What is Your Function? Static Pattern Matching on Function Behavior

Leandro Facchinetti[1], Pottayil Harisanker Menon[1], Zachary Palmer[2], Alexander Rozenshteyn[1], and Scott F. Smith[1]

[1] The Johns Hopkins University
[2] Swarthmore College

Abstract. We define a new notion of *function pattern*, which supports runtime pattern matching on functions based on their behavior. The ability to runtime dispatch on function type information enables new programmer expressiveness, including support of overloading on higher-order functions as well as other useful patterns.
We formally present a type inference system for function patterns. The system answers questions of function pattern matching by recursively invoking the type inference algorithm. The recursive invocation contains some delicate issues of self-referentiality that we address.

1 Introduction

Pattern matching is a prominent component of functional programming languages, as it is an excellent tool for case analysis and data destruction. It has also been extended in many dimensions: conjunction, disjunction, and negation patterns [3,10], combining guards with pattern matching [3], embedding computation via view patterns [14], and recursive patterns [8]. This paper extends pattern matching in another useful direction, support for *function* pattern matching. Function pattern matching is based on a function's *behavior*: the function pattern `int ~> char`, for instance, matches only those functions which will accept an `int` and return a `char`. Even in languages without subtyping, function patterns are meaningful; for example, in Haskell, the pattern `_:_ ~> Just _` could match exactly those functions which return a `Just` when passed a non-empty list. [3] Our approach is a hybrid of compile- and run- time pattern matching: the bulk of the work is done at compile time, and a runtime dispatch table aligns compile- and run-time decisions.

There are several potentially important uses of function patterns. First, they can be used to supplement the underlying type system. In this way, they are similar to refinement types [12], dynamic contracts [9], and static contracts [18]. Using function patterns to express type assertions is especially relevant in the context of subtype constraint type inference systems [1], which have notoriously unreadable types. Second, function patterns support dynamic dispatch based on the type of data an input function will be able to process. This is already possible on non-function data, especially in systems with structural subtyping: a function can essentially be overloaded by providing a case branch for each type

[3] Data types in Whayrf are expressed as unions of singleton typed records.

of input it can handle. Function patterns allow this form of dynamic dispatch to be extended to higher-order functions.

There are significant technical challenges, however, including non-local variables and function parameters. And, it is possible for function pattern matching answers to be interdependent – a naive system will contain paradoxes of self-reference ("I match this pattern if and only if I do not"). Dealing with this paradoxical case is the biggest technical challenge.

Outline In the next section, we give an informal overview of higher-order function patterns. In Section 3, we give the operational semantics and type system for Whayrf (pronounced "wharf"), a small language with function patterns. In this section we assume as given a function pattern dispatch relation answering how a given function on a given pattern will dynamically dispatch. Then, in Section 4 we present a methodology for the inferring such a dispatch relation in a way that handles the paradoxes of self-reference that arise. We review related work in Section 5 and conclude in Section 6.

An implementation of Whayrf which supports all examples in the overview can be found on Github.[4]

2 Overview

This section provides an informal overview of function patterns in both meaning and use.

2.1 Whayrf's Semantics in Brief

Whayrf is a simple core language: it includes only higher-order functions, pattern matching, records, and integers. The pattern matching semantics of Whayrf admit structural subtyping. Our formal presentation is over a further simplified version which eschews integers and only allows a degenerate form of records.

Whayrf's type system is based on subtype constraint systems, which have been shown to be expressive [1] and suitable for complex pattern matching [17]. These systems are usually characterized by the presence of a *type constraint closure* algorithm for deductively closing subtype constraints: a set of subtyping constraints are inferred for the expression to be typechecked and, if the transitive closure of those constraints produces no inconsistencies, then a sound inferred type exists.While the paper focuses on subtype inference, the general principles should be adaptable to other forms of type inference and program analysis.

2.2 Function Patterns

This paper aims to fill a gap in functional programming language designs: while functions are first-class data in functional languages, pattern matching is generally not extended to allow functions to be matched on. To recify this situation, Whayrf introduces the notion of *function patterns*. While tuple, list, and other patterns match data based on its shape, function patterns match functions based on their *behavior*: a function pattern $\pi_1 \rightsquigarrow \pi_2$ matches an argument which (1) is a function, (2) takes any argument matching the pattern π_1, and (3) returns a

[4] https://github.com/JHU-PL-Lab/whayrf

result matching the pattern π_2. We use arrow `~>` in function patterns to disambiguate function patterns from function types. Consider the following example:

```
1 let callWithDefault f = case f of
2         {} ~> int as g -> g {}
3         int ~> int as g -> g 1 in
4 callWithDefault (fun n -> n + 3)
```

Here, `callWithDefault` accepts a function and passes it an appropriate argument. The match on line 3 is invoked since that is the first case clause matching the argument's type. Observe that there is no type signature in `fun n -> n + 3` dictating this behavior. Instead, a type is inferred for the argument and that type is used to construct a runtime dispatch table for the case clause.

To construct this dispatch table we must answer what looks like a subtyping question: is the function's type a subtype of the interface's type? The problem with this general question is its complexity: since types are described by general subtype constraints, this subtyping question appears to be undecidable [13].

Our solution is to solve a simpler problem, whether the function argument can be viewed as matching the pattern π_1 `~>` π_2. We in turn compute this answer by introducing a type representing π_1 and performing a *subordinate* typecheck: the function matches the pattern iff the application of the higher-order function argument to the type representing π_1 successfully typechecks and matches π_2.

Consider in more detail the case analysis typing of the above example. In the first case, we attempt to match the argument with the pattern `{} ~> int`. For this, pattern matching creates an "unknown" type bounded above by `{}` to which we apply the function `fun n -> n + 3`. In this subordinate typechecking process a record is added, a type error, and so the pattern fails to match. So, the next case clause is considered with pattern `int ~> int`; we repeat the same process but this application typechecks and thus the pattern matches.

2.3 Self-reference

The above approach work well until self-referential functions are considered. The difficulty arises from the fact that the pattern matching and typechecking are interdependent. [5] We can then easily produce a paradoxical pattern match as in the `russell` function of Figure 2.1.

```
1 let rec selfMatch n:int =
2     case selfMatch of int ~> {} -> {}
3 in selfMatch 4
```

```
1 let rec russell n:int =
2     case russell of
3         int ~> {} -> 0
4         _ -> {}
5 in russell 4
```

Fig. 2.1. Examples of Self-Reference

In the above, a naive pattern match would proceed by typechecking the application of `russell` to an `int` argument. But this succeeds iff it fails, a paradox.

A simple resolution of the paradox would be to disallow cyclic dependencies, but this can be too strict since it prevents encoding `instanceof` on an object's

[5] This code uses syntactic sugar for recursion. It can be encoded using standard means, and subtype constraint systems are powerful enough to infer the resulting types.

own type, such as is found in Java `equals` methods. Instead, we wish to disallow cycles which do not have a consistent dispatch (like `russell`) while allowing those that do. For this purpose, we infer a dependency graph of function pattern matches in Section 4.2, and use it to rule out (fail to typecheck on) programs with paradoxes.

2.4 Applications

Here we present potential applications of function pattern matching.

Type signatures Subtype constraint systems are powerful, but they produce complex types which are difficult to read. One of the goals of function patterns is to address this shortcoming: pattern matching with function patterns is powerful enough to use pattern matching to *encode* common type assertions and restrictions. For instance, `let f x = ... : int -> int` may be encoded as:

```
let f = let g x = ... in // Assertion
        case g of int ~> int -> g
```

```
let f = let g x = ... in // Restriction
        let h x:int = let r y:int = y
        in r (g x)
        in case g of int ~> int -> h
```

Fig. 2.2. Type assertion and restriction

Here, the `case` expression matches the function `g` with exactly one pattern; if the pattern does not match, then the match is not exhaustive and a type error occurs.

Note that this example demonstrates a type *assertion* and not a type *restriction*: while we have established that `f` matches `int ~> int`, it may also be able to operate on other types. If we wish to restrict the type of our function, we can accomplish this by eta-converting with the domain part of the function pattern.

Overloading Higher-Order Functions Case analysis in a system with structural subtyping allows a form of overloading: a function accepting e.g. `int`∪`char` may act differently on one than the other. Function patterns allow overloading on *higher-order* functions which, in tandem with type inference, we believe to be novel in the literature. For instance, consider a scenario in which the following function is defined in a library:

```
let nest num:int record:{} =
    if num < 0 then {}
    else if num == 0 then record
    else {a = nest (num - 1) record}
```

```
let nest num:int arg =
    if num < 0 then {}
    else if num == 0 then
            case arg of
                {} -> arg
                {} ~> {} as fn -> fn {}
    else {a = nest (num - 1) arg}
```

Fig. 2.3. Before and after overloading

This toy function creates a record nested `num` times. A potentially desirable overloading could be to allow the function to be called with a record-generating *function*, to be called when the record is required.

Here, we case analyze on the argument to determine if it is a record or a (record-generating) function. In general, this mechanism extends to function data the same form of overloading already possible on non-function data.

Behavior similar to the above can be achieved using typeclasses [19], but type classes dispatch on *completely static* type information and not on any information at run-time. Consider the following code:

```
let inform listener =
    case listener of
        ({a:int,b:int} ~> int) as f -> f {a=4,b=8}
        ({a:int} ~> int) as f -> f {a=4}
```

A typeclass version of the above would rely upon the static knowledge of the argument's type where `inform` is called. Function patterns, on the other hand, rely upon the static type inferred for the value that is passed to `inform` at runtime. This subtle difference means that e.g. client code need not be fully aware of the type of the function it is passing to `inform` in order for the first branch to be invoked. In summary, overloading encoded by function patterns is a form of *dynamic* dispatch whereas overloading encoded by typeclasses is a *static* dispatch.

Structural `instanceof` With function patterns, Whayrf is also capable of encoding dispatch on *structural* types; we believe this to be novel in the literature. For instance, the relevant distinction between two classes may be their response to a `damage` message. One may write:

```
let collide message = case message of
      {a:{damage: {} ~> bool} as ship} -> ship.damage()
```

In essence, "`{damage: {} ~> bool}`" is the pattern of a damageable object; this is effectively a form of structural `instanceof`, like Ruby's `respond_to` or Smalltalk's `respondsTo:`.

3 Core Formalism

In this section, we present a simplified version of the Whayrf language given in the overview. Our approach to typechecking is derived from [15]. This approach is especially suited to function patterns but is somewhat unusual. For this section, we will assume that an already computed dispatch table for function pattern matches is available; we will show how to compute this table in Section 4. This approach allows us to introduce our typechecking approach separately from function pattern match inference.

To avoid distraction from the focus of this paper, we do not formalize integers or full records. For illustration purposes, we instead include so-called "degenerate" records: sets of static labels with no associated data, such as `{a,b}`.

3.1 A-Normal Form

To simplify our formal presentation, we use a shallow, A-normalized expression grammar. A-normal form (ANF) grammars are those in which all operands are trivial (i.e. variables or literal values) [11]; our A-normal form is shallow in

that only variables are permitted as operands. This form facilitates alignment between the operational semantics and the constraint-based type system; we discuss this alignment further in Section 3.3. As a further restriction, we assume that each variable is assigned at most once in the program. The combination of shallow ANF and unique variable assignments gives a one-to-one mapping between variable names and program points and allows us to avoid discussion of complexities such as variable shadowing.

$e ::= \overrightarrow{s}$	*expressions*	$s ::= x = r$	*clauses*
$r ::= v \mid x \mid x\ x \mid x \sim \pi\ ?\ f : f$	*right-hand sides*	$E ::= \overrightarrow{x = v}$	*environment*
$f ::= x \texttt{ -> } e$	*functions*	$v ::= R \mid f$	*values*
$\pi ::= R \mid \pi \texttt{ ~> } \pi$	*patterns*	$R ::= \{l, \ldots\}$	*records*
x	*variables*	l	*field labels*

Fig. 3.1. Expression Grammar

Throughout this document, we write lists of length n as $[g_1, \ldots, g_n]$, and use $||$ for list concatenation. We may write concatenation of a non-list as shorthand for singleton list concatenation when it is unambiguous: $g_1 ||[g_2, g_3] = [g_1, g_2, g_3]$. We also use $\{g, \ldots\}$ to refer to a (possibly empty) set of g. Given this notation, the grammar of our language appears in Figure 3.1.

Of note is the case analysis right-hand side of the form $x \sim \pi\ ?\ f_1 : f_2$. This expression matches the contents of the variable x with the pattern π. On success, the argument is passed to f_1; otherwise, it is passed to f_2. This binary pattern matching operation is sufficient to encode the examples presented in Section 2.

From here on, we assume that expressions are closed unless otherwise noted.

3.2 Operational Semantics

We now define an operational semantics for ANF Whayrf as a small step relation $e \longrightarrow^1 e'$. We must freshen variables as they are introduced to the expression to preserve the invariant that each variable is uniquely defined. We take an unusually formal approach to variable freshening because it considerably simplifies our proofs to closely align the operational semantics and the type system, which mandates deterministic freshening to guarantee termination. To select fresh variables in our operational semantics, we define an injective function $\boldsymbol{\alpha}(x, x') = x''$ that produces a fresh variable x'' from x' (the variable we are freshening) and x (the point in the program being freshened). For simplicity, a reader may simply interpret $\boldsymbol{\alpha}(-, -) = x''$ to read "x'' is fresh".

For convenience, we write $\boldsymbol{\alpha}(x, v)$ to indicate the freshening of all bound variables in v. We also write $\textsc{rv}(e)$ to refer to the return variable of e (that is, the variable assigned in e's last clause).

We define the operational semantics as a small step relation $e \longrightarrow^1 e'$ (Figure 3.2). The operational semantics is largely standard but, due to our use of a shallow A-normal form, it is neither precisely substitution-based nor environment-based. The aforementioned figure makes use of a *compatibility relation*, $v \backslash E \sim \pi$ (read as: "v in environment E matches π"), the rules for which appear in Figure 3.3. The compatibility relation is used to factor pattern matching out of the Conditional small step rules. In matching function patterns, the

$$\textsc{Variable Lookup}\quad \frac{E(x_2) = v}{E \,||\, x_1 = x_2 \,||\, e \longrightarrow^1 E \,||\, x_1 = v \,||\, e}$$

$$\textsc{Application}\quad \frac{E(x_2) = x_4' \texttt{->} e_1' \qquad \boldsymbol{\alpha}(x_2, x_4' \texttt{->} e_1') = x_4 \texttt{->} e_1}{E \,||\, x_1 = x_2\ x_3 \,||\, e_2 \longrightarrow^1 E \,||\, x_4 = x_3 \,||\, e_1 \,||\, x_1 = \mathrm{RV}(e_1) \,||\, e_2}$$

$$\textsc{Conditional Success}\quad \frac{f_1 = x_3 \texttt{->} e_2 \qquad E(x_2) = v \qquad v\backslash E \sim \pi}{E \,||\, x_1 = x_2 \sim \pi \,?\, f_1 : f_2 \,||\, e_1 \longrightarrow^1 E \,||\, x_3 = v \,||\, e_2 \,||\, x_1 = \mathrm{RV}(e_2) \,||\, e_1}$$

$$\textsc{Conditional Failure}\quad \frac{f_2 = x_3 \texttt{->} e_2 \qquad E(x_2) = v \qquad \neg\, v\backslash E \sim \pi}{E \,||\, x_1 = x_2 \sim \pi \,?\, f_1 : f_2 \,||\, e_1 \longrightarrow^1 E \,||\, x_3 = v \,||\, e_2 \,||\, x_1 = \mathrm{RV}(e_2) \,||\, e_1}$$

Fig. 3.2. Small Step Operational Semantics

compatibility relation defers to the *dispatch relation* $f \precsim \pi_1 \texttt{->} \pi_2$. We discuss the properties of this relation in Section 3.4 and define it in Section 4.1.

$$\textsc{Function Match}\quad \frac{f \precsim \pi}{f\backslash E \sim \pi} \qquad\qquad \textsc{Record Match}\quad \frac{\pi = R \qquad R \subseteq R'}{R'\backslash E \sim \pi}$$

Fig. 3.3. Compatibility

This dispatch relation is a key focus of this paper. One intuition is to view the dispatch relation as a dispatch table, with functions as rows and function patterns as columns. We show in Section 4 how this dispatch relation can be inferred during typechecking. The dispatch table is finitized through generalizations made by the type system; we discuss this further in Section 3.4.

It is helpful to define further notation to aid in later discussion of the operational semantics. Given the above small step relation, we define $e_0 \longrightarrow^* e_n$ to hold when $e_0 \longrightarrow^1 \ldots \longrightarrow^1 e_n$ for some $n \geq 0$. Note that $e \longrightarrow^* E$ means that computation has resulted in a final value. We write $e \not\longrightarrow^1$ iff there is no e' such that $e \longrightarrow^1 e'$; observe $E \not\longrightarrow^1$ for any E. When $e \not\longrightarrow^1$ for some e not of the form E, we say that e is *stuck*.

3.3 Type System

Recall that subtype constraint systems [1,2,17,15] first infer constraints for an expression and then perform a deductive closure of those constraints to search for inconsistencies. To simplify the presentation, we give only a monomorphic system; polymorphism is left to future work.

Initial Constraint Generation Figure 3.4 provides a grammar of the type system. Of particular note is the constraint corresponding to the conditional $\alpha \sim \pi \ ?\ \phi_1 : \phi_2$. Much of the power of function patterns comes from precise analysis of the paths that data can take through the program. The form of this

$$
\begin{array}{lll}
C ::= \{c, \ldots\} & \textit{constraint sets} & t ::= \alpha\backslash C \quad \textit{constrained types} \\
b ::= \tau \mid \alpha \mid \alpha\ \alpha \mid \alpha \sim \pi\ ?\ \phi : \phi & \textit{lower bounds} & c ::= b <: \alpha \quad \textit{constraints} \\
\phi ::= \alpha \texttt{->} t & \textit{function types} & \tau ::= \{l, \ldots\} \mid \phi \quad \textit{types} \\
\alpha & \textit{type variables} &
\end{array}
$$

Fig. 3.4. Type Grammar

constraint facilitates precise analysis by allowing us to avoid introducing spurious constraints from conditional branches that are not explored; this is somewhat reminiscent of conditional constraints [16].

We choose this constraint grammar and the A-normalized expression grammar so that they maintain a close syntactic and semantic correspondence. Each expression in the expression grammar aligns closely with some term in the type grammar: values v are paralleled by types τ, expressions e are paralleled by constrained types $\alpha\backslash C$, and so on. For each rule in the operational semantics, we define a type system analogue. As a result, the constraint closure step of type inference bears similarities to an abstract interpretation [7]. We use this fact in Section 4 to drive our reasoning about function patterns.

We formalize this alignment as a function $[\![e]\!]$ which produces a constrained type $\alpha\backslash C$ (Figure 3.5). We assume a variable alignment function $[\![x]\!] = \alpha$ which is injective for the set of variables appearing in the original program but maps each freshening of a given x aligns to the same type variable; i.e. $[\![x]\!] = [\![\boldsymbol{\alpha}(x', x)]\!]$ for every x and x'. This interaction between freshening and alignment is relevant for dispatch and is discussed further in Section 4.1.

$$
\begin{array}{rcl}
[\![\overrightarrow[n]{s}]\!] & = & \alpha_n\backslash\{c_i \mid i \in \{1..n\}, [\![s_i]\!] = \alpha_i\backslash c_i\} \\
 & & \text{where } [\![s_n]\!] = \alpha_n\backslash c_n \\
 & & \\
[\![x_0 \texttt{=} R]\!] & = & [\![x_0]\!]\backslash\ r <: [\![x_0]\!] \\
[\![x_0 \texttt{=} f]\!] & = & [\![x_0]\!]\backslash\ [\![f]\!] <: [\![x_0]\!] \\
[\![x_0 \texttt{=} x_1]\!] & = & [\![x_0]\!]\backslash\ [\![x_1]\!] <: [\![x_0]\!] \\
[\![x_0 \texttt{=} x_1\ x_2]\!] & = & [\![x_0]\!]\backslash\ [\![x_1]\!]\ [\![x_2]\!] <: [\![x_0]\!] \\
[\![x_0 \texttt{=} x_1 \sim \pi\ ?\ f_1 : f_2]\!] & = & [\![x_0]\!]\backslash\ [\![x_1]\!] \sim \pi\ ?\ [\![f_1]\!] : [\![f_2]\!] <: [\![x_0]\!] \\
 & & \\
[\![f]\!] & = & [\![x_1]\!] \texttt{->} [\![e]\!] \\
 & & \text{where } f = x_1 \texttt{->} e
\end{array}
$$

Fig. 3.5. Alignment

We will sometimes write $[\![e]\!]$ to refer to only the constraint set in the result.

Constraint Closure The initial alignment provides an initial set of inferred constraints for the program. Next, we work toward defining a constraint closure relation which propagates data forward, by paralleling the operational semantics, an easy task given the close syntactic alignment between the two systems.

In analogy with the value compatibility relation in Section 3.2, we define a *type compatibility relation* to determine whether there is any intersection between a type and a pattern. The rules for this relation are given by Figure 3.6. Type compatibility is written $\tau\backslash C \overset{\odot}{\sim} \pi$ and has four places: the type being examined (τ), the constraint set in which this examination is being done (C), the pattern the type is being matched against (π), and $\odot$, a metavariable ranging over $\bullet$

$$\textsc{Record Match}\quad \frac{\tau = \{\ldots\} \qquad \pi = \{\ldots\} \qquad \pi \subseteq \tau}{\tau \backslash C \overset{\bullet}{\sim} \pi} \qquad \textsc{Record Anti-Match 1}\quad \frac{\tau = \{\ldots\} \qquad \pi = \{\ldots\} \qquad \pi \not\subseteq \tau}{\tau \backslash C \overset{\circ}{\sim} \pi}$$

$$\textsc{Record Anti-Match 2}\quad \frac{\tau = \{\ldots\} \qquad \pi \neq \{\ldots\}}{\tau \backslash C \overset{\circ}{\sim} \pi} \qquad \textsc{Dispatch Positive}\quad \frac{\phi \precsim \pi}{\phi \backslash C \overset{\bullet}{\sim} \pi} \qquad \textsc{Dispatch Negative}\quad \frac{\neg\ \phi \precsim \pi}{\phi \backslash C \overset{\circ}{\sim} \pi}$$

Fig. 3.6. Type Compatibility Rules

$$\textsc{Transitivity}\quad \frac{\{\tau <: \alpha_2, \alpha_2 <: \alpha_1\} \subseteq C}{C \overset{\mathrm{D}}{\Longrightarrow}^1 C \cup \{\tau <: \alpha_1\}}$$

$$\textsc{Application}\quad \frac{\{\alpha_1\ \alpha_2 <: \alpha_3, \tau_1 <:\ \alpha_1, \tau_2 <: \alpha_2\} \subseteq C \qquad \tau_1 = \alpha_4 \texttt{->} \alpha' \backslash C'}{C \overset{\mathrm{D}}{\Longrightarrow}^1 C \cup C' \cup \{\tau_2 <: \alpha_4, \alpha' <: \alpha_3\}}$$

$$\textsc{Conditional Success}\quad \frac{\{\alpha_1 \sim \pi\ \texttt{?}\ \phi_1 : \phi_2 <: \alpha_2, \tau <: \alpha_1\} \subseteq C \qquad \phi_1 = \alpha_3 \texttt{->} \alpha_4 \backslash C' \qquad \tau \backslash C \overset{\bullet}{\sim} \pi}{C \overset{\mathrm{D}}{\Longrightarrow}^1 C \cup C' \cup \{\tau <: \alpha_3, \alpha_4 <: \alpha_2\}}$$

$$\textsc{Conditional Failure}\quad \frac{\{\alpha_1 \sim \pi\ \texttt{?}\ \phi_1 : \phi_2 <: \alpha_2, \tau <: \alpha_1\} \subseteq C \qquad \phi_2 = \alpha_3 \texttt{->} \alpha_4 \backslash C' \qquad \tau \backslash C \overset{\circ}{\sim} \pi}{C \overset{\mathrm{D}}{\Longrightarrow}^1 C \cup C' \cup \{\tau <: \alpha_3, \alpha_4 <: \alpha_2\}}$$

Fig. 3.7. Constraint Closure

(indicating a witness has been found which matches) and $\circ$ (indicating that a witness has been found which does not). To distinguish between $\neg\ \tau \backslash C \overset{\bullet}{\sim} \pi$ and $\tau \backslash C \overset{\circ}{\sim} \pi$, we say *anti-match* when referring to the latter. This notion of constructive failure is preferable to the alternative in that it is monotone.

Recall that we described a runtime function pattern dispatch relation $v \precsim \pi$ in Section 3.2; evaluation is defined to be parametric in this relation. Analogously, we define closure here to be parametric in a type-level function pattern dispatch relation $\tau \precsim \pi$; this is the solution to the function pattern dispatch problem, as we assumed above. By deferring to this relation, type compatibility for a particular function and function pattern relates as exactly one of $\bullet$ or $\circ$. In Section 4.1, we will construct a system which infers a dispatch relation, but the system presented here only checks the consistency of a given dispatch.

Compatibility for records is unsurprising: a record type can be proven to match a record pattern if the type has at least the fields the pattern requires; a record type can be proven to anti-match a pattern if the pattern demands a field which is absent or if the pattern is not a record pattern.

Due to the correspondence between the expression and constraint grammars, the deductive closure of constraints proceeds in analogy with the small step relation of the operational semantics. Each step of closure represents one forward

propagation of constraint information and abstractly models a single step of the operational semantics. We define the constraint closure relation as follows.

Definition 1. *We write* $C \overset{\mathrm{D}}{\Longrightarrow}^1 C'$ *to indicate a single step of constraint closure. This relation is defined by the rules in Figure 3.7. In analogy with the operational semantics, we define* $C_0 \overset{\mathrm{D}}{\Longrightarrow}^* C_n$ *to hold when* $C_0 \overset{\mathrm{D}}{\Longrightarrow}^1 \ldots \overset{\mathrm{D}}{\Longrightarrow}^1 C_n$ *for some* $n \geq 0$. *We also define* $C_0 \overset{\mathrm{D}}{\Longrightarrow}^\omega C_n$ *to hold when* $C_0 \overset{\mathrm{D}}{\Longrightarrow}^* C_n \wedge C_n \overset{\mathrm{D}}{\Longrightarrow}^1 C_{n+1}$ *only if* $C_n = C_{n+1}$ *(that is, when* C_n *is completely closed).*

The operational semantics has a definition for a "stuck" expression; the type system analogue is the immediately inconsistent constraint set, which arises when a non-function is called:

Definition 2 (Immediate Inconsistency). *A constraint set* C *is* immediately inconsistent *in one case: a non-function type is called. That is,* $\{\alpha_1\ \alpha_2 <: \alpha_3, \tau_1 <: \alpha_1, \tau_2 <: \alpha_2\} \subseteq C \wedge \tau_1 \neq \alpha \texttt{->} t \backslash C$. *A constraint set which is not immediately inconsistent is* immediately consistent.

Definition 3 (Inconsistency). *A constraint set* C *is* eventually inconsistent *or simply* inconsistent *if it closes to an immediately inconsistent constraint set. That is,* $C \overset{\mathrm{D}}{\Longrightarrow}^* C' \wedge C'$ *is immediately inconsistent. A constraint set which is not eventually inconsistent is* always consistent *or simply* consistent.

We can now define what it means for a program to be type correct:

Definition 4 (Typechecking). *A closed expression* e typechecks *iff* $[\![e]\!] = \alpha \backslash C$ *and* C *is consistent.*

We defer the discussion of the soundness of this typechecking process to Section 3.4; however, we state the decidability here.

Theorem 1 (Decidability). *Whether a closed expression typechecks is decidable if the dispatch relation is decidable.*

Lemma 1 (Confluence). *If* $C \overset{\mathrm{D}}{\Longrightarrow}^1 C_1$ *and* $C \overset{\mathrm{D}}{\Longrightarrow}^1 C_2$, *then* $\exists C'$. $C_1 \overset{\mathrm{D}}{\Longrightarrow}^* C'$ *and* $C_2 \overset{\mathrm{D}}{\Longrightarrow}^* C'$.

Theorem 2 (Fixpoint Computability). $\overset{\mathrm{D}}{\Longrightarrow}^\omega$, *the fixpoint of* $\overset{\mathrm{D}}{\Longrightarrow}^1$, *is a computable function.*

3.4 Properties of Dispatch

The dispatch relation $v \precsim \pi$ is used only when v is a function value. It can be viewed as a large lookup table: entries in this table correspond to pattern/function pairs such that any pattern match that could occur during the execution of the program has a corresponding entry. Note that we will generate this dispatch table from type information; this leads us to make necessarily conservative conclusions about function behavior but also serves to finitize the dispatch table. These conservative conclusions are not draconian; due to the nature of initial alignment, each syntactically distinct function has its own row in the dispatch table. Since there is a bijection between our function types and syntactic functions (due to monomorphism), we can use the same sort of table during type

checking as we would during execution. The type system presented in this paper is monomorphic; a polymorphic system's dispatch table would additionally distinguish between different polyinstantiations of the same function.

Above, we use the notation $\tau \precsim \pi$ to describe the form of a type-level dispatch relation. In this section, we discuss the properties of such relations; we thus use δ to index over such relations. Analogously, we use d to index over runtime dispatch relations of the form $v \precsim \pi$. Where convenient, we identify these relations with their indices. In both cases, we require the relations to be decidable.

Recall that we intend for the type system, in the course of closure, to infer $\precsim_d$ for the operational semantics to use. In particular, we will extract this relation from the final constraint set. We use γ to range over functions which may be used to extract such relations.

In Section 4.1, we will define a closure which does not rely on an external definition of dispatch, and in Section 4.1 we will define the $\precsim_d$ we intend to use; however, for this section, we will remain parametric in the dispatch relations and the dispatch extraction function so that we can discuss their properties.

What properties are desirable in dispatch? First and foremost, we would like the relations for typechecking and evaluation to correspond; otherwise, the evaluation paths that are being typechecked are not those that are being executed.

Definition 5 (Safety). *A dispatch relation δ is* safe *with respect to a dispatch relation d iff $f \precsim_d \pi \iff [\![f]\!] \precsim_\delta \pi$ whenever both are defined.*

We also want the dispatch relation to be defined at any pair that matters.

Definition 6 (Coverage). *A dispatch relation δ* covers *a dispatch relation d iff $[\![f]\!] \precsim_\delta \pi$ is defined whenever $f \precsim_d \pi$ is defined. A dispatch relation d* covers *an expression e iff $f \precsim_d \pi$ is defined for every pair of f and π that occur while executing e. We will also say that a dispatch relation δ covers an expression e if there is some d such that the above hold.*

Soundness follows directly.

Theorem 3 (Soundness). *If $e \longrightarrow^* e'$ under dispatch relation d such that e' is stuck, dispatch relation δ is safe with respect to d, δ covers d, and d covers e, then e does not typecheck under dispatch relation δ.*

And what properties are desirable in a dispatch extraction function? There are many d/δ pairs which are technically safe but not at all usable (e.g. when both relations reject all matches, leading to the anti-match of every function pattern). We desire an extraction function which conforms to the decisions we made during constraint closure:

Definition 7 (Stability). *With respect to a program e and an extraction function γ, a dispatch relation δ is* stable *iff $[\![e]\!] \overset{\mathrm{D}}{\Longrightarrow}^\omega C$ such that $\gamma(C) = \delta$. We elide e and γ if they are evident from context.*

We also want pattern match success to correspond to programmer intuitions:

Definition 8 (Sensibility). *A dispatch relation d is* sensible *w.r.t. a program e, function f, and pattern π ~> π' iff, for all values v and environments E where $f \precsim_d \pi$ ~> π', $v \backslash E \sim \pi$, $E(x_1) = f$, and $E(x_2) = v$, the following are true:*

1. $E \,||\, x_3 = x_1\ x_2$ *does not get stuck,*
2. *If* $E \,||\, x_3 = x_1\ x_2 \longrightarrow^* E' \,||\, x_3 = v'$, *then* $v' \backslash E' \sim \pi'$.

We say a dispatch relation d *is* sensible *with respect to a program* e *if it is sensible with respect to all of that program's function-pattern pairs. We elide* e *when evident from context.*

Note that sensibility does not address cases in which the dispatch relation does not hold. There is very little that can be guaranteed at runtime about a function which does not match a function pattern, since this means that *some* input causes the function to behave in a way contrary to the above expectation.

Theorem 4. *The stability property is decidable.*

4 Inferring Function Dispatch

The task of inferring dispatch for Whayrf appears cyclic: in order to dispatch functions, we need to use information gathered from type checking; but in order to typecheck a program, we must know how functions are dispatched. Fortunately, this problem is largely illusory: most matches do not have cyclic dependencies and we can typically order the closure to defer matches until their dependencies are met. For the cases with cycles we present a novel cycle resolution algorithm below. We begin in Section 4.1 by showing how the type system can be changed to infer function pattern dispatch given a dependency graph of matches in the program. In Section 4.2, we show how that dependency graph may be inferred by repurposing constraint closure.

By the end of this section, this paper will have defined three distinct closure relations. The first closure relation was defined previously in Definition 1, which unloads all function pattern work to an external relation and is discussed in Section 3.4. The second closure relation will be defined in Definition 9, which infers a dispatch table but relies on a dependency graph. Finally, the third closure relation will be defined in Section 4.2 which computes the dependency graph.

Running a program then involves the following steps (Figure 4.1). First, we determine the initial constraint set by performing initial alignment and then using dependency inference closure on that constraint set; that process generates a dependency graph. Then, we perform dispatch inference closure on the original constraint set using this dependency graph; that process infers a dispatch table, which can then be used by the operational semantics to run the program.

4.1 Dispatch Inference

In this section, we show how to construct a sensible (*a la* Definition 8) dispatch relation during the course of type inference; the relation constructed is decidable and can be used in the previous system. As a side-effect of inferring dispatch, though, we end up showing that the program does not get stuck, eliminating the need to typecheck separately (see Theorem 5). We start off by extending the grammar with auxiliary notation needed to help in construction of the relation.

We add four new constraint forms and one new type, shown in Figure 4.2. The $\alpha <: \pi$ constraint is an assertion that a type variable matches a pattern; it is

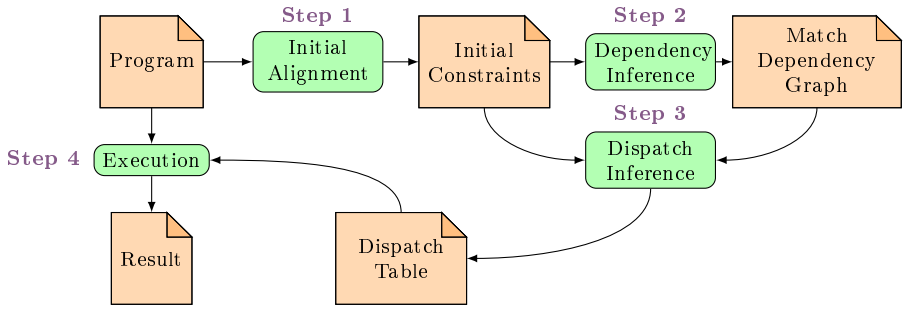

Fig. 4.1. Running a Program with Function Patterns

$$\begin{array}{lcll} \tau & ::= & \cdots \mid \bullet\lceil\pi & \textit{types} \\ c & ::= & \ldots \mid \tau \overset{+}{\sim} \pi \mid \tau \overset{-}{\sim} \pi \mid \tau \overset{\times}{\sim} \pi \mid \alpha <: \pi & \textit{constraints} \end{array}$$

Fig. 4.2. Changes to the type grammar

only added to check return values of functions during dispatch relation inference. Similarly, the *suppression constraint* $\tau \overset{\times}{\sim} \pi$ is used to guide closure as performed by auxiliary functions. The two other constraints, referred to as *dispatch constraints*, assert a type matches $(+)$ or anti-matches $(-)$ a pattern, and each such constraint can be viewed as a "cell" of the overall function dispatch table. The new *unknown* type $\bullet\lceil\pi$ represents some type which matches the pattern π.

Unknowns Suppose we are trying to determine whether a function `f` should match or a function pattern $\pi = \pi_1$ `~>` π_2. A first attempt could be to create a type from π_1, pass it into `f` as an argument, and then check that each result of `f` matches π_2. Although this is intuitive, it is also incorrect in Whayrf due to the presence of subtyping and implicit runtime type analysis. Consider the following function (written in the intuitive language presented in the overview):

```
1 let f x = case x of
2             {a:int,b:int} -> {}
3             {a:int} -> 0
```

This function should not match the pattern `{a:int} ~> int` because there are some values of type `{a:int}` (e.g. `{a=0,b=0}`) for which the function does not return an `int`. If our pattern matching process creates an argument of type `{a:int}`, the subordinate closure will not encounter this second branch and the function will appear to match. Instead of testing a specific, concrete type as an input to the function, we need a way of testing *all subtypes* of `{a:int}` at once.

We use *unknown types*, $\bullet\lceil\pi$, to solve this problem. An unknown type represents an existential type restricted by the pattern in π. For example, $\bullet\lceil\{l\}$ represents a record type which has an l field. This is distinct from the type $\{l\}$ (the record having exactly the l field), from α where $\{l\} <: \alpha$ (a type which may be such a record but might also be any other type lower bounding α), and from $\alpha <: \{l\}$ (which requires α to be a record with a l field and is inconsistent otherwise). Unknown types never produce inconsistencies as a result of the bounding π. Our unknown types are related to the $\bullet$ of [18].

Compatibility For our inference system, we define type compatibility using the rules in Figures 3.6 and 4.4, the latter of which handles function patterns (by deferring to previously inferred dispatch constraints, e.g. $\tau \overset{+}{\sim} \pi$) and unknowns.

$$\textsc{Record}\quad \frac{r \supseteq r'}{r \sqsubseteq r'} \qquad\qquad \textsc{Function}\quad \frac{\pi_1' \sqsubseteq \pi_1 \qquad \pi_2 \sqsubseteq \pi_2'}{\pi_1 \texttt{ ~> } \pi_2 \sqsubseteq \pi_1' \texttt{ ~> } \pi_2'}$$

Fig. 4.3. Pattern Subsumption

$$\textsc{Unknown Match}\quad \frac{}{\bullet\lceil \pi' \backslash C \overset{\bullet}{\sim} \pi}$$

$$\textsc{Unknown Anti-Match}\quad \frac{\pi' \not\sqsubseteq \pi}{\bullet\lceil \pi' \backslash C \overset{\circ}{\sim} \pi}$$

$$\textsc{Function Match}\quad \frac{\phi \overset{+}{\sim} \pi_1 \texttt{ ~> } \pi_2 \in C}{\phi \backslash C \overset{\bullet}{\sim} \pi_1 \texttt{ ~> } \pi_2}$$

$$\textsc{Function Anti-Match}\quad \frac{\phi \overset{-}{\sim} \pi_1 \texttt{ ~> } \pi_2 \in C}{\phi \backslash C \overset{\circ}{\sim} \pi_1 \texttt{ ~> } \pi_2}$$

Fig. 4.4. Type Compatibility Extension

The compatibility rules reflect the existential nature of unknowns. The anti-match rule states that an unknown anti-matches a pattern unless it is guaranteed to match the pattern by π. The match rule claims that an unknown matches any pattern. The reason for this is that no pattern that a type is known to match can lead us to conclude that it anti-matches some other pattern. Unknowns also exhibit the property that matching and anti-matching are not mutually exclusive. This is because both matching and anti-matching are existential claims.

Type compatibility for functions does not attempt to match functions with patterns directly; it instead merely tests for the presence of a match constraint ($\phi \overset{+}{\sim} \pi$) or an anti-match constraint ($\phi \overset{-}{\sim} \pi$) constraint in the constraint set. These constraints are introduced during closure when we are certain of whether a function matches a given pattern (see Figure 4.6 below). Note that we do not use the dispatch relation itself; instead, Section 4.1 uses these match and anti-match constraints to define that relation.

Closure This section defines the closure relation which will infer the dispatch relation for function patterns. We begin by defining a relation $\overset{\text{N}}{\Longrightarrow}^1$ to address the closure steps which do not concern function patterns. We then define a relation $\overset{\text{F}}{\Longrightarrow}^1$ to analyze function pattern matches and introduce the match and anti-match constraints described above.

$$\textsc{Unknown Application}\quad \frac{\{\alpha_1\ \alpha_2 <: \alpha_3, \bullet\lceil \pi' <:\ \alpha_1, \tau <: \alpha_2\} \subseteq C \qquad \pi_1 \texttt{ ~> } \pi_2 = \pi' \qquad \tau \backslash C \overset{\bullet}{\sim} \pi_1}{C \overset{\text{N}}{\Longrightarrow}^1 C \cup \{\bullet\lceil \pi_2 <: \alpha_3\}}$$

Fig. 4.5. Non-Function Closure Extension

Non-Function Closure We define a relation $\overset{\text{N}}{\Longrightarrow}^1$ to be $\overset{\text{D}}{\Longrightarrow}^1$ from Definition 1 extended with the rule in Figure 4.5 to handle unknowns. This UNKNOWN APPLICATION rule is needed when function patterns themselves have function domains. The philosophy of this rule is similar how higher-order functions are addressed in [18], although that work does not consider dynamic dispatch.

We define $\overset{\text{N}}{\Longrightarrow}^*$ and $\overset{\text{N}}{\Longrightarrow}^\omega$ analogously to Definition 1.

Function Rules We now consider the core issue of this paper: how does the type system know whether or not a function matches a function pattern? The answer is to simply test a simulated function application for consistency. If applying the function to all types matching the domain of the match produces a consistent constraint set in which the output of the function always matches the codomain, the function should match the pattern.

We give the rules for a new relation $\overset{\text{F}}{\Longrightarrow}^1$ in Figure 4.6. This ruleset is implicitly parametrized over $\prec$, a decidable relation which defines dependencies, which will be further discussed in Section 4.2. We use $\prec^+$ to indicate the transitive (but not reflexive) closure of $\prec$. There are three closure rules, which need four auxiliary relations in their definition. FUNPATS is a function that produces a set of function-pattern pairs ($\langle\phi, \pi\rangle$), one for each enabled function pattern match in the constraint set. FUNMATCH is a predicate that performs an isolated closure as dictated by its parameters and determines if the resulting set is consistent. The closure rules defer to FUNMATCH to determine whether to add a dispatch constraint indicating match or anti-match. Each such constraint is *de facto* building an entry in the function pattern dispatch table. READY($\langle\phi, \pi\rangle, C$) is a predicate which holds if C contains a dispatch constraint for each dependency of $\langle\phi, \pi\rangle$ as determined by the $\prec$ relation; a pair is intuitively READY if all dependencies have been resolved. Finally, BLOCKING(S) is a predicate which determines if S, a set of function-pattern pairs, contains only pairs which could never become READY during the normal course of closure due to a cycle.

There is an apparent circularity of definition here: the definition of FUNMATCH depends on the definition of *consistency* (Definition 10), which depends on $\Longrightarrow^*$ and $\Longrightarrow^1$ (Definition 9), which depend on $\overset{\text{F}}{\Longrightarrow}^1$ (Figure 4.6), which depends on FUNMATCH. However, all of the aforementioned relations are well-founded.

The FUNCTION MATCH and FUNCTION ANTI-MATCH closure rules add dispatch constraints based on whether the function matches (or anti-matches) the pattern. The CYCLE BREAKER rule adds a constraint simultaneously to each match that occurs on a dependency cycle. Cycles will never be resolved by the other two rules, since they are never READY; all elements of a cycle are simultaneously set to $+$. We choose $+$ and not $-$ for three reasons: first, we prefer to match function patterns rather than anti-match them because a match is a stronger guarantee; second, self-reinforcing cycles are common; and third, it enables us to detect paradoxical self-reference.

Once a cycle is broken, its patterns can become READY if they have any dependencies external to the cycle; it is then possible for the ANTI-MATCH rule to apply and introduce a $\tau \overset{-}{\sim} \pi$ constraint in addition to an already present $\tau \overset{+}{\sim} \pi$ constraint, a situation we will label a type inconsistency (see Figure 4.7

$$\tau \sim \pi \in C \qquad \text{iff } \{\tau <: \alpha, \alpha <: \pi\} \subseteq C \quad \text{or} \quad \{\tau <: \alpha \in C, \alpha \sim \pi \ ?\ \phi_1 : \phi_2 <: \alpha'\} \subseteq C$$
$$\text{or } \{\bullet\lceil \pi \sim\!\!> \pi' <: \alpha_1, \tau <: \alpha_2, \alpha_1\ \alpha_2 <: \alpha'\} \subseteq C$$

$$\textsc{FunPats}(C) = \{\langle \phi, \pi\rangle \mid \pi = \pi_1 \sim\!\!> \pi_2, \phi \sim \pi \in C\}$$

$$\textsc{FunMatch}(\phi, \pi, C_0) = \begin{cases} \text{true} & \text{if } C_4 \text{ is consistent} \\ \text{false} & \text{if } C_4 \text{ is inconsistent} \end{cases}$$
$$\text{where } \phi = \alpha_1 \text{->} \alpha_2 \backslash C_1 \text{ and } \pi = \pi_1 \sim\!\!> \pi_2 \text{ and } C_0 \overset{\text{N}}{\Longrightarrow}^{\omega} C_2$$
$$\text{and } C_3 = \{\tau \not\sim \pi \mid \langle \tau, \pi\rangle \in \textsc{FunPats}(C_2)\}$$
$$\text{and } C_4 = \{\bullet\lceil \pi_1 <: \alpha_1, \alpha_2 <: \pi_2\} \cup C_1 \cup C_2 \cup C_3$$

$$\textsc{Ready}(\langle \tau, \pi\rangle, C) = \forall \langle \tau', \pi'\rangle.\ \text{if } \langle \tau', \pi'\rangle \prec \langle \tau, \pi\rangle \text{ then } \tau' \overset{+}{\not\sim} \pi' \in C$$
$$\textsc{Blocking}(S) = \text{iff } \forall \langle \tau, \pi\rangle \in S.\ \langle \tau, \pi\rangle \prec^{+} \langle \tau, \pi\rangle$$
$$\text{and } \forall \langle \tau, \pi\rangle \in S.\ \text{if } \langle \tau, \pi\rangle \prec^{+} \langle \tau', \pi'\rangle \wedge \langle \tau', \pi'\rangle \prec^{+} \langle \tau, \pi\rangle \text{ then } \langle \tau', \pi'\rangle \in S$$

$$\textsc{Function Match}\quad \frac{\langle \tau, \pi\rangle \in \textsc{FunPats}(C) \qquad \tau \not\sim \pi \notin C \qquad \textsc{Ready}(\langle \tau, \pi\rangle, C) \qquad \textsc{FunMatch}(\tau, \pi, C)}{C \overset{\text{F}}{\Longrightarrow}^{1} C \cup \{\tau \overset{+}{\not\sim} \pi\}}$$

$$\textsc{Function Anti-Match}\quad \frac{\langle \tau, \pi\rangle \in \textsc{FunPats}(C) \qquad \tau \not\sim \pi \notin C \qquad \textsc{Ready}(\langle \tau, \pi\rangle, C) \qquad \neg\textsc{FunMatch}(\tau, \pi, C)}{C \overset{\text{F}}{\Longrightarrow}^{1} C \cup \{\tau \overset{-}{\sim} \pi\}}$$

$$\textsc{Cycle Breaker}\quad \frac{\langle \tau, \pi\rangle \in \textsc{FunPats}(C) \qquad \langle \tau, \pi\rangle \in S \qquad \forall \langle \tau', \pi'\rangle \in S.\ \tau' \not\sim \pi' \notin C \qquad \textsc{Blocking}(S)}{C \overset{\text{F}}{\Longrightarrow}^{1} C \cup \{\tau \overset{+}{\not\sim} \pi \mid \langle \tau, \pi\rangle \in S\}}$$

Fig. 4.6. Function Closure Rules

below). All three closure rules avoid suppressed function pattern matches $\tau \not\sim \pi$ since they are already under scrutiny by FunMatch.

As we mentioned earlier, our approach to inferring a function pattern match involves a simulated function call; for this reason, the constraints checked by FunMatch resemble those introduced by the Application rule in Figure 3.7. In particular, FunMatch simulates an application whose argument is an unknown restricted by a pattern and whose output must match a pattern. It is also important to note that FunMatch introduces suppression constraints; at minimum, such a constraint must be introduced for the pattern match being simulated (for well-foundedness), but we introduce one for every function pattern match reachable by non-function closure; this ensures that in the subordinate closures, each pattern match is considered independently of its peers and maintains confluence.

Closure relations The global closure process may now be defined.

Definition 9. *We define* $\Longrightarrow^{1}$ *as* $C_0 \Longrightarrow^{1} C_1$ *iff* $C_0 \overset{\text{N}}{\Longrightarrow}^{1} C_1 \vee C_0 \overset{\text{F}}{\Longrightarrow}^{1} C_1$

Given $\Longrightarrow^{1}$, $\Longrightarrow^{*}$ and $\Longrightarrow^{\omega}$ are defined analogously to Definition 1.

$\textsc{Ready}(\langle \phi, \pi\rangle, C)$ and $\textsc{Blocking}(S)$ depend on the relation $\prec$. A suitable relation is one that satisfies $\textsc{Ready}(\langle \phi, \pi\rangle, C) \wedge (C \Longrightarrow^{*} C')$ implies $\textsc{FunMatch}(\phi, \pi, C) = \textsc{FunMatch}(\phi, \pi, C')$; that is, the relation

expresses a dependency between two function-pattern pairs when the manner in which the first is dispatched may affect the outcome of the second.

The Function Match rule in Figure 4.6 relies on FunMatch, which is not monotone with respect to the constraint set; as a result, the order of closure operations could matter. To see why, consider any function that relies on a non-local variable (for example, `(fun x -> fun y -> x) 0`); it is possible to add a constraint which broadens the return type of this function. For this reason, the rule fires only when all dependencies of the match are have already been decided. When restricted in this way, FunMatch *is* monotone with respect to $\Longrightarrow^*$.

Another concern here is well-foundedness. Note that we are calling the full closure (indirectly, by asking if a constraint set typechecks) as a subroutine in FunMatch, and infinite descending chains of closure are not desirable. Fortunately, we can establish the following.

Lemma 2 (Well-Foundedness). *The definition of* $\Longrightarrow^1$ *is well-founded.*

Corollary 1. *The definitions of* $\Longrightarrow^*$, $\Longrightarrow^\omega$, typechecks, *and* FunMatch *are well-founded.*

$$\frac{\{\alpha_1\ \alpha_2 <: \alpha_3, \{\ \ldots\} <: \alpha_1, \tau_2 <: \alpha_2\} \subseteq C}{C \text{ is immediately inconsistent}}\ \textsc{Application Failure}$$

$$\frac{\{\alpha_1\ \alpha_2 <: \alpha_3, \bullet\lceil \pi_1 \rightsquigarrow \pi_2 <: \alpha_1, \tau <: \alpha_2\} \subseteq C \qquad \tau \backslash C \mathrel{\mathring{\sim}} \pi_1}{C \text{ is immediately inconsistent}}\ \textsc{Unknown Application Failure}$$

$$\frac{\alpha <: \pi \in C \qquad \alpha \backslash C \mathrel{\mathring{\sim}} \pi}{C \text{ is immediately inconsistent}}\ \textsc{Upper-Bounding Pattern} \qquad \frac{\tau \mathrel{\dot{\sim}} \pi \in C \qquad \tau \mathrel{\bar{\sim}} \pi \in C}{C \text{ is immediately inconsistent}}\ \textsc{Ambiguous Dispatch}$$

Fig. 4.7. Immediate Inconsistency

Inconsistency Figure 4.7 defines immediate inconsistency. The four rules are mostly straightforward. The Application Failure rule is the same as the one described in Definition 2, catching the case where a non-function value may be called. The Unknown Application Failure rule handles the case (arising only in sub-closures initiated by function patterns) that an unknown is treated as a function in a potentially unsafe way, either because it is not restricted to be a function or because it is not guaranteed to accept the argument. The Upper-Bounding Pattern rule (again, arising only in sub-closures) ensures that all values appearing at a point match a pattern. The Ambiguous Dispatch rule indicates that the closure was unable to find a consistent dispatch; this occurs when e.g. the constraint set contains a paradoxical self-reference.

Definition 10 ((In)consistency). *Analogous to Definition 3.*

Lemma 3 (Inconsistency is monotone). *If* C *is (immediately) inconsistent then so is* $C \cup C'$.

Definition 11 (Typechecking). *Analogous to Definition 4, replacing* $\overset{\text{D}}{\Longrightarrow}^*$ *with* $\Longrightarrow^*$.

Dispatch We can finally define the dispatch relation mentioned in Section 3.2.

Definition 12. *We define a canonical dispatch relation δ_C for closed, consistent constraint set C as follows: $\tau \precsim_{\delta_C} \pi$ holds iff $\tau \overset{+}{\sim} \pi \in C$. We define a canonical dispatch relation d_e for expression e as follows: $v \precsim_{d_e} \pi$ holds for v appearing in e when $[\![e]\!] \Longrightarrow^{\omega} C$ and $[\![v]\!] \precsim_{\delta_C} \pi$ holds.*

In order for this definition to make sense, we'd like d to be unique. The most obvious way to show uniqueness would be to prove that $\Longrightarrow^{\omega}$ is a function; this is only the case when its left argument is eventually consistent, but that suffices:

Lemma 4 (Determinism). *If $C \Longrightarrow^{\omega} C'$ and C is consistent, C' is unique.*

Corollary 2. *If e typechecks then d_e is unique.*

The canonical dispatch relation represents the dispatch information learned from performing constraint closure. It has the following valuable property:

Lemma 5. *If $[\![e]\!] \Longrightarrow^{\omega} C$ then δ_C is safe with respect to d_e.*

Soundness of Closure We assert closure as presented is sound. Recall that we use $\overset{\text{D}}{\Longrightarrow}^{\omega}$ to refer to the complete closure defined in Section 3.4.

Lemma 6. *For any consistent $C \Longrightarrow^{\omega} C'$, if $C \overset{\text{D}}{\Longrightarrow}^{\omega} C''$ under the dispatch relation $\delta_{C'}$, then $C'' \subseteq C'$.*

Theorem 5 (Soundness). *For any program e, if $[\![e]\!]$ is consistent then e does not get stuck when evaluated under dispatch relation d_e, provided d_e covers e.*

The rules as presented here will produce a non-covering dispatch relation for some programs where there are functions that are matched but never called; we have an extension to address this shortcoming, but we omit it for space reasons.

4.2 Inferring Dependencies

Closure as defined in the previous section relies upon a dependency analysis of function patterns: for each function pattern match, we must know the function pattern matches that might influence its result. Here we informally define such a dependency analysis to produce the $\prec$ relation upon which our constraint closure depends. We use (another) deductive closure system for this purpose.

This closure does not need to be as precise as the one we use for typing, so we drop any attempt at flow sensitivity, and we just assume that functions both match and anti-match all patterns. This lets us build a data dependency graph whose vertices are type variables and type-pattern pairs. $\langle \tau_1, \pi_1 \rangle \prec \langle \tau_2, \pi_2 \rangle$ is defined to hold iff there is a (directed) path from $\langle \tau_1, \pi_1 \rangle$ to $\langle \tau_2, \pi_2 \rangle$ whose interior vertices are all type variables.

4.3 Putting it all Together

Recall Figure 4.1 from the beginning of this section, which illustrates the steps involved in running a program. Now that we have all the definitions in hand, we can more precisely specify this process.

Starting with program e, we compute $C_0 = [\![e]\!]$, the initial constraint set of the program. From there, we compute the dependency graph of the function

pattern matches present in e, "$\prec$". We use $\prec$ to compute C_1, where $C_0 \Longrightarrow^{\omega} C_1$. If C_1 is inconsistent (equivalently, if e does not typecheck), we stop, since we cannot extract a dispatch table from an inconsistent constraint set. On the other hand, if C_1 is consistent, we extract δ_{C_1} and use it to compute d_e. We verify that d_e covers e. Finally, we run $e \longrightarrow^* e'$ in the context of d_e.

5 Related Work

We know of no precedent for fully inferred higher-order function pattern matching. Our interest in the topic was inspired by the power of runtime contracts [9]. Function patterns are more static than traditional contracts, because higher-order function contracts only enforce the contract at runtime and only on the values they were invoked on, whereas higher-order function patterns are statically enforced at all potential runtime values. Contracts, on the other hand, have the advantage of having the full dynamic program context at their disposal, and so the two approaches are complementary.

One example of contracts work which is particularly near to Whayrf is that of Symbolic PCF with Contracts [18]. That system uses a mechanism similar to the Unknown Application given here (Figure 4.5) to handle contract verification in the presence of unknown higher-order values. The work of [18] supports broader logical expressiveness in its contracts than Whayrf does in function patterns. A major contribution of Whayrf, however, is that it permits dynamic dispatch on function patterns; [18], for example, only permits function contract assertions.

Run-time conditioning on *declared* type information is a common feature; Java/C++ dynamic typecasts are a prime example. But this is simply a runtime dispatch based on a nominal type tag, so it only allows assertions around declared program types. Typed multimethods [4,6] are another runtime dispatch mechanism based on nominal types. CDuce [5] allows functions to be matched directly by a pattern which is their annotated type (or subtype thereof), so it also does not allow matching on function behavior (as with our inference approach).

One application of function patterns is to allow the overloading of higher-order functions. Type classes [19] are capable of performing a similar task but in a subtly different way. Much like the signatures of CDuce (discussed above), type classes rely on information statically gathered at the invocation site to select a type class instance for dispatch. Using function patterns, the statically inferred behavior of the *runtime value* is used to decide dispatch. This distinction is, in a sense, similar to the difference between static and dynamic dispatch for object-oriented message dispatch.

6 Conclusion

We presented a mechanism for incorporating higher-order function pattern matching into a statically-typed language. We believe that function pattern matching has significant applicability: it supports higher-order function overloading, enables a novel form of multimethod dispatch, and enriches subtype constraint type systems with a form of interface declaration. We hope to investigate further applications of this theory in future work.

Independent of practical utility, these results are interesting in the same way coding a meta-circular interpreter is interesting: sufficiently powerful pattern matching can embed type declaration within the language, and then to dynamically control the computation in ways not possible in traditional languages.

References

1. A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *FPCA*, pages 31–41, 1993.
2. A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL 21*, pages 163–173, 1994.
3. Bard Bloom and Martin Hirzel. Robust scripting via patterns. In *DLS*, pages 29–40. ACM, 2012.
4. G. Castagna, G. Ghelli, , and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.
5. Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types. part 1: Syntax, semantics, and evaluation. In *POPL*, 2014.
6. Craig Chambers. Object-oriented multi-methods in Cecil. In *ECOOP*, 1992.
7. P. Cousot. Types as abstract interpretations, invited paper. In *POPL*, January 1997.
8. Manuel Fähndrich and John Boyland. Statically checkable pattern abstractions. In *ICFP*, pages 75–84. ACM Press, 1997.
9. Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.
10. Kathleen Fisher and John Reppy. The design of a class mechanism for Moby. In *PLDI*, New York, NY, USA, 1999.
11. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, 1993.
12. Tim Freeman and Frank Pfenning. Refinement types for ml. In *PLDI*, 1991.
13. Fritz Henglein and Jakob Rehof. Constraint automata and the complexity of recursive subtype entailment. In *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, 1998.
14. Dan Licata and Simon Peyton-Jones. View patterns: lightweight views for Haskell. Haskell Café mailing list, 2007.
15. Zachary Palmer, Pottayil Harisanker Menon, Alexander Rozenshteyn, and Scott Smith. Types for flexible objects. In *Asian Programming Languages Symposium*, 2014.
16. François Pottier. A 3-part type inference engine. In *ESOP*, pages 320–335. Springer Verlag, 2000.
17. François Pottier. A versatile constraint-based type inference system. *Nordic J. of Computing*, 7(4):312–347, 2000.
18. Sam Tobin-Hochstadt and David Van Horn. Higher-order Symbolic Execution via Contracts. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2012.
19. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1989.