Type Inference for the Spine View of Data

Matthew Roberts
Macquarie University
matthew.roberts@mq.edu.au

Anthony Sloane
Macquarie University
anthony.sloane@mq.edu.au

Abstract

In this work we describe both a type checking and a type inference algorithm for generic programming using the *spine view* of data. The spine view of data is an approach to decomposing data in functional programming languages that supports generic programming in the style of Scrap Your Boilerplate and Stratego. The spine view of data has previously been described as a library in a statically typed language (as in Haskell), as a language feature in a dynamically typed language (as in Stratego), and as a calculus of patterns (as in the Pattern Calculus). The contribution of this paper is a type inference algorithm for the spine view and a type relation that underlies this inference algorithm. In contrast to all other typed implementations of the spine view, the type inference algorithm does not require any type annotations to be added in support of the spine view. This type inference algorithm is an extension of Hindley-Milner type inference, thus showing how to introduce the spine view of data as a language feature in any functional programming language based on Hindley-Milner.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—patterns, data types and structures; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—functional constructs

Keywords spine view; pattern matching; generic programming; FCP; type inference

1. Introduction

Generic programming is desirable because it reduces the amount of boilerplate code required to encode certain programs. Interest in generic programming has resulted in many implementations with many subtly different characteristics. In Haskell there are multiple libraries [2, 9, 14–16]. Stratego [19] is a language built to support generic programming via strategic programming. The bondi programming language [7] supports generic programming via advanced pattern matching. Clean has a system of generic programming [1] as an extension to the language that views data as a sum of products. Although there are many systems of generic programming in many languages, there are none that define a simple interface to the *spine view of data* [3, 5] and on which we can directly perform type inference. We provide both of these.

WGP '14, August 31 2014, Gothenburg, Sweden.
Copyright © 2014 ACM 978-1-4503-3042-8/14/08…$15.00.
http://dx.doi.org/10.1145/2633628.2633629

Defining a type inference algorithm for a language feature is desirable because it means adding that feature does not force any further type annotations. In languages like Haskell where type inference is part of the language, this is clearly very important. However, in any language it is useful to know that adding a spine view will not necessitate any *further* notational overhead. Although the type annotation overhead of generic programming has proven small, this is due to the very complex type systems in which it is typically deployed. In this work we are describing a very small extension to Hindley-Milner.

A system of generic programming requires three ingredients [4], *a generic view of data*, a system for *functions that dispatch on a type argument* (which we will refer to as "type indexed functions"), and *a run-time representation*. All are necessary for generic programming, but they are orthogonal; one can be considered in isolation of the others.

In this work we are concerned only with the generic view of data. There are multiple possible approaches to type indexed functions and run-time representation, and any of them could be combined with the work we present here. In fact, in other work, we have combined the inference algorithm in this paper with a system for type indexed functions and a run-time representation, resulting in a full generic programming language called DGEN [18]. All of the work presented in this paper has been implemented in the DGEN compiler.

Generic programs are primarily of interest in functional programming languages, where the (generalised) algebraic data type view of data normally requires lots of boilerplate for working with large data types. Furthermore, it is in statically typed functional languages where the problem is most acute because dynamically typed languages often allow run-time solutions to the problem. While generic programming is not restricted to statically typed functional languages, it is in this domain that we will consider it.

For languages based on Hindley-Milner type inference there are two functional language features that are common and well-understood, but not standard, that must be available for generic programming via the spine view to work. We have previously shown that to allow generic programming with the spine view requires both *polymorphic recursion* and *higher ranked types* [18]. These features are not difficult to achieve in general, but for type inferencing systems they are not available without either type annotations [13, 17] or witnessing constructors [10]. We will show how to add generic programming to a type inferencing functional language that uses witnessing constructors for polymorphic recursion and higher-ranked types.

In Section 1.1 we formally describe the *spine view* of data. In Section 2 we describe FCP_{ς}, our language which supports the spine view of data. Our description includes a formal operational semantics which is used in Section 3 to describe a sound type relation for this language. This type relation is simple and novel. The main contribution of this work is a type inference algorithm for

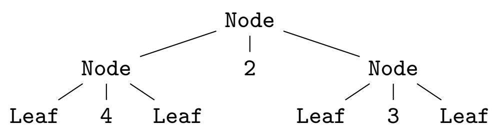

Figure 1. The fully applied constructor view of the `Tree` value `Node (Node Leaf 4 Leaf) 2 (Node Leaf 3 Leaf)`.

FCP$_\varsigma$ given in Section 4. Section 5 describes related work, Section 6 gives future directions, and Section 7 concludes the paper.

1.1 The Spine View of Data

We are going to show in this paper how to perform type inference for a particular generic view of data included in a functional language. The view of data we are concerned with is the *spine view* of data. The spine view is the mechanism that underlies the "Scrap your Boilerplate" series of papers [14–16], the pattern calculus [7], and that has been described explicitly by Hinze et.al. [3, 5]. In this section we formally describe the spine view, comparing it to the way we normally consider data in functional programming languages.

In functional programming languages we can think of a data constructor as a function that takes its arguments and returns a value of its data type. If you give that function only some of its arguments, it remains a function. However, if you give it all its arguments, you get a data value which can be pattern matched against.

For example, in Haskell, a data type definition

```
data List a = Cons a (List a) | Nil
```

introduces two constructors;

```
Nil :: List a
```

and

```
Cons :: a -> List a -> List a
```

which can be matched against in case statements and function definitions,

```
length Nil = 0
length (Cons x xs) = 1 + length xs
```

It *is not possible* under this view of data to pattern match against a partially applied constructor such as `Cons 1`. We call this the "fully applied constructor view" of data. Although some languages which use the fully applied constructor view will accept a term like `Cons 1`, that term is not a data value, it is a function which is waiting for its last argument; i.e. `\xs -> Cons 1 xs` in Haskell. According to this view, we can think of data as a tree with the constructor at the root and its arguments as its children. The number of children depends on the number of arguments for that constructor. Consider the following Haskell data type definition:

```
data Tree a = Node (Tree a) a (Tree a) | Leaf
```

`Tree` is a binary tree which stores data in the nodes. To traverse the data structure we need to know which constructor we are at and thus how many children it has. This style of traversal is achieved with standard pattern matching case expressions where one alternative is given for each possible constructor. Under this view an individual function is restricted then to operating over one (perhaps parameterised) type because we must enumerate its constructors.

In contrast, the spine view of data considers a constructor with only some of its arguments to be a data value which can be matched against. Each argument applied to the constructor is reified in the

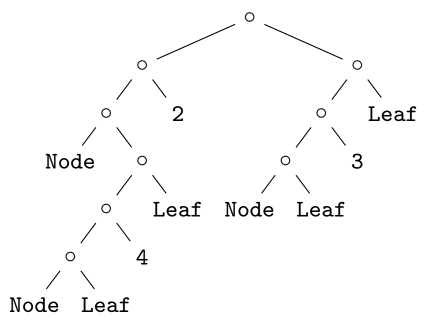

Figure 2. The spine view of the `Tree` value `Node (Node Leaf 4 Leaf) 2 (Node Leaf 3 Leaf)`.

structure as a "data application node". These data applications are just like function applications, the types work out the same, but run-time support for them is required. This approach to data is called the "spine view" because drawing the value with its application nodes creates a "spine" of applications off which the values applied to the constructor hang. This is a view of data where all data is a binary tree of data application nodes.

Figures 1 and 2 show an example data value under the two different views. We assume integers are an infinite number of nullary constructors. Under the spine view every non-leaf node has two children, regardless of how many children the constructor is defined to have. This is the central feature of the spine view, internal nodes are *tuples* (or *pairs*) and leaf nodes are nullary constructors. This means that two cases are enough to pattern match on any data using the spine view. The spine view presented here is suited to generic *consumers*, not generic *producers* and other work addresses how to extend it to that domain [3].

In this work we expose the spine view of data with a single new primitive expression, the *is-pair* expression. `ispair` d `bind` (x, y) `in` e `else` f will scrutinise the value d and if it is an internal node by the spine view ($\circ$ in Figure 2), will bind x to the left child and y to the right child, evaluating the `in` branch. If the scrutinesed value is a leaf node by the spine view (`Node`, `Leaf`, `2`, `3`, or `4` in Figure 2) no new binding is made and the result is the `else` branch. In the remainder of this paper we describe a simple language that supports the spine view via an is-pair expression, and show a type relation and a type inference algorithm for this expression.

The simplest thing you can do with the spine view of data is to pull an arbitrary value apart and put it back together again

$$\lambda d.\texttt{ispair}\ d\ \texttt{bind}\ (x, y)\ \texttt{in}\ x\ y\ \texttt{else}\ d$$

This function takes in any value and scruitinises it to see if it is a constructor applied to something. If it is, x is bound to the constructor and all its arguments but the last, y is bound to the last argument and the `in` branch is the result. For example, for the value `Node (Node Leaf 4 Leaf) 2 (Node Leaf 3 Leaf)` (shown in Figure 2), x is bound to `Node (Node Leaf 4 Leaf) 2` and y is bound to `Node Leaf 4 Leaf`. In the `in` branch, the constructor with a missing argument is applied to that argument, which reattaches the final argument to its constructor. The end result is that the initial value is returned. If the scrutinee is not a constructor applied to something, then it must be a constructor by itself and it is returned as is. Thus this function should have the type $\alpha \rightarrow \alpha$. Our type inference algorithm is able to infer this type with no type annotations.

This simple example can be extended to explore some subtleties of the spine view. If we were to try and write a function which took in any data (constructor applied to arguments) and stripped its final argument, it might look something like

$$\lambda d.\texttt{ispair}\ d\ \texttt{bind}\ (x, y)\ \texttt{in}\ x\ \texttt{else}\ ??$$

We would hope very much that such a function was severely restricted and indeed attempting to fill the `else` branch makes it clear that there is no general value we can put there which would work for all types. We can write a version where the `else` branch results in something more specific (assuming a constructor K of the type T_K with type $K\colon \mathit{Int} \rightarrow \mathit{Int} \rightarrow T_K$)

$$\lambda d.\texttt{ispair}\ d\ \texttt{bind}\ (x, y)\ \texttt{in}\ x\ \texttt{else}\ (K\ 3)$$

Now we can assign a type to the function, but it can't be a polymorphic type. An appropriate type would be $T_K \rightarrow \mathit{Int} \rightarrow T_K$. Here we see that under the spine view of data, constructors which are missing their arguments can exist. The type inference algorithm we present here will compute this type.

Of course, we must be very careful how we use constructors which have been stripped of some of their arguments, we must ensure they are only ever applied to values of the right type. Again, the type system we present here enforces this, allowing for example terms such as

$$(\lambda d.\texttt{ispair}\ d\ \texttt{bind}\ (x, y)\ \texttt{in}\ x\ \texttt{else}\ (K\ 3))\ (K\ 2\ 3)\ 4$$

and computing the type T_K for it.

1.2 The Missing Inference Algorithm

Amongst the accounts of generic programming using the spine view there is no system which performs type inference without annotation on terms. In GHC/Scrap Your Boilerplate for example, inference is done in the presence of type annotations on terms. The definition of the underlying combinator `gfoldl` includes the following type annotation:

```
gfoldl :: (forall d b. Data d =>
            c (d -> b) -> d -> c b)
        -> (forall g. g -> c g)
        -> a
        -> c a
```

The terms which encode the spine view, such as `gfoldl`, require type annotations and the GHC type inference algorithm is particularly sophisticated. It is not clear what parts of it need to be replicated for typing the spine view in particular.

In the bondi programming language an existentially quantified variable that needs to be accounted for forces the use of "aggressive assumptions" in the inference algorithm and expressions using the spine view are also annotated with their types.

Thus two questions arise, "is it even possible to do type inference on the spine view without annotation of terms?" and "exactly what additional machinery is required in a type system to support inference for the spine view?". It is these two questions which motivated the research we present here.

1.3 The Key Ideas of this Paper

1.3.1 Generic code can be expressed in a very small calculus

Generic programming is typically demonstrated in quite complex systems. This is partly because three non-trivial problems need to be solved to achieve convincing demonstrations of the idea (see section 2.1). In this paper we build on the work already done by many people who have shown the kinds of functions we can write in generic programming and we distill it down to a very small calculus. This makes it feasible to prove, rather than just demonstrate, properties of the system, such as the soundness of the type relation.

1.3.2 The spine view can be encoded with a single expression

There have been a number of expositions of the spine view (see section 5) but they are all wrapped up in much larger systems. The most notable examples being those generic programming libraries for Haskell which rely on GHC extensions, and the pattern calculus which is a full account of treating functions as data. In this work we extract just what is needed to support the spine view of data. We describe a single extension to the lambda calculus (and systems based thereon), the is-pair expression, which is sufficient to support the spine view.

1.3.3 Encoding the spine view with an is-pair expression provides a place to introduce the necessary existentially quantified variable

The spine view of data generates an existentially quantified type variable (see Section 3). In other typed accounts of the spine view, the treatment of this existentially quantified variable is unsuitable for type inference. In this work we show that by using an is-pair expression to encode the spine view, we create an artefact in the language at exactly the point where the existentially quantified variable needs to be introduced. Further, one branch of the is-pair expression is the only place that existentially quantified variable exists, so the scope created by the is-pair expression also matches with the scope of the existentially quantified type variable. It is this fact that makes type inference possible.

1.4 The Key Outcomes of this Work

1.4.1 Correction of an error in the FCP type inference algorithm

Using FCP [10] in the way we do (see Section 1.5) in this work highlights an error in the original formulation of FCP's type inference algorithm. We have corrected this error and describe the correction in this paper.

1.4.2 A type inference algorithm for the spine view of data

We describe a type inference algorithm for the spine view of data which requires only Hindley-Milner types and is a modest extension to an existing system, FCP.

1.4.3 A correctness proof for the inference algorithm

The correctness of the work in this paper is shown in two ways. The first is a working implementation in the DGEN compiler. DGEN includes an extensive set of example programs demonstrating that this particular spine view of data can encode a large number of generic programs and types for them can be inferred.

Most importantly, the correctness of the work in this paper is demonstrated with proofs of key properties of the systems described. We have proofs that:

- The type relation is sound. If you can assign a type to a term in FCP_ς, that term can be evaluated one step to a term with the same type or is a value which has that type.
- The type inference algorithm computes types for terms which are consistent with the type relation. If a term is given a type σ by the type inference algorithm, then the type relation holds for that term having the type σ and vice versa.

1.5 FCP

FCP is a language and associated type system that supports *first class polymorphism*. FCP uses Hindley-Milner types (i.e. universal quantifiers only occur at the top-level) and no type annotations are required on terms. FCP adds to the lambda calculus constructors in the style of algebraic data types. These constructors operate exactly as they do in functional languages like Caml and Haskell. Each constructor is effectively a constructor function which when given all its arguments, will construct a value of the type in question. In FCP, as in Caml and Haskell, constructors *are* annotated with a type. What FCP adds to other systems with data constructors is that the type relation and type inference rules are able to work with nested quantifiers in these constructor types.

For terms, FCP allows only Hindley-Milner types and there are no annotations. For constructors, nested quantifiers are supported. This small change is enough to support all of System F. Jones [10] describes a type-driven algorithm for converting any System F program to an FCP program and vice versa. Thus it is possible to use the first-class polymorphism of FCP to support other type system features not available in Hindley-Milner, such as higher-ranked functions and polymorphic recursion.

FCP differs from other systems which have been used to support higher-ranks, polymorphic recursion and first class polymorphism because it does not have type annotations on terms and does not extend the Hindley-Milner types. Alternatives which have seen wider usage such as Peyton Jones et al.'s practical type inference [17] or Rémy and Le Botlan's MLF [11], are simpler to program with, but require type annotations on terms. It is our primary concern to show that the spine view of data, and in particular the is-pair expression, does not require any annotations on terms and thus working from a system which included them would hamper our argument.

The Peyton Jones et. al, Rémy and DeBotlan systems also use significantly different types to those of Hindley-Milner. One of our aims in this work is to show a "baseline" for the spine view of data in functional languages. We want our work to be applicable to as many different existing (or future) programming languages/compilers as possible. Thus the fact that FCP only uses Hindley-Milner types further recommends it as a starting point for this work.

2. A Language of the Spine View

We now describe an extended lambda calculus FCP_ς that supports the spine view of data. We take as our starting point FCP [10], a language that includes the fully applied constructor view of data and that supports inference for first class polymorphism.

Figure 3 shows the syntax for types and terms of FCP_ς (pronounced FCP-spine). FCP_ς is FCP with additions for the spine view of data. Shared with FCP, FCP_ς has: monotypes for constructed values and function types; type variables; type schemes that are quantified only at the top level; lambda abstractions for parameterising a function by a variable; function applications for providing arguments to functions; variables; let expressions which allow polymorphic definitions; pattern matching lambda abstractions; constructed values built from a constructor token and its argument values; and a "pattern matching lambda" expression for pulling a constructor from a constructed value.

In addition to the above features taken from FCP, FCP_ς has the following new expressions to support the spine view of data:

Recursive let expressions Generic programming relies on recursive definitions and in particular needs polymorphic recursion. Thus FCP_ς has a recursive let expression where FCP had a non-recursive let expression.

The is-pair expression The is-pair expression is the expression that allows us to pattern match against the spine view of data. Its left branch is used if its discriminator is a tuple/pair and its right branch is used for nullary constructors.

Constructors take more than one argument FCP restricts constructors to arity 1. While this does not invalidate what we are showing here, it rather disguises it because the whole point of the spine view is to say that "no matter what the arity of your constructor, we can see it as arity 2 using the spine view". Thus FCP_ς has constructors which take multiple arguments.

$$
\begin{array}{rll}
\sigma ::= & \forall t.\sigma & \text{type scheme} \\
| & \tau & \text{monotype} \\
\tau ::= & t & \text{type variable} \\
| & \tau \to \tau' & \text{function type} \\
| & T\ \tau_1 \ldots \tau_n & \text{datatype, arity}(T) = n
\end{array}
$$

$$
\begin{array}{rll}
e, f ::= & x & \text{variable} \\
| & e\ f & \text{application} \\
| & \lambda x.e & \text{abstraction} \\
| & \texttt{letrec}\ x = e\ \text{in}\ f & \text{local definition} \\
| & K(e, \ldots, f) & \text{construction} \\
| & \lambda(K\ x_1 \ldots x_n).e & \text{decomposition} \\
| & \texttt{ispair}\ d\ \texttt{bind}\ (x, y)\ \texttt{then}\ e\ \texttt{else}\ f & \text{pair discriminator}
\end{array}
$$

Figure 3. Type and expression syntax of FCP_ς

FCP_ς achieves inference for higher ranked types and polymorphic recursion in the same way as FCP, by requiring them to be *witnessed* by constructors that wrap the necessary type. Types are wrapped with constructors and unwrapped by functions that take the constructed value and return its contents. Jones shows that this is enough to encode anything you can encode in System F while maintaining type inference [10]. The result is that FCP expressions are somewhat more complex than the equivalent in, for example, GHC's type system, but no type annotations are required.

A subset of FCP_ς expressions are designated as values (v). These are abstractions, decompositions, and constructors tagging values.

2.1 Example generic expressions in FCP_ς

We now give a number of examples of the generic functions you can write in FCP_ς. Type inference for FCP (and thus FCP_ς) occurs in an environment in which various datatypes with associated constructors have been defined. These can be created by datatype definitions that introduce a constructor with a function type from its argument type to the type of the datatype being defined. For example, the Haskell code

```
data Tagged a = Tag a
```

creates the type *Tagged* and the constructor *Tag* with the type signature:

$$Tag\colon a \to Tagged\ a$$

We can also define a function to pull the data from a $Tagged$ value:

$$Tag^{-1} = \lambda(Tag\ d).d$$

For our examples we need witnessing types for a type $\forall\alpha.\alpha \to \alpha$ and a polymorphic recursive mapping type $\forall\beta.(\forall\alpha.\alpha \to \alpha) \to \beta \to \beta$. Thus all the examples are written assuming the following constructor (K_ι, K_μ) and extractor function (K_ι^{-1}, K_μ^{-1}) definitions. The constructed types Y_ι and Y_μ are used to witness the identity type and the polymorphic recursive mapping type respectively.

$$K_\iota : (\forall a.a \to a) \to Y_\iota$$
$$K_\mu : (\forall b.Y_\iota \to b \to b) \to Y_\mu$$
$$K_\iota^{-1} = \lambda(K_\iota\ x).x$$
$$K_\mu^{-1} = \lambda(K_\mu\ x).x$$

Witnessing higher-ranked types in this way is a straightforward task, but it tends to obfuscate the meaning of our expressions. All definitions of higher ranked and polymorphic recursive functions need to be put into the appropriate constructor and all uses of these functions need to extracted from those constructors. In other work [18] we have shown how this system can be converted to one that uses type annotations, making the expressions much easier to understand.

Figure 4 defines an *FCP*$_\varsigma$ expression that applies a polymorphic argument to all the values in the data it is passed. In bondi this expression (a_β) is called *apply to all*, in Stratego it is *all bottom up*.[1] This function takes a polymorphic function (which would normally be type-indexed) and applies it to every sub-value in its second argument. It does this by calling the function on the whole value and recursively applying itself to each part of the tuple under the spine view. The recursive expression a_β is polymorphically recursive, it is applied recursively to two arguments with two different types in its own body. The first argument to a_β is applied to (possibly) three different types in the body of the expression and thus must be witnessed by a K_ι. This is a concrete example of why we need to begin with a system that supports type inference for polymorphic recursion and higher-ranked functions.

We can define a top-down version (a_τ) of this expression using the spine view,

```
letrec $a_\tau = K_\mu\ (\lambda f.\lambda d.$
    ispair $((K_\iota^{-1}\ f)\ d)$ bind $(x, y)$
    in $((K_\mu^{-1}\ a_\tau)\ f\ x)\ ((K_\mu^{-1}\ a_\tau)\ f\ y)$
    else $(K_\iota^{-1}\ f)\ d)$
```

We can also write expressions that collect a single value from a data structure, called *generic queries*. Figure 5 shows an expression, q_τ, which queries from the top-down. q_τ takes as arguments a polymorphic accumulator function, a start value and data to query. It recursively calls itself on the left-hand-side of the tuple using the query value from a recursive call of itself on the right-hand-side of the tuple as a starting value. The start value of this second recursive call is generated by calling the accumulator function on the whole value with the original start value. Again, we require witnessing constructors and extractors for the some of the types:

$$K_\kappa : (\forall a.r \to a \to r) \to Y_\kappa\ r$$
$$K_\xi : Y_\kappa\ r \to b \to r \to Y_\xi\ b\ r$$
$$K_\kappa^{-1} = \lambda(K_\kappa\ x).x$$
$$K_\xi^{-1} = \lambda(K_\xi\ x).x$$

Again we can control the order of the traversal using the spine view, the following definition is a bottom up query (q_β):

```
letrec $q_\beta = K_\xi\ (\lambda f.\lambda s.\lambda d.$
    ispair $d$ bind $(x, y)$
    in $(K_\kappa^{-1}\ f)\ ((K_\xi^{-1}\ q_\beta)\ f\ ((K_\xi^{-1}\ q_\beta)\ f\ s\ y)\ x)\ d$
    else $(K_\kappa^{-1}\ f)\ s\ d)$
```

[1] Stratego only behaves the same as *FCP*$_\varsigma$ in this instance if the strategy being passed to *all bottom up* can't fail.

```
let gfoldl $=\lambda k.\lambda z.\lambda p.$
    ispair $p$ bind $(x, y)$
    in $k$ (gfoldl $k\ z\ x)\ y$
    else $z\ x$
```

Figure 6. SYB's `gfoldl` in *FCP*$_\varsigma$

Terms like a_τ, a_β, q_τ and q_β can't be written in FCP, but they can be written in *FCP*$_\varsigma$. Since we know type inference is possible for FCP, the question we are answering in this paper is "can we define type inference for *FCP*$_\varsigma$?" If we can't then it is the extension of FCP, i.e. the spine view, which precludes type inference. In fact we show that the spine view *does not* preclude type inference.

Also, we know that we can define terms like a_τ, a_β, q_τ and q_β in type systems with type annotations and which do type checking/inference in both directions, as in algorithm M [12]. We know this because of all the GHC-based implementations of these functions. What we are answering in this paper is "can we perform type inference, with no annotation of terms, for these terms". The answer, happily, is yes.

We answer these questions by defining, and proving correct, a type relation and a type inference algorithm on *FCP*$_\varsigma$.

2.2 Comparison to Scrap Your Boilerplate

FCP$_\varsigma$ is related to Scrap Your Boilerplate (SYB) [14] quite closely. We now compare the two and use SYB primitives as a further example of *FCP*$_\varsigma$. In this section we don't include the witnessing constructors and destructors, making the examples simpler to read.

Firstly, both SYB and *FCP*$_\varsigma$ use the spine view of data to see arbitrary data in a uniform way. SYB exposes the spine view via two non-recursive combinators, `gmapQ` and `gmapT`. These non-recursive combinators are turned into recursive traversal and query functions by the library functions `everywhere` and `everything`. All of these, however, are built from one underlying combinator; `gfoldl`. The ispair operation in *FCP*$_\varsigma$ is an alternative to `gfoldl` but as we will see, it is more suitable as an extension for an underlying calculus because it is more flexible than `gfoldl`. The ispair expression is the minimal addition required to a core calculus to achieve what `gfoldl` achieves in SYB but it is also capable of directly encoding the other SYB combinators `gmapQ`, `gmapT`. To see this, we will encode all three in *FCP*$_\varsigma$.

Figure 6 shows how *FCP*$_\varsigma$ can encode `gfoldl`. Notice that one definition of `gfoldl` will work for all values of all datatypes. This contrasts with SYB where an instance of this function must be created, either by the programmer or by the compiler, for each datatype. This then enforces the need for type classes in the SYB approach. *FCP*$_\varsigma$ is both simpler and requires less compiler machinery to do the same thing.

From here we could build `gmapT`, `gmapQ` from `gfoldl`, but *FCP*$_\varsigma$ allows us an alternative. Figure 7 shows *direct* encodings of these functions in *FCP*$_\varsigma$ (assuming a list datatype with constructors C and N and a concatenation function concat).

Directly defining pattern matching against the spine view opens up the opportunity to mix spine view patterns with fully applied constructor pattens, as shown by Jay [7].

SYB also includes a system for type indexed functions with the `ext` combinators. Type indexed functions allow for specific behaviour at certain nodes in the value being operated on. The methods for defining such functions can be combined with this work, as we have shown in the DGEN compiler where an extension mechanism very similar to the one used in SYB has been combined

$$\begin{aligned}&\texttt{letrec}\ a_\beta = K_\mu\ (\lambda f.\lambda d.\\&\qquad \texttt{ispair}\ d\ \texttt{bind}\ (x,y)\\&\qquad \texttt{in}\ (K_\iota^{-1}\ f)\ ((K_\mu^{-1}\ a_\beta)\ f\ x)\ ((K_\mu^{-1}\ a_\beta)\ f\ y)\\&\qquad \texttt{else}\ (K_\iota^{-1}\ f)\ d)\end{aligned}$$

$$\begin{aligned}&\texttt{letrec}\ a_\beta : \forall b.(\forall a.a \to a) \to b \to b =\\&\qquad \lambda f : \forall a.a \to a.\lambda d.\\&\qquad \texttt{ispair}\ d\ \texttt{bind}\ (x,y)\\&\qquad \texttt{in}\ f\ ((a_\beta\ f\ x)\ (a_\beta\ f\ y))\\&\qquad \texttt{else}\ f\ d\end{aligned}$$

Figure 4. A generic function similar to Stratego's *all bottom up*. On the left is the expression in *FCP*$_\varsigma$, on the right is the same function written using type annotations in the style of GHC instead of witnessing constructors. The type annotation version is given as a guide to understanding the *FCP*$_\varsigma$ version.

$$\begin{aligned}&\texttt{letrec}\ q_\tau = K_\xi\ (\lambda f.\lambda s.\lambda d.\\&\qquad \texttt{ispair}\ d\ \texttt{bind}\ (x,y)\ \texttt{in}\ (K_\xi^{-1} q_\tau)\ f\ ((K_\xi^{-1} q_\tau)\ f\ ((K_\kappa^{-1}\ f)\ s\ d)\ y)\ x\\&\qquad \texttt{else}\ (K_\kappa^{-1}\ f)\ s\ d)\end{aligned}$$

$$\begin{aligned}&\texttt{letrec}\ q_\tau : \forall r.\forall b.(\forall a.r \to a \to r) \to r \to b \to r =\\&\qquad \lambda f : \forall a.r \to a \to r.\lambda s.\lambda d.\\&\qquad \texttt{ispair}\ d\ \texttt{bind}\ (x,y)\ \texttt{in}\ q_\tau\ f\ (q_\tau\ f\ (f\ s\ d)\ y)\ x\\&\qquad \texttt{else}\ f\ s\ d\end{aligned}$$

Figure 5. A generic query. On the left is the expression in *FCP*$_\varsigma$, on the right is the same expression written using type annotations instead of witnessing constructors.

$$\begin{aligned}&\texttt{let}\ \text{gmapT} = \lambda f.\lambda d.\\&\qquad \texttt{ispair}\ d\ \texttt{bind}\ (x,y)\\&\qquad \texttt{in}\ (\text{gmapT}\ f\ x)\ (f\ y))\\&\qquad \texttt{else}\ x\\&\texttt{let}\ \text{gmapQ} = \lambda f.\lambda d.\\&\qquad \texttt{ispair}\ d\ \texttt{bind}\ (x,y)\\&\qquad \texttt{in}\ \text{concat}(\text{gmapQ}\ f\ x, C(f\ y, N))\\&\qquad \texttt{else}\ N\end{aligned}$$

Figure 7. Direct encodings of SYB combinators in *FCP*$_\varsigma$

with *FCP*$_\varsigma$ and a run-time representation to create a full generic programming language.

2.3 Operational Semantics of *FCP*$_\varsigma$

Figure 8 provides an operational semantics for *FCP*$_\varsigma$ that captures the normal lambda calculus evaluation style, the expected operation of pattern matching lambdas, and the spine view of data via `ispair` . The evaluation rules make use of a substitution operation in which $[x/e]f$ denotes the substitution of e for free occurrences of x in f.

The evaluation rules for application, lambda abstraction, and the E-Con1 rule are shared with *FCP*. The operation of the recursive let is specific to *FCP*$_\varsigma$ but is entirely standard. The four evaluation rules added to support the spine view are:

E-Con2 If a constructed value is applied to an expression, the constructed value is treated like a function expecting an argument and it consumes the expression. The result is that the constructor is given the expression as a new final argument. This operation can only occur on constructors that have previously been stripped of one of their arguments and this constraint is enforced by the type system. This is how we "put data back together" according to the spine view.

E-IsPair1 If the discriminator of an is-pair expression is not a value, evaluate it one step.

E-IsPair2 If the discriminator of an is-pair is a value and it is a constructor with at least one argument attached, bind the constructor and all arguments but the last to x, bind the last argument to y and evaluate the left branch. This is how we "pull data apart" according to the spine view.

E-IsPair3 If the discriminator of an is-pair is a value and it is not a constructor with at least one argument attached, evaluate the right branch.

3. Typing the Spine View

Figure 9 describes the typing relation of *FCP*$_\varsigma$. $A \vdash e\colon \tau$ denotes the expression e having the type τ given the environment A. An environment is a set of variable to type scheme bindings. $A_x, x\colon \sigma$ is the environment A where any existing binding for x has been replaced by σ. The variable, application, let expression, and lambda abstraction rules are based on the equivalent rules in FCP but we use generalisation in specific places rather than have a generalisation rule. Being specific about generalisation simplifies the proofs of type system and type inference properties. A type scheme σ is a generalisation of a type τ, denoted $\sigma \succ \tau$, if there is some substitution for the bound variables of σ that gives τ. We can generalise a type τ to a type scheme with the Gen operation

$$Gen(\tau, A) = \forall \alpha_1 \dots \alpha_n.\tau \text{ where } \{\alpha_1 \dots \alpha_n\} = TV(\tau)\backslash TV(A)$$

While we need a new evaluation rule to deal with putting spine data back together, we don't need a special type rule for this because the normal function application rule T-App works perfectly.

The fact that function application and data application have the same type rules is a key property of Jay's pattern calculus [7]. Our work has validated Jay's findings.

Specific to *FCP*$_\varsigma$ is the T-IsPair rule. How can we determine the type rule for `ispair`? Our starting point is the following question

> Given d has type D, and assuming d is a tuple, what are the types of x and y in `ispair` d `bind` (x, y)`in` f `else` g?

If d is a tuple, it is a constructor applied to some arguments, x is the constructor with all the arguments but the last, so it could have the type of a function that if given that last argument, will return a value of the original type, i.e. *argtype* $\to$ D. y is exactly the argument that was peeled off, so its type is *argtype*. However, we don't know anything about *argtype*. Since we only know d has type D, we can say nothing about its last argument, we only know that there is one. We might be tempted then to make it a type variable, giving x the type $\alpha \to D$ and y the type α. However, this variable would be implicitly universally quantified and since x and y will

$$\text{E-App1}\;\frac{e \longrightarrow e'}{e\; f \longrightarrow e'\; f} \qquad \text{E-App2}\;\frac{e \longrightarrow e'}{v\; e \longrightarrow v\; e'} \qquad \text{E-Lam}\;\frac{}{(\lambda x.e)\; v \longrightarrow [x/v]e}$$

$$\text{E-LetRec1}\;\frac{e \longrightarrow e'}{\texttt{letrec}\; x = e\; \texttt{in}\; f \longrightarrow \texttt{letrec}\; x = e'\; \texttt{in}\; f} \qquad \text{E-LetRec2}\;\frac{f \longrightarrow f'}{\texttt{letrec}\; x = v\; \texttt{in}\; f \longrightarrow \texttt{letrec}\; x = v\; \texttt{in}\; f'}$$

$$\text{E-LetRec3}\;\frac{x \in FV(v_f)}{\texttt{letrec}\; x = v\; \texttt{in}\; v_f \longrightarrow \texttt{letrec}\; x = v\; \texttt{in}\; [x/v]v_f} \qquad \text{E-LetRec4}\;\frac{x \notin FV(v_f)}{\texttt{letrec}\; x = v\; \texttt{in}\; v_f \longrightarrow v_f}$$

$$\text{E-PLam}\;\frac{}{(\lambda(K\; x_1, \ldots, x_n).e)\; K(v_1, \ldots, v_n) \longrightarrow [x_1/v_1] \cdots [x_n/v_n]e}$$

$$\text{E-Con1}\;\frac{e_j \longrightarrow e_j'}{K(v_1, \ldots, v_{j-1}, e_j, e_{j+1}, \ldots, e_m) \longrightarrow K(v_1, \ldots, v_{j-1}, e_j', e_{j+1}, \ldots, e_m)} \qquad \text{E-Con2}\;\frac{}{K(v_1, \ldots, v_m)\; e \longrightarrow K(v_1, \ldots, v_m, e)}$$

$$\text{E-IsPair1}\;\frac{e \longrightarrow e'}{\texttt{ispair}\; e\; \texttt{bind}\; (x, y)\; \texttt{in}\; f\; \texttt{else}\; g \longrightarrow \texttt{ispair}\; e'\; \texttt{bind}\; (x, y)\; \texttt{in}\; f\; \texttt{else}\; g}$$

$$\text{E-IsPair2}\;\frac{m \geq 1}{\texttt{ispair}\; (K(v_1, \ldots, v_m))\; \texttt{bind}\; (x, y)\; \texttt{in}\; f\; \texttt{else}\; g \longrightarrow [x/K(v_1, \ldots, v_{m-1}), y/v_m]f}$$

$$\text{E-IsPair3}\;\frac{}{\texttt{ispair}\; K()\; \texttt{bind}\; (x, y)\; \texttt{in}\; f\; \texttt{else}\; g \longrightarrow g}$$

Figure 8. Operational Semantics ($e \longrightarrow f$) of *FCP*$_\varsigma$

$$\text{T-Var}\;\frac{x\colon \sigma \in A \qquad \sigma \succ \tau}{A \vdash x\colon \tau} \qquad \text{T-Lam}\;\frac{A_x, x\colon \rho \vdash e\colon \tau}{A \vdash \lambda x.e\colon \rho \to \tau}$$

$$\text{T-App}\;\frac{A \vdash e\colon \rho \to \tau \qquad A \vdash f\colon \rho}{A \vdash e\; f\colon \tau}$$

$$\text{T-LetRec}\;\frac{A_x, x\colon \sigma \vdash e\colon \tau' \qquad \sigma \succ \tau' \qquad A_x, x\colon \sigma \vdash f\colon \tau}{A \vdash \texttt{letrec}\; x = e\; \texttt{in}\; f\colon \tau}$$

for each K : $((\forall \alpha_1.\tau_1'), \ldots, (\forall \alpha_n.\tau_n')) \to \tau_K$
where τ_K is unique to this K

$$\text{T-Con1}\;\frac{\forall i.(A \vdash e_i\colon \tau_i') \qquad \forall i(\alpha_i \notin TV(A))}{A \vdash K(e_1, \ldots, e_n)\colon \tau_K}$$

$$\text{T-Con2}\;\frac{\forall i \in 1 \ldots m.(A \vdash e_i\colon \tau_i') \qquad m < n \qquad \forall i(\alpha_i \notin TV(A))}{A \vdash K(e_1, \ldots, e_m)\colon \tau_{m+1}' \to (\cdots \to (\tau_n' \to \tau_K) \cdots)}$$

$$\text{T-PLam}\;\frac{A_{x_1 \cdots x_n}, x_1\colon [\alpha_1/\rho_1]\tau_1', \ldots, x_n\colon [\alpha_n/\rho_n]\tau_n' \vdash e\colon \tau_e}{A \vdash \lambda(K(x_1, \ldots, x_n)).e\colon \tau_K \to \tau_e}$$

$$\text{T-IsPair}\;\frac{\begin{array}{cc} A \vdash e\colon \tau_e & A_{x,y}, x\colon \alpha \to \tau_e, y\colon \alpha \vdash f\colon \tau \\ A \vdash g\colon \tau & \alpha \notin TV(A, \tau, \tau_e) \end{array}}{A \vdash \texttt{ispair}\; e\; \texttt{bind}\; (x, y)\; \texttt{in}\; f\; \texttt{else}\; g\colon \tau}$$

Figure 9. Type Relation ($A \vdash e\colon \tau$) for *FCP*$_\varsigma$

appear separately, we lose the fact that the two α variables represent the same *something*. The solution to this problem is to existentially quantify this type variable.

Our approach differs from the compound calculus dialect of the pattern calculus, for example, because the latter separates access to the left and right parts of the pair without a scope to existentially quantify the introduced type variable. In *FCP*$_\varsigma$, x and y only occur in the `in` branch of an is-pair expression which is the ideal site at which to introduce a fresh existentially quantified type variable to constrain the use of these unpaired variables.

The T-IsPair rule in Figure 9 shows the type relation that reflects this understanding of the is-pair expression. We have proven that the type system in Figure 9 is sound. The proof is based on the small-step semantics in Figure 8 and proves both *progress* (if $A \vdash e\colon \tau$ then $e \longrightarrow e'$ or e is a value) and *preservation* (If $A \vdash e\colon \tau$ and $e \longrightarrow e'$ then $A \vdash e'\colon \tau$).

3.1 Details of the Proof

We have followed the syntactic approach of Wright and Felleisen [20] in constructing our proofs thus the proofs of type system soundness for *FCP*$_\varsigma$ are almost a superset of the equivalent proofs for FCP. For this reason we do not describe the proofs in detail, instead focussing on the is-pair expression and the treatment of partially applied constructors. A version of the full proof (including the FCP segments) has previously been published in Robert's doctoral thesis [18].

Our proof of type system soundness hinges on two main results:

Progress If $A \vdash e\colon \tau$ then $e \longrightarrow e'$ or e is a value.

Preservation If $A \vdash e\colon \tau$ and $e \longrightarrow e'$ then $A \vdash e'\colon \tau$

If both progress and preservation are proven then the type system is *sound*.

Theorem 3.1 (Progress)**.** *If $A \vdash e\colon \tau$ then $e \longrightarrow e'$ or e is a value.*

Proof. The proof proceeds by induction on the length of the type deduction for $A \vdash e\colon \tau$, with one case for each possible final deduction rule. Here we only give the cases relating to the spine view of data.

Case T-IsPair If the final deduction rule is T-IsPair and $A \vdash (\texttt{ispair}\; e\; \texttt{bind}\; (x, y)\; \texttt{in}\; f\; \texttt{else}\; g)\colon \tau$ we have

$$A \vdash e\colon \tau_e \tag{1}$$

This, with the induction hypothesis gives either e is a value or $e \longrightarrow e'$. Furthermore, if e is a value, it is either a constructed value ($K(v_1, \ldots, v_n)$) or it is not. These possibilities give rise to the following cases:

e **is not a value** By $e \longrightarrow e'$ and E-IsPair1 we have $\texttt{ispair}\; e\; \texttt{bind}\; (x, y)\; \texttt{in}\; f\; \texttt{else}\; g \longrightarrow \texttt{ispair}\; e'\; \texttt{bind}\; (x, y)\; \texttt{in}\; f\; \texttt{else}\; g$.

e **is a constructed value** Say e is the constructed value $K(e_1, \ldots, e_n)$. By E-IsPair2 we have `ispair` $K(v_1, \ldots, v_n)$ `bind` (x, y) `in` f `else` $g \longrightarrow [x/K(v_1, \ldots, v_{n-1}), y/v_n]f$.

e **is a value, but not a constructed value** Say e is the non-constructed value v_e. By E-IsPair3 we have `ispair` v_e `bind` (x, y) `in` f `else` $g \longrightarrow g$.

Case T-Con1 If $A \vdash K(e_1, \ldots, e_n) \colon \tau$, and all e_i are values, then $K(e_1, \ldots, e_n)$ is a value. If $A \vdash K(e_1, \ldots, e_n) \colon \tau$, and one of e_i is not evaluated (say e_j), then by T-Con, $A \vdash e_j \colon \tau'_j$. Hence by the induction hypothesis $e_j \longrightarrow e'_j$ and $K(e_1, \ldots, e_j, \ldots, e_n) \longrightarrow K(e_1, \ldots, e'_j, \ldots, e_n)$

□

Theorem 3.2 (Preservation). *If* $A \vdash e \colon \tau$ *and* $e \longrightarrow e'$ *then* $A \vdash e' \colon \tau$

Proof. The proof proceeds by induction on the depth of the evaluation tree, with one case for each possible final reduction $e \longrightarrow e'$. Here we only give the cases related to the spine view of data.

Case E-IsPair2 If `ispair` $K(v_1, \ldots, v_m)$ `bind` (x, y) `in` f `else` $g \longrightarrow [x/K(v_1, \ldots, v_{m-1}), y/v_m]f$ and $A \vdash$ `ispair` $K(v_1, \ldots, v_m)$ `bind` (x, y) `in` f `else` $g \colon \tau$, by T-IsPair we have

$$A \vdash K(v_1, \ldots, v_m) \colon \tau' \tag{2}$$

$$A_{x,y}, x \colon \alpha \to \tau', y \colon \alpha \vdash v' \colon \tau \tag{3}$$

where α is unique. From (2) and T-Con2 we have

$$\forall i \in 1 \ldots m. (A \vdash v_i \colon \tau'_i) \tag{4}$$

By (4) and T-Con2 again (this time in the opposite direction) we have

$$A \vdash K(v1, \ldots, v_{m-1}) \colon \tau'_m \to \tau' \tag{5}$$

Note also that (4) includes the fact that $A \vdash v_m : \tau'_m$. This with (5), (3) and Lemma 3.1 gives $A \vdash [x/K(v_1, \ldots, v_{m-1}), y/v_m)]f$: τ as required.

Case E-IsPair3 If `ispair` v `bind` (x, y) `in` f `else` $g \longrightarrow g$ and $A \vdash$ `ispair` v `bind` (x, y) `in` f `else` $g \colon \tau$, by T-IsPair we have $A \vdash g \colon \tau$ as required.

□

One particularly important lemma used in the proof is

Lemma 3.1 (Existential Instantiation). *If* $A_{x,y}, x \colon \alpha \to \tau', y \colon \alpha \vdash e \colon \tau$ *and* $\alpha \notin TV(A, \tau, \tau', \rho')$ *and* $A \vdash v' \colon \rho' \to \tau'$ *and* $A \vdash v'' \colon \rho'$ *then* $A \vdash [x/v', y/v'']e \colon \tau$ *for* any ρ'.

Proof. The proof is by induction on the length of the type deduction for $A_{x,y}, x \colon \alpha \to \tau', y \colon \alpha \vdash v \colon \tau$, with one case for each possible final deduction rule. □

4. Type Inference for the Spine View

In this section we define a type inference algorithm for *FCP*$_\varsigma$. This type inference algorithm is built from the type relation in Figure 9 and we have developed both a proof of correctness and a working implementation in the DGEN compiler.[2] As noted in Section 2, full type inference is generally not possible for higher ranks and polymorphic recursion. *FCP*$_\varsigma$ solves this problem by insisting that these types be witnessed by a constructor. Figure 11 describes the type inference algorithm for *FCP*$_\varsigma$. The type inference algorithm, denoted $TA \vdash \tau \colon \sigma \bmod V$, takes as input a type τ, a set of fixed-for-unification variables V, and an environment A. It calculates a set of substitutions T which are applied to the environment and a type σ. UT denotes a substitution U applied to a substitution T. Unification, Figure 4, $\tau \overset{U}{\sim} \rho \bmod V$ steps take in two types τ and ρ and calculate a substitution U that unifies those types. To enforce fixed-for-unification variables, *FCP*$_\varsigma$ uses the unification algorithm of FCP which keeps track of those variables (as V) in the environment that must be handled in this way and adds these variables to the occurs check.

$$\text{(id)} \quad \tau \overset{id}{\sim} \tau \bmod V$$

$$\text{(var)} \quad \left.\begin{array}{l} \alpha \overset{[\alpha/\tau]}{\sim} \tau \bmod V \\ \tau \overset{[\alpha/\tau]}{\sim} \alpha \bmod V \end{array}\right\} \alpha \notin V \cup TV(\tau)$$

$$\text{(fun)} \quad \frac{\tau \overset{U}{\sim} \nu \bmod V \qquad U\tau' \overset{U'}{\sim} U\nu' \bmod V}{(\tau \to \tau') \overset{UU'}{\sim} (\nu \to \nu') \bmod V}$$

Figure 10. Rules for unification. This version of unification reduces to normal unification when the set V is empty.

The type rules for variables, abstractions and applications are taken directly from FCP. The rule for pattern matching lambdas is derived from the equivalent FCP rule. The variable, abstraction, application and let expression rules are equivalent to the corresponding Hindley-Milner rules, but the set of fixed-for-unification variables is passed around. The constructor rule enforces the requirement that a constructor tagging values has the correct type for those values by unifying them, it also enforces the quantification of the variable α by ensuring it is unique.

The recursive let rule is specific to *FCP*$_\varsigma$ but does not require anything new because polymorphic recursion is dealt with by witnessing constructors. Inference for normal function application can deal with putting spine data back together but we describe an entirely novel algorithm for type inference of is-pair expressions. We can provide type inference rules for the existentially quantified type variable in an is-pair expression without having to provide a complete implementation of existential typing. Specifically, all we need to do is ensure that any new existential variables introduced by an is-pair expression are treated like constants for the purposes of unification in the body of the is-pair expression, which can already be done with our unification algorithm by adding that variable to the set of fixed-for-unification variables.

The type inference rule for is-pair expressions first calculates the type of the expression being tested (c). This inferred type is used as the basis for the type of the conditional in the first branch. We store two types for x and y in the environment when inferring the type for t. The first ($\beta \to \tau_c$) is the type for x anywhere in the body of the branch and the second (β) is the type for y. When we are inferring the type of the t branch, we ensure that β is treated as an existential type variable by adding it to the set of fixed-for-unification variables. In this environment we calculate the type for the first branch. This is all the hard work done and we use standard techniques to get the type for the second branch and to unify that type with the type we got for the first branch.

In this way, Figure 11 describes a type inference algorithm for a language that supports *both* the fully applied constructor view and the spine view of data. This type inference algorithm has been implemented in the generic programming language DGEN [18], a

[2] DGEN implements a variant of the type inference algorithm we present here. Specifically a type annotation version of FCP is used as the basis of type inference in DGEN. Crucially, the rules relating to the spine view are not impacted by this adjustment.

$$\text{I-Var}\frac{(x\colon \forall\alpha.\tau) \in A \qquad \beta\ \text{new}}{A \vdash^{W} x\colon [\beta/\alpha]\tau\ \text{mod}\ V} \qquad \text{I-Abs}\frac{T(A_x, x\colon \alpha) \vdash^{W} e\colon \tau\ \text{mod}\ V \qquad \alpha\ \text{new}}{TA \vdash^{W} \lambda x.e\colon T\alpha \to \tau\ \text{mod}\ V}$$

$$\text{I-App}\frac{\begin{array}{c} TA \vdash^{W} e : \tau\ \text{mod}\ V \qquad T'TA \vdash^{W} f : \tau'\ \text{mod}\ V \\ T'\tau \overset{U}{\sim} (\tau' \to \alpha)\ \text{mod}\ V \qquad \alpha\ \text{new} \end{array}}{UT'TA \vdash^{W} e\ f : U\alpha\ \text{mod}\ V} \qquad \text{I-LetRec}\frac{\begin{array}{c} T(A_x, x\colon \alpha) \vdash^{W} e\colon \tau\ \text{mod}\ V \qquad T\alpha \overset{U}{\sim} \tau\ \text{mod}\ V \qquad \alpha\ \text{new} \\ \sigma = \mathit{Gen}(UTA, U\tau) \qquad T'(UTA_x, x\colon \sigma) \vdash^{W} f\colon \rho\ \text{mod}\ V \end{array}}{T'UTA \vdash^{W} \texttt{letrec}\ x = e\ \texttt{in}\ f\colon \rho\ \text{mod}\ V}$$

$$\text{I-Con}\frac{\begin{array}{c} \sigma_K = \forall\gamma.(\forall\alpha_1.\tau_1) \to \cdots \to (\forall\alpha_n.\tau_n) \to \tau' \qquad \alpha_1, \ldots, \alpha_n, \gamma\ \text{new} \qquad U = U_1 \ldots U_n \qquad T = T_1 \ldots T_n \\ \forall i \in \{1, \ldots, n\}.(T_i A \vdash^{W} e_i\colon \rho_i\ \text{mod}\ V \qquad \rho_i \overset{U_i}{\sim} \tau_i\ \text{mod}\ (V \cup \{\alpha_i\}) \qquad \alpha_i \notin TV(U_i T_i A, U_i \tau')) \end{array}}{UTA \vdash^{W} K\ (e_1, \ldots, e_n)\colon U\tau'\ \text{mod}\ V}$$

$$\text{I-PLam}\frac{\sigma_K = \forall\gamma.(\forall\alpha_1.\tau_1) \to \cdots (\forall\alpha_1.\tau_1) \to \tau' \qquad \alpha_1, \ldots \alpha_n, \gamma\ \text{new} \qquad T(A_{\{}x_1, \ldots, x_n\}, x_1 : \tau_1, \ldots, x_n : \tau_n) \vdash^{W} e\colon \tau_e\ \text{mod}\ V}{TA \vdash^{W} \lambda(K\ x_1\ \cdots\ x_n).e\colon T\tau \to \tau_e\ \text{mod}\ V}$$

$$\text{I-IsPair}\frac{\begin{array}{c} TA \vdash^{W} c\colon \tau_c\ \text{mod}\ V \qquad T'T(A, x : \beta \to \tau_c, y\colon \beta) \vdash^{W} t\colon \tau_t\ \text{mod}\ (V \cup \{\beta\}) \qquad T''A \vdash^{W} e\colon \tau_e\ \text{mod}\ V \\ \tau_t \overset{U}{\sim} \tau_e\ \text{mod}\ V \qquad \beta\ \text{new} \qquad \beta \notin TV(UTA, U\tau_t) \end{array}}{UT''T'TA \vdash^{W} \texttt{ispair}\ c\ \texttt{bind}\ (x, y)\ \texttt{in}\ t\ \texttt{else}\ e\colon U\tau_e\ \text{mod}\ V}$$

Figure 11. Type Inference for *FCP*$_\varsigma$

language that supports both the spine view of data and a method of creating type indexed functions. The distribution of DGEN contains many example generic expressions and supports experimenting with the spine view we have described here.

4.1 Type Inference Correctness

We have proven the soundness and completeness of the type inference algorithm with respect to the type relation of Figure 9.

Theorem 4.1. *If* $TA \vdash^{W} e\colon \tau$ *mod* V *then* $(TA) \vdash e\colon \tau$

Proof. The proof proceeds by induction on the length of the type inference for $TA \vdash^{W} e\colon \tau$ mod V, with one case for each possible final step. The proof uses the same structure as the syntactic proof for type relation soundness and is routine. The fact that β is fixed-for-unification in I-IsPair and T-IsPair is important in the proof because it ensures the unification of τ_e ant τ_t will generate an appropriate unification U.

The main source of potential difficulty in this proof (and the next) is the existentially quantified variable and it's enforcement by adding fixed-for-unification variables to the unification algorithm. However, FCP itself allows existentially quantified variables and uses the fixed-for-unification variables to perform type inference for these. Thus everything required for the *FCP*$_\varsigma$ proof is already present in the equivalent proof for FCP. ☐

Theorem 4.2. *If* $(TA) \vdash e\colon \tau$ *then* $TA \vdash^{W} e\colon \tau$ *mod* V *where* $\tau' \succ \tau$

Proof. The proof proceeds by induction on the length of the type inference for $A \vdash e\colon \tau$, with one case for each possible final deduction rule. As above, the proof is a routine application of proof by structural induction. Side conditions must be used to ensure the required properties of unification and substitution application. ☐

We have also implemented a variant of type inference algorithm in the DGEN compiler. DGEN includes; the spine view of data encoded via ispair, the fully applied constructor view via case statements and function definitions, type-indexed functions via an extension primitive, and a run-time which supports both the spine view and the fully applied constructor view. Because the implementation in DGEN of the type inference algorithm predates this presentation of the work, it is not identical to the one we present here, but it is the same in all matters of substance. The interested reader can download DGEN and run the type inference algorithm over large code examples. A number of examples are included in the DGEN distribution including: generic traversal varying by traversal order and exit condition, generic show, generic equality, generic zip, generic map, and generic query [18].

4.2 FCP Correction

The algorithm in Figure 11 includes a correction to the original FCP algorithm which we now describe.

In the original formulation of FCP [10] the side condition on the I-Con rule (there called *make*) is given as $\alpha \notin TV(UTA)$. However, the algorithm with this side condition calculates the incorrect type for some terms. In particular, it will calculate the type $T\ \beta'$ (where β' is not bound in the type environment) for the term $K\ (\lambda x.x)$ in the context of the constructor K, $\sigma_K = \forall\alpha.(\forall\beta.\beta \to \alpha) \to T\ \alpha$. We have tracked the error to an implicit constraint on the type relation which is not explicitly respected in the type inference algorithm. Adding that $\alpha \notin TV(U\tau')$ to the side condition of I-Con restores the constraint, correcting the type inference algorithm.

5. Related Work

The Pattern Calculus The spine view is fundamental to the pattern calculus which is a calculus that sets pattern matching as the guiding principal of programming. We first saw data considered and pulled apart according to the spine view in Jay's early work [6] on the pattern calculus and later work [7] has taken the evaluation and typing of the spine view of data to great lengths, resulting in the pure pattern calculus [8].

Our work takes a slightly different approach to the semantics of the spine view than any of the pattern calculus formulations, but is largely in line with them. The primary difference is the restriction of the partially applied constructor to a variable bound inside an is-pair expression. It is this difference that allows us to formulate a type inference algorithm for this expression.

The pattern calculus gives a full account of evaluation using the spine view and extends it to allow *dynamic patterns*, which we do not consider here. The pattern calculus also has various type systems defined upon it. However, type inference for the pattern calculus is not described in any published work and the only documented version is in the source code of the bondi compiler. There it uses "aggressive assumptions" [7] to make inference work. What we present here is a type inference algorithm for a language very closely related to the pattern calculus, but we fully describe it, show its derivation, and prove its correctness.

The Spine View The spine view first came to broad attention through the Scrap Your Boilerplate (SYB) series of papers and libraries [14–16] although it was not known as such at the time. Work by Hinze et. al. [3, 5] exposed the spine view that underlies SYB. Our work is broadly compatible with both these systems but has a subtly different approach. In both these systems it is the evaluation mechanics and fitting the system into Haskell's type system that are of primary concern. By taking the very smallest and most fundamental version and exposing it in a simple language we have added a *specific* type system for this view, including a type inference algorithm.

Higher Ranked Type Systems In Section 2.1 we used the equivalence between System F and FCP to write functions that required higher ranked types but for which types could still be inferred. We have explained that the FCP approach is equivalent to using type annotation. There are other modern higher-ranked type systems that may be amenable to the same type of manipulation to cope with spine types, such as HMF by Leijen [13] and the practical higher ranked inference of Peyton Jones et. al. [17]. We expect that there is nothing in the spine view, or the is-pair expression, that would conflict with the type inference in either of these systems. The main challenge would be ensuring the existentially quantified type variable introduced by the is-pair expression is treated as such.

6. Future Work

DGEN describes a full implementation of generics and this work was a result of taking one part of DGEN's implementation and formalising it. We plan to repeat this process for other aspects, in particular the run-time representation and the system of type indexed functions.

Hinze and Löh [3] have shown how the spine view can be extended to support generic producers, it is our intention to apply their work to FCP_ς.

Presently we have an implementation of the ideas in this paper (in DGEN) and separately we have proofs of various properties of that system. While the simplicity of FCP_ς makes this approach tractable, it will quickly become unsupportable if the system is extended. Furthermore, it is currently not possible to prove properties of DGEN's implementation, nor is it possible to execute substantial examples in FCP_ς. A machine assisted proof of the properties in this paper with a system like Coq would allow the implementation and the theoretical basis to be unified and make future extensions of the system easier.

7. Conclusion

We have described a small extension of the lambda calculus that supports the spine view of data. We have given a type checking relation and described a type inference algorithm for this language that requires no extra type annotations to support the spine view. In this way we have created the smallest expression to date of the spine view of data and showed how it can be included in any functional programming language based on a Hindley-Milner type system. The type system has been proven sound and has been implemented in a compiler for a general purpose language.

References

[1] Artem Alimarine and Rinus Plasmeijer. A Generic Programming Extension for Clean. In *Selected Papers from the 13th International Workshop on Implementation of Functional Languages*, pages 168–185, London, UK, UK, 2002. Springer-Verlag.

[2] Ralf Hinze. Generics for the masses. *Journal of Functional Programming*, 16(4-5):451–483, 2006.

[3] Ralf Hinze and Andres Löh. "Scrap your boilerplate" Revolutions. In *In Tarmo Uustalu, editor, Mathematics of Program Construction, 2006, volume 4014 of LNCS*, pages 180–208. Springer-Verlag, 2006.

[4] Ralf Hinze and Andres Löh. Generic programming in 3D. *Science of Computer Programming*, 2009.

[5] Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. "Scrap Your Boilerplate" Reloaded. In *Functional and Logic Programming*, pages 13–29. Springer Berlin Heidelberg, 2006.

[6] Barry Jay. The Pattern Calculus. *ACM Transactions of Programming Languages and Systems (TOPLAS)*, 26(6):911–937, November 2004.

[7] Barry Jay. *Pattern Calculus*. Computing with Functions and Structures. Springer, July 2009.

[8] Barry Jay and Delia Kesner. Pure Pattern Calculus. In *Programming Languages and Systems: 15th European Symposium on Programming*, pages 100–114. Springer, 2006.

[9] Johan Jeuring, Andres Löh, and Ralf Hinze. Comparing approaches to generic programming in Haskell. Technical report, 2006.

[10] Mark Jones. First-class polymorphism with type inference. *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 483–496, 1997.

[11] Didier Le Botlan and Didier Rémy. MLF: raising ML to the power of system F. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 27–38, New York, New York, USA, August 2003. ACM Request Permissions.

[12] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):707–723, July 1998.

[13] Daan Leijen. HMF: simple type inference for first-class polymorphism. In *th International Conference on Functional Programming ICFP*, Victoria, BC, Canada, September 2008. ACM.

[14] Simon Peyton Jones and Ralf Lämmel. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the ACM SIGPLAN Workshop on Tyupes in Laguage Design and Implementation TLDI*, New Orleans, January 2003. ACM Press.

[15] Simon Peyton Jones and Ralf Lämmel. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 244–255, New York, NY, USA, 2004. ACM.

[16] Simon Peyton Jones and Ralf Lämmel. Scrap your boilerplate with class: extensible generic functions. In *th ACM SIGPLAN International Conference on Functional Programming*, pages 204–215, Tallinn, Estonia, September 2005. ACM.

[17] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1, January 2007.

[18] Matthew Roberts. *Compiled Generics for Functional Programming Languages*. PhD thesis, Macquarie University, 2011.

[19] Eelco Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, pages 216–238. Spinger-Verlag, June 2004.

[20] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1), November 1994.