Type-level Computation Using Narrowing in Ωmega.

Tim Sheard[1 ,2]

Computer Science Department
Portland State University
Portland, OR. USA

Abstract

Ωmega is an experimental system that combines features of both a programming language and a logical reasoning system. Ωmega is a language with an infinite hierarchy of computational levels. Terms at one level are classified (or typed) by terms at the next higher level. In this paper we report on using two different computational mechanisms. At the value level, computation is performed by reduction, and is largely unconstrained. At all higher levels, computation is performed by narrowing.

Keywords: Programming languages, logical systems, theorem provers, narrowing, Curry-Howard isomorphism.

1 Introduction

There has been a lot of recent interest in exploiting the Curry-Howard isomorphism in type systems for more or less traditional programming languages. Types based upon the Curry-Howard isomorphism can express precise properties of programs. Such properties can be either functional or non-functional properties. I.e. they can constrain either the output of the program, or the resources needed to produce the output, or both.

The Curry-Howard isomorphism states that types are propositions (or properties), and that programs are proofs. This isomorphism points out two views of the same phenomena. When we write $prog_1 :: type_1$, we are stating both that $prog_1$ has type $type_1$, and $prog_1$ is a proof of the property $type_1$. By generalizing the notion of type, we can use types to be precise about how programs behave.

To use this concept effectively, we have to think of types as more than just a mechanism to predict the layout or representation of data, or as a mechanism to classify values into similar groups (all the integers, all the booleans, etc.). In

[1] Work reported in this paper was supported by the National Scince Foundation by grants CCF-0541447 and CCF-0613969

[2] Email: sheard@cs.pdx.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Ωmega, the normal types such as `Int`, `String`, and the function arrow (`->`), play these roles, but programmers can also introduce new kinds of types, which look like algebraic data structures (list, trees, etc.), but which live at the type level, rather than at the value level like ordinary algebraic data. Such types play new roles, not played by traditional types. They do not classify values, but instead they are used to describe properties such as the shape of value-level data, or the resources consumed by a function application. In addition, Ωmega allows programmers to write programs that manipulate these types, and these type-manipulating programs are executed, not at run-time, but instead at type-checking time to direct the type-checking process. Declaring a type for a program states a property of that program, and checking that a program has a declared type, proves it has that property.

Our goal is to build a system in which the specification of designs, the definition of properties, the implementation of programs, and the checking that programs adhere to their properties, are all bundled in a coherent manner into a single unified system that appears to the user to be a programming language. We hope to use Ωmega as broad spectrum language, capable of handling abstract properties as well as implementation minutiae, where the connection between properties and programs is formal and precise.

The language Ωmega is our first attempt at building such a system. It is a work in progress [23,24,29,33,32,30,31,34]. While it does not yet meet all of the goals we will list below, we have made it freely available (`http://www.cs.pdx.edu/~sheard/Omega/index.html`).

2 Related work

Writing programs at both the value and type level is a strange but exciting idea indeed! The programmer writes programs at two levels, one to execute at run-time, the other to be executed by the type checker at compile-time to enforce properties. The design space for such a system is large. Explored corners of the design space include logical systems such as Inductive Families[12,15], theorem provers (Coq[39], Isabelle[25]), logical frameworks (Twelf[27], LEGO[20]), and proof assistants (ALF[22], Agda[11]).

Recently, the design space exploration has grown to include work that uses dependent types to build "practical" systems that are part language, part reasoning system. The designs of these systems vary widely. Some explore different parts of the design space, and most have some explicit design goals that the language designers were trying to accomplish. We first list our own design goals for Ωmega, and then try and place some of the other systems within the design space. In Ωmega, our design goals include:

- We want both programmers and logicians to be able to use the system. We want our system to both look and behave like a programming language, but still be a sound logic.
- When the type system disallows a program we want the programmer to be able to understand why the program was rejected, and to be able to understand how to repair the program if possible. (This is very hard!)

- There should be a strict separation between values and types. This allows the implementation to use a type erasure semantics, so that types (properties) cost nothing at run-time. This is an important instance of a more general policy.
- There should be a pay for what you use approach. Programs which need not be too precise, and hence use little or no logic, should incur little or no cost. Ones whose properties must be highly constrained, will be more costly to write and maintain.
- The language must be capable of expressing all kinds of computations. We should not throw away language features because they might make the type system unsound. Instead we should use the type system to *separate* sound features from unsound ones. The type system should clearly mark the boundaries between the two. It should spell out the obligations required to control the unsound features, and support and track how these obligations can be met.

A partial list of contemporary approaches to combining programming languages and logics include:

- **Cayenne.** Augustsson's Cayenne language[5,4] is a dependently typed version of Haskell. Like many dependently typed systems it does not distinguish between values and types.
- **Epigram.** McBride's Epigram[21] system is a contemporary attempt at combining practical programming language with reasoning system by using dependant types without a strict phase separation. The system is an integrated editor type-checker. The feel of using Epigram is more akin to using a proof editor, than writing a program.
- **RSP.** Stump's Rogue-Sigma-Pi[37,40] combines dependent types, general recursion, and imperative features in a type safe way. The emphasis is on an imperative, rather than a pure functional approach is emphasized.
- **DML.** Xi and Pfenning's Dependent ML[43,14] is an ML like language that allows the user to define type refinements. DML, like Ωmega separates types from values, but the set of indexed types cannot be extended by the programmer, and constraints are solved by a fixed decision procedure (rather than be user authored type level functions).
- **ATS.** Chen and Xi have developed the a paradigm they call Applied Type Systems[42,10,9]. It shares many of Ωmega's design goals, and is very closely related.
- **Constraint Handling Rules.** Martin Sulzman has applied his general mechanism for defining constrained type systems, Constraint Handling Rules, to build a system with similar goals[38,36]. His system has some nice completeness results which we need to learn more about.

The technical focus of this paper is to describe a previously unexplored mechanism for handling functions at the type level: narrowing. The non-technical focus is to introduce several non-trivial examples which illustrate how types can describe properties. These examples were also chosen because they work well with the narrowing approach. We will also discuss several problems for which the narrowing approach is not effective, and to suggest some future work.

3 Combining Programming Language and Logic in Ωmega

We have adopted the following structure for the Ωmega language. Ωmega is a language with an infinite hierarchy of computational levels: value, type, kind, sort, etc. Computation at the value level is performed by reduction, and is largely unconstrained. Computation at all higher levels is performed by narrowing, and is constrained in several ways. First, all "data" at the type level and above is inductively defined data (no floats or primitive data for example). Second, functions at the type level and above must be inductively sequential. This is a constraint on the form of the definition, not on the expressiveness of the language. See Section 8 for a detailed discussion of narrowing and these constraints.

Terms at each level are classified by terms at the next level. Thus values are classified by types, types are classified by kinds, kinds are classified by sorts, etc. We maintain a strict phase distinction – the classification of a term at level n cannot depend upon terms at lower levels. For example, no types can depend on values, and no kinds can depend on types. Programmers are allowed to introduce new terms and functions at every level, but any particular program will have terms at only a finite number of levels.

We formalize properties of programs by exploiting the Curry-Howard isomorphism. Terms at computational level n, are used as proofs about terms at level $n + 1$. We use indexed types to maintain a strict and formal connection between the two levels, and singleton types to maintain the strict separation between values and types (more about this in Section 5).

4 A simple example

To illustrate the hierarchy of computational levels we give the following two-level example. We introduce tree-like data (the natural numbers, `Nat`) at the type level by using the `kind` introduction form. We write a function at the type level over this data (`plus`). At the type level and higher, we distinguish function application from Constructor application by surrounding function application by braces (`{` and `}`).

```
kind Nat:: *1 where
  Z:: Nat
  S:: Nat ~> Nat

plus:: Nat ~> Nat ~> Nat
{plus Z m} = m
{plus (S n) m} = S {plus n m}
```

We then introduce data at the value level (`Seq`) using the `data` introduction form. The types of such values are indexed by the natural numbers. These indexes describe an invariant about the constructed values – their length appears in their type – consider the type of `l1`. Finally, we introduce a function at the value level over `Seq` values (`app`). The type of `app` describes one of its important properties – there is a functional relationship between the lengths of its two inputs, and the length of its output.

```
data Seq:: *0 ~> Nat ~> *0 where
  Snil :: Seq a Z
  Scons:: a -> Seq a n -> Seq a (S n)

l1 = Scons 3 (Scons 5 Snil) :: Seq Int (S(S Z))

app:: Seq a n -> Seq a m -> Seq a {plus n m}
app Snil ys = ys
app (Scons x xs) ys = Scons x (app xs ys)
```

Natural numbers at the type level have proved to be very useful. So useful, in fact, that we have added special syntactic sugar for constructing them. We sometimes write `#0` for `Z`, and `#1` for `(S z)`, and `#2` for `(S (S Z))`, etc. We may also write `#(1 + n)` for `(S n)`, and `#(2 + n)` for `(S (S n))`, etc. when `n` is a variable.

5 Introduction to Ωmega

Throughout this section we introduce the features of Ωmega by comparing and contrasting them with the features of Haskell. We assume a basic understanding of Haskell programs.

Feature: Kinds. Kinds introduce new tree-like data at the type level and higher. A kind declaration introduces both the constructors for the tree-like data and the object that classifies these structures. Consider the two new kinds `Set` and `Termination`:

```
kind Set:: *1 where
  Univ:: Set
  Deny:: Set
  Times:: Set ~> Set ~> Set
  Plus:: Set ~> Set ~> Set

kind Termination:: *1 where
  Total:: Termination
  Partial:: Termination
```

The new tree-like data at the type level are constructed by the type-constants (`Univ`, `Total`, and `Partial`), and type constructors (`Times`, and `Plus`). The kinds `Set` and `Termination` classify these structures, as shown explicitly in the declaration. For example `Univ` is classified by `Set`, and `Plus` is a constructor from `Set` to `Set` to `Set`. Think of the operator `~>` as an function arrow at the type level. Note that while `Univ`, `Total`, or `Partial` live at the type level, there are no values classified by them.

Instead, we use the structures as indexes to value level data, i.e. types like `(T Total)` and `(S Univ Int)`. These indexes will indicate static (type-checking time) properties of values. For example, a value with type `(T Total)` is statically guaranteed to have the `Total` property, and a value of type `(S Univ Int)` is statically guaranteed to have the `Univ` property. We will soon introduce two examples which follow this pattern.

The first example is a data structure representing expressions of some object language, where an Ωmega term of type `(Exp Total)` represents an object level term that always terminates when executed, and a term of type `(Exp Partial)` represents an object level term which may not terminate. Note that the index is a semantic invariant that appears in the type of every `Exp`. Judicious use of indexes in the definition of `Exp` can ensure that every well-typed term maintains the invariant.

The second example classifies values representing patterns by types like `(Pattern Univ)` or `(Pattern Deny)`. Semantically, a pattern is a predicate over some domain. I.e. some subset of that domain. For example, in Haskell we use patterns in case expressions. The pattern `(5,x)` describes elements of the domain of pairs of integers such as `(5,2)` and `(5,9)` but *not* `(3,9)` (because the pattern constant `5` is different from the value `3`).

We will use the `Set` index as a static invariant describing what sets in the domain are matched by a pattern. Consider a domain of simple objects, products, and sums. An index of `Univ` means the pattern matches every value. An index of `Deny` means their exists at least one value the pattern does not match, perhaps many. Indexes of `(Times a b)` means the pattern matches a product, and the left part of the product matches `a`, and the right part of the product matches `b`. Similar reasoning holds for disjoint sums with the `(Plus a b)` index.

Feature: GADTs. Generalized Algebraic Datatypes allow constructor functions to have more general types than the types supported by the `data` declaration in Haskell. GADTs are important because the additional generality allows programmer to express properties of types using type indexes and witnesses (or proof) objects. GADTs are the machinery that support the Curry-Howard isomorphism in Ωmega.

The `data` declaration in Ωmega defines generalized algebraic datatypes (GADT), a generalization of the algebraic datatypes available in Haskell. This is characterized by explicitly classifying constructors in a `data` declaration with their full types. The additional generality arises because the range of a constructor in a GADT is not constrained to be the type constructor applied to only type variables. For example consider the `Pattern` value level type:

```
data Pattern:: Set ~> *0 ~> *0 where
  PVar :: String -> Pattern Univ t
  PInt :: Int -> Pattern Deny Int
  PPair:: Pattern s1 a ->
          Pattern s2 b ->
          Pattern (Times s1 s2) (a,b)
  PWild:: Pattern Univ t
  PNil :: Pattern (Plus Univ Deny) [t]
  PCons:: Pattern s1 a ->
          Pattern s2 [a] ->
```

```
Pattern (Plus Deny (Times s1 s2)) [a]
```

Note, that instead of ranges like `(Pattern a b)`, where only type variables like `a` and `b` can be used as parameters, the ranges contain sophisticated instantiations such as `(Pattern Univ t)`. Note that the first index to `Pattern` (the one of kind `Set`) is used to describe an invariant about the meaning of a pattern. A variable pattern `(PVar "x"):: (Pattern Univ t)` matches all values of all types t. But, consider the nested pattern:

```
(PCons (PInt 4) (PVar "x")):: 
     Pattern (Plus Deny (Times Deny Univ)) [Int]
```

Its type tells us a a lot. It should be matched against terms of type `[Int]`. Such objects can be considered disjoint sums. The pattern definitely does not match a left injection (that is reserved for `Nil` objects). It matches some right injections (i.e. a `Cons` object) but not all. A `Cons` object can be considered a product. The right part of the product (the head of the list) only matches some things (i.e. integers with value 4, but not other integers such as 3 or 7). And the right part of the product (the tail) matches everything. All this is embedded in the type of the term! The GADT `Pattern` is an abstraction of the more complex GADT `Pat` we will see in Section 10.

Feature: Type functions. Kind declarations allow us to introduce new tree-like structures at the type level. We can use these structures to parameterize data at the value level as we did with `Pattern`, or we can compute over these tree like structures. Such functions are written by pattern matching equations, in much the same manner one writes functions over data at the value level. Several useful functions over types classified by `Termination` and `Set` are:

```
lub:: Termination ~> Termination ~> Termination
{lub Total x} = x
{lub Partial x} = Partial

allU:: Set ~> Termination
{allU Univ} = Total
{allU Deny} = Partial
{allU (Times x y)} = {lub {allU x} {allU y}}
{allU (Plus x y)} =  {lub {allU x} {allU y}}

union:: Set ~> Set ~> Set
{union Univ a} = Univ
{union (Times p1 p2) (Times q1 q2)} = Times {union p1 q1} {union p2 q2}
{union (Times a b) Deny} = Times a b
{union (Times a b) Univ} = Univ
{union (Plus p1 p2) (Plus q1 q2)} = Plus {union p1 q1} {union p2 q2}
{union (Plus p1 p2) Deny} = Plus p1 p2
{union (Plus p1 p2) Univ} = Univ
{union Deny x} = x
```

Like functions at the value level, the type functions `lub`, `allU` and `union` are expressed using equations. At the type level, we use brackets ({}) to surround function application to distinguish it clearly from constructor application at the type level (like (Tree Int) or (Pattern Univ Int)). The function lub is a binary function that combines two Terminations, and allU converts a Set into a Termination. It returns Total only if all parts of the Set are universal. Both functions are strict total (terminating) functions. The type function union unions two Sets.

Pattern: Witnesses. GADTs can be used to witness relational properties between types. This is because the parameters to types introduced using the GADT mechanism can play different roles. The natural number argument of the type constructor Seq (from Section 4) plays a qualitatively different role than type arguments in ordinary ADTs. Consider the declaration for a binary tree datatype:

```
data Tree a = Fork (Tree a) (Tree a) | Node a | Tip
```

In this declaration the type parameter a is used to indicate that there are sub components of Trees that are of type a. In fact, Trees are polymorphic. Any type of value can be placed in the "sub component" of type a. The type of the value placed there is reflected in the Tree's type. Contrast this with the n in (Seq a n). Instead, the parameter n is used to stand for an abstract property (the length of the list represented). When we use a type parameters in this way we call it a type index[41,44] rather than a type parameter.

We can use indexes to GADTs to define value level data that we can think of as proofs, or witnesses to type level properties. This is a powerful idea. Consider the introduction of two new parameterized types Even and LE. Note that these are ordinary data structures that exist at the value level, but describe properties at the type level.

```
data Even:: Nat ~> *0 where
  EvenZ:: Even Z
  EvenSS:: Even n -> Even (S (S n))
```

```
data LE:: Nat ~> Nat ~> *0 where
  LeZ:: LE Z n
  LeS:: LE n m -> LE (S n) (S m)
```

These declarations introduce value-level constants (EvenZ and LeZ) and constructor functions (EvenSS and LeS). Let's examine the types of several values constructed with these constructors.

```
EvenZ:: Even #0
(EvenSS EvenZ):: Even #2
(EvenSS (EvenSS EvenZ)):: Even #4
```

```
LeZ:: LE #0 a
(LeS LeZ):: LE #1 (S a)
(LeS (LeS LeZ)):: LE #2 (S(S a))
```

We write #0 for Z, and #1 for (S Z), etc. to emphasize that we should view LE as a relationship between two natural numbers. The important thing to notice is that we may view ordinary values with types (LE n m) and (Even n) as proofs, since the types of all legally constructed values witness only true statements about n and m. For example we cannot build a term of type (Even #1). This is the essence of the Curry-Howard isomorphism. We can view (EvenSS EvenZ):: (Even #2) as either the statement that (EvenSS EvenZ) has type (Even #2), or that (EvenSS EvenZ) is a proof of the property (Even #2). All this follows directly from the introduction of new types with tree-like structure, and the ability to *compute* over them.

Pattern: Singleton Types. Sometimes it is useful to direct computation at the type level, by writing functions at the value level. Even though types cannot depend on values, this is possible by the use of singleton types. The idea is to build a completely separate isomorphic copy of the type in the value world, but still retain a connection between the two isomorphic structures. This connection is maintained by indexing the value-world type with the corresponding type-world kind. This is best understood by example. Consider reflecting the kind `Nat` into the value-world by defining the type constructor `SNat` using a `data` declaration.

```
data SNat:: Nat ~> *0 where
  Zero:: SNat Z
  Succ:: SNat n -> SNat (S n)

three = (Succ (Succ (Succ Zero))):: SNat(S(S(S Z)))
```

Here, the value constructors of the `data` declaration for `SNat` mirror the type constructors in the `kind` declaration of `Nat`. We maintain the connection between the two isomorphic structures by the use of `SNat`'s natural number index. This type index is in one-to-one correspondence with the shape of the value. Thus, the type index of `SNat` exactly mirrors its shape. For example consider the example `three` above, and pay particular attention to the structure of the type index, and the structure of the value with that type.

We call such related types singleton types because there is only one element of any singleton type. For example only `Succ (Succ Zero)` inhabits the type `SNat(S (S Z))`. It is possible to define a singleton type for any first order type (of any kind). All Singleton types always have kinds of the form `I ~> *0` where `I` is the index we are reflecting into the value world. We sometimes call singleton types *representation types*. We cannot over emphasize the importance of the singleton property. Every singleton type completely characterizes the structure of its single inhabitant, and the structure of a value in a singleton type completely characterizes its type. Thus we can compute over a value of a singleton type, and the computation at the value level can express a property at the type level.

By using singleton types we completely avoid the use of dependent types where types depend on values[35,28].

Pattern: A pun: Nat'. We now define the type `Nat'`, which is in all ways isomorphic to the type `SNat`. The type `Nat'` is also a singleton type representing the natural numbers, but it relies on an anomaly of the Ωmega type system. In Ωmega (as in Haskell) the name space for values is separate from the name space for types. Thus it is possible to have the same name stand for two things. One in the value space, and the other in the type space. We exploit this ability by defining the pun `Nat'`.

```
data Nat':: Nat ~> *0 where
  Z:: Nat' Z
  S:: Nat' n -> Nat' (S n)

three' = (S(S(S Z))):: Nat'(S(S(S Z)))
```

The value constructors `(Z:: Nat' Z)` and `(S:: Nat' n -> Nat' (S n))` are ordinary values whose types mention the type constructors they pun. In `Nat'`, the singleton relationship between a `Nat'` value and its type is emphasized even more strongly, as witnessed by the example `three'`. Here the shape of the value, and the type index appear isomorphic.

We further exploit this pun, by extending the syntactic sugar for writing natural numbers at the type level (`#0`, `#1`, etc.) to their singleton types at the value level. Thus we may write `(#2:: Nat' #2)`.

Pattern: Computing Programs and Properties Simultaneously. We can write programs that compute an indexed value along with a witness that the value has some additional property. For example, when we add two static length lists, the resulting list has a length that is related to the lengths of the two input lists, and we can simultaneously produce a witness to this relationship.

```
data Sum:: Nat ~> Nat ~> Nat ~> *0 where
  PlusZ:: Sum Z m m
  PlusS:: Sum n m z -> Sum (S n) m (S z)

app1:: Seq a n -> Seq a m -> exists p . (Seq a p,Sum n m p)
app1 Snil ys = Ex(ys,PlusZ)
app1 (Scons x xs) ys =
   case (app1 xs ys) of { Ex(zs,p) ->  Ex(Scons x zs,PlusS p) }
```

`Ex` is the "pack" operator [8]. It turns a normal type `(Seq a p,Plus n m p)` into an existential type `(exists p.(Seq a p,Plus n m p))`.

Feature: Tags and Labels. Many systems have a notion of label. Usually, labels are unique names that support a notion of equality. Many times it is convenient to think of every label has having unique type in a family of related types. As a *first approximation*, consider the finite kind `Tag` and its singleton type `Label`:

```
kind Tag = A | B | C | D
```

```
data Label:: Tag ~> *0 where
  A:: Label A
  B:: Label B
  C:: Label C
  D:: Label D
```

Here, we again deliberately use the value/type name space overloading we used in the example `Nat'`. The names `A`, `B`, and `C` are defined in both the value and type name spaces. They name different, but related objects in each space. At the value level, every `Label` has a type index that reflects its value. I.e. `A::Label A`, and `B::Label B`, and `C::Label C`. So in the value name space `A` is an ordinary value, and because it is a singleton, its type `(Label A)` reflects its structure. In the type `(Label A)`, the `A` is different (but related) object that lives at the type level, and is classified by the kind `Tag`. This is a very useful pun. Every label at the value level has a different type. Thus we can distinguish statically that two labels differ (or are the same) by observing their type.

The problem with the approach above is that there are only four different labels. We would like a countably infinite set of labels. We can't define this explicitly, but we can build such a type as a primitive inside of Ωmega. At the type level, every legal identifier whose name is preceded by a back-tick (‘) is a type classified by the kind `Tag`. For example, at the type level `‘abc` is classified by `Tag`. At the value level, every such symbol `‘abc` is also reflected, but here it is classified by the singleton type `(Label ‘abc)`. We write: `‘abc :: Label ‘abc :: Tag`. We say the value `‘abc` is classified by the singleton type `(Label ‘abc)` which is classified by the kind `Tag`. Labels are singleton types, and as such their values are uniquely determined by their type. Their values are reflected in their types. This is a powerful mechanism we will exploit in the next section.

Pattern: Binding in object languages. Binding of variables is a perennial problem when modeling rich languages. Our approach to binding is to use a two prong approach, using both labels and de Bruijn indices. Such an approach is related to the notion of record structures. To define records, we proceed in two steps. We define a new kind at the type level called rows, then we define a new data structure (records) at the value level that is classified by rows.

```
kind Row:: *1 ~> *1 where
  RNil:: Row a
  RCons:: a ~> Row a ~> Row a
kind HasType = Has Tag *0
```

Rows are nothing more than polymorphic list like structures at the type level. Examples of `Rows` include: `(RCons Int (RCons Bool RNil))` a row of types and `(RCons #5 RNil)` a row of `Nat`, and `(RCons (Has ‘name String) RNil)` a row of `HasType`. Such types are kinded by `Row`. The kind `Row` is a higher order kind, and is classified by `(*1 ~> *1)`. Thus `Row` must be applied to a kind (such as `*0`, `Nat`, or `HasType`) to be well formed (the argument indicates the kind of the types stored in the row).

A `HasType` is nothing more than a pair of types. The the first component of the pair must be a `Tag` and the second component must be an ordinary type (like `Int` or `String`). To define record data at the value level we proceed as follows.

```
data Record:: Row Hastype ~> *0 where
  RecNil :: Record RNil
  RecCons:: Label a -> b -> Record l -> Record (RCons (Has a b) l)

record1 :: Record (RCons (Has ‘name String) (RCons (Has ‘age Int) RNil))
record1 = RecCons ‘name "Tim" (RecCons ‘age 21 RecNil)
```

Record data is constructed by applying the constructors `RecCons` and `RecNil` to appropriate values. See the example `record1` above. Note the use of labels (`‘name` and `‘age`) at the value level, whose kinds are reflected at the type level in the record type.

Labeled de Bruijn indices are implemented by using exactly the same structure. We build a value level data structure `Exp` indexed by `(Row HasType)`. This index uniquely identifies the variables appearing in the term, in the terms type. If the language has binding structure (i.e. any kind of scoping mechanism like lambda or let), then the binding location of a variable is determined by its location in the environment (this is what makes it de Bruijn-like). For example let the value level data `Exp` have two indexes: `Exp:: Row HasType ~> *0 ~> *0`. Interpret a term with type `(Exp r t)` as a term with type `t` whose free variables are described by `r`. For example, consider a term with the following type:

```
Exp (RCons (Has ‘f (b -> t)) (RCons (Has ‘x b) d)) t
```

This term has two free variables, named by `‘f` and `‘x`, with types `(b -> t)` and `b`, respectively. This term should occur inside a larger context where the variable `‘f` is the most closely nested variable, and the variable `‘x` is the the second most closely nested variable. This hybrid approach accomplishes two things. The labels are there for the reader, but the location or de Bruijn index is what counts. This makes alpha-renaming unnecessary when formally manipulating the terms, but the labels provides the reader with some visual clues when inspecting terms and their types. `Tags`, `Labels`, `Rows`, and `Hastype` build objects that look very much like typing environments.

`Row` and `HasType` are so useful we have built special syntactic sugar for printing them. For example, `Rec(RCons (Has ‘x Int) (RCons (Has ‘a Bool) RNil))` prints as `Rec {‘x:Int,‘a:Bool}`. The syntactic sugar for `Row` and `HasType` replaces `RCons` and `RNil` with squiggly brackets, and replaces `Has` with colon. A type classified by `Row` whose (ultimate) tail is not `RNil` (i.e. a type variable) prints with a trailing semi-colon. For example, `Rec(RCons (Has ‘x Int) (RCons (Has ‘a Bool) w))` prints as `Rec {‘x:Int,‘a:Bool; w}`. Using this notation, our earlier example looks like: `Exp {‘f: b->t, ‘x: b; d} t`

Now that we understand the mechanism, lets use it to define object-languages with binding structures that track their free variables in their meta-level types. The object-language `(Lam env t)` represents the simply typed lambda calculus.

```
data Lam:: Row HasType ~> *0 ~> *0  where
  Var   :: Label s -> Lam (RCons (Has s t) env) t
  Shift :: Lam env t -> Lam (RCons (Has s q) env) t
  Abs   :: Label a -> Lam (RCons (Has a s) env) t -> Lam env (s -> t)
  Apply :: Lam env (s -> t) -> Lam env s -> Lam env t
```

The first index to `Lam`, `env` is a Row tracking its variables, and the second index, `t` tracks the object-level type of the term. For example, a term with variables `x` and `y` might have type `Lam {‘x:Int, ‘y:Bool; u} Int`. The key to this approach is the use of `Row` and `Hastype` in the typing of functions (`Abs`) and variables (`Var`). Consider the `Var` constructor function. To construct a variable we simply apply `Var` to a label, and its type reflects this. For example, here is the output from a short interactive session with the Ωmega interpreter.

```
prompt> Var ‘name
(Var ‘name)::
forall a (u:Row HasType) . Lam {‘name:a; u} a

prompt> Var ‘age
```

```
(Var ‘age)::
forall a (u:Row HasType) . Lam {‘age:a; u} a
```

Variables behave like Bruijn indices. Variables created with `Var` are like the natural number 0. A variable can be lifted to the next natural number by the successor operator `Shift`. To understand why this is useful consider that the two examples have different names in the same index position. The two variables would clash if they were both used in the same lambda term. To shift the position of variable to a different index, we use the `Shift:: Lam u a -> Lam {v:b; u} a` constructor. Rather than counting with natural numbers (as is done with de Bruijn indices) we "count" with rows, recording both its symbolic name and its type. Here is how we could define two variables `x` and `y` for use in the same environment.

```
x :: Lam {‘x:a; u} a
x = Var ‘x

y :: Lam {u:a,‘y:b; v} b
y = (Shift (Var ‘y))
```

The type system now tracks the variables in an expressions.

```
z :: Lam {‘x:a -> b,‘y:a; u} b
z = (Apply (Var ‘x) (Shift (Var ‘y)))
```

We have found many other useful patterns that exploit the features of Ωmega. There are also several features of Ωmega we have not discussed, notably the use of witness types as static constrains and the constrained type system that manages them. But in the rest of this paper, we focus our attention on the uses of narrowing in Ωmega.

6 The structure of type checking

Ωmega uses a combination of type checking and type inference. Type inference, as implemented in Ωmega, can only assign Hindley-Milner types to terms, and only works over algebraic datatypes (as opposed to generalized algebraic datatypes) so type checking is the normal mode of typing in many Ωmega programs.

During type checking, information can flow in two directions. We can compute the type of a term, then check if it is consistent with its declared type, or we can use the declared type to suggest a type for a term, and then traverse the term computing a type consistent with that suggestion.

In some systems every term has exactly one principal type, all other types for that term are "instances" of that principal type. But such systems may be too restrictive. Checking discovers if a term has a particular type, even when it might also have additional types. The type declaration states which of the multiple types should be chosen as the type for that term. Thus every term is assigned a single type (and hence a single meaning), even if it can have more than one type.

When type checking we often ask if two types (b and c) are mutually consistent

$(b \approx c)$. Consider a "generic" type rule for function application.

$$\frac{\Gamma \vdash f : c \to d \quad \Gamma \vdash x : b \quad b \approx c}{\Gamma \vdash f \;\; x : d}$$

Type checkers perform computation to answer consistency problems. Different languages ask different kinds of questions. Some questions have simple yes/no answers, others search to find particular values meeting some criteria, or reduce a complicated term into a simple normal form. For example, in a monomorphic language like Pascal, mutually consistent means structurally equal. In a polymorphic language like Haskell, mutually consistent means that b and c are unifiable. In a language with subtyping, like Java, mutually consistent means that b is a subtype of c. In Ωmega, consistent means semantically equivalent. Because of the use of type level functions, syntactic equivalence is no longer sufficient. Types are no-longer static free algebras, but dynamic computations. A very simple example of this phenomena is the `app` function from Section 4. Consider type-checking the second clause.

```
app:: Seq a n ->    Seq a m -> Seq a {plus n m}
app   (Scons x xs)  ys      =  Scons x (app xs ys)
```

From the declaration, we know the pattern `(Scons x xs)` has type `(Seq a (S b))` where (`n = (S b)`), so we must show that the right-hand-side has the type: `Seq a {plus (S b) m}`. Computing the type of the right-hand-side we get: `Scons x (app2 xs ys):: Seq a (S{plus b m})`. The types `(Seq a {plus (S b) m})` and `(Seq a (S {plus b m}))` are consistent only if we can find bindings for `b` and `m` that make `{plus (S b) m}` semantically equal to `(S {plus b m})`. If we allow computation at the type level we will need a computational mechanism that can answer equivalence questions such as `(Seq a {plus (S b) m})` ≈ `(Seq a (S {plus b m}))`.

We know of several choices. First, one can use a combination of reduction and unification. Second, one can use entailment. Third, one can use constraint solving, and Fourth, one can use narrowing.

- **Reduction & unification**. To check for consistency, we reduce both terms to normal form and then unify the normal forms. This is the approach taken by Morrisett [13]. Unfortunately, simple consistency checks such as (`{plus b m}` ≈ `m`) cannot be answered since both sides are in normal form, and unifying does not find the correct solution `b` = `Z`.
- **Entailment**. By expressing computation at the type level logically, we can use entailment mechanisms (such as resolution or higher order pattern matching) to solve consistency checks. This is the case in the logical framework Twelf [26]. This choice is inconsistent with our desire for uniformity in the mechanism used to define functions at the value and type levels.
- **Constraint Solving**. General purpose constraint solving systems can be used to decide equivalence. One example, where constraint solvers is applied to type checking is Sulzman's CHR system[38,36].
- **Narrowing**. Narrowing combines the power of reduction and unification. Narrowing is a primary computational mechanism of the functional logic language

Curry [17,16]. Unlike unification the terms being compared can contain functions, and unlike reduction the terms being simplified can contain variables. Narrowing finds bindings for some free variables in its terms. Narrowing is a special purpose constraint solving system, that meshes quite well with the equivalence questions asked when there are functions at the type level.

One of our design goals is *uniformity*. The beauty of an integrated system is that users need learn only one tool. If it has many different modes, each requiring different skills, then the benefit of integration is lost. So even if we use a different mechanism to execute programs at compile-time and run-time, we want the user interface to the two to be the same. Narrowing is an excellent choice for execution at type checking time. The interface for both run-time and compile-time computation is the writing of recursive equations.

7 How Narrowing works

Narrowing combines the power of reduction and unification. Narrowing finds bindings for some free variables in the term being narrowed, once instantiated, these bindings allow the term to reduce to a normal form.

If a term contains constructors, function symbols, and variables, it often cannot be reduced. Usually because function calls within the term do not match any left-hand side of any of their definitions. The failure to match is caused by either variables or other function calls in positions where the function definitions have only constructor patterns. Consider narrowing `(plus a Z == Z)` (checking whether `{plus a Z}` is equal to `Z`).

This cannot be reduced because `plus` inducts over its first argument with the patterns `Z` and `(S n)`. But in `(plus a Z == Z)`, the first argument position is a variable `a`. Narrowing proceeds by guessing instantiations for the variable `a` – either `{a -> Z}` or `{a -> (S m)}`, and following both paths.

`plus:: Nat ~> Nat ~> Nat`	guess `{a -> Z}`	guess `{a -> (S m)}`
`{plus Z m} = m`	`({plus Z Z} == Z)`	`({plus (S m) Z} == Z)`
`{plus (S n) m}= S {plus n m}`	`(Z == Z)`	`(S {plus m Z} == Z)`
	Success !	Failure.

The returned solutions are the bindings obtained in every successful path. In the example above we get a list of one solution: `[{a -> Z}]`. Some problems have no solutions, some have multiple, and some even have infinite solutions. Consider `{plus x #2}`, we get `#2` when `{x -> #0}`, and `#3` when `{x -> #1}`, and `#4` when `{x -> #2}` etc.

Narrowing works best when we have a problem with many constraints. The constraints prune the search path resulting in few solutions. If we're type checking with narrowing we hope there is exactly one solution. Consider narrowing `({plus x #3} == #5)`, Guessing `{x -> 0}` and `{x -> S z1}` we get the two paths:

```
1)    { #3 == #5 }
2)    { #(1+{plus z1 #3}) == #5 }
```

The first path fails, on the second path we take a single reduction step leaving:

```
{ {plus z1 #3} == #4 }
```

Guessing `{z1 -> 0}` and `{z1 -> S z2}` we get the two paths:

```
1)   { #3 == #4 }
2)   { #(1+{plus z2 #3}) == #4 }
```

The first path fails, on the second path we again take a single reduction step leaving:

```
{ {plus z2 #3} == #3 }
```

Guessing `{z2 -> 0}` and `{z2 -> S z3}` we get the two paths:

```
1)   { #3 == #3 }
2)   { #(1+{plus z3 #3}) == #3 }
```

The first succeeds, and the second eventually fails, leaving us with only one solution `{ x -> #2 }`. We have found narrowing to be a efficient and understandable mechanism for directing computation at the type level.

8 Narrowing Strategies

While narrowing is non-deterministic, it is both sound and complete with an appropriate strategy[3]. All answers found are real answers, and if there exists an answer, good strategies will find it. When a question has an infinite number of answers, a good implementation will produce these answers lazily. In our type-checking context, finding 2 or more answers is a sign that a program being type checked has an ambiguous type and needs to be adjusted. In the rare occurrence that narrowing appears to diverge on a particular question, we can safely put resource bounds on the narrowing process, declaring failure if the resource bounds are exceeded. The consequence of such a declaration, is the possibility of declaring a well- typed function ill-typed. In our experience this rarely happens.

We restrict the form of function definitions at the type level to be inductively sequential[1]. This ensures a sound and complete narrowing strategy for answering type-checking time questions. The class of inductively sequential functions is a large one, in fact every Haskell function has an inductively sequential definition. The inductively sequential restriction affects the form of the equations, and not the functions that can be expressed. Informally, a function definition is inductively sequential if all its clauses are non-overlapping. For example the definition of `zip1` is not-inductively sequential, but the equivalent program `zip2` is.

```
zip1 (x:xs) (y:ys) = (x,y): (zip1 xs ys)
zip1 xs ys = []

zip2 (x:xs) (y:ys) = (x,y): (zip2 xs ys)
zip2 (x:xs) []     = []
zip2 []     ys     = []
```

The definition for `zip1` is not inductively sequential, since its two clauses overlap. In general any non-inductively sequential definition can be turned into an inductively sequential definition by duplicating some of its clauses, instantiating variable patterns with constructor based patterns. This will make the new clauses non-overlapping. We do not think this burden is to much of a burden to pay, since it is applied only to functions at the type level, and it supports sound and complete narrowing strategies.

We pay for the generality of narrowing over unification and reduction by a modest increase in overhead. Narrowing uses a general purpose search algorithm rather than a special purpose unification or reduction engine. Narrowing is Turing complete, so we can solve any problem that can be solved by reduction, and many more.

9 Narrowing and type checking

Ωmega uses a combination of type inference and type checking. Type inference is useful when we are writing programs that do not utilize GADTs or type-level computation. I.e. all the programs we used to write in Haskell. When we use either of these new features Ωmega uses a type checking approach. If a function is given a prototype, declaring its type, it is type checked. If it not given a prototype, we attempt to infer a Hindley-Milner type for it. Our type system is based in large part upon the system described in the paper *Simple unification-based type inference for GADTs* by Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. We also borrow ideas from the paper *Practical type inference for arbitrary-rank type*[18] by Simon Peyton Jones and Mark Shields. Type level functions and narrowing are accommodated with the following extensions.

- In both papers, type checking is described as a monadic computation over the structure of program terms. We strengthen the monad so it is an accumulating monad. I.e. a monad in which a computation can emit an equality constraint between a type and a type function. All such emissions are accumulated into a list of equalities as the computation proceeds.
- Equality constraints between types and type functions are emitted during unification when an attempt is made to unify two terms, at least one of which is a function application at the type level. This is the only source of such constraints.
- At generalization points (at `let` and at top level) the constraints are solved by narrowing. The resulting set of bindings is a unifier, and this is composed with the unifier obtained in the normal type checking process.
- If narrowing fails to return a binding, type checking fails. If narrowing returns more than one binding, the type is ambiguous. Further use of type annotations can often be chosen to pick a unique type from amongst the set of possible types.

10 Tracking termination: An example with lots of narrowing

It is important for proof objects in Ωmega to attest to true things. If Ωmega is regarded as a logic, it should be sound. The only inhabitant of witnesses and proofs should be one of the well formed objects built from its constructors. This requires that expressions with witness types never result in non-terminating or error computations. As we stated in the introduction we hope to use the type system to separate error producing programs from total ones. In this section we introduce an example which is both a rich source of narrowing examples and a framework for future extensions to Ωmega that will separate terminating programs from possibly non-terminating ones using types.

Our strategy is to statically compute both a termination behavior and a type for every Ωmega term. For this we use the kinds `Set` and `Termination` introduced earlier. To track termination across function calls, the type constructor arrow is tagged with a termination behavior. Note that the arrow type constructor would have kind `(->):: *0 ~> Termination ~> *0 ~> *0`. In this future version of Ωmega, we will either write `((->) a b c)` or `(a -b-> c)`. For example:

```
add:: Int -Total-> Int -Total-> Int
divide:: Int -Total-> Int -Partial-> Int
```

where `(-Total->)` is a terminating function, and `(-Partial->)` is a possibly divergent function.

In some preliminary work, which we describe below, we have built a rich model in Ωmega of this future type system for Ωmega. It captures most of the language features in Ωmega, including pattern matching, algebraic data, primitive functions, and recursion. It makes heavy use of indexed GADTs, type functions, and narrowing.

Our strategy is to track both the type and termination behavior as an index of a GADT representing programs terms. An expression of type `(Exp t env p)` is interpreted as representing a program with type `t` in an environment with shape `env`, with termination behavior `p`. An expression with type `(Pat c t e1 e2)` is interpreted as representing a pattern with type `t`, which matches the set of values `c` (a `Set`), and maps an environment with shape `e1`, to one with shape `e2`. Non-termination stems from three causes: non-exhaustive pattern matching, the use of the `undefined` term (think of this as a call to an error function), and recursive functions. The specification makes extensive use of functions at the type level and of narrowing. In our Ωmega model we write `(Arr dom t rng)` for `(dom -t-> rng)`.

```
data Pat:: Set ~> *0 ~> Row HasType ~> Row HasType ~> *0 where
  Pvar :: Label a -> Pat Univ t r (RCons (Has a t) r)
  Ppair:: Pat s1 a g1 g2 -> Pat s2 b g2 g3 -> Pat (Times s1 s2) (a,b) g1 g3
  Pwild:: Pat Univ t r r
  Pnil :: Pat (Plus Univ Deny) [a] r r
  Pcons:: Pat s1 a g1 g2 ->
          Pat s2 [a] g2 g3 ->
          Pat (Plus Deny (Times s1 s2)) [a] g1 g3


data Exp:: *0 ~> Row HasType ~> Termination ~> *0 where
  Var::   Label a -> Exp t (RCons (Has a t) r) Total
  Shift:: Exp t g m -> Exp t (RCons (Has a b) g) m
```

```
  Const:: Int -> Exp Int g Total
  Pair::  Exp x g m -> Exp y g n -> Exp (x,y) g {lub m n}
  App::   Exp (Arr x m b) g n -> Exp x g p -> Exp b g {lub {lub p n} m}
  Abs::   Cls xx (Arr a i b) g1 -> Exp (Arr a {lub {allU xx} i} b) g1 tt
  Nil::   Exp [a] g Total
  Cons::  Exp a g m -> Exp [a] g n -> Exp [a] g {lub m n}
  Error:: Exp a g Partial
  Let::   Label f ->
          Exp (Arr a Partial b) (RCons (Has f (Arr a Partial b)) r) m ->
          Exp c (RCons (Has f (Arr a Partial b)) r) n ->
          Exp c r {lub n m}

data Cls:: Set ~> *0 ~> Row HasType ~> *0 where
  Last:: (Pat s1 a g1 g2,Exp b g2 m) -> Cls s1 (Arr a m b) g1
  Next:: (Pat s1 a g1 g2,Exp b g2 m) ->
         Cls s (Arr a n b) g1 ->
         Cls {union s1 s} (Arr a {lub m n} b) g1
```

Note as in the `Lam` example we have used `(Row Hastype)` to track the shape of environments. Second, note the use the type functions `lub`, `allU`, and `union` to track termination behavior. The latter gives rise to a number of interesting narrowing problems. First lets see how the types of the constructors track both the type and the termination behavior of the term they construct. Consider:

```
--  (fn [] => 0 | (x:_) => x)

(Abs (Next (Pnil,(Const 0))
     (Last ((Pcons (Pvar `x) Pwild),(Var `x)))))

:: Exp (Arr [Int] Total Int) a Total
```

In the comment we give an ML-style lambda with multiple pattern matching clauses to describe the concrete syntax of the term. Next we give its actual construction in terms of the constructors of `Exp`, and finally we give its computed type. To compute this type, we need to solve several narrowing problems. First, study the types of the four components. The two patterns, and the two right-hand sides.

```
Pnil::                   Pat (Plus Univ Deny) [a] b b
Pcons (Pvar `x) Pwild:: Pat (Plus Deny (Times Univ Univ)) [c] d {`x:c; d}
Const 0::                Exp Int e Total
Var `x::                 Exp f {`x:f; g} Total
```

The type rules for `Next` and `Last` indicate we need `union` the `Sets` of the two patterns: `{union (Plus Univ Deny) (Plus Nil (Times Univ Univ))}` to get `(Plus Univ (Times Univ Univ))`. Next we must `lub` the termination behavior of the two right hand sides `{lub Total Total}` to get `Total`. So the sub term that is the argument to `Abs` has the type: `Cls (Plus Univ (Times Univ Univ)) (Arr [Int] Total Int) a`

The type of `Abs` then narrows `{allU (Plus Univ (Times Univ Univ))}` to get

`Total`. This tells us that the patterns are exhaustive. Combined with the totality of the right-hand sides we narrow `{lub Total Total}` to get `Total`. Hence we arrive at the type of the term: `(Exp (Arr [Int] Total Int) a Total)`. The free type variable `a` tells us the term has this type in any environment (because it is a closed term with no free variables).

We can track non-termination from non-exhaustive patterns, the use of error, or recursion.

```
--  (fn [] => 0 | (x:(y:ys)) => x)
(Abs (Next (Pnil,(Const 0))
     (Last ((Pcons (Pvar ‘x) (Pcons (Pvar ‘y) (Pvar ‘ys))
:: Exp (Arr [Int] Partial Int) a b

-- (fn _ => error "oops")
(Abs (Last (Pwild,Error))) :: Exp (Arr a Partial b) c d

-- letrec f = (fn x => 1 + (f x))  in (f 3)
Let ‘f (abs ‘x (App (App (Oper Add) (Const 1))
                    (App (Shift (Var ‘f)) (Var ‘x))))
            (App (Var ‘f) (Const 3))
:: Exp Int g Partial
```

With this small language, every well-typed meta-level expression computes both the type and a termination behavior of the object program it represents. The computed termination is exact, except in the case of recursion where it over approximates, declaring every recursive function to be partial. One of the research question is how to build such a model into the Ωmega implementation, and how to address the fact that it is too conservative for recursive functions. Our approach will be to assume all recursive programs are partial unless the user indicates they are not by giving a total function type in the functions prototype. Then we will incorporate more precise strategies for modelling the termination behavior of recursion[19,6,7].

11 Ambiguous types

Its possible to compute ambiguous solutions while narrowing. When we were exploring the design space for tracking termination, we initially declared the types of the variable and application constructor of `Exp` as follows.

```
data Exp:: *0 ~> Row HasType ~> Termination ~> *0 where
  Var::   Label a -> Exp t (RCons (Has a t) r) p
  App::   Exp (Arr x m b) g n -> Exp x g p -> Exp b g {lub {lub p n} m}
  . . .
```

The idea was that we could make the termination behavior of a variable polymorphic (the `p` type variable in `(Exp t (RCons (Has a t) r) p)`) rather than `Total`. After all a polymorphic termination could be `Total` if we needed it to be. Unfortunately, this under constrains the problem. For example consider declaring the term (in concrete syntax (`f y`)) to be partial.

```
ambig:: Exp Int
            (RCons (Has `f (Arr Int m Int)) (RCons (Has `y Int) RNil))
            Partial
ambig = App (Var `f) (Shift (Var `y))
```

Here the partiality of the term could arise from either the `f` or the `y`. Typing this term requires solving the narrowing problem `{ {lub {lub a b} c} == Partial }`. But there are two solutions to this problem `{ a -> Total, b ->Partial }` and `{ a -> Partial }`. The type is ambiguous and is disallowed. In a context where either the `f` or the `y` is known (either `Total` or `Partial`), there is a unique solution. It is better to tightly constrain the problem. After all, in a strict language like Ωmega, every variable is total.

Another kind of ambiguity problem also sometimes arises when type checking. Consider narrowing `{ {lub x x} == x }`. This has two solutions `{ x -> Total }` and `{ x -> Partial }`. In some sense, since these two bindings exhaustively cover the space of possible bindings for `x`, the identity substitution (binding no variables) is also a solution. In some sense the ambiguity introduced by narrowing in this example is spurious.

We can also have spurious ambiguity in an infinite set of solutions. Consider narrowing `{ {plus n #0} == n }`. This is true for all `n`, and narrowing finds an infinite set of bindings `{ x -> #0 }`, `{ x -> #1 }`, `{ x -> #2 }` etc. But, just as in the previous example this set of bindings exhaustively covers the space of all possible bindings, so the identity substitution is also a solution.

After writing and type checking many programs, we find that the spurious ambiguity problem arising from an infinite number of soultions is the biggest weakness of using narrowing to perform compile-time computation. To avoid this problem we could recoginize it when it occurs, and then generalize the infinite number of solutions to a unique more general solution. In essence, recognizing when this occurs, is a theorem proving problem. The is an area for future research, but there is some hope.

In the paper, *Narrowing the Narrowing Space*[2], Antoy and Ariola identify the same problem and propose an ingenious solution. By memoizing previous narrowing steps, narrowing can recognize instances of a previous goals. The algorithm can build a graph representation of the narrowing space. They show that such a graph is a finite state automaton, and hence can be reformulated as a regular expression. In regular expression form, the spurious ambiguous solutions are recognizable and can be removed. Antoy reports that he knows of no implementation of this result. We are evaluating whether to build one.

12 Is narrowing effective?

Narrowing has often been legitimately criticized for two reasons. (1) Problems for which narrowing is appropriate involve search. If the search space is too large, the performance of narrowing degrades. The search space becomes unmanageable when the number of variables becomes too large, and when the search space is not effectively pruned by constraints on the variables. (2) Narrowing is not applicable to computations over non-algebraically defined data (such as arithmetic over floating point numbers). Neither of these criticisms hold in the context of using narrowing to answer type-checking questions. Type-checking questions are always over inductively defined algebraic data, and the number of free variables in a type-checking question is related to the size of the program being type checked. People do not write programs that generate large narrowing problems. Typical questions have only a handful of variables, well within the range of effective solution using narrowing.

Narrowing is a powerful technique whose power we have just begun to explore. It seems to provide a good tradeoff between expressive power and amenablility to equation solving for writing functions at the type level, and it has a well studied theory upon which we can rely. In addition, it also allows programmers to use a uniform means of expressing computation at all levels – the writing of recursive equations.

References

[1] S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157, Volterra, Italy, September 1992. Springer LNCS 632.

[2] S. Antoy and Z. M. Ariola. Narrowing the narrowing space. *Lecture Notes in Computer Science*, 1292:1–??, 1997. 9th International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97).

[3] Sergio Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, July 2005.

[4] Lennart Augustsson. Cayenne — a language with dependent types. *ACM SIGPLAN Notices*, 34(1):239–250, January 1999.

[5] Lennart Augustsson. Equality proofs in cayenne, July 11 2000.

[6] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, New York, 1979.

[7] Jürgen Brauburger. *Automatic Termination Analysis for Functional and Imperative Programs*. PhD thesis, Technische Universität Darmstadt, 1999.

[8] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACMCS*, 17(4):471–522, December 1985.

[9] Chen and Xi. Combining programming with theorem proving. In *Proceedings of the 2005 ACM/SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 66 – 77, September 2005.

[10] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *ICFP 2005*, 2005. http://www.cs.bu.edu/ hwxi/.

[11] Catarina Coquand. Agda is a system for incrementally developing proofs and programs. Web page describing AGDA: `http://www.cs.chalmers.se/~catarina/agda/` .

[12] T. Coquand and P. Dybjer. Inductive definitions and type theory an introduction (preliminary version). *Lecture Notes in Computer Science*, 880:60–76, 1994.

[13] Karl Crary, Stephanie Weirich, and J. Gregory Morrisett. Intensional polymorphism in type-erasure semantics. In *ICFP*, pages 301–312, 1998.

[14] Rowan Davies. A refinement-type checker for Standard ML. In *International Conference on Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[15] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. *Lecture Notes in Computer Science*, 1581:129–146, 1999.

[16] Michael Hanus and Ramin Sadre. An abstract machine for curry and its concurrent implementation in java. *Journal of Functional and Logic Programming*, 1999(Special Issue 1), 1999.

[17] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, March 28, 2006.

[18] Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types. Technical report, Microsoft Research, "December" 2003. `http://research.microsoft.com/Users/simonpj/papers/putting/index.htm`.

[19] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92, 2001.

[20] Zhaohui Luo and Robert Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, Computer Science Dept., University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, May 1992. Updated version.

[21] Connor McBride. Epigram: Practical programming with dependent types. In *Notes from the 5th International Summer School on Advanced Functional Programming*, August 2004. Available at: `http://www.dur.ac.uk/CARG/epigram/epigram-afpnotes.pdf` .

[22] Bengt Nordstrom. The ALF proof editor, March 20 1996.

[23] Emir Pasalic. *The Role of Type Equality in Meta-programming*. PhD thesis, OGI School of Science & Engineering at OHSU, October 2004. Available from: http://www.cs.rice.edu/~pasalic/thesis/body.pdf.

[24] Emir Pasalic and Nathan Linger. Meta-programming with typed object-language representations. In *Generative Programming and Component Engineering (GPCE'04)*, pages 136 – 167, October 2004. LNCS volume 3286.

[25] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[26] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide*, 1.2 edition, September 1998. Available as Technical Report CMU-CS-98-173, Carnegie Mellon University.

[27] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *LNAI*, pages 202–206, Berlin, July 7–10, 1999. Springer-Verlag.

[28] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. *ACM SIGPLAN Notices*, 37(1):217–232, January 2002.

[29] Tim Sheard. Languages of the future. *Onward Track, OOPSLA'04. Reprinted in: ACM SIGPLAN Notices, Dec 2004*, 39(10):116–119, October 2004.

[30] Tim Sheard. Playing with types. Technical report, Portland State University, 2005. `http://www.cs.pdx.edu/~sheard`.

[31] Tim Sheard. Putting Curry-Howard to work. In *Proceedings of the ACM SIGPLAN 2005 Haskell Workshop*, pages 74–85, 2005.

[32] Tim Sheard, James Hook, and Nathan Linger. GADTs + extensible kind system = dependent programming. Technical report, Portland State University, 2005. `http://www.cs.pdx.edu/~sheard`.

[33] Tim Sheard and Nathan Linger. Programming with static invariants in omega, September 2004. Available from: `http://www.cs.pdx.edu/~sheard/` .

[34] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Logical Frameworks and Meta-Languages workshop*, July 2004. Available at: `http://cs-www.cs.yale.edu/homes/carsten/lfm04/`.

[35] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, Boston, Massachusetts, January 19–21, 2000.

[36] Peter J. Stuckey and Martin Sulzmann. Type inference for guarded recursive data types, February 2005. Available from: `http://www.comp.nus.edu.sg/~sulzmann/` .

[37] Aaron Stump. Imperative LF meta-programming. In *Logical Frameworks and Meta-Languages workshop*, July 2004. Available at: `http://cs-www.cs.yale.edu/homes/carsten/lfm04/`.

[38] Martin Sulzmann and Meng Wang. A systematic translation of guarded recursive data types to existential types, February 2005. Available from:
`http://www.comp.nus.edu.sg/~sulzmann/` .

[39] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 7.4*. INRIA, 2003. http://pauillac.inria.fr/coq/doc/main.html.

[40] Edwin Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct inperative programming. Technical report, Washington University in St. Louis, 2005. Available at: `http://cl.cse.wustl.edu/`.

[41] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1997.

[42] Hongwei Xi. Applied type systems (extended abstract). In *In post-workshop proceedingds of TYPES 2003*, pages 394 – 408, 2004. Springer-Verlag LNCS 3085.

[43] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *ACM SIGPLAN Notices*, 33(5):249–257, May 1998.

[44] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 214–227, New York, NY, USA, 1999. ACM Press.