Interner Bericht

Best-Fit Pattern Matching

Dipl.-Inform. Reinhard Eppler

Interner Bericht 252 / 94

Fachbereich Informatik

Universität Kaiserslautern · Postfach 3049 · D-67653 Kaiserslautern

Best-Fit Pattern Matching

Dipl.-Inform. Reinhard Eppler

Interner Bericht 252 / 94

Herausgeber: AG Programmiersprachen und Compiler, Universität Kaiserslautern
Kaiserslautern, September 1994

Contents

Abstract

This report shows that dispatching of methods in object oriented languages is in principle the same as best fit pattern matching. A general conceptual description of best fit pattern matching is presented. Many object oriented features are modelled by means of the general concept. This shows that simple methods, multi methods, overloading of functions, pattern matching, dynamic and union types, and extendable records can be combined in a single comprehensive concept.

Keywords: best-fit pattern matching, method dispatching, case analysis

1. Motivation

One major topic of object oriented programming is specializing method definitions for derived classes. Given a set of method definitions for some type hierarchy and a method application to a specific object, the method with the most specific domain which fits the type of the object is selected for execution. This is exactly the same as is done in best-fit pattern matching known from the functional language Hope [Bur80]. In both domains we find the principle of specialization for combining several function definitions. In this report we present some general aspects of best-fit pattern matching and model essential concepts of object oriented systems which fit the presented pattern matching scheme.

Denotational semantics of guarded expressions that may, e.g. appear in the form of if-clauses or overloaded functions, is modelled by 'adding' the semantic functions of the alternatives in some appropriate way. We can identify several ways to do this which we will call disjoint sum, overriding sum and specializing sum. We characterize these operations as follows:

Definition (1):

Let f and g be some functions.

1. Disjoint function sum

$$(f + g) = \begin{cases} f \cup g & ; \text{if } dom(f) \cap dom(g) = \varnothing \\ \bot & ; \text{else} \end{cases}$$

1a. Extended disjoint function sum

$$(f + g) = \begin{cases} f \cup g & ; \forall x \in dom(f) \cap dom(g): f(x) = g(x) \\ \bot & ; \text{else} \end{cases}$$

2. Overriding function sum

$$(f \circledR g)(x) = \begin{cases} g(x) & ; \text{if } x \in dom(g) \\ f(x) & ; \text{else} \end{cases}$$

3. Specializing function sum

$$(f \oplus g) = \begin{cases} g \circledR f & ; \text{if } dom(f) \subset dom(g) \\ f \circledR g & ; \text{else} \end{cases}$$

Each variant has great impact to the style of programming and is more or less suitable for programming from a software engineering point of view. The disjoint function sum is the only

operation which commutes. Hence, for clarity of programs, it would be nice only to use this kind in programming languages, i.e. to have distinct cases in guarded expressions, case expressions and in pattern matching. E.g., the factorial function could be defined as *fac(n) = if (n=0) then 1 elsif (n≠0) then ... fi* rather than using the keyword else for the second branch. Unfortunately, it is not decidable in general whether predicates are defined on disjoint domains. Restriction to predicates with decidable domains may overcome this problem, at least for most expressions with alternatives. Patterns (without guards) are an example for such predicates and constitute tests on the structure of values. With respect to extendability of programs disjoint sums require a lot programming overhead in adding new alternatives. E.g., adding a new alternative for *(n=7)* to the previously defined function *fac* requires the predicate *(n≠0)* to be replaced with *(n≠0 and n≠7)*. It is clear that the new alternative shall replace the old definition. Hence, it is superfluous to repeat this intention in the predicate. The overriding sum which is the basis for alternative expressions in nearly all programming languages (if-statements, first-fit pattern matching) gives a suitable solution for this problem, but requires the programmer to give the alternatives in the right order. E.g., in the expression *if (n≠0) then ... elsif (n=7) then ... fi* the first alternative would totally mask the second one. This problem can be avoided with the specializing sum, as is actually done in redefinitions of methods in object oriented languages and in best-fit pattern matching. As we will point out, best-fit pattern matching offers a scheme for analysing and testing the semantics of alternative expressions and is a technique that can be used in compilers or preprocessors to transform best-fit into first-fit expressions. We will investigate how to make the specializing function sum a commute operation and show how this can be used in reasoning about guarded expressions.

2. Properties of patterns

Let us first have a look at the properties we expect patterns to have. From an algebraic point of view a pattern is a term that contains free variables. Pattern matching is the process of finding a substitution for a pair *(term, pattern)* such that the substitution applied to *pattern* equals *term*. Formally:

Definition (2):

Let v be some term, p a pattern, σ a substitution. Finding the result of *match* is called pattern matching, where *match* is defined as follows:

$$\text{match}(v,p) = \begin{cases} \sigma & ; \text{if } (\exists\sigma : \sigma(p) =_m v) \\ \bot & ; \text{else} \end{cases}$$

where $=_m$ is some equivalence relation.

Since we are going to use pattern matching in a programming language which we expect to be deterministic, *match* must be decidable and its result unique. Suppose matching the pattern *i+j* with value *3*. Since there is more than one solution for *i+j=3*, matching would be nondeterministic. Hence, the equation *f(i+j) = i* would not define a unique result.

Because we want to reason about the domains of patterns, we require a set *Pat* of patterns to fulfil some properties which we will motivate in this section. All required properties are listed

in definition(7). First we define the domain of a pattern:

Definition (3):

Let p be a pattern. Then the domain of p is

$dom(p) = \{ v \mid match(v,p) \neq \perp \}$

For semantic tests variable names are irrelevant with respect to nonlinearities. For this reason we introduce the following

Definition (4):

Let p_1 and p_2 be patterns. p_1 equals p_2 with respect to renaming of variables (written $p_1 \approx p_2$), if there exists some substitution σ with $\sigma(p_1) = p_2$ and

$\forall x,y \in dom(\sigma): \sigma(x)=\sigma(y) \Rightarrow x=y$

Throughout the rest of the report we regard sets of patterns with respect to renaming of variables and use appropriate set operations. We use the term "structural equivalence" for $\approx$ as a synonym.

As we will see later, problems arise from patterns with overlapping domains. We introduce the terms overlap, overlap pattern and critical overlap. These terms are well known from term rewrite systems [KB70],[Hue80] and need no further explanation here.

Definition (5):

Let p, p_1 and p_2 be patterns. A set d is called an overlap of p_1 and p_2, iff

$d \subseteq dom(p_1) \cap dom(p_2) \neq \emptyset$

p is called an overlap pattern of p_1 and p_2, iff

$dom(p) \subseteq dom(p_1) \cap dom(p_2) \neq \emptyset$

An overlap of p_1 and p_2 is critical, iff

$\neg(dom(p_1) \subseteq dom(p_2) \vee dom(p_2) \subseteq dom(p_1))$

We write $crit(p_1, p_2, p)$, iff p is a critical overlap pattern of p_1 and p_2

We will see in the next chapter that problems arise only from patterns with overlapping domains. These problems are solved if for every critical overlap there is a pattern covering the overlap. Such sets of patterns are called to be complete:

Definition (6):

A set P of patterns is complete, iff

$(\forall p_1, p_2 \in P: p_1 \text{ and } p_2 \text{ have critical overlap } d \Rightarrow (\exists P' \subseteq P: \bigcup_{p \in P} dom(p) = d))$

Decidability of inclusion and intersection of domains of patterns is the most important criterion for patterns. Match must be uniquely decidable because nondeterminism destroys referential transparency in functional languages. Patterns with empty domain have no effect on function definitions and therefore are not allowed. Domain subsumption must be decidable such that the specializing function sum can effectively be computed, and determination of overlaps is needed to guarantee that the specializing sum is deterministic, as we will see in the next chapter. Two rather complicated criterions (4 and 5) in definition(7) address the topic of pattern set completion. Every set of patterns that is not complete should be completable by some finite set of patterns. This is necessary to allow incremental definitions of pattern sets. Otherwise there could

be some function definitions that cannot be extended.

Definition (7):

A set *Pat* is a legal set of patterns, iff all following properties hold:

1. Matching is decidable and unique:
   $\forall p \in Pat: \forall v \in U: \forall \sigma_1,\sigma_2$ substitution: $(match(v,p)=\sigma_1 \wedge match(v,p)=\sigma_2) \Rightarrow (\sigma_1 = \sigma_2)$
2. There are no patterns with empty domain:
   $\forall p \in Pat: dom(p) \neq \emptyset$ is decidable
3. Domain subsumption is decidable:
   $\forall P,P' \subseteq Pat: (\bigcup_{p\in P} dom(p)) \subseteq (\bigcup_{p'\in P'} dom(p'))$ is decidable
4. There are finitely many overlaps of two patterns:
   $\forall p,p' \in Pat: (dom(p) \cap dom(p') = d \neq \emptyset) \Rightarrow$
   $(\exists P \subseteq Pat: \bigcup_{p\in P} dom(p) = d) \wedge |P| \in \mathbb{N}$ and P is decidable
5. There are no infinite sequences of critical overlaps
   $\neg(\exists((p_0, p_0'), (p_1, p_1'), (p_2, p_2'), ...) \in (Pat \times Pat)^*:$
   $\forall i \geq 0: crit(p_i, p_i', p_{i+1}) \wedge crit(p_i, p_i', p_{i+1}')$

Example (1): The set *Term* of all well formed terms in an one-sorted algebra is a legal pattern set.

Sketch of proof: The construction of dom(t) for any term t is straight forward, matching is decidable and unique, with every ground substitution σ there is at least $\sigma(t)$ in dom(t), i.e. dom(t) is not empty for all t, domain subsumption can be determined with an appropriate extension of term subsumption (similar to the one we present in Chapter 5), and overlaps can be uniquely computed with unification. This also implies that property (5) holds, since for every pair $(p_i, p_i') \in (Pat \times Pat)^*$ that have a critical overlap there is exactly one pattern covering the overlap. Since there are at least two patterns required for building an infinite sequence of overlaps, there is no such sequence.

In the next example we give a set of patterns that allow infinite sequences.

Example (2):

Let $Pat = \{ p_i \mid i \geq 0 \} \cup \{p_i' \mid i \geq 0 \}$ where
$dom(p_i) = \{ 2i \} \cup \mathbb{N}_{2(i+1)}$
$dom(p_i') = \{ 2i + 1 \} \cup \mathbb{N}_{2(i+1)}$
$\mathbb{N}_j = \{ i \mid i \in \mathbb{N} \wedge i > j \}$
Then two elements of Pat have at most two overlapping patterns that cover the overlap and $((p_i, p_i'))_i$ is an infinite sequence of critical overlaps.

We show that properties 4 and 5 of definition(7) are sufficient to guarantee that every set of patterns can be completed with finitely many patterns. For this we introduce some more terms and give a theorem that describes the process of determining all missing patterns of a set of patterns.

Definition (8):

The subset of most general patterns of any set of patterns P is defined as

$\max(P) = \{ p \mid p \in P \wedge \neg(\exists p' \in P: \mathrm{dom}(p) \subset \mathrm{dom}(p')) \}$

The set of most general critical overlaps of any set of patterns P is defined as

$\mathrm{crit}(P) = \max\{ p \mid \exists p_1, p_2 \in P: \mathrm{crit}(p_1, p_2, p) \}$

Let P be a set of patterns and P_i

$P_0 = P$

$P_{i+1} = \mathrm{crit}(P_i)$

Then P and ΔP are defined as

$P_\infty = (\bigcup_{i \in \mathbb{N}} P_i)$

$\Delta P = P_\infty - P$

Theorem (1):

Let Pat be a set of legal patterns, P a finite subset of Pat and P_i, P_∞ and ΔP defined as in definition(8). Then the following holds:

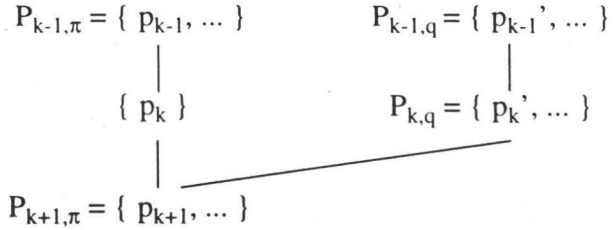
(1) $\exists i: P_i = \varnothing$

(2) P_∞ is complete

(3) $|P_\infty| \in \mathbb{N}$

Proof:

(1)Suppose $\forall i: P_i \neq \varnothing$. Let $P_{i,p} = \{p' \mid p' \in P_i \wedge \mathrm{dom}(p') \subseteq \mathrm{dom}(p)\}$ for all $p \in P$. Then there must be some $\pi \in P$ with $\forall i: P_{i,\pi} \neq \varnothing$, because all critical pairs of $P_{i,x}$ are members of $P_{i+1,x}$ and thus $(P_{i,x} = \varnothing) \Rightarrow (\forall j > i: P_j = \varnothing)$. The following holds: $\forall i > 0: |P_{i,\pi}| \geq 2$, because otherwise there would be some $k > 0$ with $P_{k,\pi} = \{p_k\}$. Since there is some $p_{k+1} \in P_{k+1,\pi}$, there must be a $p_k' \in P_{k,q}$ for some $q \neq \pi$ with $\mathrm{crit}(p_k, p_k', p_{k+1})$. This situation is illustrated as a graph in which every edge denotes domain subsumption of patterns:



With $p_{k-1} \in P_{k-1,\pi}$ and $p_{k-1}' \in P_{k-1,q}$ which both must exist since p_k and p_k' are both critical overlaps of some patterns, $\mathrm{crit}(p_{k-1}, p_{k-1}', p_k')$ holds which implies that $p_{k+1} \in P_{k,\pi}$. This contradicts $P_{k,\pi} = \{p_k\}$. Hence $\forall i > 0: |P_{i,\pi}| \geq 2$.

Now it is easy to construct an infinite sequence according to point 5 in definition(7) which contradicts to Pat being a legal pattern set. Hence, $\exists i: P_i = \varnothing$ holds.

(2)Let $p_i \in P_i$ and $p_j \in P_j$ have critical overlap p. If $i = j$ then $p \in P_{i+1}$ by definition. If $i \neq j$ then there is some $p_i' \in P_i$ with $\mathrm{dom}(p_j) \subset \mathrm{dom}(p_i')$. There must be some $p_{i+1} \in P_{i+1}$ with

$dom(p_j) \subset dom(p_{i+1})$. This patterns solves the critical overlap, hence P_∞ is complete.
(3) P_0 is finite. Since two critical overlaps are solved by finitely many patterns and finitely many patterns have finitely many overlaps, each P_i is finite. Since there is some $k \in \mathbb{N}$ with $P_k = \emptyset$ which implies that $\forall j > k$: $P_j = \emptyset$, P_∞ is a union of finitely many finite sets which implies that P_∞ is finite.

❑

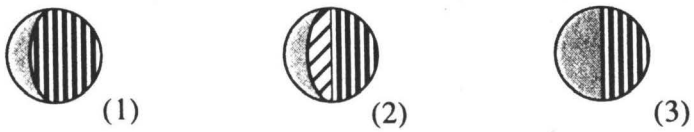
Remark:

The theorem(1) states that the completion algorithm is correct and terminates, that P_∞ is a complete, finite superset of P and hence, every finite subset of Pat is subset of a complete finite subset of Pat.

3. Function equations and overlapping domains of equations

In this section we are going to define the semantics of a set of equations that define a function. Such a set is called a function definition. An equation $f\,p = b$ consists of a function identifier f, a pattern p and a body b which constitutes the value of an appropriate function application. The domain of the equation is determined by the domain of the pattern. For this reason we will use the terms pattern and equation interchangeably. We use the specializing function sum for defining the semantics of a function definition. If we have a set e of equations, a semantic function S and a value v to which the function defined by e shall be applied, we expect $S[\![\, e \,]\!]v$ to be equal to $S[\![\, eq \,]\!]v$, iff $eq \in e$ with $\neg(\exists eq' \in e: dom(S[\![\, eq' \,]\!]) \subset dom(S[\![\, eq \,]\!]))$. Unfortunately, simple adding of functions does not always give a unique and correct result. E.g. suppose three equations A, B, and C with overlapping domains as illustrated in the following picture:

It is obvious that A and B have an overlap and C covers the overlapping domain. There are three ways to combine A, B and C with $\oplus$:



(1) is a result of $(A \oplus C) \oplus B$ or $(C \oplus A) \oplus B$, (2) is a result of $(A \oplus B) \oplus C$ or $(B \oplus A) \oplus C$, and (3) is a result of $(B \oplus C) \oplus A$ or $(C \oplus B) \oplus A$. (2) is the combination we want to have since it is the only one in which all three equations have an effect. We have to extend the specializing sum to be n-ary in a way that it combines functions with largest domains first:

Definition (9):

Let e be a set of functions and

$fmax(e) = \{ eq \mid eq \in e \wedge \neg(\exists eq' \in e: dom(eq) \subset dom(eq')) \}$

Then the specialized sum of all functions of e is defined as

$\oplus e = (\circledR_{f \in fmax(e)} f) \circledR (\oplus(e - fmax(e)))$

It should be clear that $\circledR_{f \in fmax(e)} f$ does not denote some unique value in definition(9). Since we expect function definitions in programs to be non-ambiguous, we need some criterion that guarantees the uniqueness of $\oplus$. Ambiguities in the definition of $\oplus$ are introduced by functions with partially overlapping domains. These ambiguities disappear if for every critical overlap of some set of equations *e* there is an equation in *e* covering exactly the overlapping domain, i.e. ambiguities disappear if *e* is complete. We define the best-fit semantics for complete sets of function equations and give a theorem for the soundness of the definition.

Definition (10):

Let $e = \{ f(p)=b \mid f \in F, p \in Pat(P,F,V) \}$ be a set of equations defining some function f ,

S be a semantic function and

$fs = \{ S[\![f(p)=b]\!] \mid (f(p)=b) \in e \}$.

An equation e_1 is said to be more specific than an equation e_2, iff $dom(e_1) \subseteq dom(e_2)$.

$S[\![e]\!]$ is defined as

$S[\![e]\!] = \oplus fs$; if e is complete

$= \perp$; else

Theorem (2):

Let e be a complete set of equations. Then $S[\![e]\!]$ is well defined and the following holds:

1. $dom(S[\![e]\!]) = \cup_{eq \in e} dom(S[\![eq]\!])$
2. $v \in dom(S[\![e]\!]) \Rightarrow (\exists eq \in e: v \in dom(eq) \wedge (S[\![e]\!]v = S[\![eq]\!]v) \wedge \neg(\exists eq' \in e: v \in dom(eq') \wedge dom(S[\![eq']\!]) \subset dom(S[\![eq]\!])))(*)$

Proof:

1. $dom(S[\![e]\!]) = \cup_{eq \in e} dom(S[\![eq]\!])$ by construction
2. Let $v \in dom(S[\![e]\!])$ be some value. According to (1) there exists an $eq \in e$ such that $v \in dom(eq) \wedge (S[\![e]\!]v = S[\![eq]\!]v)$ holds. We show (*) and that $S[\![e]\!]$ is well defined by induction over the recursive definition of $\oplus$.

   (a) If fmax(e) = e, then (*) holds by definition of fmax. We now show that eq is the only equation with $v \in dom(eq)$. Suppose $eq' \in e$ with $v \in dom(eq')$. Since e is complete, eq = eq', because eq and eq' having a critical overlap would cause e-fmax(e) to be non-empty, which is a contradiction to the hypothesis. Hence, $S[\![e]\!]$ is well defined.

   (b) Let the theorem hold for mf=(e - fmax(e)). This is possible since removing some maximal element from a complete set of functions does not destroy the property of completeness. The proof is left to the reader.

   If $v \in dom(\oplus mf)$, then the theorem holds since $\oplus mf$ overrides the definitions of fmax(e).

If $v \in dom(fmax(e)) - dom(\oplus mf)$, then (*) holds. Suppose $eq' \in e$ with $v \in dom(eq')$. Since e is complete, eq = eq', because eq and eq' having a critical overlap would cause v to be an element of $dom(\oplus mf)$, which is a contradiction to the hypothesis. Hence, the theorem holds.

❑

4. Hidden equations

In the last chapter we discussed problems with equations that are more specific than others. We prevented such equations from being overridden by using a semantic that takes the domains of the equations into account. A similar overriding problem exists for equations that are totally masked by some more specific equations. Formally

Definition (11):

Let $P \subseteq Pat$. $p \in P$ is called hidden in P, iff

$\exists P' \subseteq P: \forall p' \in P': dom(p') \subset dom(p) \wedge \bigcup_{p' \in P} dom(p') = dom(p)$

The set hidden(P) denotes the subset of all hidden patterns in P

Fortunately, hidden equations have no effect on the defined function and the completeness property of function definitions; they are dead code in programs and should be avoided. In theorem(4) we show that removing hidden patterns has no impact on the defined function. Hence, not considering hidden equations does not change the semantics of programs. A short example demonstrates the problem:

Example (3):

Let bool = {true, false} and Pat = {boolp, truep, falsep} with
dom(boolp) = bool, dom(truep) = {true} and dom(falsep) = {false}. Let
$(=_m)$ = { (boolp,true), (boolp,false), (truep, true), (falsep, false) }.
Then Pat is legal and the definition

{ f truep = 1, f boolp = 2, f falsep = 0 }

is complete, but the equation *f boolp = 2* is totally masked by the other ones.

Theorem (3):

Let e be a complete set of equations. Then

$S[\![e]\!] = S[\![e - hidden(e)]\!]$

Proof:

i) Let v be some value such that there is no $e_1 \in hidden(e)$ with $v \in dom(e_1)$. Then $S[\![e]\!]\ v = S[\![e\text{-}hidden(e)]\!]\ v$ holds.

ii) Let $v \in dom(e_1)$ for some $e_1 \in hidden(e)$. Then there is an $e_2 \in e$ with $v \in dom(e_2)$ and $dom(e_2) \subset dom(e_1)$. Since e is complete, there is some $e_3 \in e$ with $v \in dom(e_3)$ and $dom(e_3) \subseteq dom(e_2)$, such that there is no $e_4 \in e$ with $dom(e_4) \subset dom(e_3)$ and $v \in dom(e_4)$. Hence, $S[\![e]\!]\ v = S[\![e_3]\!]\ v$ and $e_3 \notin hidden(e)$, which implies $S[\![e]\!]\ v = S[\![e\text{-}hidden(e)]\!]\ v$.

❑

5. Patterns in a type hierarchy

In this section we discuss some syntactic and semantic features of patterns and model essential concepts of hierarchic type systems. We regard our notation to be abstract in the sense that we expect the presented concepts to be written in a particular language in a more elegant way. We address the topics of defining subtypes by extending record types [Wir88] and defining some sort of union types. We show how type compatibility by type name and type structure can be modelled and investigate typechecking patterns, finding normal forms of patterns, determining domain inclusion of patterns, and computing overlaps of patterns.

We start with algebraic structures in a sort hierarchy which allow computation of overlaps by unification [SN87] and introduce some modifications to guarantee decidability of unification, domain inclusion, and uniqueness of matching. For decidability, we will only regard regular type hierarchies, i.e. type systems in which for every term there exists exactly one most special type.

We will match patterns with ground terms that consist only of constructor functions. Hence, we will have constructor functions in our patterns. Each constructor function may have several, at least one declaration of the same arity. In this way we allow for generic constructors. For variables resp. (generic) constructors we provide type annotations in patterns to restrict their domain. If *tp* is some type, *x* a variable and *f* a constructor, then x_{tp} and $f_{tp}(...)$ are patterns that match appropriate values of type *tp*.[1] This is equivalent to introducing sort *tp* in some algebra with variable x_{tp} and function symbol f_{tp} being of sort *tp*. We permit nonlinear patterns (variables may appear more than once in a pattern) and so called as-patterns which allow aliasing of variables with patterns. If *p* is a pattern and *v* a variable then *v@p* is an as-pattern. The domains of *p* and *v@p* are the same. As-patterns are a useful tool for programming although they do not increase the expressive power of patterns.

We define the set of all patterns as follows:

Definition (12):

Let P be a set of type predicates, F be a set of constructor functions and V be a set of variables.

Let $v \in V$, $p \in P$, $f \in F$. Then Pat(P,F,V) is defined as

1. Variable pattern

   $v_p \in Pat(P,F,V)$

1. In a very general setting every monadic predicate can be seen as a type annotation. However, we have to distinguish the terms predicate and type, since types resp. sorts are introduced in the algebra which defines the value space of our programs, whereas the mentioned predicates are not. They are even not part of ground terms defined by the algebra. We can additionally allow for type predicates that constitute union types. Actually, we do some research in allowing predicates in general that fit definition(7).

2. Structured pattern

   $f_p(p_1, ..., p_n) \in Pat(P,F,V)$; if $p_i \in Pat(P,F,V) \wedge f$ is n-ary

3. As-pattern

   $v_p@f_p(p_1, ..., p_n) \in Pat(P,F,V)$; if $f_p(p_1, ..., p_n) \in Pat(P,F,V)$

Remark:

We introduce some shorthand notations for patterns. We define the predicate $\tau(x)$ to be a tautology. For a variable v and some constructor f we simply write v resp. f to denote v_τ and f_τ. With $p_i \in Pat$, $p \in P$ and tuple constructor functions *tup2*, *tup3* etc. we write $p(p_1,p_2)$ *resp.* $p(p_1,p_2,p_3)$ to denote tuple patterns $tup2_p(p_1,p_2)$ and $tup3_p(p_1,p_2,p_3)$ and so on. Again, we omit p if $p = \tau$. We use standard list constructors *cons* and *nil*, and use the infix operator :: for *cons* and brackets *[* and *]* to build lists.

Now we will take a closer look at finding the matching substitution. Matching a variable with some value yields a substitution that maps the value to the variable. Matching a constructor pattern with a value of appropriate type is continued in matching the values combined by the constructor to the corresponding subpattern. I.e., the inverse function of the constructor is applied to the value, which is matched with the pattern. A matching substitution exists only if the resulting substitutions for each constructor component can be combined to a single substitution with extended disjoint function sum. In this way nonlinear patterns can be allowed.

Definition (13):

Let $proj_i(v_1, ..., v_n) = v_i$ for $1 \le i \le n$ and $proj_{i,f} = proj_i \circ f^{-1}$.

Let $id \in V$, $p \in P$ and $f \in F$.

$match(v, id_p) = \{ (id \leftarrow v) \}$; if $p(v)$

$match(v, f_p(p_1, ..., p_n)) = +_{i=1..n}\, match(proj_{i,f}(v), p_i)$; if $v \in dom(f) \wedge p(v)$

$\perp$; else

$match(v, id_p@f_p(p_1, ..., p_n)) = +_{i=1..n}\, match(proj_{i,f}(v), p_i) \cup \{(id \leftarrow v)\}$; if $v \in dom(f) \wedge p(v)$

$\perp$; else

As motivated and proved in detail in [Hue80], critical overlaps can be determined by unification. Since we have many sorted patterns we will use order sorted unification [SN87]. In contrast to the systems mentioned above we do not test critical pairs for confluence. Instead, we give an equation that specializes the critical overlap. Both, unification and domain inclusion will be defined over the structure of patterns. Hence, unifiable patterns and overlapping patterns must have a similar structure. Unfortunately, this is not necessarily the case for as-patterns as demonstrated in the following example:

Example (4):

Let x and y be variables. It is obvious that the following equality holds:

$dom(\, y@(0::x) :: y\,) = dom(\, (0::x) :: 0 :: x\,)$.

To overcome this problem we introduce a normal form for patterns. Unifying a pattern $pat \in P(\varnothing,F,V)$ with a variable x that does not occur in *pat* gives a solution for the aliased patterns in *pat* and thus defines a normal form. Hence, we can remove as-patterns which shows

that as-patterns are a syntactic feature. Failure of unification indicates that *dom(pat)* = ∅, i.e. *pat* is not a legal pattern. Therefore, unification can be used to determine consistency of patterns. In the following discussion we will only regard type consistent patterns that do not contain aliasing.

The introduced notation for patterns is sufficient to model structural and name equivalence of types, union types for which the type dynamic that comprises all values of all types is a prominent example, and extendable record types in an implicit and explicit type hierarchy. For name equivalence we just have to introduce some type predicate that does not overlap with other predicates. Structural equivalence holds for all patterns that are not constrained by such a predicate. A union type is introduced by some predicate that is composed of some other type predicates. We will use union types to model explicit record hierarchies. For doing this we first model extendable records in an implicit type hierarchy. We regard the presented solution to be some practical subset of the record theory introduced by Cardelli [CM91]. In particular we remark that our concept gives a solution for the method specializing concept that Cardelli mentioned to be still missing in his theory. For all records we introduce one constructor function *rec* who's arity equals the number of different record fields used in the whole program, each record field corresponding to some fixed position of the constructor. We represent an absent field by a fresh variable, a present field by its value applied to a special unary constructor function *fld*. We regard this representation to be a theoretical model of some more efficient implementation.

Example (5):

Let a,b and c be all record fields used in some program. Then the two record patterns

{ a=3, b=4, ..} and { b=x, c=x, ..}

written in ML style [Mil85] correspond to the patterns

rec(fld(3), fld(4), y) resp. rec(z, fld(x), fld(x)).

It is obvious that unifying the two patterns yields record { a=3, b=4, c=4 } which is the only record that matches both patterns.

For modelling named record types in an explicit type hierarchy there need only some predicates additionally to be defined for the types. The predicates of supertypes comprise all subtype predicates and thus form some sort of union type. To make the predicates decidable, each instance of a named record type (don't mess this up with record patterns) is tagged by a bit vector which encodes record type membership of the record. For each record type there exists an entry in that vector indicating whether the record belongs to that type or not.

Example (6):

Let types R and T be the only record types of some program and defined as follows (using Oberon2 notation):

TYPE R = RECORD a:INT END;

TYPE T = RECORD (R) b:INT END;

Then R and T are predicates that we can use in patterns. Type consistent patterns are then

rec_R(fld(3), x) and rec_T(fld(y), fld(4)).

A 2-bit vector is needed to encode record type membership. In this example we choose the first position of that vector to correspond to type R. Instances of R and T might then be for example:

rec((1,0), 3, *) (in ML style: R{a=3})

rec((1,1), 5, 4) (in ML style: T{a=5, b=4})

As we have seen, many features of object oriented languages can be modelled with terms in a many sorted ordered algebra. We can identify overlaps of two patterns with order-sorted unification if the algebra which defines our patterns is regular, i.e. there exists exactly one most special type for each pattern and the type hierarchy is a partial order. The first property holds by definition of Pat. The properties listed in definition(7) have also to be fulfilled. Uniqueness of matching is guaranteed if we provide only constructor functions. The set of most general unifiers in order sorted logic is finite [SN87] which fits criterion 4. If we consider only finitely many types there are no infinite sequences of overlaps, hence criterion 5 holds. Computation of domain inclusion can be defined on the term structure of patterns. Problems arise from variables that appear more then once in a pattern. E.g., consider the patterns tup(x,y) and tup(x,x) which are structural equivalent, but have different domains, because the nonlinear variable x restricts the domain of the second pattern. This motivates that nonlinearities must separately be considered. We will do this by comparing sets of pattern positions, that are syntactically equivalent. Formally:

Definition (14):

Let $p \in Pat(P,F,V)$ without aliasing. Then the set O(p) is the set of all positions of p:

1. $O(v) = \{0\}$ where $r/0 = v$, if $v \in V$
2. $O(f_q(p_1, \dots, p_n)) = \{ i.x \mid 1 \le i \le n \wedge x \in O(p_i) \} \cup \{0, \dots, n\}$

where $f_q(p_1, \dots, p_n)/0 = f_q$ and $f_q(p_1, \dots, p_n)/i \equiv p_i$ for $1 \le i \le n$

if $f \in F$ and $q \in P$ and $p_i \in Pat(P,F,V)$

Definition (15):

Let $p \in Pat(P,F,V)$ without aliasing. The set same(p) of positions of p that are syntactically equivalent is defined as follows:

$same(p) = \{ (i,j) \mid \forall i,j \in O(p): p/i = p/j \wedge i \neq j \}$

We can now define domain inclusion recursively on the structure of a pattern.

Definition (16):

Let $v, v' \in V$, $f \in F$ and $p, p' \in P$. Structural inclusion $\le_s$ for two patterns without aliasing holds if one of the following cases holds:

1. $v_p \le_s v'_{p'}$ iff $dom(p) \subseteq dom(p')$
2. $f_p(p_1, \dots, p_n) \le_s v'_{p'}$ iff $dom(p) \subseteq dom(p')$
3. $f_p(p_1, \dots, p_n) \le_s f_{p'}(p'_1, \dots, p'_n)$ iff $(\forall i: 1 \le i \le n: p_i \le_s p'_i) \wedge dom(p) \subseteq dom(p')$

Theorem (4):

Let p1, p2 ∈ Pat(P,F,V) without aliasing. Then the following holds:
dom(p1) ⊆ dom(p2) iff p1 ≤ p2 ∧ same(p2) ⊆ same(p1)

Proof: (sketch)

Since variable-patterns v_p match all values val with val ∈ dom(p), the theorem holds for all cases according to points 1 and 2 in definition(16).

Let $p1 = f_p(p_1, \ldots, p_n)$ and $p2 = f_{p'}(p'_1, \ldots, p'_n)$. If dom(p1) ⊆ dom(p2) then it is easy to see that p1 ≤ p2 holds. In addition, there must be at least as much equal subpatterns in p1 than in p2 because otherwise there would be some value matching p2 and not p1. On the other hand, if p1 ≤ p2 and same(p2) ⊆ same(p1) then dom(p1) ⊆ dom(p2) holds.

❑

6. Related and Future Work

Best-fit pattern matching was introduced by Field [Bur80], [FHW88]. He provides very few and simple concepts in patterns. Multi methods are described in [Bob86]. There is a lot of work addressing method dispatching for simple and multi methods. The principle of dispatching simple methods is described by Wirth in [Wir88]. This scheme become more complicated if multiple inheritance is provided. This topic is addressed by Stroustrup in [Str87]. Dispatching of multi methods is even worse and described for example in [Ing86]. Ingalls provides a scheme to model multi methods with simple methods. His idea is the basis for the implementation of the concepts presented in Chapter 5 which is described by Hofmann in [Hof94]. A lambda calculus for overloaded functions is presented in [Tsu94]. Implementing classes with (extendable) records is described by Wirth [Wir88]. A theoretical concept for records and operations on them is presented by Cardelli [CM91]. Records with subtyping can also be found in [JM88].

The concepts presented in Chapter 5 are integrated in the functional language Refus [Epp92]. This language provides type annotations only in a very traditional way. We suggest to introduce type predicates in a more general way such that e.g. predicates *odd* and *even* can be used in patterns. Another extension would be to allow users to define predicates which they can use in patterns. The user would be required to give axioms that state that the predicates are suitable for patterns. A new unification algorithm and normalization algorithm will have to be developed, if predicates depend on relationships between parts of structured values. A simple example of such a relationship are nonlinear patterns which made normalization of patterns necessary in the case of aliased subpatterns. Nonlinearities had also effect on determining domain inclusion of patterns. Furthermore, defined functions could be provided in patterns which would make knowledge of the corresponding inverse function necessary. An example are '*n+k*' patterns in Haskell [HJW92] which match on all numbers *v* greater than *k* binding variable *n* to the value *v-k*.

7. Conclusion

We presented a concept for best-fit pattern matching that subsumes the concepts for dispatching

multi methods, specializing methods and overloading functions in general. We showed that essential object oriented concepts fit the presented scheme. We gave a concise denotational semantics for specialized and overloaded functions which suffices to denotationally describe the semantics of nearly all object oriented languages. The implementation of the presented concepts in context of the functional language Refus [Sey94] shows that best-fit pattern matching is practical. However, there is some compile time overhead emerging from the required analysis of overlaps. We suggest to provide this analysis optionally at least for case expressions to transform such expressions given in best-fit style to first-fit style on programmers demand. In such a system the analysis method would serve as a tool to discover not considered or hidden cases. We point up that object oriented method overloading and dispatching can only given in best-fit style.

8. References

[Bob86] **Bobrow D G, et al**; *CommonLoops: Merging Lisp and Object-Oriented Programming*. In OOPSLA'86 Conference Proceedings, SIGPLAN Notices 21(11), 1986.

[Bur80] **Burstall R M, et al**; *Hope: An Experimental Applicative Language*, CSR-62-80, Department of Computer Science, University of Edinburgh, 1980.

[Epp92] **Eppler R**; *Refus Sprachreport*, Universität Kaiserslautern, Arbeitspapier, 1992.

[FHW88] **Field A, Hunt S, While L**; *The Semantics and Implementation of Various "Best Fit" Pattern Matching Schemes for Functional Languages*, Report Imperial College of Science and Technology, London, 1988.

[CM91] **Cardelli L, Mitchell C**; *Operations on Records*, Math Structure in Comp Science, 1991.

[Gol83] **Goldberg A, Robson D**; *Smalltalk-80, The Language and its Implementation*, Addison-Wesley, 1983.

[Hof94] **Hofmann B**; *Effiziente Mustervergleiche in einer funktionalen Sprache mit objektorientierten Konzepten (Efficient Pattern Matching in a Functional Language with Object Oriented Concepts)*, Universität Kaiserslautern, Diplomarbeit, 1994.

[HJW92] **Hudak P, Jones S P, Wadler P**; *Report on the Programming Language Haskell*, SIGLPLAN Notices 27(5), 1992.

[Hue80] **Huet G**; *Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems*, J. ACM 27, 1980.

[Ing86] **Ingalls D H H**; *A Simple Technique for Handling Multiple Polymorphism*. In OOPSLA'86 Conference Proceedings, SIGPLAN Notices 21(11), 1986.

[JM88] **Jategaonkar L A, Mitchell J C**; *ML with Extended Pattern Matching and Subtypes*, Proceedings ACM Conference on Lisp and Functional Programming, 1988.

[KB70] **Knuth D E, Bendix P B**; *Simple Word Problems in Universal Algebras*, Computational Problems in Abstract Algebra, ed.: J Leech, Pergamon Press, 1970.

[Mil85] **Milner R**; *The Standard ML Core Language*, Polymorphism, 2(2), 1985.

[MW91] **Mössenböck H, Wirth N**; *The Programming Language Oberon-2*, Institut für Computersysteme, ETH Zürich, 1991.

[Sey94] **Seyfried M**; *Entwurf und Implementierung eines Übersetzers für Refus (A Compiler for Refus: Design and Implementation)*, Universität Kaiserslautern, Diplomarbeit, 1994.

[SN87] **Smolka G, Nutt W, Goguen J A, Meseguer J**; *Order-Sorted Equational Computation*, in H Aït-Kaci, M Nivat: Resolution of Equations in Algebraic Structures, Academic Press, 1987.

[Str87] **Stroustrup B**; *Multiple Inheritance for C++*, Proceedings of the Spring'87 EUUG Conference. Helsinki, May 1987.

[Tsu94] **Tsuiki H**, *A Normalizing Calculus with Overloading and Subtyping*, In: Theoretical Aspects of Computer Software, Proceedings, 1994.

[Wir88] **Wirth N**; *Type extensions*, ACM Transactions on Programming Languages and Systems, Vol. 10, Nr. 2, 1988.