Free Variables and Subexpressions in Higher-Order Meta Logic*

Chuck Liang

Department of Computer Science
Trinity College
300 Summit Street
Hartford, CT 06106-3100, USA
chuck.liang@mail.trincoll.edu

Abstract. This paper addresses the problem of how to represent free variables and subexpressions involving λ-bindings. The aim is to apply what is known as higher-order abstract syntax to higher-order term rewriting systems. Directly applying β-reduction for the purpose of subterm-replacement is incompatible with the requirements of term-rewriting. A new meta-level representation of subterms is developed that will allow term-rewriting systems to be formulated in a higher-order meta logic.

Introduction

Higher-order logic, and specifically the technique of *higher-order abstract syntax* has been shown to be a useful paradigm in the formulation of object-level systems[2]. These range from automated theorem proving to polymorphic type inferencing and to program transformation. Compared to first-order systems, higher-order logic based on the λ-Calculus can represent variables and abstractions in the object theory in a more natural manner. Issues such as the renaming of bound variables are eliminated by α-equivalence classes of the meta logic. Many techniques have been developed on this basis.

The existing techniques, however, are still insufficient for using higher-order logic as a generic framework (and meta-programming language) for dealing with the wide range of problems encountered in representing object-level systems. Many operations required by these systems are seemingly inconsistent with characteristics of the meta logic. In many object-level systems, extracting a subexpression or *subterm* from an expression is a common procedure. In term-rewriting systems in particular, we need to be able to substitute some occurrence of a subterm in an expression with another term. In the λ-calculus, substitution is synonymous with β-reduction. β-reduction alone, however, does not suffice to formulate all aspects of term replacement required in implementing and reasoning about term-rewriting systems. This is especially true when the *object-level*

* This paper appears in " Theorem Proving in Higher Order Logics, 11th International Conference" Springer-Verlag LNCS Vol. 1479. September 1998.

[2] See [17, 4, 10, 5] for background and sample work on higher-order abstract syntax.

rewriting system is itself higher-order. Higher-order rewriting requires substitution to be regarded in the broadest sense, one in which the scopes of bound variables are not necessarily respected. We are required to consider x as, in some sense, a *subterm* of $\lambda x.x$. This is the central problem we shall address here.

This paper also complements the work of Felty [3], which showed how higher-order term rewriting can be implemented in a logic programming language supporting higher-order abstract syntax. The techniques presented in this previous work does not address the problem of subterms with free variables directly, and are therefore limited in their capacity as a *meta-theory* for reasoning about various aspects of higher-order writing systems (Nipkow's higher-order critical pairs [15] in particular). We shall develop a technique that is compatible with those of [3], but which allows for full flexibility in reasoning about individual subterms.

1 Substitutions and Contexts in Higher-Order Rewriting

In the λ-calculus, β-reduction alone is incompatible with substitution required in term-rewriting. First of all, β-reduction will universally replace all occurrences of a variable, whereas in term rewriting we may only wish to replace a particular occurrence. This can be solved by using λ-abstraction to represent a *context* that identifies the precise location in the term structure where the substitution is to take place.

Definition 1. A **context** is a term of the form $\lambda c.D$ where c appears free exactly once in D.

For example, given a term $(f\ a\ (f\ a\ b))$, the "context" $C = \lambda c.(f\ a\ (f\ c\ b))$ identifies the second occurrence of a as the one to be replaced, so that the β-reduction of $(C\ S)$ will replace that occurrence of a with the term S. The replacement of subterms using this higher-order representation is therefore a three-place relation involving a context in addition to the original and final terms.

There remains, however, many problems with such a formulation. The first of which is control and computational feasibility. Consider again term-rewriting. Given an arbitrary term T and a rewrite rule of the form $lhs \longrightarrow rhs$, we need to be able to determine what, if any subterms of T, along with their contexts, can be rewritten using this rule. That is, what context C and subterm S exist such that S is an instance of lhs and $(C\ S) = T$. One may be tempted to specify this relation using Huet's formulation of higher-order unification [7]. Such a specification, however, is computationally unacceptable since higher-order unification is undecidable and the set of unifiers for a problem is generally too large. That is, only a few of the multitude of unifiers are acceptable as correct solutions for S and C, because we require C to be a context.

An even more serious problem occurs when the object-theory itself may involve λ-terms, and this forms the major focus of this paper. In [15], Nipkow extended the critical pair formulation used in the well-known Knuth-Bendix

Completion procedure to a rewriting system involving (a restricted class of) λ-terms. In this system, subterms that are subject to rewrite rules may contain bound variables. For example, assume that g is an (object-level) unary operator and f some binary operator. Given a sample term $T = \lambda x.(f\ (\lambda x.x)\ \lambda y.(g\ x))$, we would like to consider $(g\ x)$ as a subterm of T. This is because it is valid to apply a rewrite rule, such as $(g\ Z) \longrightarrow (h\ Z)$, to $(g\ x)$, yielding the term $(h\ x)$. We also need to graft $(h\ x)$ *back* into the context for $(g\ x)$ in T to form the final term $\lambda x.(f\ (\lambda x.x)\ \lambda y.(h\ x))$. But in both $(h\ x)$ and $(g\ x)$ x is a free variable. This process therefore requires the *recapturing* of the free variable x by λ-abstraction. But capturing a *free* variable under λ-abstraction is in direct opposition to substitution as β-reduction. Furthermore, we also must ensure that x is captured under the original abstraction that bound it in T, namely the outermost λx, and not the inner λy. The names of bound variables in T are now critical. It may therefore appear that α-equivalence classes, as well as β-reduction, are both inconsistent with higher-order term-rewriting systems. Yet $\alpha\beta$-equivalence forms the basis of higher-order abstract syntax. This apparent inconsistency induces the abandonment of higher-order logic as meta-language. We would need to use an essentially *first* order theory as the meta logic of a *higher* order system. This approach was in fact adopted by Nipkow in his formulation of higher-order critical pairs.

The inconsistency described above is only one of many which may arise in meta-programming. Related to the problem of substitution is the issue of how to represent object-level variables in the meta-logic. The naive approach of using free variables in the meta-logic to represent free variables in the object-logic, known as the "non-ground representation," is insufficient for all but the most trivial systems[3]. In order for a meta-logic to properly represent the object-theory, it can be enriched with new features that will allow it to address certain issues directly. Alternatively, we can try to preserve the meta-logic and address the same issue by *changing the representation of the object-theory* in the meta-logic. In [9] for example, it was shown how meta-level bound variables can be used to represent free type variables in an object-level typing system. The problem of determining subterms is similar in that meta-level substitution (i.e, β-reduction) must not be used to represent subterm replacement at the object-level. In order to preserve the λ-calculus as meta-theory, we must first adopt a different representation of a "subterm" of an expression in the λ-calculus, and then redefine the subterm-replacement relation using this new representation. α-equivalence classes can be preserved (with all the benefits of adopting them) using this new representation. β-reduction will still be used to implement term-replacement, but in a restricted and indirect manner.

2 Term Trees with λ-expressions

Term-rewriting requires the notion of "subterm" to be regarded in the widest possible sense. In [15], Nipkow showed that the traditional treatment of first-

[3] This is well documented (see [6]).

order term trees can be extended to λ-terms by simply treating each occurrence of λx as a unary first-order term constructor. Each subterm of an expression or term is associated with a unique sequence representing the *position* of the subterm in the tree. The empty sequence ϵ represents the position of the "root" of the tree, or the term itself. Each member of the sequence is an integer starting from 1 to the maximum branching factor (or the maximum arity of the constructors) of the expression. We will use $T_{@}p$ to represent the subterm at position p in T. Inductively, $T_{@}\epsilon = T$ and for any n-ary operator op, $(op\ X_1, \ldots, X_n)_{@}i.p = X_{i@}p$. For example, position 1 represents the leftmost subtree of the root. Position 1.3 represents the third descendent of the first subtree of the root, and so on. The notation $T[S]/_{@}p$ is used to represent T with S replacing the subterm at position p. Each λ-binding λx represents one node in the term tree (the x is not a separate node). If we confine ourselves to pure λ-terms with only application and abstraction, then positions are sequences over $\{1, 2\}$. The term tree for $\lambda x.(f\ (\lambda x.x)\ \lambda y.(g\ x))$ is given in Figure 1. The subterm $\lambda x.x$ is at position 1.1.

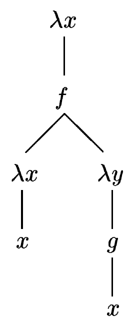

Fig. 1. Term tree for $\lambda x.(f\ (\lambda x.x)\ \lambda y.(g\ x))$

Under this formulation, subexpressions with bound variables can be extracted by "pruning" off a subtree. Expressions with free variables can be inserted or "grafted" into a tree - *allowing* for the capture of free variables under λ-bindings. α-equivalence is therefore not preserved under such an interpretation. In particular, x can be considered a subterm of (for example) $\lambda x \lambda y.(f\ y\ x)$ because $x = (\lambda x \lambda y.(f\ y\ x))_{@}1.1.2$. We shall call x a *free subterm* when considered in this manner.

Definition 2. S is a **free subterm** of T if for some sequence p, $T_{@}p = S$.

We emphasize that x is a free subterm of $\lambda x.x$ but *not* of $\lambda y.y$ although the two λ-expressions are α-equivalent. Nipkow used free subterms and term trees in formulating higher-order critical pairs. The clarity of this formulation suffers from the need to explicitly maintain consistent names for free and bound variables - something that a higher-order meta logic should provide for automatically. A reformulation of the representation of subterms and substitution is required - one that preserves α-equivalence classes and yet respects the first-order notion of replacing subterms in a term tree. This formulation is necessarily different from traditional β-reduction because of reasons stated above.

3 Subterm Redefined

The meta-language we shall use in the reformulation is the λ-Calculus with α-equivalence. Concerning β-reduction, only a restricted form is required. This is the "β_0" reduction of Miller [12]. A β_0-redex is a β-redex of the form $(\lambda x.A)y$ where y is a λ-bound variable - or equivalently, an arbitrary variable not appearing free elsewhere[4]. Given α-convertibility, β_0-reduction can be simplified to $(\lambda x.A)x =_{\beta_0} A$. We shall write E^{β_0} to represent the β_0-reduced form of E. β_0-reduction is terminating since no new redeces can be introduced. The meta language need not involve types (though our result will usually be applied to object-level $\beta\eta$-long normal forms in practice).

Our general approach can be summarized by the following:

> *The* free *variables in a free subterm should be represented by* bound *variables at the meta-level.*

If α-equivalence is to be preserved, some mechanism must be applied to associate the free variables in a free subterm with the bound variables in the parent term. Consider the term represented by the tree in Figure 1. The free occurrence of x in the free subterm $(g\ x)$ must be associated with the outermost λ-abstraction, and not with either of the two other λ-abstractions occurring in the parent term. This relationship between variables in free subterms and the scopes of λ-abstractions they fall under can be cleanly represented by λ-abstraction itself (at the meta-level). For each λ-bound variable in the parent term, we *include* the λ-abstraction in representing the free subterm. The free subterm $(g\ x)$ of the term in Figure 1 will be represented by (the α-equivalence class of) $\lambda x \lambda y.(g\ x)$.

We shall call $\lambda x \lambda y.(g\ x)$ a *Λ-subterm* of its parent term. The ordering of the λ-abstraction prefix of a Λ-subterm, *not* the names of bound variables, preserves the relationship between variables in the subterm and λ-bindings in the parent term.

In the following definition of Λ-subterms we do *not* distinguish meta-level λ-abstraction from object-level λ-abstraction. Although in practice this distinction is usually made, the result we prove is more general without the distinction. We show how to add the distinction in discussing the implementation of this technique in Section 5.

Definition 3. (Λ-subterms)
The three-place relation $\Lambda_{subterm}$ is inductively defined over the third argument (representing the context) on all terms A, B, D, X and $X_1, \ldots, X_n$ as follows:

1. $\Lambda_{subterm}\ X\ X\ \lambda c.c.$

[4] See [13] for the formulation of this equivalence.

2. $\Lambda_{subterm}\ A\ (op\ X_1 \ldots X_i \ldots X_n)\ \lambda c.(op\ X_1 \ldots (D\ c)^{\beta_0} \ldots X_n)$ if and only if $\Lambda_{subterm}\ A\ X_i\ D$, where *op* is any n-ary operator symbol (including application) that is not of the form λx.[5]

3. $\Lambda_{subterm}\ A\ B\ \lambda c.\lambda x.(D\ x\ c)^{\beta_0}$ if and only if $\Lambda_{subterm}\ (A\ x)^{\beta_0}\ (B\ x)^{\beta_0}\ (D\ x)^{\beta_0}$ where x is an arbitrary variable not appearing free in A, B or $\lambda x.(D\ x)^{\beta_0}$.

The intuitive meaning of $\Lambda_{subterm}\ S\ T\ C$ is that S is a Λ-subterm of T under the context C. The context replaces the position sequence in first order term trees. We say that *S is a Λ-subterm of T* if $\Lambda_{subterm}\ S\ T\ C$ holds for some context C[6].

3.1 Examples

Some sample derivations of Λ-subterms are provided below:

1. The first example does not involve λ-bindings:
$$\Lambda_{subterm}\ a\ \ (f\ (g\ a)\ b)\ \ \lambda c.(f\ (g\ c)\ b)$$
holds if, by the second clause of the definition of $\Lambda_{subterm}$, with $i = 1$,
$$\Lambda_{subterm}\ a\ \ (g\ a)\ \ \lambda c.(g\ c)$$
holds. Again using clause two, this holds if
$$\Lambda_{subterm}\ a\ a\ \lambda c.c$$
holds. But $\Lambda_{subterm}\ a\ a\ \lambda c.c$ holds by clause one.

2. For the second example,
$$\Lambda_{subterm}\ \lambda x\lambda y.(g\ x)\ \ \lambda x.(f\ (\lambda x.x)\ \lambda y.(g\ x))\ \ \lambda c.\lambda x.(f\ (\lambda x.x)\ \lambda y.c)$$
holds since (by clause three with $D = \lambda x\lambda c.(f\ (\lambda x.x)\ \lambda y.c)$ and arbitrary variable z)
$$\Lambda_{subterm}\ \lambda y.(g\ z)\ \ (f\ (\lambda x.x)\ \lambda y.(g\ z))\ \ \lambda c.(f\ (\lambda x.x)\ \lambda y.c)$$
holds since (by clause two)
$$\Lambda_{subterm}\ \lambda y.(g\ z)\ \ \lambda y.(g\ z)\ \ \lambda c.\lambda y.c$$
holds since (by clause three with $D = \lambda y\lambda c.c$ and arbitrary variable x)
$$\Lambda_{subterm}\ (g\ z)\ (g\ z)\ \lambda c.c$$
holds by clause one. The reader is invited to verify that $\lambda y\lambda x.(g\ x)$ would be a Λ-subterm of $\lambda x.(f\ (\lambda x.x)\ \lambda y.(g\ y))$, but *not* of $\lambda x.(f\ (\lambda x.x)\ \lambda y.(g\ x))$.

[5] For pure λ-terms, *op* would just be application and this case would be for $(X_1\ X_2)$. We generalize this case to arbitrary operators in order to better connect the definition with arbitrary term trees.

[6] $\Lambda_{subterm}$ is defined so that C must be a context if $\Lambda_{subterm}\ S\ T\ C$ holds (see Theorem 6).

3. The final example demonstrates the generality of the $\Lambda_{subterm}$ relation. It is possible to consider y as a free subterm of $\lambda x \lambda y.(y\ x)$ by treating *op* implicitly as the application (*app*) operation of the λ-calculus. $(y\ x)$ can then be read as $(app\ y\ x)$. As a *Λ-subterm,* y becomes $\lambda x \lambda y.y$:

$$\Lambda_{subterm}\ \lambda x \lambda y.y\ \ \lambda x \lambda y.(y\ x)\ \ \lambda c.\lambda x \lambda y.(c\ x)$$

holds since (by clause three with $D = \lambda x \lambda c \lambda y.(c\ x)$ and arbitrary variable u)

$$\Lambda_{subterm}\ \lambda y.y\ \ \lambda y.(y\ u)\ \ \lambda c.\lambda y.(c\ u)$$

holds since (again by clause three with $D = \lambda y \lambda c.(c\ u)$ and arbitrary variable v)

$$\Lambda_{subterm}\ v\ (v\ u)\ \lambda c.(c\ u)$$

holds since (by clause two)

$$\Lambda_{subterm}\ v\ v\ \lambda c.c$$

holds by clause one.

It is worthwhile to note that, as in the second example, the $\Lambda_{subterm}$ relation will also hold under different contexts. Specifically, $\lambda x \lambda y.(g\ x)$ is also a Λ-subterm of $\lambda x.(f\ (\lambda x.x)\ \lambda y.(g\ x))$ under the context $\lambda c.\lambda x.(f\ (\lambda x.x)\ c)$. In this case, $\lambda x \lambda y.(g\ x)$ would correspond to the free subterm $\lambda y.(g\ x)$ and not $(g\ x)$. This ambiguity as to how to interpret the λ-bindings is resolved by attaching the precise context to the $\Lambda_{subterm}$ relation. It can also be resolved by separating meta-level λ-abstraction from object-level λ-abstraction, which we do in Section 5.

3.2 Rewriting with Λ-subterms

The $\Lambda_{subterm}$ relation can be used for term-rewriting because we can now safely extract the subterm to be replaced. A rewrite rule can be applied by "looking inside" the λ-bindings for the subterm to be replaced. That is, we can define the application of rewrite rules in higher-order abstract syntax as follows:

Definition 4. Given a rewrite rule $R = lhs \longrightarrow rhs$, we say that B **replaces** A under R if either

1. $A \longrightarrow B$ is an instance of $lhs \longrightarrow rhs$, or
2. $(B\ x)^{\beta_0}$ replaces $(A\ x)^{\beta_0}$ under R for an arbitrary variable x not appearing free in A, B or R.

Thus $\lambda u \lambda v.(h\ u)$ *replaces* $\lambda x \lambda y.(g\ x)$ under the rule $(g\ Z) \longrightarrow (h\ Z)$ (where Z is a free variable). That is, the "replaces" relation rewrites a Λ-subterm to a term that preserves the λ-abstraction prefix. The rewritten term can be grafted into a context to form a new term. We can define rewriting for general λ-terms as follows:

Definition 5. A **rewrites** to B under rule R if:

1. for some term S_a, $\Lambda_{subterm}\ S_a\ A\ C$ holds for some (context) C.
2. for some term S_b, S_b replaces S_a under R.
3. $\Lambda_{subterm}\ S_b\ B\ C$ holds.

That is, if S_a is a Λ-subterm of A and S_b is a Λ-subterm of B under the same context, and such that S_b replaces S_a under R, then A rewrites to B under R. Thus $\lambda x.(f\ (\lambda x.x)\ \lambda y.(g\ x))$ rewrites to $\lambda u.(f\ (\lambda x.x)\ \lambda v.(h\ u))$ under the rule $(g\ Z) \longrightarrow (h\ Z)$ because:

1. $\Lambda_{subterm}\ \lambda u \lambda v.(g\ u)\ \ \lambda x.(f\ (\lambda x.x)\ \lambda y.(g\ x))\ \ \lambda c.\lambda x.(f\ (\lambda x.x)\ \lambda y.c)$ holds
2. $\lambda u \lambda v.(h\ u)$ replaces $\lambda x \lambda y.(g\ x)$ under $(g\ Z) \longrightarrow (h\ Z)$.
3. $\Lambda_{subterm}\ \lambda u \lambda v.(h\ u)\ \ \lambda u.(f\ (\lambda x.x)\ \lambda v.(h\ u))\ \ \lambda c.\lambda x.(f\ (\lambda x.x)\ \lambda y.c)$ holds.

The reader is invited to verify that this example is preserved under α-equivalent classes of terms.

The technique used in Definition 4 of looking inside λ-bindings to find a matching instance of a rewrite rule is similar to how higher-order rewriting is formulated by Felty in [3]. Instead of extracting a subterm together with its context explicitly, Felty's method essentially *combines* the stripping away of λ-bindings with the finding of a subterm that can be rewritten. While this approach suffices for the *implementation* of a higher-order rewriting system, it can not be used for the meta-level *reasoning* about properties of such systems. Nipkow's higher-order critical pairs provide the best example, for it requires a greater degree of flexibility in reasoning about individual subterms and their contexts. Consider the rewrite rules $\lambda y.(h\ (g\ (f\ y))) \longrightarrow a$ and $(g\ X) \longrightarrow (b\ X)$. A critical pair is formed by the terms a and $\lambda y.(h\ (b\ (f\ y)))$. By Nipkow's definitions this pair is formed by *extracting* the free subterm $(g\ (f\ y))$ together with its position and *grafting* the term $(b\ (f\ y))$ into the same position. Notice that the variable y becomes *recaptured* under the original λ-binding. This process can not be formulated using only the "tacticals" of [3]. The subterm and context must be extracted explicitly, but in a way that preserves α-equivalence classes. As shown above, both the extracting and grafting of subterms can be formulated using the $\Lambda_{subterm}$ relation.

4 Correctness Theorem

In this section we show that this higher-order formulation of subterms is correct in the sense that it is consistent with the term-tree representation.

Theorem 6. *For any terms S, T, and C, $\Lambda_{subterm}\ S\ T\ C$ holds if and only if:*

1. *$S =_\alpha \lambda x_1 \ldots \lambda x_n.S'$ for some $n \geq 0$ (if $n = 0$ then $S =_\alpha S'$), such that*
2. *for some position p, $T_@p = S'$ where*
3. *$x_1, \ldots, x_n$ are all the λ-bound variables in T that includes $T_@p$ in their scope, and such that if $i < j$ then λx_i includes λx_j in its scope, and*

4. C is a context, and $(C\ x)^{\beta_0} =_\alpha T[x]/_{@}p$ for an arbitrary variable x not occurring in S, T or C.

Proof: both the forward and reverse directions are proved by induction on the structure of the context C:

Forward Direction:

Base Case: if $C = \lambda c.c$: assume $\Lambda_{subterm}\ S\ S\ \lambda c.c$ holds. Here, let $S' = S$, $p = \epsilon$ and so the first and third conditions holds vacuouly. If $p = \epsilon$ then $S_{@}\epsilon = S = S'$ and the second condition is satisfied. Finally, for an arbitrary x, $(\lambda c.c)x =_{\beta_0} x$, and $S[x]/_{@}\epsilon = x$ and so the fourth condition is also satisfied.

Inductive Case for $C = \lambda c.op\ X_1 \ldots (D\ c)^{\beta_0} \ldots X_n$: T must have the form

$$op\ X_1 \ldots X_i \ldots X_n$$

By definition of $\Lambda_{subterm}$, $\Lambda_{subterm}\ S\ X_i\ D$ holds, which by inductive hypotheses yields:

1. $S =_\alpha \lambda x_1 \ldots \lambda x_n.S'$
2. there's a sequence p such that $X_{i@}p = S'$
3. $x_1, \ldots, x_n$ are all bound variables in X_i that contain S' in their scopes are for $i < j$ λx_j is in the scope of λx_i.
4. D is a context, and for an arbitrary x, $(D\ x)^{\beta_0} =_\alpha X_i[x]/_{@}p$

Now we need to show that each condition holds for the larger context:

1. S remain the same and $S =_\alpha \lambda x_1 \ldots \lambda x_n.S'$
2. Let $p' = i.p$. Then since $T_{@}i = X_i$, $T_{@}p' = X_{i@}p = S'$.
3. Since $x_1, \ldots, x_n$ satisfies the third condition with respect to S' in X_i, then $x_1, \ldots, x_n$ also satisfies the condition with respect to S' in T.
4. for an arbitrary x,

$$(\lambda c.op\ X_1 \ldots (D\ c)^{\beta_0} \ldots X_n)x =_{\beta_0} op\ X_1 \ldots (D\ x)^{\beta_0} \ldots X_n$$

   By part 4 of the inductive hypothesis,

$$op\ X_1 \ldots (D\ x)^{\beta_0} \ldots X_n = op\ X_1 \ldots X_i[x]/_{@}p \ldots X_n = T[x]_{@}i.p$$

   Finally, C is a context since D is a context.

Inductive Case for $C = \lambda c.\lambda x.(D\ x\ c)$: assume that $\Lambda_{subterm}\ S\ T\ C$ holds. Then by definition of $\Lambda_{subterm}$,

$$\Lambda_{subterm}\ (S\ x)^{\beta_0}\ (T\ x)^{\beta_0}\ (D\ x)^{\beta_0}$$

holds. Application of the inductive hypothesis yields:

1. $(S\ x)^{\beta_0} =_\alpha \lambda x_1 \ldots \lambda x_n.S'$
2. for some sequence p, $(T\ x)^{\beta_0}{}_{@}p = S'$

3. $x_1, \ldots, x_n$ are all bound variables in $(T\ x)^{\beta_0}$ that contain S' in their scopes are for $i < j$ λx_j is in the scope of λx_i.
4. $(D\ x)^{\beta_0}$ is a context and for an arbitrary y, $(D\ x\ y)^{\beta_0} =_\alpha (T\ x)^{\beta_0}[y]/_@p$

Again we need to show that these four conditions are preserved:

1. Since $(S\ x)^{\beta_0} =_\alpha \lambda x_1 \ldots \lambda x_n.S'$, $S =_\alpha \lambda x \lambda x_1 \ldots \lambda x_n.S'$
2. Let $p' = 1.p$. T can be written as $\lambda x.(T\ x)^{\beta_0}$. Since $(T\ x)^{\beta_0}{}_@p = S'$, we have
$$\lambda x.(T\ x)^{\beta_0}{}_@p' = S'.$$
3. Since $x_1, \ldots, x_n$ satisfies the third condition with respect to S' in $(T\ x)^{\beta_0}$, $x, x_1, \ldots, x_n$ satisfies the condition with respect to S' in $\lambda x.(T\ x)^{\beta_0}$.
4. For an arbitrary y, $(\lambda c.\lambda x.(D\ x\ c)^{\beta_0})y =_{\beta_0} \lambda x.(D\ x\ y)^{\beta_0}$. By inductive hypothesis, $(D\ x\ y)^{\beta_0} =_\alpha (T\ x)^{\beta_0}[y]/_@p$, so $\lambda x.(D\ x\ y)^{\beta_0} =_\alpha \lambda x.[(T\ x)^{\beta_0}[y]/_@p]$. But $\lambda x.[(T\ x)^{\beta_0}[y]/_@p] = \lambda x.(T\ x)^{\beta_0}[y]/_@p'$. Finally, $\lambda c.\lambda x.(D\ x\ c)^{\beta_0}$ is a context since $(D\ x)^{\beta_0}$ is a context implies that c occurs exactly once in $\lambda x.(D\ x\ c)^{\beta_0}$.

Reverse Direction:

Base Case: assuming $C =_\alpha \lambda c.c$ (since C must be a context) and the four conditions of the theorem holds for C. By the fourth condition, $T[x]\_@p = (C\ x)^{\beta_0} = x$. We can derive from this that $p = \epsilon$. Now by the second condition, $T_@p = T = S'$ where (by condition one) $S =_\alpha \lambda x_1 \ldots \lambda x_n.S'$. Now by condition three, $n = 0$ ($x_1, \ldots, x_n$ is an empty sequence) since $x_1, \ldots, x_n$ are variables in T that contain T, which is equal ot $T_@p$, in their scopes. Thus we have $S = S' = T$, and
$$\Lambda_{subterm}\ S\ S\ C$$
holds by definition.

Inductive Case for $C = \lambda c.op\ X_1 \ldots X_m$: since by the fourth condition c is a context, c appears in exactly one X_i among $X_1, \ldots, X_m$. Let $D = \lambda c.X_i$. D is therefore also a context. We need to show that all four conditions holds for D in order to apply the inductive hypothesis. We can write
$$C = \lambda c.op\ X_1 \ldots (D\ c)^{\beta_0} \ldots X_m.$$
Condition four also yields
$$T[x]/_@p =_\alpha (C\ x)^{\beta_0} = op\ X_1 \ldots (D\ x)^{\beta_0} \ldots X_m$$
for some arbitrary variable x. Since x can occur only in $(D\ x)^{\beta_0}$, $p = i.p_2$ for some sequence p_2. Let $T_2 = T_@i$. Then $T[x]/_@p = op\ X_1 \ldots T_2[x]/_@p_2 \ldots X_m$. Now we have $(D\ x)^{\beta_0} =_\alpha T_2[x]/_@p_2$ (condition four is satisfied for D). Since by condition two $T_@p = S'$, $T_{2@}p_2 = S'$. Conditions one and three gives us that $S =_\alpha \lambda x_1 \ldots \lambda x_n.S'$ where $x_1, \ldots, x_n$ are all bound variables in T containing S' in their scope. But this also means that $x_1, \ldots, x_n$ are all bound variables in T_2

containing S' in their scope (there are no more λ-abstractions outside of T_2). Now we are read to apply the inductive hypothesis, which yields that

$$\Lambda_{subterm}\ S\ T_2\ D$$

holds. Then by definition of the $\Lambda_{subterm}$ relation, $\Lambda_{subterm}\ S\ T\ C$ also holds.

Inductive Case for $C = \lambda c.\lambda x.D'$: let $D = \lambda x.\lambda c.D'$. Then $C = \lambda c.\lambda x.(D\ x\ c)^{\beta_0}$ We are allowed to assume that the four conditions hold for S, T and C. We need to show that they also hold for $(S\ x)^{\beta_0}$ $(T\ x)^{\beta_0}$ and $(D\ x)^{\beta_0}$ in order to apply the inductive hypothesis. Since C is a context, $(D\ x)^{\beta_0}$ must be a context. Condition four also yields $T[y]/_{@}p =_\alpha (C\ y)^{\beta_0} = \lambda x.(D\ x\ y)^{\beta_0}$ for some arbitrary variable y. This implies $p = 1.p_2$ for some p_2. Since $T[y]/_{@}p =_\alpha \lambda x.(D\ x\ y)^{\beta_0}$. T is an abstraction of the form $\lambda x.T_{@}1$, and in particular $(T\ x)^{\beta_0} = T_{@}1$. But then

$$\lambda x.(D\ x\ y)^{\beta_0} =_\alpha T[y]/_{@}p = \lambda x.T_{@}1[y]/_{@}p_2.$$

This implies that

$$(D\ x\ y)^{\beta_0} =_\alpha T_{@}1[y]/_{@}p_2 = (T\ x)^{\beta_0}[y]/_{@}p_2$$

And so the fourth condition is satisfied for $(T\ x)^{\beta_0}$. By condition two on $T_{@}p$,

$$(T\ x)^{\beta_0}{}_{@}p_2 = T_{@}1_{@}p_2 = T_{@}p = S'$$

so condition two is also satisfied for $(T\ x)^{\beta_0}$. If $S =_\alpha \lambda x_1 \ldots \lambda x_n.S'$ and $x_1, \ldots, x_n$ statisfies condition three, then $x_1 = x$ since $T = \lambda x.T_{@}1$. This means

$$(S\ x)^{\beta_0} =_\alpha \lambda x_2 \ldots \lambda x_n.S'$$

and $x_2, \ldots x_n$ satisfies condition three with respect to $(T\ x)^{\beta_0}$ since $T_{@}1 = (T\ x)^{\beta_0}$. We can now apply the inductive hypothesis on $(S\ x)^{\beta_0}$, $(T\ x)^{\beta_0}$ and $(D\ x)^{\beta_0}$ so that

$$\Lambda_{subterm}\ (S\ x)^{\beta_0}\ (T\ x)^{\beta_0}\ (D\ x)^{\beta_0}$$

holds. So by definition of the $\Lambda_{subterm}$ relation we have that

$$\Lambda_{subterm}\ S\ T\ C$$

also holds.

□

We summarize the result of this theorem in the following corollary, which is directly derivable. It formalizes the relationship between Λ-subterms and free subterms:

Corollary 7. *For all terms S and T:*

1. *if S is a Λ-subterm of T then (for some $n \geq 0$) $S =_\alpha \lambda x_1 \ldots \lambda x_n.S'$ such that S' is a free subterm of T.*

2. *if S is a free subterm of T and $x_1, \ldots, x_n$ are all the bound variables in T that contain a common occurrence of S in their scope, and such that for $i < j$ λx_j is in the scope of λx_i, then $\lambda x_1 \ldots \lambda x_n.S$ is a Λ-subterm of T.*

5 Declarative Implementation

The inductive definition of $\Lambda_{subterm}$ suggests that, in addition to the λ-calculus, the meta-logic must support some form of logical inference, such as natural-deduction style theorem proving. Many systems, such as Coq [1, 2] and Isabelle [16], meet this criteria. Furthermore, to use the $\Lambda_{subterm}$ relation as a *meta-programming* device requires a system that involves a form of automated *proof search.* A higher-order logic programming interpreter in the sense of [14] meets this criteria. In particular, the $\Lambda_{subterm}$ relation has a direct formulation in the logic programming language L_λ [12], which is the subset of the better known λProlog language [11] that involves only *higher-order patterns*. A higher-order pattern is a term in β-normal form where in every occurrence of subterms of the form $(F\ X_1 \ldots X_n)$ with F a free variable, $X_1, \ldots, X_n$ must be a distinct list of bound variables. In other words, solutions to higher-order patterns can only involve β_0-redeces. Unification of higher-order patterns is decidable and yields most-general unifiers. Furthermore, L_λ supports negative occurrences of universal quantifiers. The operational interpretation of a logic programming query of the form $\forall x A$ is to prove A with a fresh variable x not occurring elsewhere. L_λ is the simplest known language capable of supporting meta-programming in higher-order abstract syntax. A formulation in L_λ is therefore also a formulation in a wide variety of systems in which L_λ is embedded.

It is appropriate in describing the implementation of $\Lambda_{subterm}$ in L_λ to separate the meta-language from the object-language. Object-level application and λ-abstraction can be represented at the meta-level with a pair of higher order constants *app* and *abs* respectively. Although the meta-language need not be typed, it is helpful to think of *app* as having type $\tau \to \tau \to \tau$ and *abs* as having type $(\tau \to \tau) \to \tau$, where τ is the type of object-level expressions. The object-level term $\lambda x.(x\ y)$ is represented as $(abs\ \lambda x.(app\ x\ y))$ at the meta-level. We also need another constant in order to eliminate the ambiguity mentioned in Section 3.1. The outermost λ-bound variables of Λ-subterms intuitively represent *free* variables at the object level. We therefore introduce the constant fv (of type $(\tau \to \tau) \to \tau$) to "label" a meta-level λ-bound variable as representing an object-level free variable. Λ-subterms can be modified to use fv-abstractions. A Λ-subterm such as $(fv\ \lambda x.x)$ is a Λ-subterm of $(abs\ \lambda x.x)$ under *only* the context $\lambda c.(abs\ \lambda x.c)$, and not $\lambda c.c$.

Figure 2 contains the logic programming concrete-syntax[7] reformulation of the $\Lambda_{subterm}$ relation for object-level λ-terms (*app* and *abs* terms). The symbol `pi` represents universal quantification and `x\e` represents $\lambda x.e$. As is common in logic programming, upper-case letters represent universally quantified variables over the entire clause.

The closure under the `lsubterm` clauses represents the $\Lambda_{subterm}$ relation with separated meta- and object-language. As a logic program, these clauses can be used to implement a higher-order term rewriting system. Given a term τ, the query `lsubterm S` τ `C` (where `S` and `C` are free logic variables) will return a

[7] This syntax is actually for λProlog, which directly embeds L_λ.

```
lsubterm X X (c\c).
lsubterm S (app A B) (c\(app (D c) B) :- lsubterm S A D.
lsubterm S (app A B) (c\(app A (D c)) :- lsubterm S B D.
lsubterm (fv S) (abs T) (c\(abs x\(D x c))) :-
                        pi x\(lsubterm (S x) (T x) (D x)).
```

Fig. 2. $\Lambda_{subterm}$ as logic program clauses

solution instantiating S and C such that S is a Λ-subterm of τ under context C. Conversely, given a context γ and a term σ, solving `lsubterm` σ T γ will, if possible, instantiate T as the term with σ "grafted" into the position indicated by context γ. Finally, given σ and τ and with C as a free logic variable, the query `lsubterm` σ τ C will succeed if and only if σ is *some* Λ-subterm of τ.

A slightly modified form of the `lsubterm` clauses (among other techniques out of scope here) was used in giving a declarative implementation of Nipkow's formulation of higher-order critical pairs. This implementation is described in [8].

6 Conclusion

Higher-order term rewriting systems require a degree of flexibility in reasoning about free variables and subterms. This flexibility, however, is seemingly incompatible with substitution as β-reduction in the λ-calculus, which forms the basis of many meta-theoretic frameworks such as theorem provers and logic programming languages. Term trees provide an unsatisfactory meta-level representation of λ-terms because α-equivalence classes are lost.

We have shown that the λ-calculus can be preserved as the basis of a meta-theory for reasoning about higher-order term rewriting systems. Substitution can be reformulated by replacing β-reduction with the simpler β_0-reduction combined with the ability to deduce $\Lambda_{subterm}$ relations. This combination exists in many systems. The simplified logic programming language L_λ in particular can give a direct implementation of Λ-subterms.

For future work, we hope to apply the Λ-subterm concept to more generic problems than higher-order term rewriting systems.

Acknowledgments

The author wishes to thank Dale Miller and Amy Felty for helpful discussions.

References

1. Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

2. Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138. Springer-Verlag Lecture Notes in Computer Science, April 1995.
3. Amy Felty. A logic programming approach to implementing higher-order term rewriting. In Lars-Henrik Eriksson, Lars Hallnas, and Peter Schroeder-Heister, editors, *Second International Workshop on Extensions to Logic Programming*, pages 135–161. Springer-Verlag, 1992. Volume 596 of Lecture Notes in Artificial Intelligence.
4. Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.
5. John Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, April 1993.
6. P. M. Hill and J. G. Gallagher. Meta-programming in logic programming. Technical Report Report 94.22, University of Leeds, hill@scs.leeds.ac.uk, August 1994. To appear in Vol. 5 of the *Handbook of Logic in Artificial Intelligence and Logic Programming,* Oxford University Press.
7. Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
8. Chuck Liang. *Substitution, Unification and Generalization in Meta-Logic.* PhD thesis, University of Pennsylvania, September 1995.
9. Chuck Liang. Let-polymorphism and eager type schemes. In *TAPSOFT '97: Theory and Practice of Software Development*, pages 490–501. Springer Verlag LNCS Vol. 1214, 1997.
10. R. McDowell and D. Miller. A logic for reasoning with higher-order abstract syntax. In *Symposium on Logic in Computer Science.* IEEE, 1997.
11. Dale Miller. Abstractions in logic programming. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.
12. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
13. Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, pages 321–358, 1992.
14. Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
15. Tobias Nipkow. Higher-order critical pairs. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 342–349. IEEE, July 1991.
16. Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, September 1989.
17. Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

This article was processed using the LaTeX macro package with LLNCS style