Discovering Needed Reductions Using Type Theory [*]

Philippa Gardner [†]
University of Edinburgh

Abstract

The identification of the *needed redexes* in a term is an undecidable problem. We introduce a (partially decidable) type assignment system, which distinguishes certain redexes called the *allowable* redexes. For a well-typed term e, allowable redexes are needed redexes. In addition, with principal typing, *all* the needed redexes of a normalisable term are allowable. Using these results, we can identify all the needed reductions of a principally typed normalisable term. Possible applications of these results include strictness and sharing analysis for functional programming languages, and a reduction strategy for well-typed terms which satisfies Lévy's notion of optimal reduction.

1 Introduction

Barendregt et al. [2] show that the identification of the needed redexes in a λ-term is an undecidable problem. A redex r in λ-term e is *needed* if a residual of r is contracted in every reduction of e to normal form. For example, in the term $(\lambda x.(\lambda y.z)x)(Ie)$, the redexes $(\lambda y.z)x$ and $(\lambda x.(\lambda y.z)x)(Ie)$ are needed, whereas the redex (Ie) is not. We introduce a (partially decidable) type assignment system, which is able to distinguish *all* the needed redexes of a normalisable term. Possible applications of these results include strictness and sharing analysis for functional programming languages, and a reduction strategy for well-typed terms which satisfies Lévy's notion of optimal reduction [15].

We use a type assignment system to identify needed redexes. Typically, a type assignment system constructs typed λ-terms by checking that the application of terms makes sense with respect to the typing information. Implicit in this construction is information regarding the use of variables in the typed term. We use techniques from intersection types [4], adapted to incorporate ideas about resource from linear and relevant logics [10] [7], to make this information explicit by retaining tight control of variables as the terms are being constructed. In this paper we concentrate on a type theory based on intersection types, in order to type as many terms as we can. It should be possible, however, to adapt our ideas to other type assignment systems.

Entailments of our type system have shape $\Gamma, x : T_1 \wedge \ldots \wedge T_n \vdash e : T$, where $T_1 \wedge \ldots \wedge T_n$ denotes a multiset of types. A key property of our system is that, with principal typing, the length of the multiset assigned to x identifies the number of times x is *used* in e: that is, the number of free occurrences of x in the normal form of e when it exists. Our system,

[*]This paper appeared in *Theoretical Aspects of Computer Software*, Sendai, Japan, 1994.

[†]This research was supported by a SERC Research Grant. Address: Department of Computer Science, University of Edinburgh, The King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, UK. E-mail: pag@dcs.ed.ac.uk. Fax: (+) 44-31-667-7206.

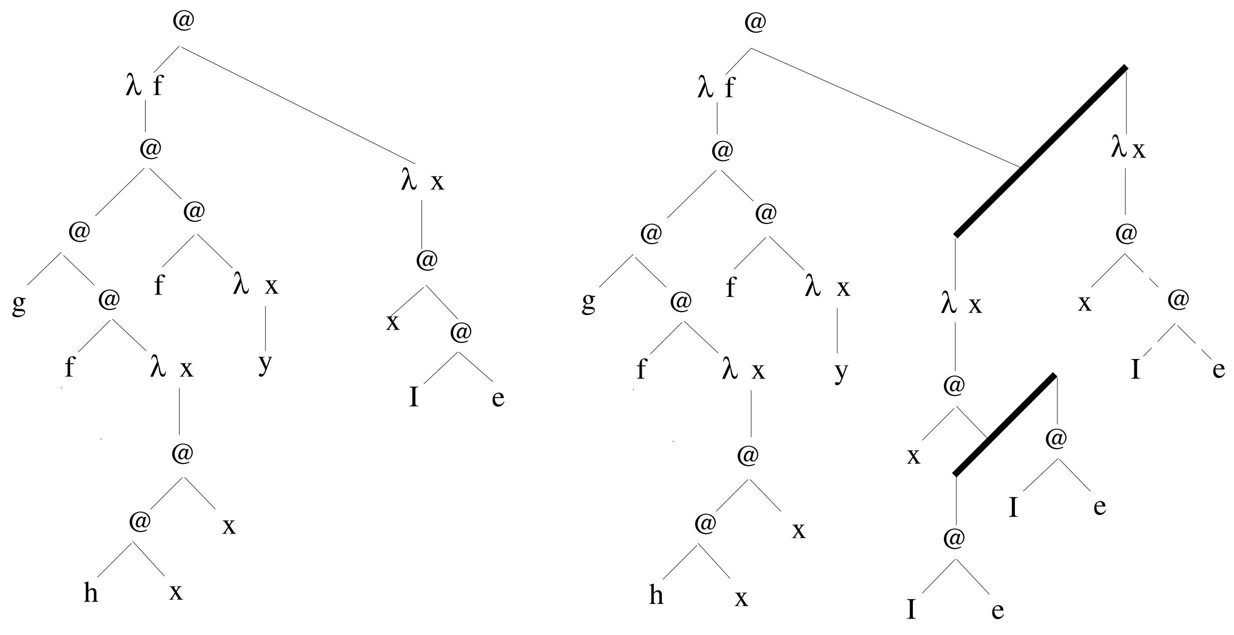

Figure 1: The 2-dimensional and 3-dimensional graphs of the labelled term $(\lambda f.g(f(\lambda x.hxx))(f(\lambda x.y)))(\lambda x.x(Ie))$.

which we call the *labelled resource calculus*, labels the application nodes and variables with type information. Using the labelled application nodes, we are able to distinguish the allowable redexes. We prove that allowable redexes are needed redexes, and, for normalisable terms which have been principally typed, prove that all needed redexes are allowable. The problem of identifying needed redexes has also been addressed by Barendregt et al. [2]. They distinguish the so-called (generalised) spine redexes, and show that these form a *proper* subset of the set of needed redexes. Since this approach does not identify *all* needed redexes, it is unlikely that these results can be used for the applications we have in mind.

Needed reductions are those reductions which contract needed redexes. To capture all the needed reductions, we have to keep track of which residuals of a needed redex are needed after a reduction step. For example, consider the reduction

$$(\lambda f.g(f(\lambda x.hxx))(f(\lambda x.y)))(\lambda x.x(Ie)) \to g((\lambda x.x(\underline{Ie}))(\lambda x.hxx))((\lambda x.x(\underline{\underline{Ie}}))(\lambda x.y)).$$

The needed redex Ie in the original term has two residuals resulting from this reduction. The residual $\underline{Ie}$ is needed, whereas $\underline{\underline{Ie}}$ is not. Such information is retained using the type annotations accompanying the variables. We define the notion of *allowable* reduction, which only contracts allowable redexes. We prove that every allowable reduction is a needed reduction. In the presence of principal typing, it also follows that the needed reductions of normalisable terms are allowable.

We have stressed that the labelled resource calculus retains extra information regarding the use of variables and arguments as the typing derivation proceeds. This extra information can be expressed pictorally using what we call the *3-dimensional graphs* of λ-terms. To explain this concept, consider the standard graph of term $(\lambda f.g(f(\lambda x.hxx))(f(\lambda x.y)))(\lambda x.x(Ie))$ shown on the left-hand side of figure 1. With principal tying, the corresponding labelled term gives rise to the 3-dimensional graph on the right-hand side of figure 1. This 3-dimensional graph should be viewed as a ‘mobile’, with the thick edges, or *beams*, at right-angles to the plane of the paper. There is an obvious projection from the 3-dimensional graph to the 2-dimensional graph, which is analogous to forgetting the typing information in labelled terms. The extra information in the type annotations corresponds to the beam structure of the 3-dimensional graph. Intuitively, the beam of length 2 provides the information that the bound variable f is used twice in $(\lambda f.g(f(\lambda x.hxx))(f(\lambda x.y)))$. The dotted lines in the 3-dimensional graph cature the fact that one residual of Ie is not used in the reduction of $(\lambda f.g(f(\lambda x.hxx))(f(\lambda x.y)))(\lambda x.x(Ie))$ given above. The notion of 3-dimensional graphs provides a useful picture for motivating the definitions and results of this paper.

Summary of the Paper. In section 2 we give the concepts and terminology required to make this paper self-contained. Section 3 introduces the type system for constructing well-typed labelled terms, and defines allowable redexes. Substitution and β-reduction for labelled terms are defined in section 4. The notion of allowable reduction is also introduced in this section, and we show that allowable reductions are finite. Our main technical results are given in section 5: specifically, we show that allowable reductions are needed reductions, and show that, in the presence of principal typing, all the needed reductions of normalisable terms are allowable. We finish in section 6 by indicating how these results can be applied to strictness and sharing analysis, and to optimal reductions à la Lévy [15].

2 Preliminaries

In this section we introduce the concepts required to make the paper self-contained. Unless otherwise stated, the details can be found in [1].

2.1 Definition The set Λ of *untyped λ-terms* is defined by the abstract grammar

$$\Lambda \qquad\qquad e ::= x \mid \lambda x.e \mid e_1 e_2,$$

with $x \in Var$, where Var is a countably infinite set of variables.

The usual notions of α-conversion, free variables and substitution apply. We let $\rightarrow$ denote a one-step β-reduction, and $\rhd$ its reflexive and transitive closure. A term of the form $e_1 e_2$ is an *application.* We call e_2 the *argument* of the application. A term $r = (\lambda x.r_1)r_2$ is called a *redex.* We sometimes write $r : t \rightarrow s$ to emphasise that (a particular occurrence of) r is the contracted redex, and write $r_1, \ldots, r_n : e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_n$ to denote the reduction sequence where r_i is the contracted redex in reduction step $e_{i-1} \rightarrow e_i$. Sometimes we make the application nodes explicit, and write $r = (\lambda x.r_1)@r_2$ instead. The application node given here is distinguished as the *leading application node* of the redex. A term which does not contain a redexe is in *normal form.* We say that e isa *normalisable* term if there is a reduction $e \rhd NF(e)$, where $NF(e)$ denotes that normal form of e when it exists. We write $NF(e)$ for the normal form of e when it exists. A term is in *head-normal form* if it has shape $\lambda x_1 \ldots \lambda x_n.\, ye_1 \ldots e_m$ for variable y and arbitrary terms $e_1, \ldots, e_m$. It is useful to distinguish the *leftmost* redex of a term. The *leftmost* reduction sequence is the one in which each contracted redex is leftmost.

Let r be a redex in e, and let $\rho : e \rhd f$ denote a particular reduction sequence. The redex r can be copied, modified, eliminated or contracted during ρ. The set of redexes in f that descend from r is called the *residual set* of r under ρ, and is denoted by $res(r/\rho)$. One of the simplest ways of defining $res(r/\rho)$ is to uniquely mark all the application nodes in e. Then $res(r/\rho)$ is the set of all redexes in f whose leading application nodes have the same mark as the leading application node for r.

We focus in this paper on *needed* redexes (see for example [2]).

2.2 Definition

1. Let r be a redex in e. Redex r is *needed in* e (or just 'needed' when the term e is apparent) if every reduction sequence of e to normal form reduces some residual of r.
2. A reduction sequence $r_1, \ldots, r_n : e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_n$ is a *needed reduction* if redex r_i is needed in e_{i-1} for each $i \in \{1, \ldots, n\}$.

Thus, in the term $(\lambda u.(\lambda x.y)u)(Ie)$, the redex Ie is not needed. In the introduction we showed that, given a needed redex r, it is not necessarily the case that all the residuals of r are needed. It is, however, always the case that at least one residual of a needed redex is needed.

2.3 Lemma Let e be normalisable, and let r be a needed redex in e. Suppose $\rho : e \rhd f$ is a reduction which does not reduce a residual of r. Then r has a needed residual in f.

This lemma is proved by Barendregt et al. [2], who also prove the following important results.

2.4 Theorem A leftmost reduction is always a needed reduction.

2.5 Theorem It is undecidable whether a given redex in a term is needed.

3 The Type Theory

In this section we introduce the type theory used to identify needed redexes. Our type theory uses techniques from the intersection calculus $\vdash_{\wedge\omega}$ [4], adapted to incorporate explicit information about resource using ideas from linear and relevant logics [10] [7]. We choose to base this paper on intersection types in order to type as many terms as we can; however, it should be possible to apply our approach to other type assignment systems. The type theory is introduced in two stages. First, we provide the core type theory, which we call the *resource calculus*. Then, we adapt the type theory to incorporate type information in the λ-terms. This information is used to distinguish the needed redexes.

3.1 Resource Calculus

Entailments in our resource calculus have the shape $\Gamma, x : T_1 \wedge \ldots \wedge T_n \vdash e : T$, where $T_1 \wedge \ldots \wedge T_n$ can be viewed as a multiset of types. One intuition is that, with principal typing, the multiset assigned to x captures the fact that x occurs free n times in the normal form of e when it exists. The syntax of the resource calculus is defined using the following abstract grammars, where VAR and Var denote countably infinite sets of type variables and term variables respectively:

$$\begin{array}{llllll}
\text{base types} & T & ::= & X & \mid & M \multimap T \\
\text{resource types} & M & ::= & \bot & \mid & T_1 \wedge \ldots \wedge T_n, \qquad n \geq 1 \\
\text{contexts} & \Gamma & ::= & \langle\, \rangle & \mid & \{\Gamma, x : M\} \\
\text{terms} & e & ::= & x & \mid & \lambda x.e \quad \mid \quad e_1 e_2
\end{array}$$

where $X \in VAR$ and $x \in Var$. At this stage, the resource type $T_1 \wedge \ldots \wedge T_n$ denotes a list of base types. The idea that it can be viewed as a multiset arises from the typing rules given in definition 3.1. We typically write $\Gamma, x : M$ for the set $\{\Gamma, x : M\}$. We assume that $\wedge$ takes precedence over $\multimap$: that is, the type $T_1 \wedge T_2 \multimap T_3$ is equivalent to $(T_1 \wedge T_2) \multimap T_3$. Let $dom(\Gamma)$ denote the set of variables declared in context Γ. We combine contexts using list concatenation, as follows:

$$x : T_1 \wedge \ldots \wedge T_n \in conc(\Gamma, \Delta) \text{ if and only if } x : T_1 \wedge \ldots \wedge T_i \in \Gamma \text{ and } x : T_{i+1} \wedge \ldots \wedge T_n \in \Delta,$$

where $i \in \{0, \ldots, n\}$. We let $conc(\Gamma_1, \ldots, \Gamma_n)$ denote $conc(conc(\ldots(conc(\Gamma_1, \Gamma_2), \Gamma_3), \ldots), \Gamma_n)$. This combination of contexts allows us the assign different types to different occurrences of a variable x. For example, we shall see that term xx is well-typed in context $x : (T_1 \multimap T_2) \wedge T_1$.

The resource calculus can be viewed as a fragment of the intersection calculus $\vdash_{\wedge\omega}$ [4], with more control of the use of variables.

3.1 Definition The *resource calculus* is defined by the following rules:

AXIOM $\quad x : T \vdash x : T$

PERM $\quad \dfrac{\Gamma, x : T_1 \wedge \ldots \wedge T_n \vdash e : T \qquad \pi \text{ is a permutation of } \{1, \ldots, n\}}{\Gamma, x : T_{\pi(1)} \wedge \ldots \wedge T_{\pi(n)} \vdash e : T}$

$\lambda 1$ $\quad \dfrac{\Gamma, x : M \vdash e : T \qquad x \notin dom(\Gamma)}{\Gamma \vdash \lambda x.e : M \multimap T}$

$\lambda 2$ $\quad \dfrac{\Gamma \vdash e : T \qquad x \notin dom(\Gamma)}{\Gamma \vdash \lambda x.e : \bot \multimap T}$

APP1 $\quad \dfrac{\Gamma \vdash e : T_1 \wedge \ldots \wedge T_n \multimap T \quad \Delta_i \vdash f : T_i \qquad i \in \{1, \ldots, n\}, n \geq 1}{conc(\Gamma, \Delta_1, \ldots, \Delta_n) \vdash ef : T}$

APP2 $\quad \dfrac{\Gamma \vdash e : \bot \multimap T}{\Gamma \vdash ef : T}$

The $\lambda 2$- and APP-rules require some explanation. The AXIOM- and $\lambda 1$-rule force the term $\lambda x.y$ to have type $\bot \multimap T$ in context $y : T$. Using the APP2-rule, the type of $\lambda u.((\lambda x.y)u)$ is also $\bot \multimap T$. Our intension is that the type $\bot \multimap T$ indicates the fact that the term $\lambda x.y$ is expecting an argument which it does not use. For an arbitary typing, the type $\bot \multimap T$ does not necessarily indicate this. For example, the term fx can be well-typed in the context $f : \bot \multimap T$. However, an intuitively more meaningful typing of this term is in the context $f : T_1 \multimap T_2, x : T_1$. This intuition is captured by the notion of *principal typing*. Principal typing for the resource calculus can be easily adapted from principal typing for intersection types, which has been studied extensively in the literature [4] [5] [16] [17]. With principal typing, the type $\bot \multimap T$ does indeed indicate that the argument is not used, as intended.

The concatenation of contexts in the APP1-rule allows variables to be assigned more than one type. A simple example to illustrate this is the derivation

$$\frac{x : T_1 \multimap T_2 \vdash x : T_1 \multimap T_2 \qquad x : T_1 \vdash x : T_1}{x : (T_1 \multimap T_2) \wedge T_1 \vdash xx : T_2}$$

Observe that the term xx is also well-typed in the contexts $x : \bot \multimap T$ and $x : (T_1 \wedge T_2 \multimap T_3) \wedge T_1 \wedge T_2$. Principal typing ensures that the length of the resource type assigned to a variable corresponds to the number of times that variable x occurs free in the normal form of a term when it exists. We shall see that, in our analysis of needed redexes, arbitrary typing identifies needed redexes, and for normalisable terms principal typing identifies *all* the needed redexes.

3.2 Lemma [Substitution]

1. $\Gamma, x : T_1 \wedge \ldots \wedge T_n \vdash e : T$ and $\Delta_i \vdash f : T_i$, for $n \geq 1$ and $i \in \{1, \ldots, n\}$, imply $conc(\Gamma, \Delta_1, \ldots, \Delta_n) \vdash e[f/x] : T$;
2. $\Gamma \vdash e : T$, for $x \notin dom(\Gamma)$, and $f \in \Lambda_{[\,]}$ imply $\Gamma \vdash e[f/x] : T$.

3.3 Lemma $\Gamma \vdash e : T$ and $e =_\beta e'$ imply $\Gamma \vdash e' : T$.

3.4 Corollary If e has a head-normal form then e can be well-typed in the resource calculus.

Proof It is easy to show that head-normal forms can be well-typed. The result follows from lemma 3.3. □

3.2 Labelled Resource Calculus

The information regarding which redexes are needed redexes is implicit within the derivations of the resource calculus. We adapt the calculus to give an explicit account of this information by annotating the well-typed terms with type information. We annotate application nodes in order to distinguish which redexes are *allowable*, and variables in order to control the type information during β-reduction. In this section, we define the *labelled resource calculus* which incorporates this type information. Our motivation centres on the annotation of the application nodes; the motivation for the variable annotation is given in section 4.

First we introduce the set Λ_L of *labelled* λ-terms defined by the following abstract grammars:

Λ_L	e	$::=$	$x^U \mid$	$\lambda x.e \mid e_1 @_V e_2$	
List(T)	U	$::=$	$T \mid$	$[U_1, \ldots, U_n]$	$n \geq 0$
List(M)	V	$::=$	$M \mid$	$[V_1, \ldots, V_m]$	$m \geq 0$

where T denotes a base type and M a resource type, and $x \in Var$.

We motivate the well-typed labelled terms by appealing to the 3-dimensional graphs discussed in the introduction. We shall see that the labelled term[1] $(\lambda x.gxx)@_{T_1 \wedge T_2}(I@_{[T_1,T_2]}y)$ is well-typed in the labelled resource calculus. The corresponding 3-dimensional graph is given in figure 2. The application node $@_{T_1 \wedge T_2}$ provides *local* information: it indicates that the 3-dimensional graph of the argument $I@_{[T_1,T_2]}y$ begins with a beam of length two. Intuitively, this local information indicates how many times the argument is used in the application. The application node $@_{[T_1,T_2]}$ provides *global* information: it informs us that the 3-dimensional graph of the subterm $I@_{[T_1,T_2]}y$ consists of two graphs connected by a beam of length two. The sub-graphs are given by the terms $I@_{T_1}y$ and $I@_{T_2}y$, where the application nodes provide the local information that y is used once in each application $I@_{T_i}y$. Hence, variable y is used twice in the overall term. This global information, given to us by annotations of the form $[V_1, \ldots, V_m]$

[1]For clarity when discussing examples, we only label the application nodes and variables of particular interest to the point under discussion.

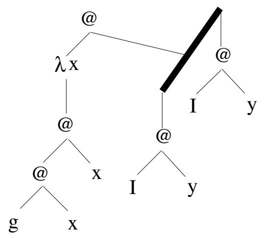

Figure 2: The 3-dimensional graph of $(\lambda x.gxx)@_{T_1 \wedge T_2}(I@_{[T_1,T_2]}y)$.

for $m \geq 0$, provides us with essential information for preserving the type annotations during β-reduction. This point is made clearer in section 4.

There is an obvious projection of the 3-dimensional graph of a labelled term onto its standard 2-dimensional counterpart, which discards the beam information. It is useful to define such a concept for the labelled λ-terms themselves. We define the projection function $proj : \Lambda_L \rightharpoonup \Lambda_{[\,]}$, where $\Lambda_{[\,]}$ is defined by the abstract grammar $e ::= x^{[\,]} \mid \lambda x.e \mid e_1 @_{[\,]} e_2$. The function $proj : \Lambda_L \rightharpoonup \Lambda_{[\,]}$ is defined inductively on the structure of labelled λ-terms as follows:

$$\begin{aligned} proj(x^U) &= x^{[\,]} \\ proj(\lambda x.e) &= \lambda x.proj(e) \\ proj(e_1 @_V e_2) &= proj(e_1)@_{[\,]}proj(e_2). \end{aligned}$$

The application nodes annotated with resource types can be defined directly by the typing in the resource calculus. In order to incorporate more complicated annotations of the form $[V_1, \ldots, V_n]$, we require the partial functions $merge_n : \underbrace{\Lambda_L \times \ldots \times \Lambda_L}_{n} \rightharpoonup \Lambda_L$ for each $n \geq 1$, which essentially package up the subgraphs on a beam into one. In the example above, the merge functions are used to create $I@_{[T_1,T_2]}y$ from $I@_{T_1}y$ and $I@_{T_2}y$ respectively. The partial function $merge_n : \underbrace{\Lambda_L \times \ldots \times \Lambda_L}_{n} \rightharpoonup \Lambda_L$, for arbitrary $n \geq 1$, is defined inductively on the structure of labelled λ-terms as follows:

$$\begin{aligned} merge_n(x^{U_1}, \ldots, x^{U_n}) &= x^{[U_1,\ldots,U_n]} \\ merge_n(\lambda x.e_1, \ldots, \lambda x.e_n) &= \lambda x.merge_n(e_1, \ldots, e_n) \\ merge_n(e_1 @_{V_1} f_1, \ldots, e_n @_{V_n} f_n) &= merge_n(e_1, \ldots, e_n)@_{[V_1,\ldots,V_n]}merge_n(f_1, \ldots, f_n). \end{aligned}$$

$merge_n(e_1, \ldots, e_n)$ is undefined when $proj(e_i) \neq proj(e_j)$ for some $i, j \in \{1, \ldots, n\}$.

We now define the *labelled resource calculus* by adapting the resource calculus of section 3.1 to incorporate labelled λ-terms.

3.5 Definition The *labelled resource calculus* is defined by the following rules:

AXIOM $\quad x : T \vdash_L x^T : T$

PERM $$\frac{\Gamma, x : T_1 \wedge \ldots \wedge T_n \vdash_L e : T \qquad \pi \text{ is a permutation of } \{1, \ldots, n\}}{\Gamma, x : T_{\pi(1)} \wedge \ldots \wedge T_{\pi(n)} \vdash_L e : T}$$

$\lambda 1$ $$\frac{\Gamma, x : M \vdash_L e : T \qquad x \notin dom(\Gamma)}{\Gamma \vdash_L \lambda x.e : M \multimap T}$$

$\lambda 2$ $$\frac{\Gamma \vdash_L e : T \qquad x \notin dom(\Gamma)}{\Gamma \vdash_L \lambda x.e : \bot \multimap T}$$

APP1 $$\frac{\Gamma \vdash_L e : T_1 \wedge \ldots \wedge T_n \multimap T \quad \Delta_i \vdash_L f_i : T_i \qquad proj(i) = proj(j), n \geq 1}{conc(\Gamma, \Delta_1, \ldots, \Delta_n) \vdash_L e@_{T_1 \wedge \ldots \wedge T_n} merge_n(f_1, \ldots, f_n) : T}$$

APP2 $$\frac{\Gamma \vdash_L e : \bot \multimap T \qquad f \in \Lambda_{[\,]}}{\Gamma \vdash_L e@_\bot f : T}$$

As observed in the previous section, an untyped term can have many corresponding labelled terms. Principal typing distinguishes the unique 'correct' one. For example, the correct labelled term of the term $(\lambda x.y)u$ is $(\lambda x.y)@_\bot u$. We have already remarked that $(\lambda x.gxx)(Ie)$ has the correct labelled term $(\lambda x.gxx)@_{T_1 \wedge T_2}(I@_{[T_1,T_2]}e)$. The term $(\lambda x.x(Iy)(\lambda y.y))(\lambda p.\lambda q.p)$ has the correct labelled term

$$(\lambda x.x@_T(I@_{[T]}y)@_\bot(I))@_{T \multimap \bot \multimap T}(\lambda p.\lambda q.p).$$

This last term is especially interesting as it is the example cited by Barendregt et al. [2] to illustrate the fact that they cannot identify all needed redexes. In particular, they cannot show that the redex $(I@_{[T]}y)$ is a needed redex. By contrast, we are able to show that the redex is needed using the type annotations of its leading application node. To make this precise, we require the flattening function $|\,.\,| : List(M) \to List(M)$, defined inductively on the structure of $List(M)$ as follows:

$$\begin{aligned} |L| &= [L] \\ |[\,]| &= [\,] \\ |[V_1, \ldots, V_n]| &= conc(|V_1|, \ldots, |V_n|), \end{aligned}$$

where $conc(|V_1|, \ldots, |V_n|)$ denotes the standard list concatenation of $|V_1|, \ldots, |V_n|$.

Using this flattening function, we define the allowable redexes of a well-typed term. In theorem 5.6, we show that allowable redexes are needed redexes.

3.6 DEFINITION An *allowable* redex of a well-typed labelled term t is a subterm of the form $(\lambda x.r_1)@_V r_r$ such that $|V| \neq [\,]$.

Thus, the redex $(I@_{[T]}y)$ in the above example is an allowable, and hence a needed, redex.

There is an important connection between the declarations of variables in the contexts, and the type annotations of the free occurrences of these variables in well-typed labelled terms. This connection is fundamental to our notions of substitution and reduction given in section 4. First we define the list of free labelled occurrences of x in labelled term e, denoted by $l_x(e)$, as follows:

$$\begin{aligned} l_x(x^T) &= [x^T] \\ l_x(x^{[\,]}) &= [\,] \\ l_x(x^{[U_1,\ldots,U_n]}) &= conc(l_x(x^{U_1}), \ldots, l_x(x^{U_n})) \end{aligned}$$

$$\begin{array}{rcl} l_x(y^U) & = & [\,] \\ l_x(\lambda x.t) & = & [\,] \\ l_x(\lambda y.t) & = & l_x(t) \\ l_x(t_1 @_V t_2) & = & conc(l_x(t_1), l_x(t_2)) \end{array}$$

Given a well-typed term e in context Γ, there is a precise connection between $l_x(e)$ and the declaration of x in Γ, as the following lemma states.

3.7 LEMMA $\Gamma, x : T_1 \wedge \ldots \wedge T_n \vdash_L e : T$ implies $l_x(e) = [x^{T_{\pi(1)}}, \ldots, x^{T_{\pi(n)}}]$ for some permutation π of $\{1, \ldots, n\}$.

4 Allowable β-reduction

In this section we define labelled substitution and labelled β-reduction. We define allowable β-reduction as the restriction of labelled β-reduction to allowable redexes. In section 5, we show that allowable redcutions are needed reductions.

We have already observed in the introduction that residuals of needed redexes are not necessarily needed. We therefore have to be careful to ensure that β-reduction preserves the typing information. This preservation of the typing information would not be possible if we simply annotated the application nodes; we must also annotate the variables. The astute reader will have noticed that the type information annotating the application nodes has been flattened. For example, we have the well-typed labelled term

$$(\lambda y.f(I @_{[T_1]} y^{[[T_1]]})((\lambda x.gxx) @_{[T_2 \wedge T_3]} y^{[[T_2,T_3]]})) @_{T_1 \wedge T_2 \wedge T_3} e^{[T_1,T_2,T_3]}.$$

We shall see that the reduction of the outermost redex replaces the first occurrence of y by $e^{[[T_1]]}$, and the second occurrence of y by $e^{[[T_2,T_3]]}$. The information regarding the splitting of $e^{[T_1,T_2,T_3]}$ into $e^{[[T_1]]}$ and $e^{[[T_2,T_3]]}$ is not given by the type annotations of e. Such information would result in the contraction of a redex having a global effect on the type annotations of a term, which is clearly an undesirable property when defining labelled substitution. For example, consider the reduction

$$(\lambda y.f(I @_{[T_1]} y^{[[T_1]]})((\lambda x.gxx) @_{[T_2 \wedge T_3]} y^{[[T_2,T_3]]})) @_{T_1 \wedge T_2 \wedge T_3} e^{[T_1,T_2,T_3]} \rightarrow$$
$$(\lambda y.f(I @_{[T_1]} y^{[[T_1]]})(gy^{[[T_2]]} y^{[[T_3]]})) @_{T_1 \wedge T_2 \wedge T_3} e^{[T_1,T_2,T_3]},$$

where the inner redex $(\lambda x.gxx) @_{[T_2 \wedge T_3]} y^{[[T_2,T_3]]}$ has been reduced. If we now reduce the outermost redex of this resulting term, we see that the three occurrences of variable y are replaced by $e^{[[T_1]]}$, $e^{[[T_2]]}$ and $e^{[[T_3]]}$. Hence, the contraction of the inner redex has resulted in a *different* splitting of $e^{[T_1,T_2,T_3]}$. This difference is accounted for in the type annotations of the occurrences of y.

4.1 Labelled Substitution

Consider a well-typed labelled redex of the form $(\lambda x.e) @_{T_1 \wedge \ldots \wedge T_n} f$. We know from lemma 3.7, and by inspecting the rules of the labelled resource calculus, that $f = merge_n(f_1, \ldots, f_n)$ and $l_x(e) = [x^{T_{\pi(1)}}, \ldots, x^{T_{\pi(n)}}]$. Our definition of labelled substitution replaces each $x^{T_{\pi(i)}}$ by the appropriate $f_{\pi(i)}$ for $i \in \{1, \ldots, n\}$.

4.1 DEFINITION Let $e \in \Lambda_L$ with $l_x(e) = [x^{T_1}, \ldots, x^{T_n}]$. Let $f_1, \ldots, f_n \in \Lambda_L$, such that $proj(f_i) = f \in \Lambda_{[\,]}$. Define $e[f_1, \ldots, f_n, f / x^{T_1}, \ldots, x^{T_n}, x]$ by induction on the structure of e as follows:

1. if e is x^{T_1} then $x^{T_1}[f_1, f/x^{T_1}, x] = f_1$;
2. if e is $x^{[\,]}$ then $x^{[\,]}[f/x] = f$;
3. if e is $x^{[U_1,\ldots,U_m]}$ for $m \geq 1$ then $x^{[U_1,\ldots,U_m]}[f_1,\ldots,f_n, f/x^{T_1},\ldots,x^{T_n}, x] = \mathit{merge}_m(g_1,\ldots,g_m)$, where each g_i is $x^{U_i}[f_{n_{i-1}+1},\ldots,f_{n_i}, f/x^{T_{n_{i-1}+1}},\ldots,x^{T_{n_i}}, x]$ for $l_x(x^{U_i}) = [x^{T_{n_{i-1}+1}},\ldots,x^{T_{n_i}}]$, and $n_0 = 0$, $n_m = n$, and $i < j$ implies $n_i \leq n_j$;
4. if e is y^U then $y^U[f/x] = y^U$;
5. if e is $(\lambda x.e)$ then $(\lambda x.e)[f/x] = \lambda x.e$;
6. if e is $(\lambda y.e)$ then $(\lambda y.e)[f_1,\ldots,f_n, f/x_1,\ldots,x_n, x]$ is $(\lambda z.e[z^{S_1},\ldots,z^{S_m}, z/, y^{S_1},\ldots,y^{S_m}, y][f_1,\ldots,f_n, f/x_1,\ldots,x_n, x])$, where $l_y(e) = [y^{S_1},\ldots,y^{S_m}]$ and $z \notin \{x,y\} \cup fv(f) \cup fv(e)$;
7. if e is $(e_1 @_V e_2)$ then $(e_1 @_V e_2)[f_1,\ldots,f_n, f/x^{T_1},\ldots,x^{T_n}, x]$ is $e_1[f_1,\ldots,f_i, f/x^{T_1},\ldots,x^{T_i}, x] @_V e_2[f_{i+1},\ldots,f_n, f/x^{T_{i+1}},\ldots,x^{T_n}]$, where for $i \in \{0,\ldots,n\}$ we have $l_x(e_1) = [x^{T_1},\ldots,x^{T_i}]$ and $l_x(e_2) = [x^{T_{i+1}},\ldots,x^{T_n}]$.

It is to be expected that the definition of labelled substitution is notationally more complicated than that of standard substitution, since we have to be careful to preserve the typing annotations. The complication arises from having to treat variable occurrences separately, and from having to unpack the lists annotating these variables. Treating variable occurrences separately is routine. Unpacking the lists annotating the variables is not a problem, since the embedding of the lists has the same structure as the embedding of the application nodes.

4.2 Lemma [Labelled Substitution]

1. $\Gamma, x : T_1 \wedge \ldots \wedge T_n \vdash_L e : T$ and $\Delta_i \vdash_L f_i : T_i$, where $\mathit{proj}(f_i) = f \in \Lambda_{[\,]}$ for all $i \in \{1,\ldots,n\}$, imply $\mathit{conc}(\Gamma, \Delta_1,\ldots,\Delta_n) \vdash_L e[f_{\pi(1)},\ldots,f_{\pi(n)}, f/x^{T_{\pi(1)}},\ldots,x^{T_{\pi(n)}}, x] : T$, where $l_x(e) = [x^{T_{\pi(1)}},\ldots,x^{T_{\pi(n)}}]$.
2. $\Gamma \vdash e : T$, where $x \notin \mathit{dom}(\Gamma)$, and $f \in \Lambda_{[\,]}$ imply $\Gamma \vdash e[f/x] : T$.

4.2 Labelled Reduction

In this section, we define labelled β-reduction and distinguish the *allowable* reduction steps. A key issue regarding labelled reduction is to preserve the typing annotations of labelled terms during reduction. To explain our intuition, we return to the 3-dimensional graph of the λ-term

$$(\lambda f.g(f(\lambda x.hxx))(f(\lambda x.y)))@_{((T_1\wedge T_2\multimap T_3)\multimap T_3)\wedge((\bot\multimap T_4)\multimap T_4)}(\lambda x.x@_{[T_1\wedge T_2,\bot]}(I@_{[[T_1,T_2],[\,]]}e))$$

shown in figure 3.

Consider what happens to the 3-dimensional graph when the outermost redex is contracted. One branch (marked 1) of the beam is substituted for the first occurrence of variable f (marked f_1), and the other branch is substituted for the second occurrence. If the redex Ie is contracted first, then all the branches corresponding to Ie are contracted at once, as the box around the branches in figure 3 indicates. The mechanism for reducing the outer redex in the above example is to unpack the argument to obtain the components $\lambda x.x@_{T_1\wedge T_2}(I@_{[T_1,T_2]}e)$ and $\lambda x.x@_{\bot}(I@_{[\,]}e)$. Then the first occurrence of f is replaced by the first component, and the second occurrence

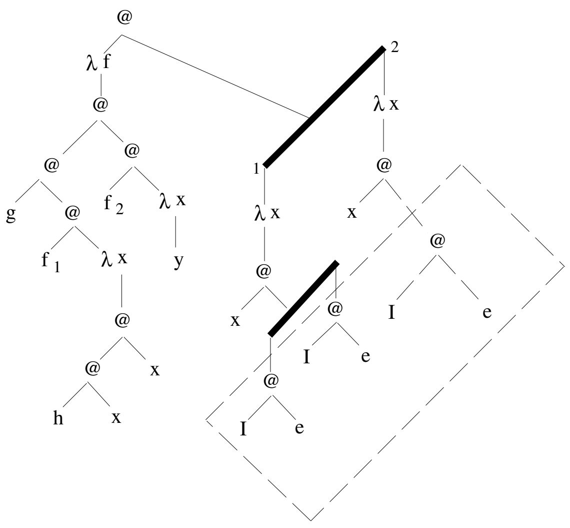

Figure 3: The 3-dimensional graph of the labelled term $(\lambda f.g(f(\lambda x.hxx))(f(\lambda x.y)))@_{((T_1\wedge T_2\multimap T_3)\multimap T_3)\wedge((\bot\multimap T_4)\multimap T_4)}(\lambda x.x@_{[T_1\wedge T_2,\bot]}(I@_{[[T_1,T_2],[\,]]}e))$.

of f by the second component. The mechanism for reducing $I@_{[[T_1,T_2],[\,]]}e$ is more complicated. First we unpack the redex into its fundamental components $I@_{T_1}e$, $I@_{T_2}e$ and $I@_{[\,]}e$. Then we reduce all these redexes to obtain e^{T_1}, e^{T_2} and $e^{[\,]}$. Finally, we repack in the same way as we unpacked to obtain $e^{[[T_1,T_2],[\,]]}$.

In order to define labelled β-reduction, we define the partial projection functions $\pi_i^n : \Lambda_L \rightharpoonup \Lambda_L$, which split a beam of length n into its components. These functions are defined, for $n \geq 1$ and $i \in \{1,\ldots,n\}$, by induction on the structure of labelled terms as follows:

$$\begin{aligned}
\pi_i^n(x^{[U_1,\ldots,U_n]}) &= x^{U_i}\\
\pi_i^n(\lambda x.e) &= \lambda x.\pi_i^n(e)\\
\pi_i^n(e_1@_{[V_1,\ldots,V_n]}e_2) &= \pi_i^n(e_1)@_{V_i}\pi_i^n(e_2).
\end{aligned}$$

π_i^n is undefined in all other cases.

Our definition of labelled β-reduction $\rightharpoonup_L$ is given for the so-called well-typed preterms. Notice that β-reduction for arbitrary labelled terms would not make sense. For example, in the labelled term $(\lambda x.fxx)@_{T_1\wedge T_2}y^{[T_1,T_2,T_3]}$ we have two free occurrences of x but three components to $y^{[T_1,T_2,T_3]}$. Hence, the substitution of y for x in fxx is not defined. We are also not able to restrict labelled β-reduction to well-typed terms, since the subterms of well-typed terms are not necessarily well-typed. For example, in the example of figure 3 the subterm $I@_{[[T_1,T_2],[\,]]}e$ is not well-typed.

4.3 Definition Let e be a labelled term. We define the notion of n-level preterm for $n \geq 0$ by induction on n:

1. e is a 0-*level preterm* if e is well-typed;

2. e is a n-*level preterm* for $n \geq 1$ is $e = merge_n(e_1,\ldots,e_n)$ and each e_i is a $(n-1)$-level preterm;

3. e is a *n-level preterm* for $n \geq 0$ is $e \in \Lambda_{[\,]}$.

The labelled term e is a *well-typed preterm* if it is an n-level preterm for some $n \geq 0$.

The above subterm $I @_{[[T_1,T_2],[;]]} e$ is a 3-level preterm, since it equals $merge_2(merge_2(I @_{T_1} e, I @_{T_2} e), I_{[\,]} e)$. In fact, all subterms of well-typed preterms are also well-typed preterms. This result means that defining labelled β-reduction for well-typed preterms is enough for us to reduce any redex in w well-typed term.

4.4 Lemma Let e be a well-typed preterm and let f be a subterm of e. Then f is a well-typed preterm.

The definition of labelled reduction consists of two parts. First we define the *redex reduction* $\rightarrow_r$, which unpacks the redexes into their underlying components, reduces the components, and then packs up the results. Using this redex reduction, we then define the labelled reduction $\rightarrow_L$ which extends the redex reduction to arbitrary terms.

4.5 Definition The *redex reduction* $\rightarrow_r$ is a binary relation on well-typed preterms which is defined by induction on the structure of the annotations of the leading application nodes of redexes as follows:

$$(\lambda x.e) @_{T_1 \wedge \ldots \wedge T_n} f \rightarrow_r e[\pi^n_{\sigma(1)}(f), \ldots, \pi^n_{\sigma(n)}(f), proj(f)/x^{T_{\sigma(1)}}, \ldots, x^{T_{\sigma(n)}}, x]$$

where $n \geq 1$ and $l_x(e) = [x^{T_{\sigma(1)}}, \ldots, x^{T_{\sigma(n)}}]$[2].

$$(\lambda x.e) @_{\perp} f \rightarrow_r e[f/x]$$

$$(\lambda x.e) @_{[\,]} f \rightarrow_r e[f/x]$$

$$\frac{\lambda x.\pi^n_i(e) @_{V_i} \pi^n_i(f) \rightarrow_r g_i \qquad i \in \{1, \ldots n\}, n \geq 1}{(\lambda x.e) @_{[V_1,\ldots,V_n]} f \rightarrow_r merge_n(g_1, \ldots, g_n)}$$

4.6 Definition Labelled β-reduction $\rightarrow_L$ is a binary relation on well-typed preterms defined by induction on the structure of labelled terms as follows:

$$* \qquad \frac{(\lambda x.e) @_V f \rightarrow_r g}{(\lambda x.e) @_V f \rightarrow_L g}$$

$$\frac{e \rightarrow_L e'}{\lambda x.e \rightarrow_L \lambda x.e'}$$

$$\frac{e \rightarrow_L e'}{e @_V f \rightarrow_L e' @_V f}$$

$$\frac{e \rightarrow_L e'}{f @_V e \rightarrow_L f @_V e'}$$

[2]We adopt the convension that if $T_i = T_j$ for $i < j$ then $\sigma(i) < \sigma(j)$.

We let $\rhd_L$ denote the reflexive, transitive closure of $\rightarrow_L$.

4.7 Lemma Labelled β-reduction is well-defined.

4.8 Lemma The relation $\rhd_L$ on well-typed preterms satisfies the Church-Rosser property.

4.9 Lemma $\Gamma \vdash_L e : T$ and $e =_L e'$ implies $\Gamma \vdash_L e' : T$.

The proofs of the above lemmas are technical but not difficult; details can be found in the full paper [8]. We let $r : e \rightarrow_L f$ denote the derivation of $e \rightarrow_L f$ such that r is the contracted redex[3]. Also let $r_1, \ldots, r_n : e_0 \rightarrow_L e_1 \rightarrow_L \ldots \rightarrow_L e_n$ denote the finite reduction path such that r_i is the redex contracted in the reduction step $e_{i-1} \rightarrow_L e_i$. Let $\rho : e \rhd_L f$ denote an arbitrary reduction path. The notions of residual sets, needed redexes and needed reductions lift immediately from the definitions given for the untyped λ-calculus in section 2.

4.3 Allowable β-reduction

One of the main motivations for studying the labelled resource calculus is that it enables us to define *allowable* β-reduction. Allowable reduction restricts labelled reduction to reducing only allowable redexes.

4.10 Definition Allowable β-reduction is a binary relation on well-typed preterms, denoted by $\rightarrow_a$, which is constructed using the same rules as those for $\rightarrow_L$, except that the rule $*$ is restricted as follows:

$$\frac{(\lambda x.e)@_V f \rightarrow_r g \qquad |V| \neq [\,]}{(\lambda x.e)@_V f \rightarrow_a g}$$

Again we let $\rhd_a$ denote the reflexive, transitive closure of $\rightarrow_a$. Note that allowable β-reduction does not satisfy the Church-Rosser property. For example, the term $(\lambda x.f@_\perp x^{[\,]}@_T x^{[T]})@_T(I@_{[T]}z^{[[T]]})$ reduces to the terms $f@_\perp(I@_{[\,]}z^{[\,]})@_T z^{[T]}$ and $f@_\perp z^{[\,]}@_T z^{[T]}$, which are both in allowable normal form.

The concluding part of this section establishes a close connection between the type annotations of application nodes of well-typed preterms, and the length of an allowable reduction to *allowable* normal form. A well-typed preterm e is in *allowable normal form*, denoted by $ANF(e)$, if there are no allowable redexes in e. In particular, we show that all allowable reduction paths to allowable normal form have the same length. We also use these results to show that well-typed labelled terms have head-normal forms.

4.11 Definition Let e be a well-typed preterm. The length of the derivation $(\lambda x.r_1)@_V r_2 : e \rightarrow_a f$, denoted by $|(\lambda x.r_1)@_V r_2 : e \rightarrow_a f|$, is the length of the list $|V|$.

We let $l(|V|)$ denote the length of list $|V|$.

Given a reduction path $r_1 \ldots r_n : e_0 \rightarrow_a e_1 \rightarrow_a \ldots \rightarrow_a e_n$, we define

$$|r_1 \ldots r_n : e_0 \rightarrow_a e_1 \rightarrow_a \ldots \rightarrow_a e_n| = \sum_{i=1}^{n} |r_i : e_{i-1} \rightarrow_a e_i|.$$

We now define the notion of the *number of labelled application nodes* of a labelled term. Intuitively, for well-typed preterms this number corresponds to the number of application nodes that are annotated with types in the 3-dimensional graph of the term.

[3]The notion of contracted redex can be defined precisely since every derivation of $e \rightarrow_L f$ contains precisely one use of the $*$ rule.

4.12 Definition The *number of labelled application nodes* in labelled term $e \in \Lambda_L$, denoted by $\#(e)$, is defined inductively on the structure of e as follows:

$$\begin{array}{rcl}\#(x^U) & = & 0 \\ \#(\lambda x.e) & = & \#(e) \\ \#(e_1 @_V e_2) & = & \#(e_1) + \#(e_2) + l(|V|).\end{array}$$

The following lemma states that, during substitution, the number of labelled application nodes remains constant.

4.13 Lemma Let $e \in \Lambda_L$ with $l_x(e) = [x^{T_1}, \ldots, x^{T_n}]$, and let $f_1, \ldots, f_n \in \Lambda_L$ such that $proj(f_i) = f \in \Lambda_{[\,]}$ for all $i \in \{1, \ldots, n\}$. Then

$$\#(e[f_1, \ldots, f_n, f/x^{T_1}, \ldots, x^{T_n}, x]) = \#(e) + \sum_{i=1}^{n} \#(f_i).$$

We can now state a connection between the type annotations of application nodes, and the length of an allowable reduction to allowable normal form.

4.14 Lemma Let $r : e \rightarrow_a f$. Then $\quad \#(e) = |r : e \rightarrow_a f| + \#(f)$

4.15 Corollary Let $\rho : e \rhd_a ANF(e)$ and $\sigma : e \rhd_a ANF(e)$ denote two reduction paths to allowable normal form. Then $|\rho : e \rhd_a ANF(e)| = |\sigma : e \rhd_a ANF(e)|$.

4.16 Corollary All allowable reduction paths from well-typed preterm e have finite length.

From these results, it is possible to show that all well-typed terms have head-normal forms.

4.17 Lemma $\Gamma \vdash_L e : T$ implies e has a head-normal form.

Proof By corollary 4.16, we know that the length of an allowable reduction is finite. Hence, there is a finite allowable reduction to the allowable normal form of e. By lemma 4.9, we know that $\Gamma \vdash_L ANF(e) : T$. By analysing the typing rules, we see that $ANF(e)$ is in head-normal form. $\square$

5 Results

This section contains the main technical results of this paper. We show that all allowable reductions are needed reductions. We also show that, for normalisable terms which have been principally typed, needed reductions are allowable reductions.

A key property of an allowable reduction is that an allowable redex is either used in the reduction, or its residual set for this reduction is non-empty and at least one of its elements is allowable. Of course, this property does not hold for arbitrary labelled reductions: for example, it does not hold for the labelled reduction $\quad (\lambda x.y)@_\perp(I@_{[\,]}e) \rightarrow_L \lambda x.y$.

5.1 Lemma Let e be a well-typed preterm, and let r be an allowable redex in e. Let $s : e \rightarrow_a f$. Then either s is r, or $res(r/s) \neq \emptyset$ and at least one of its elements is allowable.

Proof (Sketch) The proof is by induction on the structure of the derivation of $s : e \rightarrow_a f$. The interesting case is

$$(\lambda x.e)@_{T_1 \wedge \ldots \wedge T_n} merge_n(f_1, \ldots, f_n) \rightarrow_a e[f_1, \ldots, f_n, proj(f_1)/x^{T_1}, \ldots, x^{T_n}, x],$$

where redex r is an allowable redex in $merge_n(f_1, \ldots, f_n)$. By definition of allowable redexes, there is at least one corresponding allowable redex r_i in f_i. From the definition of substitution, it can be shown that each f_i is contained in $e[f_1, \ldots, f_n, f/x^{T_1}, \ldots, x^{T_n}, x]$. (The details for defining when a term is contained in another term are technical, but not difficult.) It is therefore possible to prove that redex r_i is contained in $e[f_1, \ldots, f_n, f/x^{T_1}, \ldots, x^{T_n}, x]$. The details of this proof can be found in the full paper [8]. □

5.2 Corollary Let r be an allowable redex in well-typed preterm e, and let $\rho : e \rhd_a ANF(e)$. Then a residual of redex r is used in ρ.

We now show that the *leftmost allowable reduction* is a needed reduction, using the result of Barendregt et al. [2] that the leftmost reduction is a needed reduction. As one might expect, the leftmost allowable reduction is the reduction where the leftmost allowable redex is contracted at each stage. To prove that this reduction is a needed reduction, we require the following lemma which provides conditions under which redexes needed in subterms are also needed in the whole term.

5.3 Lemma

1. Redex r is needed in well-typed preterm $\lambda x.e$ if r is needed in e.
2. Redex r is needed in well-typed preterm $x@_{V_1}e_1@_{V_2}\ldots@_{V_n}e_n$ if r is needed in e_i for some $i \in \{1, \ldots, n\}$.
3. Redex r is needed in well-typed preterm $merge_n(e_1, \ldots, e_n)$ if there exists an $i \in \{1, \ldots, n\}$ such that $\pi_i^n(r)$ is needed in e_i.

5.4 Lemma The leftmost allowable redex of well-typed preterm e is a needed redex in e.

Proof Assume that e is an n-level preterm for $n \geq 0$. The proof follows by induction on n. The interesting case is when $n = 0$ and e is well-typed. For this case, preceed by induction on the structure of e. The interesting case is when e is an application. If e is $(\lambda x.e_0)@_{M_1}e_1@_{M_2}\ldots@_{M_n}e_n$, then by inspecting the typing rules we see that M_1 must have shape $T_1 \wedge \ldots \wedge T_m$ or $\perp$. Hence, the leftmost allowable redex in this case is the leftmost redex. By theorem 2.4, the leftmost redex is a needed redex. If e is $x@_{M_1}e_1@_{M_2}\ldots@_{M_n}e_n$, then we know that e_i for some $i \in \{1, \ldots, n\}$ contains the leftmost allowable redex r of e. Again we know that M_i is $T_1 \wedge \ldots \wedge T_m$ or $\perp$. The latter case cannot hold as e_i contains allowable redex r. Hence $\pi_j^m(e_i)$ is well-typed for each $j \in \{1, \ldots m\}$. We know that $r_j = \pi_j^m(r)$ is an allowable redex for some j. By the induction hypothesis, we know that r_j is needed in $\pi_j^m(e_i)$. Using lemma 5.3, we know that r is needed in e_i, and that r is needed in $x@_{M_1}e_1@_{M_2}\ldots@_{M_n}e_n$. □

5.5 Corollary The leftmost allowable reduction is a needed reduction.

Now we can prove our first main theorem, which says that allowable reductions are needed reductions.

5.6 Theorem Let r be an allowable redex in well-typed preterm e. Then r is a needed redex.

Proof Consider the leftmost allowable reduction $\rho : e \rhd_a ANF(e)$. By corollary 5.5, the leftmost allowable reduction is a needed reduction. By corollary 5.2, redex r is used in ρ. Hence, redex r is needed. $\square$

5.7 Corollary Let $\rho : e \rhd_a f$ denote an allowable reduction. Then $\rho : e \rhd_a f$ is a needed reduction.

We have already mentioned that, for an arbitrary typing of a term, we have not captured all the needed redexes. A simple example that illustrates this point is the term $(\lambda x.x)(Ie)$, one of whose labelled terms is $(\lambda x.x)@_\perp(I@_{[\,]}e)$. In this labelled term, the redex $I@_{[\,]}e$ is not allowable, but it is needed. The 'correct' labelled term is $(\lambda x.x)@_T(I@_{[T]}e)$, which is obtained by principal typing. For our purposes, it is enough to define principal typing for normalisable terms in the labelled resource calculus, via a definition of principal typing for their normal forms. With this definition, we prove that if a normalisable term is principally typed then every needed reduction from e is an allowable reduction. A more direct definition of principal typing is given in [8], based on the notion of principal typing for the intersection calculus $\vdash_{\wedge\omega}$, which has been studied extensively in the literature [4] [5] [16] [17].

5.8 Definition Let e be a labelled term in normal form. The entailment $\Gamma \vdash_L e : T$ is *principally typed* in the labelled resource calculus if:

1. e is x^X and $(\Gamma, T) = (\{x : X\}, X)$, where X is a type variable;
2. e is $\lambda x.e'$ and $\Gamma' \vdash_L e' : T'$ is principally typed, and either
   (a) $x \notin dom(\Gamma')$ and $(\Gamma, T) = (\Gamma', \perp \multimap T')$; or
   (b) $x : M \in \Gamma'$ and $(\Gamma, T) = (\Gamma' - \{x : M\}, M \multimap T')$;
3. e is $x@_{T_1}merge_1(e_1)@_{T_2}\ldots@_{T_n}merge_1(e_n)$ for $n \geq 1$, and each $\Gamma_i \vdash_L e_i : T_i$ is principally typed, and $(\Gamma, T) = (conc(\{x : T_1 \multimap \ldots \multimap T_n \multimap X\}, conc(\Gamma_1, \ldots, \Gamma_n)), X)$, where X is a new type variable.

For normalisable term e, we say that $\Gamma \vdash_L e : T$ is *principally typed* if $\Gamma \vdash_L NF(e) : T$ is principally typed.

Recall that allowable reduction does not satisfy the Church-Rosser property. A key property of principal typing is that, if $\Gamma \vdash_L e : T$ is principally typed in the labelled resource calculus for normalisable term e, then every allowable normal form of e equals the normal form.

5.9 Lemma Let $\Gamma \vdash_L e : T$ be principally typed such that e is a normalisable term. Then $e = NF(e)$.

Proof Since allowable normal forms are in head-normal form, we know that e has shape $\lambda x_1 \ldots \lambda x_n.x^T@_{M_1}e_1 \ldots @_{M_m}e_m$. We prove by induction on the structure of e that $e = NF(e)$. By the Church-Rosser property for labelled reduction and by inspecting the definition of $\rhd_L$, we know that $NF(e) = \lambda x_1 \ldots \lambda x_n.x^T@_{M_1}e'_1 \ldots @_{M_m}e'_m$ and $e_i \rhd_L e'_i$ for each i. We also know that $\Gamma \vdash_L NF(e) : T$ is principally typed. By definition 5.8, we know that each M_i is a type, each $e'_i = merge_1(f'_i)$ and $\Gamma_i \vdash_L f'_i : M_i$ is principally typed for some context Γ_i. It is easy to see that $e_i \rhd_L merge_1(f'_i)$ implies $e_i = merge_1(f_i)$ and $f_i \rhd_L f'_i$. By definition 5.8, it follows that $\Gamma_i \vdash_L f_i : M_i$ is principally typed. Since f_i is in allowable normal form, we know by the induction hypothesis that $f_i = f'_i$ for each i, and hence that $e = NF(e)$. $\square$

5.10 Corollary Let $\Gamma \vdash_L e : T$ be principally typed such that e is a normalisable term. Then the leftmost allowable reduction is the leftmost reduction.

5.11 Theorem Let $\Gamma \vdash_L e : T$ be principally typed such that e is a normalisable term. Then every needed reduction beginning with e is an allowable reduction.

Proof Let $n : e \rhd_L f$ denote a needed reduction. By induction on the length of n, it is enough to show that the redex r contracted in the first step of n is an allowable redex. Redex r is needed by assumption, and hence it is used in the leftmost reduction path of e to $NF(e)$. By lemma 5.10, the leftmost reduction and the leftmost allowable reduction are the same. Hence redex r is an allowable redex. □

6 Conclusions and Future Work

We have advocated the technique of extracting extra information during type assignment to retain tight control of the use of variables. In particular, we have shown that techniques from intersection calculi, adapted using ideas about resource from linear and relevant logics, can be used to create the labelled resource calculus which achieves this tight control of variables. Using the labelled resource calculus, we have shown how to distinguish allowable redexes. We have proved that allowable redexes are needed redexes. Moreover, in the presence of principal typing for normalisable terms, all needed redexes are allowable. We believe that it should be possible to adapt these ideas to other type assignment systems. An important issue is to provide a principal typing algorithm for the labelled resource calculus by adapting the algorithm for intersection types [16] [17], and to investigate the complexity of such an algorithm.

We close by mentioning two potential applications of these results. The first is to strictness and sharing analysis of functional programming languages, and centres on the non-trivial notion of arguments being *used* [3]. Consider the term $\lambda f.\lambda x.f(f(x))$ which can be principally typed in the resource calculus to obtain an entailment of the form $\langle\ \rangle \vdash \lambda f.\lambda x.f(f(x)) : (T_2 \multimap T_3) \wedge (T_1 \multimap T_2) \multimap T_1 \multimap T_3$. The typing of this term indicates that variable f is used twice, as one would expect. The typing also indicates that any argument will be used in a non-trivial way. It does not, however, provide complete information regarding the use of an argument, since that information changes depending on the argument under consideration. For example with the application $(\lambda f.\lambda x.f(f(x)))(KI)$, the argument KI is used once, as is apparent when the term is reduced to normal form. This information can, in fact, be obtained from the typing since, when $\lambda f.\lambda x.f(f(x))$ is applied to KI, the principal typing of the function is forced to have shape $(\perp \multimap T_3) \multimap \perp \multimap T_3$. The notion of an argument being used is defined in [3] using a variant of Klop [13] labelling, and a similar notion of use is defined in [3]. Initial investigations suggest that the number of times an argument a is used in term $f(a)$, where f has principal type $M \multimap T$, corresponds to the length of M.

The second application is to searching for an optimal strategy for reducing λ-terms. Lévy defines optimal reductions to be the so-called "complete call-by-need reductions" [15]. Intuitively, complete reductions are reductions which involve the maximum amount of sharing. Lévy points out that the standard sharing mechanisms for graph reduction coupled with the leftmost reduction strategy (which ensures that the reduction is needed) does not always yield an optimal reduction. Lamping [14], and Gonthier, Abadi and Lévy [11] [12] translate λ-graphs to graphs which essentially capture Lévy's account of sharing. Using the leftmost reduction strategy for their graphs, they obtain an optimal reduction strategy, though this approach is extremely complex. We have shown how to identify precisely the needed reductions, and are therefore not restricted to the leftmost reduction strategy. An interesting challenge is to seek an optimal reduction strategy by combining our identification of needed reductions with the standard sharing mechanisms for graph reduction. Of course, our approach would only apply to principally typed terms, whereas the approach of Lamping, and of Gonthier, Lévy and Abadi applies to all λ-terms. In many applications, however, working with principally typed terms would not be a significant restriction.

Acknowledgements

My thanks go to Benjamin Pierce for many interesting discussions during the initial stages of this work, and for bringing to my attention the suggestion of Mariangiola Dezani-Ciancaglini of applying these ideas to intersection types. I would also like to thank John Reynolds for his help and encouragement. In particular, he suggested the notion of 3-dimensional graphs to aid in the motivation of the labelled resource calculus.

References

[1] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, 1984.

[2] H.P. Barendregt, J.R. Kennaway, J.W. Klop and M.R. Sleep. Needed Reduction and Spine Strategies for the Lambda Calculus, *Information and Computation*, Vol. 75, pp 191–231, 1987.

[3] C. Baker-Finch. *Relevance and Contraction: A Logical Basis for Strictness and Sharing Analysis*, Information Sciences Technical Report, University of Canberra, 1993.

[4] F. Cardone and M. Coppo. Two Extensions of Curry's Type Inference System, *Logic and Computer Science*, ed. P. Odifreddi, pp 19-75, 1990.

[5] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal Type Schemes and Lambda Calculus Semantics, in [18], 1980.

[6] A.J.T.Davie. *An Introduction to Functional Programming Systems using Haskell*, Cambridge University Press, 1992.

[7] J.M. Dunn. Relevant Logic and Entailment, *Handbook of Philosophical Logic*, eds. D. Gabbay and F. Guenthner, Vol. 3, pp 117–224, 1984.

[8] P.A. Gardner. *Discovering Needed Reductions Using Type Theory*, full paper, in preparation, 1993.

[9] P.A. Gardner. *Strictness and Sharing Analysis using Type Theory*, in preparation, 1993.

[10] J.Y. Girard. Linear Logic, *Theoretical Computer Science*, Vol. 50, pp 1-102, 1987.

[11] G. Gonthier, M. Abadi and J-J. Lévy. The Geometry of Optimal Lambda Reduction, *Ninteenth Annual ACM Symposium of Principles of Programming Languages*, pp 15-26, 1992.

[12] G. Gonthier, M. Abadi and J-J. Lévy. Linear Logic Without Boxes, *Logic in Computer Science*, 1992.

[13] J.W. Klop. Term Rewriting Systems, *Handbook of Logic in Computer Science*, Vol. 2, pp 1–116, 1992.

[14] J. Lamping. An Algorithm for Optimal Lambda Calculus Reduction, *Seventeenth Annual ACM Symposium on Principals of Programming Languages*, pp 16–30, 1990

[15] J-J. Lévy. *Optimal Reductions in the lambda calculus*, in [18], pp 159–192, 1980.

[16] S. Ronchi della Rocca. Principal Type Scheme and Unification for Intersection Type Discipline, *Theoretical Computer Science*, Vol. 59, pp 181–209, 1988.

[17] S. Ronchi della Rocca and B. Venneri. Principal Type Schemes for an Extended Type Theory, *Theoretical Computer Science*, Vol. 29, pp 151–209, 1984.

[18] J.P. Seldin and J.R. Hindley, editors. *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 1980.

[19] D.A. Turner. Miranda–a Non-strict Language with polymorphic types, *Functional Programming Languages and Computer Architecture*, Springer LNCS 201, pp 1–16, 1985.