ELSEVIER

Theoretical Computer Science 269 (2001) 135–162

Theoretical Computer Science

www.elsevier.com/locate/tcs

Higher-order matching for program transformation

Oege de Moor*, Ganesh Sittampalam

Oxford University, Computing Laboratory, Programming Research Group, Wolfson Bldg., Parks Road, OX1 3QD Oxford, UK

Received July 1999; revised September 2000; accepted September 2000
Communicated by M. Sintzoff

Abstract

We present a simple, practical algorithm for higher-order matching in the context of automatic program transformation. Our algorithm finds more matches than the standard *second order matching* algorithm of Huet and Lang, but it has an equally simple specification, and it is better suited to the transformation of programs in modern programming languages such as Haskell or ML. The algorithm has been implemented as part of the MAG system for transforming functional programs. © 2001 Elsevier Science B.V. All rights reserved.

1. Background and motivation

1.1. Program transformation

Many program transformations are conveniently expressed as higher order rewrite rules. For example, consider the well-known transformation that turns a tail recursive function into an imperative loop. The pattern

```
f x = if  p x
      then g x
      else f (h x)
```

is rewritten to the term

```
f x = |[ var r;
         r := x;
         while ¬(p r) do
          r := h r;
         r := g r;
         return r
      ]|
```

* Corresponding author.
E-mail address: oege@comlab.ox.ac.uk (O. de Moor).

Carefully consider the pattern in this rule: it involves two bound variables, namely f and x, and three free variables, namely p, g and h. When we match the pattern against a concrete program, we will have to find instantiations for these three free variables. Finding such instantiations involves the 'invention' of new function definitions. For example, here is the function that sums the digits of a number, in tail recursive form:

$$\begin{aligned} \mathit{sumdigs}\ (x,s) = {} & \text{if } x < 10 \\ & \text{then } s + x \\ & \text{else } \mathit{sumdigs}\ (x \operatorname{div} 10,\ s + x \bmod 10). \end{aligned}$$

Matching this recursive definition against the above pattern should result in the substitution:

$$\begin{aligned} & p\,(x,s) = x < 10, \\ & g\,(x,s) = s + x, \\ & h\,(x,s) = (x \operatorname{div} 10,\ s + x \bmod 10). \end{aligned}$$

This paper is concerned with an algorithm for finding such substitutions. Because the construction of these substitutions involves the synthesis of new functions, it is sometimes called *higher-order matching*. This contrasts with ordinary *first-order matching*, where we only solve for variables of base types such as *Int* or *Bool*.

1.2. Higher-order matching

Abstracting from the particular programming language in hand, we are led to consider the following problem. Given λ-expressions P (the pattern) and T (the term), find a substitution ϕ such that

$$\phi P = T.$$

Here equality is taken modulo renaming (α-conversion), elimination of redundant abstractions (η-conversion), and substitution of arguments for parameters (β-conversion). A substitution ϕ that satisfies the above equation is said to be a *match*. Later on, we shall refine the notion of a match.

Unlike ordinary first-order matching, there is no canonical choice for ϕ. For example, let

$$P = f\ x \quad \text{and} \quad T = 0.$$

Possible choices for ϕ include:

$$\begin{aligned} f &:= (\lambda a\,.\,a) \text{ and } x := 0, \\ f &:= (\lambda a\,.\,0), \\ f &:= (\lambda g\,.\,g\,0) \text{ and } x := (\lambda a\,.\,a), \\ f &:= (\lambda g\,.\,g\ (g\,0)) \text{ and } x := (\lambda a\,.\,a), \\ & \ldots \end{aligned}$$

All these matches are *incomparable* in the sense that they are not substitution instances of each other.

1.3. Second-order matching

Clearly a potentially infinite set of matches is undesirable in the context of automatic program transformation. In a trail-blazing paper [14], Huet and Lang suggested restricting attention to matching of *second-order* terms.

This is a condition on types: a base type (for example Int) is *first* order. The order of a derived type is calculated by adding one to the order of the argument type and taking the maximum of this value and the order of the result type. So for example $Int \rightarrow Bool$ is second order. The order of a term is simply the order of its type.

This simple restriction guarantees that there are only a finite number of incomparable matches. Huet and Lang's algorithm is the de facto standard for higher order matching in program transformation. In the example shown above, we have to give simple types to our variables to apply Huet and Lang's algorithm, for example

$$f :: Int \rightarrow Int \quad \text{and} \quad x :: Int.$$

Now the only matches found are

$$f := (\lambda a\,.\,a) \quad \text{and} \quad x := 0,$$
$$f := (\lambda a\,.\,0).$$

Note that we do not apply evaluation rules for constants; so for example

$$f := (\lambda a\,.\,a \times a) \quad \text{and} \quad x := 0$$

is not a match. Of course there are other second-order matches, such as

$$f := (\lambda a\,.\,0) \quad \text{and} \quad x := 1,$$

but all of these are specialisations (substitution instances) of the matches returned by Huet and Lang's algorithm. Note that none of the other matches we quoted before qualifies as second order, because there the variable f has type $(Int \rightarrow Int) \rightarrow Int$.

Despite its success, Huet and Lang's algorithm suffers from a number of disadvantages:

- The restriction to second-order terms is not reasonable in modern programming languages that feature functions as first-class values. For example, the *fusion* transformation [3] is routinely applied to higher order arguments: implementing that rule via Huet and Lang's algorithm would severely limit its use (see Section 5 for an example).
- Huet and Lang's algorithm only applies to simply typed terms: it needs to be modified for polymorphically typed terms. For example, if we allowed type variables, it would be natural to assign the following types in the example above:

$$f :: \alpha \rightarrow Int \quad \text{and} \quad x :: \alpha.$$

We now have to complicate the definition of allowable matches to prevent α being instantiated to function types such as $Int \rightarrow Int$. (It is not impossible however: in [17], it is shown how Huet's higher-order unification algorithm may be adapted to polymorphic typing. The same techniques apply to matching.)

- The Huet and Lang algorithm requires all terms to be in η-expanded, uncurried form, which means we are forced to work with typed terms.

The purpose of this paper is to present a new matching algorithm that does not suffer these drawbacks. In particular, our algorithm shares the property that it returns a well-defined, finite set of incomparable matches. It furthermore is guaranteed to give at least the second-order matches, but possibly more. Finally, its implementation is simple and efficient.

2. Preliminaries and specification

We start by introducing some notation, and then pin down the matching problem that we intend to solve. Users of our algorithm (for instance those who wish to understand the operation of the MAG system [10]) need to know only about this section of the paper.

2.1. Expressions

An *expression* is a constant, a variable, a λ-abstraction or an application. There are two types of variables: bound ("local") variables and free ("pattern") variables. We shall write a, b, c for constants, x, y, z for local variables, p, q, r for pattern variables, and use capital identifiers for expressions. Furthermore, function applications are written $F\,E$, and lambda abstractions are written $\lambda x\,.\,E$. As usual, application associates to the left, so that $E_1\,E_2\,E_3 = (E_1\,E_2)\,E_3$.

It is admittedly unattractive to make a notational distinction between local and pattern variables, but the alternatives (De Bruijn numbering or explicit environments) would unduly clutter the presentation. In the same vein, we shall ignore all problems involving renaming and variable capture, implicitly assuming that identifiers are chosen to be fresh, or that they are renamed as needed. Equality is modulo renaming of bound variables. For example, we have the identity

$$(\lambda x\,.\,\lambda y\,.\,a\,x\,(b\;x\;y)) = (\lambda y\,.\,\lambda z\,.\,a\;y\,(b\;y\;z)).$$

Besides renaming, we also consider equality modulo the elimination of superfluous arguments. The *η-conversion* rule states that $(\lambda x\,.\,E\,x)$ can be written as E, provided x is not free in E. An expression of this form is known as an *η-redex*. We shall write

$$E_1 \simeq E_2,$$

to indicate that E_1 and E_2 can be converted into each other by repeated application of η-conversion and renaming. For example,

$$(\lambda x\,.\,\lambda y\,.\,a\;x\;y) \simeq a,$$

but it is *not* the case that

$$(\lambda x\,.\,\lambda y\,.\,a\ \ y\ x) \simeq a.$$

Since reduction with the η-conversion rule is guaranteed to terminate (the argument becomes smaller at each step), we have a total function *etaNormalise* which removes all η-redexes from its argument. It follows that

$$E_1 \simeq E_2 \equiv \mathit{etaNormalise}\ E_1 = \mathit{etaNormalise}\ E_2.$$

The *β-conversion* rule states how arguments are substituted for parameters: $(\lambda x\,.\,E_1)\,E_2$ is converted to $(x := E_2)E_1$. A subexpression of this form is known as a *β-redex*. The application of this rule in a left-to-right direction is known as *β-reduction*. Unlike η-reduction, repeated application of β-reduction is not guaranteed to terminate.

An expression is said to be *normal* if it does not contain any η-redex or β-redex as a subexpression. An expression is *closed* if all the variables it contains are bound by an enclosing λ-abstraction.

Some readers may find it surprising that we have chosen to work with untyped λ-expressions, instead of committing ourselves to a particular type system. Our response is that types could be represented explicitly in expressions (as in Girard's second order λ-calculus, which forms the core language of the Haskell compiler *ghc* [20]). Our algorithm can be adapted accordingly to expressions in which types are explicit in the syntax. However, as with the unification algorithm presented in [16], it does not depend on a particular typing discipline for its correctness.

2.2. Parallel β-reduction

We now introduce the operation that is the key both to the specification and implementation of our matching algorithm.

The function *step* performs a bottom-up sweep of an expression, applying β-reduction wherever possible. Intuitively, we can think of *step* as applying one *parallel* reduction step to its argument. Formally, *step* is defined by

$$\begin{aligned}
\mathit{step}\ c &= c,\\
\mathit{step}\ x &= x,\\
\mathit{step}\ p &= p,\\
\mathit{step}\,(\lambda x\,.\,E) &= \lambda x\,.\,(\mathit{step}\ E),\\
\mathit{step}\,(E_1\,E_2) &= \text{case } E_1' \text{ of}\\
&\qquad (\lambda x\,.\,B) \rightarrow (x := E_2')B\\
&\qquad - \qquad\quad\ \ \rightarrow (E_1'\ E_2'),
\end{aligned}$$

where

$$E_1' = step\, E_1,$$
$$E_2' = step\, E_2.$$

Clearly *step* always terminates, as it proceeds by recursion on the structure of terms. It is not quite the same, therefore, as the operation that applies β-reduction exhaustively, until no more redexes remain. To appreciate the difference, consider

$$step\ ((\lambda x\,.\,\lambda y\,.\,a(x\ b)y)\ (\lambda z\,.\,c\ z\ z)\ d) = a\ ((\lambda z\,.\,c\ z\ z),\, b)\ d.$$

It is worthwhile to note that our definition of *step* does not coincide with similar notions in the literature. A more common approach is to define a parallel reduction step by underlining all β-redexes in the original term, and reducing all underlined β-redexes. According to that definition, we have for example

$$\begin{aligned}&((\lambda x\,.\,x)(\lambda x\,.\,x))\ ((\lambda x\,.\,x)(\lambda x\,.\,x))\\ \to\ &(\lambda x\,.\,x)(\lambda x\,.\,x)\\ \to\ &\lambda x\,.\,x.\end{aligned}$$

in two parallel steps. By contrast, with our definition of *step*, we have

$$step\ [((\lambda x\,.\,x)(\lambda x\,.\,x))\ ((\lambda x\,.\,x)(\lambda x\,.\,x))] = \lambda x\,.\,x,$$

in one step. We are grateful to Mike Spivey and Zena Ariola, who independently pointed out this subtlety.

The operation of *step* can be a little difficult to understand. In a certain sense, it represents an approximation of *betanormalise*, the function that exhaustively applies β-reduction: if *betanormalise* E exists then there exists some positive integer n such that $step^n\, E = betanormalise\, E$. However it is not always the case that *betanormalise* E exists, since in the untyped lambda-calculus, exhaustive β-reduction is not guaranteed to terminate.

If E does not contain a λ-abstraction applied to a term containing another λ-abstraction, then $step\, E = betanormalise\, E$. In particular, this condition will be satisfied in a typed setting by all terms which only contain subterms of second-order or below. This claim will be further articulated in Section 5, where it is shown that our matching algorithm returns all second-order matches.

2.3. Substitutions

A *substitution* is a total function mapping pattern variables to expressions. Substitutions are denoted by Greek identifiers. We shall sometimes specify a substitution by listing those assignments to variables that are not the identity. For instance,

$$\phi = \{p := (a\,r),\ q := (\lambda y\,.\,b\,y\,x)\}$$

makes the indicated assignments to p and q, but leaves all other variables unchanged.

Substitutions are applied to expressions in the obvious manner. Composition of substitutions ϕ and ψ is defined by first applying ψ and then ϕ:

$$(\phi \circ \psi)E = \phi(\psi E).$$

We say that one substitution ϕ is *more general* than another substitution ψ if there exists a third substitution δ such that

$$\psi = \delta \circ \phi.$$

When ϕ is more general than ψ, we write $\phi \leqslant \psi$. Intuitively, when $\phi \leqslant \psi$, the larger substitution ψ substitutes for variables that ϕ leaves alone, or it makes more specific substitutions for the same variables. For example, with ϕ as above and ψ specified by

$$\psi = \{p := (a\,(b\,c)),\ q := (\lambda y\,.\,b\,y\,x),\ r := (b\,c)\},$$

we have $\phi \leqslant \psi$ because $\psi = \delta \circ \phi$ where

$$\delta = \{r := (b\,c)\}.$$

If two substitutions ϕ and ψ are equally general, they only differ by renaming: that is, we can find a substitution δ that only renames variables so that $\phi = \delta \circ \psi$.

A substitution is said to be *normal* if all expressions in its range are normal, and *closed* if any variables that it changes are mapped to closed expressions.

2.4. Rules

A *rule* is a pair of expressions, written $(P \rightarrow T)$, where P does not contain any η-redexes, and T is normal, with all variables in T being local variables, *i.e.* they occur under an enclosing λ-abstraction. The matching process starts off with T closed, but because it proceeds by structural recursion it can generate new rules which do not have T closed. In such a rule, a variable is still regarded as being local if it occurred under an enclosing λ-abstraction in the original rule. We call P the *pattern* and T the *term* of the rule. Rules are denoted by variables X, Y and Z. Sets of rules are denoted by Xs, Ys and Zs.

The *measure* of a rule is a pair of numbers: the first component is the number of pattern variables in the pattern, and the second component is the total number of symbols in the pattern (where the space representing function application is taken to be a symbol). The *measure* of a set of rules is defined by pairwise summing of the measures of its elements. When Xs and Ys are sets of rules, we shall write $Xs \ll Ys$ to indicate that in the lexicographic comparison of pairs, the measure of Xs is strictly less than the measure of Ys. Note that $\ll$ is a well-founded transitive relation. We shall use this fact to prove termination of our matching algorithm, and also in an inductive proof about its result.

A substitution ϕ is said to be *pertinent* to a rule $(P \rightarrow T)$ if all variables it changes are contained in P. Similarly, a substitution is pertinent to a set of rules if all variables it changes are contained in the pattern of one of the rules.

A rule $(P \rightarrow T)$ is *satisfied* by a normal substitution ϕ if

$$\mathit{step}\,(\phi P) \simeq T.$$

The substitution ϕ is then said to be a *one-step match*. Note that we take equality not only modulo renaming, but also modulo η-conversion. A normal substitution *satisfies* a set of rules if it satisfies all elements of that set. We write $\phi \vdash X$ to indicate that ϕ satisfies a rule X, and also $\phi \vdash Xs$ to indicate that ϕ satisfies a set of rules Xs.

The notion of a one-step match contrasts with that of a *general* match in that it restricts the notion of equality somewhat; a normal substitution ϕ is said to be a general match if $\mathit{betanormalise}(\phi P) \simeq T$. For convenience we shall refer to a one-step match simply as a *match*.

It is worth noting that if $\phi \vdash Xs$ and $\phi \leqslant \psi$, then $\psi \vdash Xs$. To see this, suppose $\phi \vdash P \rightarrow T$ and $\phi \leqslant \psi$, so that there exists δ such that $\psi = \delta \circ \phi$. Then $\mathit{step}\,(\phi P) \simeq T$, so $\mathit{step}\,(\phi P)$ does not contain any pattern variables. Therefore, any pattern variables in ϕP are removed when *step* is applied. If they are first substituted, this will still be the case, and so $\mathit{step}\,(\delta(\phi P)) = \mathit{step}\,(\phi P) \simeq T$. Therefore $\psi \vdash P \rightarrow T$. Clearly this result can be easily extended to a set of rules.

The application of a substitution to a rule is defined by $\sigma(P \rightarrow T) = \sigma P \rightarrow T$ (since T is closed there is no point in applying a substitution to it). The obvious extension of this definition to a set of rules applies.

2.5. Match sets

Let Xs be a set of rules. A *match set* of Xs is a set $\mathscr{M}$ of normal substitutions such that:

- For all normal ϕ: $\phi \vdash Xs$ if and only if there exists $\psi \in \mathscr{M}$ such that $\psi \leqslant \phi$.
- For all $\phi_1, \phi_2 \in \mathscr{M}$: if $\phi_1 \leqslant \phi_2$, then $\phi_1 = \phi_2$.

The first condition is a soundness and completeness property. The backwards direction is soundness; it says that all substitutions in a match set satisfy the rules. The forwards implication is completeness; it says that every match is represented. The second condition states that there are no redundant elements in a match set.

For example, if $Xs = \{p\ q \rightarrow a\}$, then

$$\begin{aligned} \{\ &\{p := (\lambda x\,.\,a)\},\\ &\{p := (\lambda x\,.\,x),\quad q := a\}\ \} \end{aligned}$$

is a match set.

Note that since

$$\mathit{betanormalise}\,((\lambda f\,.\,f\ a)\ (\lambda x\,.\,x)) = a$$

we have that

$$\{p := (\lambda f\,.\,f\;a),\;\; q := (\lambda x\,.\,x)\}$$

is a general match. However since

$$step\,((\lambda f\,.\,f\;a)\;(\lambda x\,.\,x)) \neq a,$$

it is not a member of the match set.

In general, match sets are unique up to pattern variable renaming, and consequently we shall speak of *the* match set of a set of rules.

In the remainder of this paper, we present an algorithm that computes match sets. Although it is not part of the definition, the matches returned by our algorithm are in fact pertinent to the relevant set of rules. Furthermore, in Section 5 it is shown that match sets include all second-order matches.

3. Outline of an algorithm

Our matching algorithm operates by progressively breaking down a set of rules until there are none left to solve. This section does not spell out the algorithm in full detail. Instead, we outline its structure, and give a specification for the function *resolve* that provides the means of breaking down an individual rule. We show that if that specification is met, the algorithm produces a match set. Then, in the next section, we set about deriving the function *resolve* that was left unimplemented.

3.1. Matching

The function *matches* takes a set of rules and returns a match set. It is defined recursively (using the notation of Haskell [4]):

$$\begin{aligned} matches &:: [Rule] \to [Subst] \\ matches\,[\,] &= [idSubst] \\ matches\,(X : Xs) &= [\,(\phi \circ \sigma) \mid (\sigma, Ys) \leftarrow resolve\,X, \\ &\qquad\qquad \phi \leftarrow matches\,(\sigma(Ys \mathbin{+\!\!+} Xs))]. \end{aligned}$$

That is, the empty set of rules has the singleton set containing the identity substitution as a match set. For a non-empty set of rules $(X : Xs)$, we take the first rule X and break it down into a (possibly empty) set of smaller rules Ys together with a substitution σ which makes Ys equivalent to X. We then combine the Ys with Xs, the remainder of the original rules, apply σ, and return the results of a recursive call to *matches* combined with σ.

The function that breaks up X into smaller rules is called *resolve*. Readers who are familiar with the logic programming paradigm will recognise it as being analogous to the concept of “resolution”.

Clearly it would be advantageous to arrange the rules in such a manner that we first consider rules where *resolve X* is small, perhaps only a singleton. There is no particular reason why we should take the union of *Ys* and *Xs* by list concatenation: we could place 'cheap' rules at the front, and 'expensive' rules at the back.

We shall not implement

$$resolve :: Rule \to [(Subst, [Rule])]$$

as yet, because that involves quite a long and tricky case analysis. Instead, we specify its behaviour through three properties. Let

$$[(\sigma_0, Ys_0), (\sigma_1, Ys_1), \ldots, (\sigma_k, Ys_k)] = resolve\ X.$$

We require that

- For all normal substitutions ϕ:

$$(\phi \vdash X) \equiv \bigvee_i (\phi \vdash Ys_i \wedge \sigma_i \leqslant \phi).$$

- For all normal substitutions ϕ and indices i and j:

$$(\phi \vdash Ys_i) \wedge (\phi \vdash Ys_j) \Rightarrow i = j.$$

- For each index i, σ_i is pertinent to X, closed and normal.
- The pattern variables in Ys_i are contained in the pattern variables of X.
- For each index i:

$$Ys_i \ll X.$$

The first of these is a soundness and completeness condition: it says that all relevant matches can be reached via *resolve*, and that *resolve* stays true to the original set of rules. The second condition states that *resolve* should not return any superfluous results. The third and fourth conditions are technical requirements we need to prove the non-redundancy of *matches*. Finally, the last condition states that we make progress by applying *resolve*; *i.e.* that the process of breaking down the set of rules will eventually terminate.

3.2. Proof of correctness

To prove that *matches* does indeed return a match set, we have to verify two properties:

- For each normal substitution ϕ:

$$(\phi \vdash X : Xs) \equiv \exists \psi \in matches(X : Xs) : \psi \leqslant \phi.$$

- For all normal substitutions ϕ and ψ:

$$(\phi, \psi \in matches(X : Xs) \wedge \phi \leqslant \psi) \Rightarrow (\phi = \psi).$$

The first requirement is a soundness and completeness property, while the second asserts non-redundancy. Let

$$[(\sigma_0, Ys_0),\ (\sigma_1, Ys_1), \ldots, (\sigma_k, Ys_k)] = \mathit{resolve}\ X.$$

For soundness and completeness, we proceed by induction on $\ll$:

$\phi \vdash X : Xs$

$\equiv$ {definition of $\vdash$ on list of rules}

$\phi \vdash X \wedge \phi \vdash Xs$

$\equiv$ {soundness and completeness of *resolve*}

$\left(\bigvee_i \phi \vdash Ys_i \wedge \sigma_i \leqslant \phi\right) \wedge \phi \vdash Xs)$

$\equiv$ {since $(\wedge)$ distributes over $(\vee)$}

$\bigvee_i (\phi \vdash Ys_i \wedge \sigma_i \leqslant \phi \wedge \phi \vdash Xs)$

$\equiv$ {definition of generality $(\leqslant)$}

$\bigvee_i (\phi \vdash Ys_i \wedge (\exists \delta : \phi = \delta \circ \sigma_i) \wedge \phi \vdash Xs)$

$\equiv$ {predicate logic}

$\bigvee_i (\exists \delta : \phi \vdash Ys_i \wedge \phi = \delta \circ \sigma_i \wedge \phi \vdash Xs)$

$\equiv$ {since $\delta \circ \sigma_i \vdash Xs \equiv \delta \vdash (\sigma_i\, Xs)$}

$\bigvee_i (\exists \delta : \phi = \delta \circ \sigma_i \wedge \delta \vdash (\sigma_i\, Ys_i) \wedge \delta \vdash (\sigma_i\, Xs))$

$\equiv$ {definition of $\vdash$ on list of rules}

$\bigvee_i (\exists \delta : \phi = \delta \circ \sigma_i \wedge \delta \vdash (\sigma_i\, (Ys_i \mathbin{+\!\!+} Xs)))$

$\equiv$ {progress of *resolve*; $Ys_i \mathbin{+\!\!+} Xs \ll (X : Xs)$; induction}

$\bigvee_i (\exists \delta : \phi = \delta \circ \sigma_i \wedge (\exists \chi \in \mathit{matches}\ (\sigma_i\, (Ys_i \mathbin{+\!\!+} Xs)) : \chi \leqslant \delta))$

$\equiv$ {predicate logic}

$\bigvee_i (\exists \chi \in \mathit{matches}\ (\sigma_i\, (Ys_i \mathbin{+\!\!+} Xs)) : \exists \delta : \phi = \delta \circ \sigma_i \wedge \chi \leqslant \delta)$

$\equiv$ {property of generality $(\leqslant)$, see below}

$\bigvee_i (\exists \chi \in \mathit{matches}\ (\sigma_i\, (Ys_i \mathbin{+\!\!+} Xs)) : \chi \circ \sigma_i \leqslant \phi)$

$\equiv$ {definition of *matches*, *take* $\psi = \chi \circ \sigma_i$}

$\exists \psi \in \mathit{matches}\, (X : Xs) : \psi \leqslant \phi$

In the penultimate step, we used a property of the generality preorder ($\leqslant$), namely

$$(\exists \delta : \phi = \delta \circ \sigma \wedge \chi \leqslant \delta) \equiv \chi \circ \sigma \leqslant \phi.$$

The proof of this property is a simple exercise in applying the definition of generality, and we omit details.

It remains to prove non-redundancy of *matches Xs*. We first prove by induction over the measure of a rule set that the substitutions returned by *matches Xs* are closed and pertinent to *Xs*. Clearly the identity substitution is closed and pertinent, so the base case for the empty rule set is satisfied. For the case of *matches* $(X : Xs)$, we know that the substitution σ returned by *resolve* is closed, and pertinent to X, and by the induction hypothesis ϕ is closed and pertinent to $\sigma(Ys +\!\!+ Xs)$. Since the pattern variables in *Ys* are also in *X*, $(\phi \circ \sigma)$ must be closed and pertinent to $(X : Xs)$, thus completing the proof.

Let us now move on to the proof of non-redundancy, which also proceeds by induction on the measure of a rule set. The base case is trivially satisfied since we only return a single substitution for the empty rule set. For *matches* $(X : Xs)$, let ϕ and ψ be elements of *matches* $(X : Xs)$. Furthermore, assume that $\phi \leqslant \psi$. By definition of *matches*, there are normal substitutions ϕ' and ψ', and indices i and j such that

$$\phi = \phi' \circ \sigma_i \quad \text{and} \quad \psi = \psi' \circ \sigma_j,$$

with (σ_i, Ys_i) and (σ_j, Ys_j) in *resolve* X. By the soundness of *resolve*, we have

$$\phi \vdash Ys_i \quad \text{and} \quad \psi \vdash Ys_j.$$

Now because $\phi \leqslant \psi$, we have

$$\psi \vdash Ys_i \quad \text{and} \quad \psi \vdash Ys_j.$$

It follows that $i = j$ by the non-redundancy of *resolve*.

Since $\phi \leqslant \psi$, there exists δ such that $\delta \circ \phi = \psi$. Therefore, $\delta \circ \phi' \circ \sigma_i = \psi' \circ \sigma_i$. We know that ϕ' and ψ' are pertinent to $\sigma_i(Ys +\!\!+ Xs)$, so they cannot make any changes to variables which are changed by σ_i.

Construct δ' by restricting the domain of δ to variables not changed by σ_i. Now consider a pattern variable p. If p is changed by σ_i, it cannot be changed by ϕ', ψ' or δ', and so $(\delta' \circ \phi')p = \psi' p$. If p is not changed by σ_i, then $\sigma_i p = p$ and so $(\delta' \circ \phi')p = \psi' p$.

Therefore $\delta' \circ \phi'$ and ψ' are equal on all pattern variables. This equality can be extended to all terms by structural induction, so $\delta' \circ \phi' = \psi'$ and thus $\phi' \leqslant \psi'$. By the induction hypothesis, $\phi' = \psi'$ and so $\phi = \psi$.

4. Implementing *resolve*

The function *resolve* breaks down a rule into smaller rules, recording substitutions along the way. It does so by syntactic analysis of the shape of the argument rule. In all there are seven cases to consider, and these are summarised in the table below. The intention is that the first applicable clause is applied. The reader is reminded of the notational distinction we make between variables: x and y represent local variables, a and b constants, and p a pattern variable.

X	*resolve* X
$x \to y$	$[(id, [])]$, if $x = y$
	$[\,]$, otherwise
$p \to T$	$[(p := T, [])]$, if T is closed
	$[\,]$, otherwise
$a \to b$	$[(id, [])]$, if $a = b$
	$[\,]$, otherwise
$(\lambda x.P) \to (\lambda x.T)$	$[(id, [P \to T])]$
$(\lambda x.P) \to T$	$[(id, [P \to (T\,x)])]$
$(F\,E) \to T$	$[(id, [(F \to T_0), (E \to T_1)]) \mid (T_0\,T_1) = T] +\!\!+$
	$[(id, [(F \to T_0), (E \to T_1)]) \mid (T_0, T_1) \leftarrow apps\,T] +\!\!+$
	$[(id, [F \to (\lambda x.T)])]$, x fresh
$P \to T$	$[\,]$

Let us now examine each of these clauses in turn.

The first clause says that two local variables match only if they are equal.

The second clause says that we can solve a rule $(p \to T)$ where the pattern is a pattern variable by making an appropriate substitution. Such a substitution can only be made, however, if T does not contain any local variables occurring without their enclosing λ: since the original term cannot contain any pattern variables, any variables in T must have been bound in the original term and so the substitution would move these variables out of scope.

The third clause deals with matching of constants a and b. These only match when they are equal.

Next, we consider matching of λ-abstractions $(\lambda x\,.\,P)$ and $(\lambda x\,.\,T)$. Here it is assumed that the clauses are applied modulo renaming, so that the bound variable on both sides is the same, namely x. To match the λ-abstractions is to match their bodies.

Recall, however, that we took equality in the definition of matching not only modulo renaming, but also modulo η-conversion. We therefore have to cater for the possibility that the pattern contains a λ-abstraction, but the term (which was assumed to be normal) does not. This is the purpose of the clause for matching $(\lambda x\,.\,P)$ against a term T that is not an abstraction: we simply expand T to $(\lambda x\,.\,T\,x)$ and then apply the previous clause.

The sixth clause deals with matching where the pattern is an application $(F\,E)$. This is by far the most complicated clause, and in fact the only case where *resolve* may return a list with more than one element. It subsumes the projection and imitation steps of Huet's algorithm; in essence, it attempts to write the term T as an application in three different ways. The first and simplest way is to leave T unchanged: of course this only gives an application if $T = T_0\, T_1$ (for some T_0 and T_1) in the first place. If that condition is satisfied, we match F against T_0, and E against T_1.

Another way of writing T as an application is to take (T_0, T_1) from *apps* T. This function returns all pairs of normal expressions (T_0, T_1) such that:

$$\begin{array}{ll} \exists B : (& T_0 = \lambda x\,.\,B \\ & \wedge\ (x := T_1)B = T \\ & \wedge\ x \text{ occurs in } B,\ x \text{ fresh}). \end{array}$$

For example, a correct implementation of *apps* would return

$$\begin{array}{ll} \mathit{apps}\,(a+a) = [& (\lambda x\,.\,x+x, a) \\ & (\lambda x\,.\,x+a, a) \\ & (\lambda x\,.\,a+x, a) \\ & (\lambda x\,.\,x, a+a) \\ & (\lambda x\,.\,x\,a, ((+)\,a)) \\ & (\lambda x\,.\,x\,a\,a, (+))\ ]. \end{array}$$

We require that x occurs in B because otherwise the value of T_1 would not matter: it could be absolutely anything. The most general choice for T_1 would then be a fresh free variable – but introducing such a variable would go against our dictum that the term in a rule must be closed: substitutions are applied to the pattern, but not to the term. We therefore deal with the case of x not occurring in B separately: in that case, all we need to do is match F against $(\lambda x\,.\,T)$, and the argument E in the pattern is ignored.

The final clause in the definition of *resolve* says that if none of the earlier clauses apply, the pattern does not match the term, and the empty list is returned.

To implement *resolve*, all that is needed is an effective definition of *apps*. The function *apps* T can in fact be implemented by abstracting subexpressions from T, in all possible ways. This is fairly easy to program, and we omit details.

4.1. Proof of correctness

We now turn to a proof of the correctness of *resolve*. First we examine a way of simplifying the proof obligations, and then we examine the case of matching against an application in detail.

4.1.1. Simplifying the proof obligation

It is difficult to reason in terms of equality modulo η-conversion, so we aim to eliminate $(\simeq)$ from the definition of $\vdash$. Let ϕ be a normal substitution, and $P \to T$ a rule. First observe that

$$\begin{aligned}
& \phi \vdash P \to T \\
\equiv\ & \{\text{definition of } \vdash\} \\
& \mathit{step}\,(\phi\, P) \simeq T \\
\equiv\ & \{\eta\text{-conversion, and } T \text{ normal}\} \\
& \mathit{etaNormalise}\,(\mathit{step}\,(\phi\, P)) = T.
\end{aligned}$$

Now recall the definition of rules, which states that the pattern P should have no η-redexes, and that the substitution ϕ is required to be normal. It follows that ϕP has no η-redexes. On an argument E that has no η-redexes, we have

$$\mathit{etaNormalise}\,(\mathit{step}\,E) \equiv \mathit{step}'\,E,$$

where step' is defined as follows:

$$\begin{aligned}
\mathit{step}'\,c &= c, \\
\mathit{step}'\,x &= x, \\
\mathit{step}'\,p &= p, \\
\mathit{step}'\,(\lambda x.B) &= \mathit{etaRed}\,(\lambda x.(\mathit{step}'\,B)), \\
\mathit{step}'\,(E_1\,E_2) &= \text{case } E_1' \text{ of} \\
&\qquad (\lambda x.B) \to (x := E_2')B \\
&\qquad - \qquad\ \ \to (E_1'\ E_2'),
\end{aligned}$$

where

$$\begin{aligned}
E_1' &= \mathit{step}'\,E_1, \\
E_2' &= \mathit{step}'\,E_2.
\end{aligned}$$

Here the function *etaRed* removes top-level η-redexes. This property of *step* is easily proved: the only case in which $\mathit{step}\,E$ can introduce new η-redexes is when it is applied to a λ-abstraction. If we remove such newly introduced η-redexes, the result is η-contracted.

In summary, we have argued that

$$(\phi \vdash P \to T) \equiv (\mathit{step}'(\phi\, P) = T).$$

This is the characterisation of $\vdash$ that we shall use below. We stress once more that its proof depends on the assumption that the substitution ϕ is normal, and the data type invariant of rules, which states that the pattern P is free of η-redexes, and the term T is normal. The proof that our algorithm maintains this invariant can be found in an appendix.

4.1.2. Matching against an application

To prove that the above implementation of *resolve* is indeed correct, we need to verify that each of the four requirements in its specification is satisfied. Such a proof is long and tedious, as it involves a case analysis, split into 7 cases for each requirement, making a total of 28 subproofs. Fortunately most of these are easy, and below we concentrate on the most difficult part, namely matching against a pattern that is an application.

To prove soundness and completeness in this case, we have to show:

$$\begin{aligned}\phi \vdash (F\ E \to T) \equiv\ & (\exists T_0, T_1 : (T_0 T_1) = T \wedge \phi \vdash \{F \to T_0, E \to T_1\})\\ & \vee\ (\exists (T_0, T_1) \in \mathit{apps}(T) : \phi \vdash \{F \to T_0, E \to T_1\})\\ & \vee\ (\phi \vdash \{F \to \lambda x.T\}).\end{aligned}$$

To prove this equivalence, we shall first massage its left-hand side, aiming to separate the case that T is an application from when it is not:

$$\begin{aligned}& \phi \vdash (F\ E \to T)\\ \equiv\ & \{\text{definition of } \vdash\}\\ & \mathit{step}'(\phi(F\ E)) = T\\ \equiv\ & \{\text{definition of substitution application}\}\\ & \mathit{step}'((\phi F)\ (\phi E)) = T\\ \equiv\ & \{\text{definition of } \mathit{step}',\ \text{let } F' = \mathit{step}'(\phi F) \text{ and } E' = \mathit{step}'(\phi E)\}\\ & \begin{pmatrix}\text{case } F' \text{ of} & \\ (\lambda x.B) & \to (x := E')B\\ \text{-} & \to F'\ E'\end{pmatrix} = T\\ \equiv\ & \{\text{semantics of case}\}\\ & \quad (\exists B : F' = \lambda x.B \wedge (x := E')B = T)\\ & \vee\ (\neg(\exists B : F' = \lambda x.B) \wedge (F'\ E' = T))\end{aligned}$$

We continue with the two disjuncts separately. The second disjunct is easy:

$$\begin{aligned}& \neg(\exists B : F' = \lambda x.B) \wedge (F'\ E' = T)\\ \equiv\ & \{T \text{ is normal}\}\\ & F'\ E' = T\\ \equiv\ & \{\text{definition } \vdash\}\\ & \exists T_0, T_1 : T_0 T_1 = T \wedge \phi \vdash \{F \to T_0,\ E \to T_1\}\end{aligned}$$

For the other disjunct, we argue

$$\begin{aligned}& \exists B : F' = \lambda x.B \wedge (x := E')B = T\\ \equiv\ & \{\text{predicate logic}\}\end{aligned}$$

$$\begin{aligned}
&\exists B_0, B_1 : \quad F' = \lambda x.B_0 \\
&\qquad\quad \wedge\ B_1 = E' \\
&\qquad\quad \wedge\ (x := B_1)B_0 = T \\
\equiv\ &\{\text{property of substitution}\} \\
&\quad (\exists B_0, B_1 : \quad F' = \lambda x.B_0 \\
&\qquad\qquad\quad \wedge\ B_1 = E' \\
&\qquad\qquad\quad \wedge\ (x := B_1)B_0 = T \\
&\qquad\qquad\quad \wedge\ x \text{ occurs in } B_0 \\
&\quad \vee\ F' = \lambda x.T) \\
\equiv\ &\{\text{definitions of } F',\ E',\ \mathit{apps} \text{ and } \vdash\} \\
&\quad \exists (T_0, T_1) \in \mathit{apps}\, T : \phi \vdash \{F \to T_0, E \to T_1\} \\
&\quad \vee\ \phi \vdash (F \to \lambda x.T).
\end{aligned}$$

In the forward implication of the last step, we need to know that B_1 and $\lambda x.B_0$ are normal in order to apply the definition of *apps*. Normalness of B_1 follows from normalness of T (as B_1 is a subexpression of T). Similarly, normalness of B_0 follows from normalness of T. Because B_0 is normal, the abstraction $\lambda x.B_0$ can only fail to be normal by being an η-redex; but F' is not an η-redex (because it results from *step*$'$), and $\lambda x.B_0 = F'$.

This completes the proof of soundness and completeness, in the case of a pattern that is an application. In this case, the progress condition is clearly satisfied since "F" and "E" together have one less symbol than "$F\ E$", so it remains to prove that

$$\mathit{resolve}(F\ E \to T) = [(\sigma_0, \mathit{Ys}_0), (\sigma_1, \mathit{Ys}_1), \ldots, (\sigma_k, \mathit{Ys}_k)]$$

does not contain any redundant elements and that the σ_is are closed and pertinent to $F\ E \to T$. Because each of the σ_i is the identity substitution, they must be closed, normal and pertinent, and the non-redundancy proof obligation comes down to

$$\phi \vdash \mathit{Ys}_i \wedge \phi \vdash \mathit{Ys}_j \Rightarrow i = j.$$

This implication follows because each Ys_i contains a rule whose left-hand side is F:

$$(F \to E_i) \in \mathit{Ys}_i.$$

Now observe that

$$\begin{aligned}
&\phi \vdash \mathit{Ys}_i \wedge \phi \vdash \mathit{Ys}_j \\
\Rightarrow\ &\{\text{since } (F \to E_i) \in \mathit{Ys}_i\} \\
&\phi \vdash (F \to E_i) \wedge \phi \vdash (F \to E_j) \\
\equiv\ &\{\text{definition of } (\vdash)\} \\
&E_i = \mathit{step}'(\phi\, F) = E_j.
\end{aligned}$$

However all the E_i are distinct, so $i=j$. To see that the E_i are distinct, recall that there are three cases to consider:

$$\begin{array}{ll} & (E_i, Y) \in \mathit{apps}\ T \\ \text{or} & E_j = \lambda x\,.\,T \text{ where } x \text{ is fresh} \\ \text{or} & E_k\,T_1 = T. \end{array}$$

To show that these three cases are mutually exclusive, we argue:

$i \neq j$: Note that $E_i = \lambda x\,.\,B_x$ for some B_x, with x occurring in B_x. It follows that $E_i = \lambda x\,.\,B_x \neq \lambda x\,.\,T = E_j$.

$j \neq k$: If $E_j = E_k$, we have $E_j = \lambda x\,.\,T = \lambda x\,.\,E_k\,T_1 = \lambda x\,.\,(E_j\,T_1)$. An expression cannot occur inside itself, so this is a contradiction.

$i \neq k$: If $E_i = E_k$, then $E_i = (\lambda x\,.\,B_x) = E_k$ for some B_x. But this implies that $T = E_k T_1 = (\lambda x\,.\,B_x)\,T_1$ contains a β-redex. That contradicts the normalness of T.

4.2. Necessity of preconditions

Let us now return to the specification of the matching problem, and examine the preconditions we imposed on the input, namely that the pattern is free of η-redexes, and that the term is normal. Are these conditions merely to facilitate the above proof, or does the algorithm go wrong when they are not satisfied?

Consider the following pattern that contains an η-redex:

$$P = (\lambda f\,.\,(\lambda y\,.\,f\,y))\ (\lambda x\,.\,0)\ 5.$$

When we apply our algorithm to match this pattern against the term 0, it reports the identity substitution as a match. But we have

$$\mathit{step}\ P = (\lambda x\,.\,0)\ 5,$$

which is *not* equal to 0 under α/η-equality. We conclude that our algorithm is not sound for patterns that contain η-redexes.

Now consider matching the pattern $\lambda x\,.x$ against $\lambda x\,.(\lambda y\,.x\,y)$. Here our algorithm does not find any matches, and yet it is the case that

$$\mathit{step}\ (\lambda x\,.x) = (\lambda x\,.x) \simeq \lambda x\,.(\lambda y\,.x\,y).$$

We conclude that our algorithm is not complete for terms that contain η-redexes.

If we match the pattern $p\ 0$ against the term $(\lambda x\,.x)\,0$, the algorithm reports $\{p := (\lambda x\,.x)\}$. However, we have

$$\mathit{step}((\lambda x\,.x)\ 0) = 0 \not\simeq ((\lambda x\,.x)\ 0),$$

so our algorithm is not sound for terms that contain β-redexes. It follows that all three components of the precondition are necessary for our algorithm to be sound and complete.

5. Inclusion of all second-order matches

As remarked earlier, our algorithm does not depend on a particular typing discipline for its correctness. However, if we use the simply typed lambda calculus (and run the algorithm ignoring the type information), the algorithm does return all matches of second-order or lower, so long as the pattern does not contain any β-redexes. For the purpose of this section, we regard a substitution as a finite set of (variable, term) pairs. The order of a substitution (or a match) is the maximum of the order of the terms it contains.

To show that our algorithm returns all matches of second-order or lower, consider a rule $P \to E$, where P does not contain any β-redexes. (Recall that in a rule, the term E is always normal and therefore free of β-redexes.) Let ϕ be a match between P and E:

$$\mathit{betanormalise}\ (\phi P) \simeq E.$$

Furthermore assume that ϕ does not contain any terms of order greater than 2. We aim to show that ϕ is in the match set of $P \to E$; the proof is by contradiction.

Suppose that ϕ is not represented in the match set. Then by completeness, we have $\mathit{step}\ (\phi P) \not\simeq E$. Since ϕ is a match and E does not contain any β-redexes, $\mathit{reduce}\ (\phi P) \simeq E$, and so $\mathit{reduce}\ (\mathit{step}\ (\phi P)) \not\simeq \mathit{step}\ (\phi P)$. Therefore $\mathit{step}\ (\phi P)$ must contain a β-redex, the left-hand side of which must be of at least second-order (since a first-order term cannot occur on the left-hand side of a function application). In fact, more can be said about the orders:

Lemma. *If step T contains* a *β-redex $(\lambda x \,.\, B)\ C$ where $\lambda x \,.\, B$ is of order n, then T contains a β-redex with an $(n+1)$th-order term as the function part.*

Proof. From the definition of *step*, either a subterm of $\mathit{step}\ T$ contains the β-redex in which case we proceed by induction on the size of $\mathit{step}\ T$, or $T = T_0\ T_1$ where $\mathit{step}\ T_0 = \lambda y \,.\, B'$, $\mathit{step}\ T_1 = C'$ and $(y := C')B' = \mathit{step}\ T$.

If B' or C' contain the β-redex, we proceed by induction on the size of $\mathit{step}\ T$ as before. Since substitution does not affect the order of a lambda term, we can also do this if B' contains $(\lambda x \,.\, B)\ D$ as a subterm, where $(y := C')D = C$.

Otherwise, it must be the case that B' contains a subterm of the form $y\ C$, and that $C' = \lambda x \,.\, B$. Since C' is a term of order n, T_1 must be too. Also since $\mathit{step}\ T_0 = \lambda y \,.\, B'$, T_0 must either be a λ-abstraction or it must be a β-redex with a λ-abstraction as its head. In either case, the λ-abstraction has a parameter of order n so it must be of order $n+1$ or greater, and so T contains a $(n+1)$th or higher order term on the left-hand side of a β-redex, as required. □

Returning to our main proof, ϕP contains a third-order term on the left-hand side of a β-redex. But since P does not contain any β-redexes, a third-order term must occur in ϕ, which contradicts the assumption that it is a second-order match. Therefore *step* $(\phi P) \simeq E$, and therefore there exists a ψ in the match set of $(P \to E)$ such that $\psi \leqslant \phi$. This completes the proof that our algorithm subsumes second-order matching.

As we stated earlier, our algorithm also returns some matches which have an order greater than two. We see a rather trivial example of this if we match $p\ (\lambda x\,.\,x+1)$ against $\lambda x\,.\,x+1$; we get the match $p := \lambda y\,.\,y$, which is of type $(Int \to Int) \to Int \to Int$ and therefore a third-order function.

A more practically relevant example comes from using term rewriting to transform functional programs. The naive quadratic time program for reversing a list can be expressed as a "fold":

$$\begin{aligned} reverse &= foldr\ (\lambda x\,.\,\lambda xs\,.\,xs \mathbin{+\!\!+} [x])\ [], \\ foldr\ (\oplus)\ e\ []\ &= e, \\ foldr\ (\oplus)\ e\ (x : xs) &= x \oplus (foldr\ (\oplus)\ e\ xs). \end{aligned}$$

We can then define *fastrev xs ys* = *reverse xs* $\mathbin{+\!\!+}$ *ys* and transform this to a more efficient linear time program using the "fold fusion" law:

$$f\ (foldr\ (\oplus)\ e\ xs) = foldr\ (\otimes)\ (f\ e)\ xs \text{ if } \lambda x\ y\,.\,x \otimes (f\ y) = \lambda x\ y\,.\,f(x \oplus y).$$

Application of this law proceeds as follows. First the left-hand side is matched with the expanded definition of *fastrev*, giving the substitutions

$$\{f := (\mathbin{+\!\!+}),\ (\oplus) := (\lambda x\,.\,\lambda xs\,.\,xs \mathbin{+\!\!+} [x]),\ e := []\}.$$

We apply this substitution to the right-hand side of the side condition, and rewrite it as far as possible, using the associativity of concatenation. We now need to solve the rule

$$\lambda x\ y\,.\,x \otimes (f\ y) \to \lambda x\ y\ ys\,.\,y \mathbin{+\!\!+} (x : ys).$$

This is where higher-order matching comes in. The definition that needs to be generated is $(\otimes) = \lambda x\,.\,\lambda g\,.\,\lambda ys\,.\,g\ (x : ys)$, which is a third-order function (since g is a function). We have applied our algorithm to many similar examples with the MAG system [10]; that paper gives a much more detailed account of the way higher-order matching is applied in the context of program transformation. In particular it shows how the above transformation can be done without first needing to express *reverse* as a fold.

Finally, we stress that the algorithm does not find all third-order matches. For example, when matching $p\ q$ against 0, we do not get the match $p := \lambda g\,.\,g\,0,\ q := \lambda a\,.\,a$. As we remarked in the introduction, the set of third-order matches is potentially infinite.

6. Implementation notes

The above presentation may be clear, but if the algorithm is implemented exactly as stated, it is rather inefficient. The call tree of *matches* contains many paths that end in failure, so we need ways of pruning out such unsuccessful paths. In this section, we describe three ways of achieving that objective. Together, these three techniques yield a program whose performance is on a par with Huet and Lang's algorithm.

6.1. Flexible versus non-flexible heads

The *head* of an expression is the first non-application found when following left-hand branches of applications:

$$\begin{aligned} \mathit{head}\ (F\ E) &= \mathit{head}\ F, \\ \mathit{head}\ E &= E \quad \text{if } E \text{ is not an application.} \end{aligned}$$

An expression is said to be *flexible* if its head is a pattern variable or a λ-abstraction.

A rule whose term is a λ-abstraction can only be successfully solved if the pattern is flexible. In particular, when we match against a pattern $(F\ E)$, there is no point in considering all but the first rules returned by *resolve*, unless the pattern is flexible. It follows that the only case where *resolve* needs to generate multiple results is when matching against a pattern that is a flexible application.

6.2. Selection rule

When we presented the algorithm for breaking down sets of rules, we already remarked that one can vary the order in which rules are considered. In the light of the preceding subsection, it makes sense to defer the consideration of *hard* rules, that is those whose pattern is a flexible application. This leads to the following program for *matches*:

$$\begin{aligned} \mathit{matches} &:: [\mathit{Rule}] \to [\mathit{Subst}], \\ \mathit{matches}\ [\,] &= [\mathit{idSubst}], \\ \mathit{matches}\ (X : \mathit{Xs}) &= [(\phi \circ \sigma) \mid\ (\sigma, \mathit{Ys}) \leftarrow \mathit{resolve}\ X, \\ &\qquad\qquad\quad \phi \leftarrow \mathit{matches}\ (\sigma(\mathit{Ys}_1 \mathbin{+\!\!+} \mathit{Xs} \mathbin{+\!\!+} \mathit{Ys}_0)), \\ &\qquad\qquad\quad (\mathit{Ys}_0, \mathit{Ys}_1) = \mathit{partition}\ \mathit{hard}\ \mathit{Ys}]. \end{aligned}$$

Here the function *partition hard Ys* partitions its argument list into those that do satisfy the predicate *hard*, and those that do not.

6.3. Viability test

A common technique for speeding up traditional, first-order matching algorithms is to test whether the constants in the pattern also occur in the term before even attempting to match. In the context of higher-order matching, that does not quite work, because it can happen that a match actually removes some constants from the pattern, by substituting a projection function for a flexible head. Nevertheless, we can identify those constants that cannot be removed by any match: we shall call such constants *rigids*. Formally the rigids of an expression are defined as follows:

$$\begin{aligned}
\mathit{rigids}\ c &= [c],\\
\mathit{rigids}\ x &= [],\\
\mathit{rigids}\ p &= [],\\
\mathit{rigids}\ (\lambda x\,.E) &= \mathit{rigids}\ E,\\
\mathit{rigids}\ (E_1\ E_2) &= \mathit{rigids}\ E_1 +\!\!+ \mathit{rigids}\ E_2, \quad \text{if } \neg \mathit{flexible}\ (\mathit{head}\ E_1)\\
&= [], \text{otherwise.}
\end{aligned}$$

A match between P and T can only be successful if *rigids* P is a subsequence of the constants of T.

A dual condition applies to the bound variables in the term that do not occur under a binding λ: the only way these can be successfully matched is by matching them against the same bound variables in the pattern. Again, we can only find a match between P and T if the local variables of T are a subset of the local variables of P.

If both these tests succeed, we say that a rule is *viable*: we filter the result of *resolve* to remove all non-viable rules.

6.4. Comparison with Huet and Lang's algorithm

In preliminary computational experiments, we implemented a version of our algorithm that keeps track of polymorphic types, and we similarly adapted Huet and Lang's algorithm to cope with polymorphic typing. Measuring absolute times, Huet and Lang's algorithm is usually faster, because it finds far less matches. For instance, on the examples quoted in [9], our algorithm returns about 7 times more matches. When we measure the average time taken *per match*, the situation is reversed and our algorithm beats the particular implementation of Huet and Lang's algorithm. Because of the preliminary nature of these experiments, and the lack of a solid base of benchmarks, we cannot draw any firm conclusions from these results, except that our algorithm is not inherently more expensive than existing methods, despite its additional flexibility. We hope to report on a more definitive comparison in a forthcoming paper that focuses on implementation aspects.

7. Discussion

Higher-order matching allows many program transformations to be concisely expressed as rewrite rules. Two examples of systems that have incorporated its use are KORSO [15] and MAG [10]. The Ergo system is based on higher-order unification [21]. Despite the conceptual advantages offered by higher-order matching, there also exist very successful transformation systems that do not incorporate its use, for example

Kids [23] and APTS [19]. There are two significant objections to the use of higher-order matching. First, even second-order matching is known to be NP-hard [7, 25], so a truly efficient implementation is out of the question. Second, higher-order matching algorithms are restrictive, in particular in the typing discipline that they require. In this paper, we have demonstrated how that second objection can be eliminated, by giving an algorithm that operates on untyped terms.

Although there is a clear specification for the set of matches the algorithm returns, it is sometimes not quite obvious why a particular match was not produced. This contrasts with Huet and Lang's algorithm, where the reason for failed matches is crystal clear: it takes some time to gain an intuition of what the function *step* does, whereas it is easy to see whether a function is second-order or not. In our experience with the MAG system [10] there seem to be a handful of techniques to deal with failed matches (for instance 'raising' a rule by introducing explicit abstractions), so we feel that the disadvantage is not too serious.

There is a wealth of related work on higher-order matching and unification [5, 7, 11–13, 16, 18, 24, 25], to name just a few. One important concept identified in some of these works (in particular [16, 18]) is that of a restricted notion of higher-order pattern. To wit, a *restricted pattern* is a normal term where every occurrence of a free function variable is applied to a list of distinct local variables, and nothing else. For such restricted patterns, much simpler and more efficient matching and unification algorithms are possible. Our algorithm returns all higher-order matches for rules where the pattern satisfies the above restriction; in fact there is at most one such match. We have not yet investigated the efficiency of our algorithm in this important special case.

There are a number of specialised *pattern languages* for the purpose of program inspection and transformation e.g. [1, 2, 8]. Often these do not include higher-order patterns, and it would be interesting to see what primitives suggested in these languages can be profitably combined with higher-order matching.

The work reported here is part of a larger effort to produce a convenient meta-language for describing transformations in the *Intentional Programming* (IP) system, now under development at Microsoft Research [22]. The IP system is an environment for rapid prototyping of domain-specific language constructs and optimising transformations. For IP to be successful, the specification of new transformations has to be as painless as possible — we believe that higher-order matching is indispensable from that perspective. The IP system maintains the program as an abstract syntax tree, where the only primitive notion is that of binding. Indeed, one can think of IP's internal representation of programs as untyped λ-terms, and we have good hopes that our algorithm can be incorporated in IP.

It remains to be seen whether we can overcome the second objection to higher-order matching in program transformation, namely its inherent inefficiency. We are currently investigating to what extent techniques for fast implementation of first-order matching [6] can be applied here. Preliminary experiments show that the efficiency of our algorithm is comparable to that of the algorithm by Huet and Lang.

Acknowledgements

Ganesh Sittampalam would like to thank Microsoft Research for its generous financial support. We would like to thank Charles Simonyi and Will Aitken (both of Microsoft Research) for helpful discussions on the topic of this paper, and its relation to other work on IP. Zena Ariola, Henk Barendregt, and David Wolfram kindly answered our questions, and pointed us to related work. Richard Bird, Tobias Nipkow, and Luke Ong commented on an earlier draft.

Appendix A. Verification of the other cases of *resolve*

In our proof of the correctness of *resolve*, we left out several "easy" subproofs, along with the justification that the invariant (that in a rule $P \rightarrow T$, P does not contain any η-redexes and T is normal) is maintained. These are included here for the sake of completeness.

Note that the non-redundancy condition is trivially satisfied in all these cases since they all return at most one result. Similarly in each case it is obvious that the progress condition is satisfied. Since we return the identity substitution in all cases except when matching against a pattern variable, the "closed and pertinent" property is also trivially satisfied in these cases. We never introduce new pattern variables, so the pattern variables in the rule sets returned must be in the original rule.

A.1. Matching local variables

To prove soundness and completeness, we argue:

$$
\begin{aligned}
& \phi \vdash x \rightarrow y \\
\equiv \quad & \{\text{property of } \vdash \text{ deduced earlier}\} \\
& \mathit{step}'(\phi\, x) = y \\
\equiv \quad & \{x \text{ is a local variable and cannot be substituted}\} \\
& \mathit{step}'\ x = y \\
\equiv \quad & \{\text{definition of } \mathit{step}'\} \\
& x = y.
\end{aligned}
$$

Clearly the invariant is maintained as we do not generate any new rules or bindings.

A.2. Matching against a pattern variable

To prove soundness and completeness, we argue:

$$\phi \vdash p \rightarrow T$$

$\equiv$ {property of $\vdash$}

$step'(\phi\ p) = T$

$\equiv$ {ϕ is a normal substitution}

$\phi\ p = T$

$\equiv$ {property of substitution}

$((p := T) \leqslant \phi) \wedge$ T is closed.

Since T is normal, the invariant is maintained.

Since T is not allowed to contain any pattern variables, and we insist that $locals(T) = \emptyset$, the substitution $(p := T)$ must be closed and pertinent to $(p \to T)$.

A.3. Matching constants

To prove soundness and completeness, we argue:

$\phi \vdash a \to b$

$\equiv$ {property of $\vdash$}

$step'\ (\phi\, a) = b$

$\equiv$ {a is a constant and cannot be substituted}

$step'\ a = b$

$\equiv$ {definition of $step'$}

$a = b.$

Clearly the invariant is maintained as we do not generate any new rules or bindings.

A.4. Matching λ-abstractions

To prove soundness and completeness, we argue:

$\phi \vdash (\lambda x\,.\,P) \to (\lambda x\,.\,T)$

$\equiv$ {definition of $\vdash$}

$step\,(\phi(\lambda x\,.\,P) \simeq \lambda x\,.\,T$

$\equiv$ {property of substitution}

$step\,(\lambda x\,.\,(\phi\, P)) \simeq \lambda x\,.\,T$

$\equiv$ {definition of $step$}

$\lambda x\,.\,step\,(\phi\, P) \simeq \lambda x\,.\,T$

$\equiv$ {property of $\simeq$}

$step\,(\phi\, P) \simeq T$

$\equiv$ {definition of $\vdash$}

$\phi \vdash P \rightarrow T.$

Since $\lambda x\,.\,P$ does not contain any η-redexes, neither does P, similarly since $\lambda x\,.\,T$ is normal so is T.

A.5. Matching against a λ-abstraction

Let T be a normal expression that is *not* a λ-abstraction. To prove soundness and completeness, we argue:

$\phi \vdash (\lambda x\,.\,P) \rightarrow T$

$\equiv$ {definition of $\vdash$}

$\mathit{step}\,(\phi(\lambda x\,.\,P) \simeq T$

$\equiv$ {property of substitution}

$\mathit{step}\,(\lambda x\,.\,(\phi\,P)) \simeq T$

$\equiv$ {definition of *step*}

$\lambda x\,.\,\mathit{step}\,(\phi\,P) \simeq T$

$\equiv$ {property of $\simeq$}

$\lambda x\,.\,\mathit{step}\,(\phi\,P) \simeq \lambda x\,.\,T\,x$

$\equiv$ {property of $\simeq$}

$\mathit{step}\,(\phi\,P) \simeq T\,x$

$\equiv$ {definition of $\vdash$}

$\phi \vdash P \rightarrow (T\,x).$

Since we know that T is normal and not a λ-abstraction, $T\,x$ is normal. Also $\lambda x\,.\,P$ does not contain any η-redexes, so neither does P.

A.6. Matching against an application

In this case, it only remains to show that the invariant is maintained. Referring to the definition of *resolve*, we note that since $F\,E$ does not contain any η-redexes, neither do F or E. In addition, it follows from the definition of *apps* and the fact that T is normal that T_0 and T_1 are always normal.

A.7. Failure to match

$\phi \vdash P \rightarrow T$

$\equiv$ {property of $\vdash$}

$\mathit{step}'\,(\phi\,P) = T$

$$
\begin{array}{ll}
\Rightarrow & \{\text{definition of } \mathit{step}'\} \\
& (\exists c : \phi P = c \wedge T = c) \\
& \vee\ (\exists x : \phi P = x \wedge T = x) \\
& \vee\ (\exists B : \phi P = \lambda x\,.B) \\
& \vee\ (\exists E_1, E_2 : \phi P = E_1\ E_2) \\
\Rightarrow & \{\text{property of substitution}\} \\
& (\exists p : P = p) \\
& \vee\ (\exists c : P = c \wedge T = c) \\
& \vee\ (\exists x : P = x \wedge T = x) \\
& \vee\ (\exists B : P = \lambda x\,.B) \\
& \vee\ (\exists E_1, E_2 : P = E_1\ E_2).
\end{array}
$$

Since each of these cases has already been covered, we have the completeness property. Since the set we generate is empty, we automatically have soundness.

References

[1] M. Alt, C. Fecht, C. Ferdinand, R. Wilhelm, The TrafoLa-H Subsystem, in: B. Hoffmann, B. Krieg-Brückner (Eds.), Program Development by Specification and Transformation, Lecture Notes in Computer Science, vol. 680, Springer, Berlin, 1993, pp. 539–576.

[2] T. Biggerstaff, Pattern matching for program generation: a user manual, Tech. Report MSR TR-98-55, Mircosoft Research, 1998. Available from URL: [http://www.research.microsoft.com/~tedb/Publications.htm.]

[3] R.S. Bird, The promotion and accumulation strategies in functional programming, ACM Trans. Programming, Languages Systems 6 (4) (1984) 487–504.

[4] R.S. Bird, Introduction to Functional Programming in Haskell, International Series in Computer Science, Prentice-Hall, Englewood Cliffs, NJ, 1998.

[5] R.J. Boulton, A restricted form of higher-order rewriting applied to an HDL semantics, in: J. Hsiang (Ed.), Rewriting Techniques and Applications: 6th Internat. Conf., Lecture Notes in Computer Science, vol. 914, Springer, Berlin, 1995, pp. 309–323.

[6] J. Cai, R. Paige, R.E. Tarjan, More efficient bottom-up multi-pattern matching in trees, Theoret. Comput. Sci. 106 (1) (1992) 21–60.

[7] H. Comon, Higher-order matching and tree automata, in: M. Nielsen, W. Thomas (Eds.), Proc. Conf. on Computer Science Logic, Lecture Notes in Computer Science, vol. 1414, Springer, Berlin, 1997, pp. 157–176.

[8] R.F. Crew, A language for examining abstract syntax trees, in: C. Ramming (Ed.), Proc. Conf. on Domain-Specific Languages, Usenix, 1997.

[9] R. Curien, Z. Qian, H. Shi, Efficient second-order matching, 7th Internat. Conf. on Rewriting Techniques and Applications, Lecture Notes in Computer Science, vol. 1103, Springer, Berlin, 1996, pp. 317–331.

[10] O. de Moor, G. Sittampalam, in: Generic program transformation, Third Internat. Summer School on Advanced Functional Programming, Lecture Notes in Computer Science, Springer, Berlin, 1998.

[11] D.J. Dougherty, Higher-order unification via combinators, Theoret. Comput. Sci. 114 (1993) 273–298.

[12] G. Dowek, A second-order pattern matching algorithm for the cube of typed lambda calculi, in: A. Tarlecki (Ed.), Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, vol. 520, Springer, Berlin, 1991, pp. 151–160.

[13] G. Dowek, Third-order matching is decidable, in: M. Nielsen, W. Thomas (Eds.), Logic in Computer Science (LICS), IEEE, New York, 1992, pp. 2–10.

[14] G. Huet, B. Lang, Proving and applying program transformations expressed with second-order patterns, Acta Inform. 11 (1978) 31–55.

[15] B. Krieg-Brückner, J. Liu, H. Shi, B. Wolff, Towards correct, efficient and reusable transformational developments, in: M. Broy, S. Jähnichen (Eds.), KORSO: Methods, Languages, and Tools for the Construction of Correct Software, Lecture Notes in Computer Science, vol. 1009, Springer, Berlin, 1995, pp. 270–284.

[16] D. Miller, A logic programming language with lambda-abstraction, function variables, and simple unification, J. Logic Comput. 1 (1991) 479–536.

[17] T. Nipkow, Higher-order unification, polymorphism, and subsorts, in: S. Kaplan, M. Okada (Eds.), Proc. 2nd Internat. Workshop on Conditional and Typed Rewriting Systems, Lecture Notes in Computer Science, vol. 516, Springer, Berlin, 1990, pp. 436–447.

[18] T. Nipkow, Functional unification of higher-order patterns, 8th IEEE Symp. on Logic in Computer Science, IEEE Computer Society Press, Silver Spring, MD, 1993, pp. 64–74.

[19] R. Paige, Viewing a program transformation system at work, in: M. Hermenegildo, J. Penjam (Eds.), Joint 6th Internat. Conf. on Programming Language Implementation and Logic Programming, and 4th Internat. Conf. on Algebraic and Logic Programming, Lecture Notes in Computer Science, vol. 844, Springer, Berlin, 1994, pp. 5–24.

[20] S.L. Peyton-Jones, A.L.M. Santos, A transformation-based optimiser for Haskell, Sci. Comput. Programming 32 (1–3) (1998) 3–48.

[21] F. Pfenning, C. Elliott, Higher-order abstract syntax, Proc. SIGPLAN ’88 Conf. on Programming Language Design and Implementation, ACM, New York, 1988, pp. 199–208.

[22] C. Simonyi, Intentional programming: innovation in the legacy age, Presented at IFIP Working group 2.1. Available from URL [http://www.research.microsoft.com/research/ip/], 1996.

[23] D.R. Smith, KIDS: a semiautomatic program development system, IEEE Trans. Software Eng. 16 (9) (1990) 1024–1043.

[24] J. Springintveld, Third-order matching in the presence of type constructors, in: M. Dezani-Ciancaglini, G.D. Plotkin (Eds.), Type Lambda Calculi and Applications, Lecture Notes in Computer Science, vol. 902, Springer, Berlin, 1995, pp. 428–442.

[25] D.A. Wolfram, The Clausal Theory of Types, Cambridge Tracts in Theoretical Computer Science, vol. 21, Cambridge University Press, Cambridge, 1993.