\ k \to \mathit{SubHeap}\ n\ k \to \mathit{Heap}} \\
\mathit{zipHeap}\ \mathit{Top} & h = \mathit{Heap}\ h \\
\mathit{zipHeap}\ (\mathit{Skip}'\ p) & h = \mathit{zipHeap}\ p\ (\mathit{Skip}\ h) \\
\mathit{zipHeap}\ (\mathit{Cons}'\ t\ p) & h = \mathit{zipHeap}\ p\ (\mathit{Cons}\ t\ h)
\end{array}
$$

$$
\begin{aligned}
&close :: Path\ k \to Heap\\
&close\ Top \qquad\quad = Empty\\
&close\ (Skip'\ p) \quad = close\ p\\
&close\ (Cons'\ t\ p) = zipHeap\ p\ (Last\ t)
\end{aligned}
$$

$$
\begin{aligned}
&fill :: Path\ k \to Maybe\ (SubHeap\ n\ (Succ\ k)) \to Heap\\
&fill\ p\ Nothing = close\ p\\
&fill\ p\ (Just\ h) = zipHeap\ p\ (Skip\ h)
\end{aligned}
$$

9.2.2 The Focus type.

$$
\mathbf{data}\ Focus = \exists\ n\ k.Focus\ (Path\ k)\ (T\ k)\ (Maybe\ (SubHeap\ n\ (Succ\ k)))
$$

$$
\begin{aligned}
&unfocus :: Focus \to Heap\\
&unfocus\ (Focus\ p\ t\ mh) = merge\ (heapify\ t)\ (fill\ p\ mh)
\end{aligned}
$$

$$
\begin{aligned}
&key :: Focus \to Elem\\
&key\ (Focus\ \_\ t\ \_) = top\ t
\end{aligned}
$$

9.2.3 Heapifying a tree.

$$
\begin{aligned}
&heapify :: T\ k \to Heap\\
&heapify\ (Tree\ i\ f) = reverse1\ f\\
&reverse1 :: F\ k \to Heap\\
&reverse1\ (FNil) \qquad\quad = Empty\\
&reverse1\ (FCons\ t\ f) \quad = reverse2\ f\ (Last\ t)\\
&reverse2 :: F\ k \to SubHeap\ n\ k \to Heap\\
&reverse2\ (FNil) \qquad\quad h = Heap\ h\\
&reverse2\ (FCons\ t\ f)\ h = reverse2\ f\ (Cons\ t\ h)
\end{aligned}
$$

9.2.4 Finding the minimum Focus.

$$
\begin{aligned}
&minFocus_1 :: Path\ k \to SubHeap\ n\ k \to Focus\\
&minFocus_1\ p\ (Last\ t) \qquad = Focus\ p\ t\ Nothing\\
&minFocus_1\ p\ (Skip\ h) \qquad = minFocus_1\ (Skip'\ p)\ h\\
&minFocus_1\ p\ (Cons\ t\ h) \quad = minFocus_2\ (Cons'\ t\ p)\ h\\
&\qquad\qquad\qquad\qquad\qquad\qquad (Focus\ p\ t\ (Just\ h))\\
&minFocus_2 :: Path\ k \to SubHeap\ n\ k \to Focus \to Focus\\
&minFocus_2\ p\ (Last\ t) \quad\ \ x = minBy\ key\ x\ (Focus\ p\ t\ Nothing)\\
&minFocus_2\ p\ (Skip\ \quad h)\ x = minFocus_2\ (Skip'\ p)\ h\ x\\
&minFocus_2\ p\ (Cons\ t\ h)\ x = minFocus_2\ (Cons'\ t\ p)\ h\\
&\qquad\qquad\qquad\qquad\qquad\qquad (minBy\ key\ x\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (Focus\ p\ t\ (Just\ h)))
\end{aligned}
$$