Usage Analysis with Natural Reduction Types

David A. Wright
Department of Computer Science
University of Tasmania
Hobart 7001
AUSTRALIA

Clement A. Baker-Finch
Information Sciences and Eng.
University of Canberra
Canberra 2616
AUSTRALIA

Abstract

In a functional program the value of an expression may be required several times. If a *usage analysis* can determine how many times it will be required, certain optimisations are possible, such as converting *lazy* parameter passing to call-by-name or call-by-value, compile-time garbage collection and in-place update. This paper presents a method for deducing usage information in the λ-calculus, based on a type logic employing *reduction types*. A system is presented wherein function type constructors are annotated with expressions over the natural numbers to indicate the usage behaviour of λ-terms. This system is shown to be correct by interpreting the type language over a semi-model of the λ-calculus and demonstrating soundness and completeness. Furthermore, we show how the Curry-Howard interpretation naturally relates such types to relevant logic.

1 Introduction

Modern implementations of functional languages rely heavily on compile-time program transformation techniques to obtain acceptable run-time efficiency. In their turn, program transformation techniques depend on knowing detailed information about the program being transformed. Obtaining this information is the province of much ongoing research worldwide. The main approaches to obtaining information about programs can be categorised by the representation of this information, as follows: abstract lambda terms, see for example Cousot and Cousot [12], Mycroft [26], Burn et al [8] and Hudak and Young [20]; types, as used by Kuo and Mishra [25], Wright [32, 33, 34], Coppo [11] and Wadler [31]; and relations, see Mycroft and Jones [27]. In general, types have the advantage of being first-order notations which describe the higher-order properties of functions in a succinct and clear manner.

Reduction types ([32, 33, 34]) are a novel way of expressing various properties of λ-terms. The main idea is to distinguish between sets of terms based on an extended notion of type. In particular, an instance of reduction types is defined by associating a *separate function type constructor with each class of terms to be distinguished.* For example, functions strict on their argument might have a type constructor $\rightarrow_{\perp}$ associated with them, whereas functions lazy on their argument would then have a different

type constructor, say $\rightarrow_?$, so as to distinguish them from those functions definitely known to be strict. The main advantages of using the reduction type framework are that information about higher-order terms is captured very precisely (and very naturally) and that advantage may be taken of the well established implementation technologies existing for types and for unification (see Siekmann [30]).

In this paper we utilise the reduction type framework to develop an analysis of *usage* information in λ-terms. We define usage information for a redex subterm to be the number of times that copies of the redex are reduced during the head reduction of a term. This information is particularly useful to the optimisation phases of a compiler as it allows the compiler to implement "update of structures in place" and "compile-time garbage collection". In addition to the above, a discussion is given of how the logic of our instance of reduction types is related to a non-standard version of *relevant logic*, via the well known Curry-Howard isomorphism.

2 Preliminaries

The reader is assumed to be familiar with the basics of the λ-calculus (Barendregt [4]). Some possibly less familiar notions concerning the λ-calculus are outlined in this section.

β-reduction will be the main form of reduction considered, and so unless otherwise qualified "reduction" will mean β-reduction, "redex" will mean β-redex and "normal form" will mean β-normal form.

The concept of *head normal form* plays a central role in the λ-calculus, in particular, a term has a head normal form iff it is *solvable*, see Barendregt [4] pp41–42 for discussion. Associated with the notion of head normal form is a particular reduction strategy known as *head reduction*. Later, head reduction will be used to define a property called *strong head neededness*.

Definition 2.1 *A subterm N of a term M is at the* head *of M if*

- $M \equiv N$, *or*
- $M \equiv \lambda x.N'$ *and N is at the head of N', or*
- $M \equiv N_1 N_2$ *and N is at the head of N_1.*

Suppose M is not in normal form. The *leftmost* redex of M is the redex whose binder is to the left of the binder of every other redex in M. The leftmost redex, R, of M is a *head* redex of M if R is at the head of M. M is in *head normal form* if it has no head redex. The *head reduction path* of a term M is a sequence of reduction steps in which every redex which is reduced is a head-redex.

Suppose $M \twoheadrightarrow_\beta N$, then the *descendants* of some subterm M' of M can be found in N (if any exist), by marking M' and following it through the reduction from M to N. Each time a marked redex is reduced during this reduction its mark is eliminated. (A more formal definition is available on page 19 of Klop [24]).

Any descendent of a redex, R, is itself a redex, and is called a *residual*. This is so since if R is contracted in $M \rightarrow_\beta N$, then clearly R has no descendants in N.

3 Strong Heed Neededness

Strong head neededness is a variation of the idea of *head neededness* introduced by Barendregt et al [6]. (Barendregt et al [6] show that head neededness and strictness are equivalent concepts).

Definition 3.1 (Barendregt et al [6]) *Suppose R is a redex of M, then R is* head needed *in M if every reduction path of M to head normal form reduces a residual of R.*

Our variation on this concept is that we insist that a (sub)term is contracted in order for it to be considered to be strongly head needed. In contrast, all redex subterms of a term without head normal form are considered to be head needed by Barendregt et al [6]. Thus, strong head neededness gives more detailed information than head neededness about terms which have no head normal form. This has a practical impact on the implementation of functional languages on parallel machines as it reduces the possibility that a non-terminating term will be selected for evaluation—thus moderating the chance that the machine may become overloaded with the computation of non-terminating tasks (this is further discussed in Wright [34]). Strong head neededness can be defined as follows:

Definition 3.2 *Suppose R is a redex subterm of $M \in \Lambda$, then R is* strongly head needed *in M if the head reduction path of M reduces a residual of R.*

In this paper we describe an extension to this idea of strong head neededness. This extension determines more detailed information than the previous definition by recording the *precise number of times* that the residuals of a redex are contracted. This is exactly the information which a usage analysis of terms seeks to deduce.

Definition 3.3 *Let R be a redex subterm of M. R is a* strongly head needed redex of degree *n in M ($n \geq 0$) if exactly n residuals of R are reduced on the head reduction path of M. We say that R is an* irrelevant *redex of M if R is a strongly head needed redex of degree 0 in M.*

4 Reduction Types

Reduction types are a class of notations for specifying the reduction behaviour of λ-terms (Wright [34]). These notations are all a form of type in which a *set* of function type constructors is introduced to depict a particular kind of reduction behaviour.[1] The key idea can thus be summarised by the phrase "functions should be classified by *function type constructors*". For example, in [32, 33] *boolean reduction types* were introduced to describe strong head neededness information. Boolean reduction types are built from a Boolean algebra of function type constructors in which the "true" function type constructor (written as $\Rightarrow$) constructs sets of functions which strongly head need their argument.[2] In contrast, the "false" function type constructor (written as $\nrightarrow$) builds sets of functions which ignore their argument. Thus the term $\lambda x.x$ would be assigned a type of the form $\sigma \Rightarrow \sigma$, where σ is any boolean reduction type. Similarly, the term $\lambda xy.x$ would be assigned a type of the form $\sigma \Rightarrow \tau \nrightarrow \sigma$, for any types σ and τ.

[1] Typically, the set of function type constructors will form an algebra.

[2] For a complete description of the semantics of boolean reduction types, see Wright's thesis.

Higher-order terms may also be treated in this manner. For these, dummy variables must be introduced, as is illustrated by the following example. Consider the term $\lambda fx.fx$, in which a term variable is treated as a function. Writing dummy variable function type constructors ("arrow variables") as $\rightarrow_i$, $\rightarrow_j$ and $\rightarrow_k$, we can express the type of this term as being:

$$(\sigma \rightarrow_i \tau) \Rightarrow \sigma \rightarrow_i \tau,$$

for any types σ and τ. Such arrow variables are intended to be understood as being implicitly universally quantified over the algebra of ground function type constructors. Of course, under the standard laws for any Boolean algebra there are only two such ground terms, namely the values "true" and "false" (here represented by $\Rightarrow$ and $\nrightarrow$).

In general, an algebra of function type constructors must be introduced to describe the behaviour of terms. This necessitates the introduction of *function type constructor builders*. For example, an appropriate type for the $\mathbf{S} \equiv \lambda fgx.fx(gx)$ combinator is

$$(\rho \rightarrow_i \sigma \rightarrow_j \tau) \Rightarrow (\rho \rightarrow_k \sigma) \rightarrow_j \rho\ (\rightarrow_i \vee\ (\rightarrow_j \wedge \rightarrow_k))\ \tau.$$

See [34] for an in depth investigation of several systems based on boolean reduction types.

In this paper we extend boolean reduction types to *natural reduction types* which use an algebra of function type constructors over the natural numbers (rather than the truth values as in boolean reduction types). This allows us to express strong head neededness *of degree n* information. Then by simply following the framework of Wright we produce a system of logic for deducing natural reduction types for λ-terms.

4.1 Natural Reduction Types

In natural reduction types the exact number of uses of a subterm are kept, rather than just "none" ($\nrightarrow$) or "more than zero" ($\Rightarrow$). This corresponds to the *use-count* generalisation of strictness analysis (see Sestoft [29], Jensen and Mogensen [23] and Goldberg [15]). The set of function type constructors that we use is now defined.

Definition 4.1 *Let $\Delta_v = \{\rightarrow_i, \rightarrow_k, \rightarrow_j, \ldots\}$ be a sufficiently large set of* arrow variables*, where $i, j, k, \ldots$ are dummy variables over the natural numbers, and let $\Delta_g = \{\rightarrow_0, \rightarrow_1, \ldots\}$ be a set of* ground arrows*, (one for each natural number). Now some operators over sets of arrows are defined. Let $\rightarrow_i + \rightarrow_j = \rightarrow_{i+j}$ and $\rightarrow_i \times \rightarrow_j = \rightarrow_{i \times j}$. In the right-hand sides of these definitions $+$ is ordinary addition and $\times$ is ordinary multiplication (both operators are defined only over the natural numbers). The set of* natural arrow expressions *is the algebra of arrows generated by $\times$ and $+$ over $\Delta = \Delta_g \cup \Delta_v$. Let the meta-variables over arrow expressions be $\mathrm{b}, \mathrm{b}', \ldots$. Let the meta-variables for elements of Δ_g be $\rightarrow_m, \rightarrow_n, \ldots$. Occasionally the multiplication symbol ($\times$) will be replaced by juxtaposition.*

The intention is that irrelevance is now represented by the ground arrow $\rightarrow_0$ and the various other degrees of strong head neededness are represented by the arrows $\rightarrow_1, \rightarrow_2, \ldots$.

Various sets of types may now be constructed from these function type constructors. In this paper we consider the set of *intersection natural reduction types*. (Intersection types are described in several places, for example, Coppo [10] and Barendregt et al [5]).

Definition 4.2 *Let τ_v denote a set of* type variables *with sufficiently many elements. The set of* intersection natural reduction types *is the set T_I defined by:*

1. *$\alpha \in \tau_v$ implies $\alpha \in T_I$, and*
2. *$\sigma, \tau \in T_I$ and $\mathfrak{b} \in \Delta$ implies $\sigma \,\mathfrak{b}\, \tau \in T_I$ and $\sigma \cap \tau \in T_I$.*

As in Barendregt et al [5], it is natural to introduce an ordering on the types which intuitively corresponds to a notion of subset.

Definition 4.3 1. *The relation $\leq$ on T_I is inductively defined to be the least relation satisfying: $\tau \leq \omega$; $\omega \leq \omega \,\mathfrak{b}\, \omega$; $\tau \leq \tau$; $\tau \leq \tau \cap \tau$; $\sigma \cap \tau \leq \sigma$, $\sigma \cap \tau \leq \tau$; $(\sigma \,\mathfrak{b}\, \rho) \cap (\sigma \,\mathfrak{b}\, \tau) \leq \sigma \,\mathfrak{b} (\rho \cap \tau)$; $\sigma \leq \tau \leq \rho$ implies $\sigma \leq \rho$; $\sigma \leq \sigma'$, $\tau \leq \tau'$ implies $\sigma \cap \tau \leq \sigma' \cap \tau'$; and $\sigma \leq \sigma'$, $\tau \leq \tau'$ implies $\sigma' \,\mathfrak{b}\, \tau \leq \sigma \,\mathfrak{b}\, \tau'$.*

2. *$\sigma = \tau$ iff $\sigma \leq \tau \leq \sigma$.*

As stated in Barendregt et al [5], the pre-order $\leq$ defined above is a partial order when the set of types, T_I, is factored by $=$. In the following it will be assumed that T_I is indeed factored by $=$.

4.2 A Type Deduction System: The Intersection System

A *type assumption* is a statement of the form $x : \tau$, where $x \in X$ and $\tau \in T_I$. An *assumption set* is simply a set of type assumptions, with the restriction that no term variable occurs more than once in the assumption set. The letter A (possibly with subscripts) will be used to denote an arbitrary assumption set. Write A_x for the assumption set equal to A except that any occurrence of a type assumption containing the variable x in A is removed.

In order to construct a type assignment system the notion of *variable strong head neededness function* is required. A variable strong head neededness function (or, for conciseness, *variable neededness function*) is a function of type $X \to \Delta$, which denotes to what degree a free term variable is strongly head needed in a term (that is, the degree to which the term resulting by abstracting that free variable strongly head needs its argument). Variable neededness functions are denoted by the letter V (possibly with subscripts). Let V_0 denote the variable neededness function with the property that $\forall x \in X. V_0(x) = \rightarrow_0$. For example, the expression x should have the variable neededness function $V_0[x := \rightarrow_1]$ assigned to it, since $\lambda x.x : \sigma \rightarrow_1 \sigma$ and for all variables $y \not\equiv x$, $\lambda y.x : \tau \rightarrow_0 \sigma$.

A *typing statement* is a quadruple of an assumption set A, a variable neededness function V, a term M, and a type τ. A typing statement will be written as $A \vdash^I_V M : \tau$. (The I reminds us that the reduction types employed are *intersection* natural reduction types). The term logic for our system appears as Figure 1.

4.2.1 An Example

Consider the term $Twice \equiv \lambda f x.f(fx)$. This term has the following deduction using the type assignment system of Figure 1. Let $A = \{f : (\alpha \rightarrow_i \beta) \cap (\beta \rightarrow_j \gamma), x : \alpha\}$, then using VAR and LEQ we obtain deductions of: $A \vdash^I_{V_0[x := \rightarrow_1]} x : \alpha$, $A \vdash^I_{V_0[f := \rightarrow_1]} f : \alpha \rightarrow_i \beta$ and $A \vdash^I_{V_0[f := \rightarrow_1]} f : \beta \rightarrow_j \gamma$.

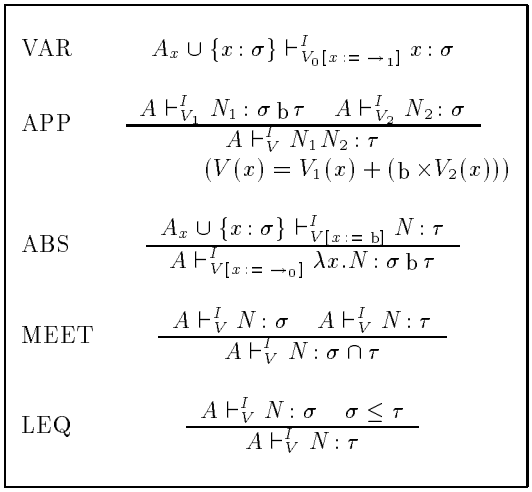

Figure 1: The Rules for deducing Intersection Natural Reduction Types

Now two instances of the APP rule apply. The interesting part of this is the construction of the variable neededness functions in these two instances of APP. In the deduction of a type for fx the calculation is, for f: $\rightarrow_1 + (\rightarrow_i \times \rightarrow_0) = \rightarrow_{1+i\times 0} = \rightarrow_1$, and for x: $\rightarrow_0 + (\rightarrow_i \times \rightarrow_1) = \rightarrow_{0+i\times 1} = \rightarrow_i$. In the deduction of a type for $f(fx)$, the calculations are, for f: $\rightarrow_1 + (\rightarrow_j \times \rightarrow_1) = \rightarrow_{1+j\times 1} = \rightarrow_{j+1}$, and for x: $\rightarrow_0 + (\rightarrow_j \times \rightarrow_i) = \rightarrow_{0+i\times j} = \rightarrow_{i\times j}$. Consider the final type for *Twice*:

$$((\alpha \rightarrow_i \beta) \cap (\beta \rightarrow_j \gamma)) \rightarrow_{j+1} \alpha \rightarrow_{i\times j} \gamma.$$

Firstly, notice that the strong head neededness of f is of non-zero degree. Moreover, if the first occurrence of f in $f(fx)$ strongly head needs fx a total of j times then f is strongly head needed a total of $j+1$ times—once for the first occurrence of f in $f(fx)$ and j times for each use of the second occurrence of f in $f(fx)$.

Similarly, the strong head neededness given for x reflects the intuition that *both* occurrences of f must strongly head need x in order for x to be strongly head needed in the overall expression.

5 An Isomorphism with Relevant Logic

The aim here is to establish a logical foundation for the term logic employing reduction types by applying the Curry-Howard isomorphism [19] to *relevant logic* [1, 14]. The implicational fragment of relevant logic is denoted $R_{\rightarrow}$. This system was also studied by Church [9] who called it *weak implicational logic*. It turns out that the '$\rightarrow$' of relevant logic corresponds to the '$\Rightarrow$' of boolean reduction types. Without trying to be precise about terminology, our claim is that relevance types are a form of reduction type.

The reasons for expecting some connections between relevance and neededness can be outlined as follows. The essence of relevant logic is that, in deducing $A \rightarrow B$,

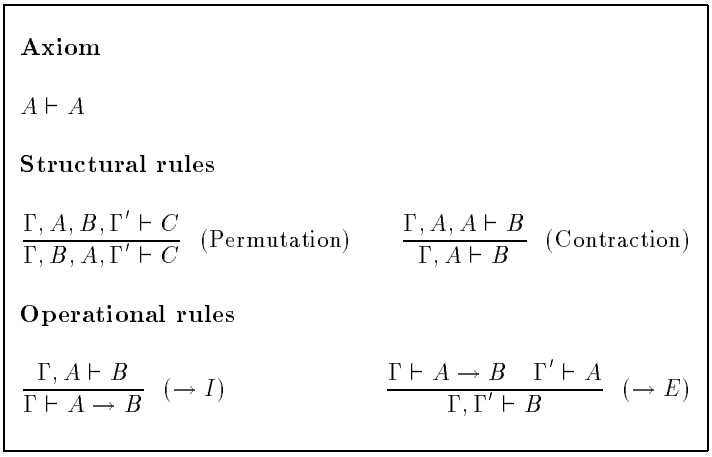

Figure 2: Natural Deduction System for $R_{\rightarrow}$

the antecedent A *really is used in* (and hence is *relevant* to) the derivation of the consequent B. The idea, generally attributed to Heyting, that a proof of $A \rightarrow B$ is some 'function' f taking each proof p of A to some proof $f(p)$ of B, can also be applied to relevant logic. However, for f to qualify as a relevant proof of $A \rightarrow B$, f must use the proof p of A; that is, f *must depend on its argument.*

Type assignment systems are generally given as formulations corresponding to natural deduction. In relevant logic, as in all other 'resource' logics, the structural rules are of paramount importance and must be explicitly stated. Figure 2 is a natural deduction system for $R_{\rightarrow}$, in sequent form.

It is reasonably well-known that the terms with relevant types are exactly the Curry typable subset of the λI-terms. It is easily demonstrated that the $R_{\rightarrow}$-typable terms are the 'hereditarily strict' ones, recalling the close correspondence with λI-calculus. That is, no sub-term of a hereditarily strict λ-expression is erasable [17, 16].

Relevant types alone contribute little to our enterprise since the aim of the analysis is to *distinguish* functions which strongly head need their arguments from constant functions, whereas the effect above is to merely *exclude* the constant ones.

The Deduction Theorem for relevant logic [9] is as follows: if $A_1, A_2, \ldots, A_n \vdash B$, then either $A_1, A_2, \ldots, A_{n-1} \vdash A_n \supset B$ or $A_1, A_2, \ldots, A_{n-1} \vdash B$. This suggests the concept of a deduction being *relevant with respect to a given hypothesis.* Dunn [14] formalises this concept by 'tagging' a hypothesis, say with a #, and passing the tag along with each application of modus ponens. Then, if the tag turns up on the conclusion, the deduction is relevant with respect to that hypothesis. This idea of tagging carries over to natural deduction and allows us to distinguish relevant and *irrelevant* hypotheses. By then permitting either tagged or untagged items to be discharged, we may distinguish relevant and irrelevant implications, respectively denoted $\Rightarrow$ and $\nrightarrow$. Thus there are two conditional proof rules and two versions of modus ponens. The resulting system is denoted $R_{\nrightarrow}$ and dubbed *irrelevant logic.* Fundamentally, the difference is that in $R_{\nrightarrow}$ weakening is permitted but it introduces only untagged hypotheses. This system closely corresponds to boolean reduction

types [3]. In the term logic using reduction types, the tagging is denoted by the variable head-neededness function V, presented separately from the assumption set.

As indicated earlier in section 4, the extension from neededness to a usage analysis is straightforward and the logical framework extends in a corresponding way. The contraction rule is the key here. To use two identical hypotheses and then discharge them together requires that they first be contracted to a single hypothesis. This indicates more than one occurrence of the bound variable in the body of a λ-expression. However, to get information about *usage*, relevance must also be taken into account. For example, $\lambda x.\mathbf{K}xx$ corresponds to a proof where the hypothesis labelled x is contracted before being discharged. Nevertheless only one of those is relevant so only one is *used*.

This leads to the following decisions regarding the monitoring of usage in proofs. The natural deduction system is given in Figure 3. First, each relevant use of a hypothesis attracts a use-count annotation of 1. This is handled in the axiom. Second, each weakening introduces an irrelevant hypothesis, hence it is tagged with a 0. Finally, contraction takes the *sum* of the annotations on the two hypotheses and, on discharging a hypothesis the annotation is transferred to the introduced arrow. The notation $m \times \Gamma$ indicates that all annotations in Γ are to be multiplied by m.

Axiom

$$A^1 \vdash A$$

Structural rules

$$\frac{\Gamma, A^m, B^n, \Gamma' \vdash C}{\Gamma, B^n, A^m, \Gamma' \vdash C} \text{ (Permutation)} \qquad \frac{\Gamma, A^m, A^n \vdash B}{\Gamma, A^{m+n} \vdash B} \text{ (Contraction)}$$

$$\frac{\Gamma \vdash A}{\Gamma, B^0 \vdash A} \text{ (Weakening)}$$

Operational rules

$$\frac{\Gamma, A^m \vdash B}{\Gamma \vdash A \rightarrow_m B} \ (\rightarrow I) \qquad \frac{\Gamma \vdash A \rightarrow_m B \quad \Gamma' \vdash A}{\Gamma, m \times \Gamma' \vdash B} \ (\rightarrow E)$$

Figure 3: Natural Deduction System, monitoring uses

By now permitting dummy variables as arrow annotations in the hypotheses we get a logic which, via the Curry-Howard interpretation, corresponds to the term logic of Figure 1 (without the rules MEET and LEQ).

6 A Semantics for Natural Reduction Types

The unique aspect of the semantics for reduction types is the need to look at the behaviour of a function as it is transformed by applying it to a sequence of arguments.

Both the number of arguments and the properties of these arguments are used to collect together functions into the interpretation of types. Thus the semantics for reduction types is *context-sensitive*, a fact which we feel makes them of particular interest.

6.1 Semi-models of the λ-calculus

Plotkin [28] has introduced the notion of *semi-model* in order to conduct a semantic analysis of Curry's original system of F-deducibility [13]. The key step in this work is the emphasis on modelling *reduction* rather than conversion. In order to avoid introducing an EQ rule in the manner of Hindley [18] we will follow Plotkin's approach to interpreting terms.

A $\lambda\beta$-semi-model of the λ-calculus is a triple, $\langle D, \bullet, [\![\cdot]\!] \rangle$, where D is a partial order, $\bullet : D \times D \to D$ is a map called *application* and $[\![\cdot]\!]$ is another map which assigns each term $M \in \Lambda$ to an element $[\![M]\!]_\mu \in D$, for an environment $\mu : X \to D$. The ordering on D will be written as $\leq$ and for two environments μ and μ' write $\mu \leq \mu'$ iff $\forall x \in X . \mu(x) \leq \mu'(x)$. Furthermore, the map $[\![\cdot]\!]$ must satisfy, for each $M \in \Lambda$ and each environment $\mu : X \to D$, the following list of properties (Plotkin [28]):

1. $[\![x]\!]_\mu = \mu(x)$,
2. $[\![MN]\!]_\mu = [\![M]\!]_\mu \bullet [\![N]\!]_\mu$,
3. $[\![\lambda x.M]\!]_\mu \bullet d \leq [\![M]\!]_{\mu[x := d]}$, for every $d \in D$,
4. if $[\![x]\!]_\mu = [\![x]\!]_{\mu'}$ for all $x \in \mathrm{FV}(M)$, then $[\![M]\!]_\mu = [\![M]\!]_{\mu'}$,
5. if $y \notin \mathrm{FV}(M)$, then $[\![M[x := y]]\!]_\mu = [\![M]\!]_{\mu[x := \mu(y)]}$, and
6. if $\forall d \in D . [\![M]\!]_{\mu[x := d]} \leq [\![N]\!]_{\mu[x := d]}$, then $[\![\lambda x.M]\!]_\mu \leq [\![\lambda x.N]\!]_\mu$.

Theorem 6.1 (Plotkin [28]) *$M \twoheadrightarrow_\beta N$ iff in all semi-models and for every environment μ, $[\![M]\!]_\mu \leq [\![N]\!]_\mu$*

Semi-models do indeed model the syntactic process of reduction (but not conversion). This means that they are an appropriate choice for testing the correctness of the information deduced by our term logic. Furthermore, Plotkin's semi-models allow the type deduction systems to be shown to give *complete* information with respect to reduction.

6.2 A Semantic Notion of Strong Head Neededness

The following definition provides a semantic analogue of the syntactic notion of strong head neededness of degree m.

Definition 6.2 *Let $\langle D, \bullet, [\![\cdot]\!] \rangle$ be a semi-model of the λ-calculus. For any interpretation of term variables μ, write $[\![M]\!]_\mu$ strongly head needed to degree m in $[\![N]\!]_\mu$ if*

$$\left.\begin{array}{l} [\![N]\!]_\mu \\ < [\![\lambda x_1 x_2 \ldots x_{n_1} . M N_1 \ldots N_{k_1}]\!]_\mu \\ < [\![\lambda y_1 y_2 \ldots y_{n_2} . M P_1 \ldots P_{k_2}]\!]_\mu \\ \qquad\qquad \vdots \\ < [\![\lambda z_1 z_2 \ldots z_{n_m} . M Q_1 \ldots Q_{k_m}]\!]_\mu \end{array}\right\} m \text{ times.}$$

$$\mathcal{I}[\![\cdot]\!] : T_I \to \mathbf{TEnv} \to \mathcal{P}(D)$$
$$\mathcal{I}[\![\sigma]\!]_\nu = \bigcap_{\sigma_i \in G_I(\sigma)} \mathcal{G}_I[\![\sigma_i]\!]_\nu$$

$$\mathcal{G}_I[\![\cdot]\!] : T_I \to \mathbf{TEnv} \to \mathcal{P}(D)$$
$$\mathcal{G}_I[\![\sigma \cap \tau]\!]_\nu = \mathcal{G}_I[\![\sigma]\!]_\nu \cap \mathcal{G}_I[\![\tau]\!]_\nu$$
$$\mathcal{G}_I[\![\alpha]\!]_\nu = \nu(\alpha)$$
$$\mathcal{G}_I[\![\sigma_0 \to_m \sigma_1 \, \mathrm{b}_1 \, \sigma_2 \ldots \sigma_n \, \mathrm{b}_n \, \alpha]\!]_\nu$$
$$= \{d \in D | \forall d_i \in \mathcal{G}_I[\![\sigma_i]\!]_\nu, 0 \le i \le n.$$
$$d_0 \text{ strongly head needed to degree } m \text{ in } d \bullet d_0 \bullet d_1 \bullet \ldots \bullet d_n;$$
$$d \bullet d_0 \in \mathcal{G}_I[\![\sigma_1 \, \mathrm{b}_1 \, \sigma_2 \ldots \sigma_n \, \mathrm{b}_n \, \alpha]\!]_\nu\}$$

Figure 4: The Semantics of Intersection Natural Reduction Types

Theorem 6.3 *$[\![M]\!]_\mu$ strongly head needed to degree m in $[\![N]\!]_\mu$ iff M strongly head needed to degree m in N.*

6.3 What is a Type Interpretation?

A type interpretation is a pair $\mathcal{T}y = \langle \mathrm{Ty}, [\![\cdot]\!] \rangle$, where Ty is a set of subsets of the domain of the chosen (semi-) model of terms which satisfies certain closure properties and $[\![\cdot]\!]$ is an interpretation function from types and type environments to elements of Ty.

The interpretation of types is given modulo the interpretation of type variables. A *type environment* is a function ($\in \tau_v \to \mathcal{P}(D)$) which maps each type variable to an element of the (semi-) model of terms. The letter ν is used to range over type environments.

Definition 6.4 *Let $\mathcal{M}$ denote a model or semi-model of terms. The notion of* semantic satisfaction *can be defined in the following manner:*

1. *$\rho, \nu, \mathcal{M}, \mathcal{T}y \models M : \sigma$ if $[\![M]\!]_\rho \in [\![\sigma]\!]_\nu$,*
2. *$\rho, \nu, \mathcal{M}, \mathcal{T}y \models A$ if for each $x : \sigma \in A$, $\rho, \nu, \mathcal{M}, \mathcal{T}y \models x : \sigma$, and*
3. *$A \models_V M : \sigma$ if $\rho, \nu, \mathcal{M}, \mathcal{T}y \models A$ implies $\rho, \nu, \mathcal{M}, \mathcal{T}y \models M : \sigma$ and $\forall x \in X. \rho, \nu, \mathcal{M}, \mathcal{T}y \models \lambda x.M : A(x) \, V(x) \, \sigma$.*

6.4 The Semantics of Natural Reduction Types

Definition 6.5 *Write $\sigma \trianglelefteq_I \tau$ if there is a substitution, R, of arrow expressions for arrow variables homomorphically extendible such that $\tau = R(\sigma)$. Let $G_I(\sigma) = \{\tau | \sigma \trianglelefteq_I \tau; \tau \in T_I\}$.*

Figure 4 contains the semantics of intersection natural reduction types. Note that this covers all cases as $\rho \, \mathrm{b} (\sigma \cap \tau) = (\rho \, \mathrm{b} \, \sigma) \cap (\rho \, \mathrm{b} \, \tau)$.

Lemma 6.6 *$\sigma \le \tau$ implies $\forall \nu. \mathcal{I}[\![\sigma]\!]_\nu \subseteq \mathcal{I}[\![\tau]\!]_\nu$.*

Let $\mathrm{Ty} = \mathcal{P}(D)$, where D is the domain of a semi-model of terms, and $\mathcal{I}[\![\cdot]\!]$ be as defined in Figure 4, then the following Theorem holds.

Theorem 6.7 (Correctness) *$A \vdash_V^I M : \tau$ iff $A \models_V M : \tau$.*

7 Conclusion and Related Work

We have introduced an original method for reasoning about *usage* in the λ-calculus. This method is based on a new instance of Wright's reduction type framework and employs a polynomial algebra over the natural numbers to deduce type expressions which describe the usage behaviour of terms. We have presented a term logic for these types and shown this logic to be semantically correct by first interpreting natural reduction types over a semi-model of the λ-calculus and then providing a correctness proof. Furthermore, we have shown that a natural extension of *relevant logic* can be related to our term logic via the Curry-Howard isomorphism.

During the development of this work, we became aware of Wadler's investigation of the use of linear logic as a basis for sharing analysis (see Wadler [31]). Wadler introduces *linear types* in which the modal '!' operator of linear logic is interpreted as indicating a shared argument. The relation between relevant and linear logic is well-known (for example [2]) so a comparison of natural reduction types with linear types seems appropriate.

At the level of the logics, the only difference is that linear logic disallows contraction except in the scope of a '!'. At the level of the types themselves, it is clear that natural reduction types contain strictly greater information than linear types, since the linear function type constructor (on its own) corresponds to the type constructor $\rightarrow_1$ and the shared argument operator '!' (which is always used in conjunction with the linear function type constructor) is modelled by the use of a variable arrow.

The basic shortcoming with linear types is that '!' indicates a non-linear type, thus confusing *sharing* with *absence*, though there has been recent work on separating '!' into relevant and affine versions [21]. Also, through the promotion and dereliction rules, even linear arguments can be annotated with '!'. In fact, at best '!' indicates several *occurrences* rather than several uses, as in $\lambda x.\mathbf{K}xx : !\sigma \multimap \sigma$. In comparison, the natural reduction type for this term is $\sigma \rightarrow_1 \sigma$, correctly indicating that the argument is *used* exactly once (ideally the linear type should be $\sigma \multimap \sigma$). It also appears that the promotion and dereliction rules complicate the notion of principal type.

Less closely related to our work is the work of Kuo and Mishra [25] and the similar method of Coppo [11]. These works are based on introducing certain non-standard constants to represent the set of strict functions and the set of all functions, and then constructing types over these constants using the standard function type constructor. Of course, no attempt is made in these works to treat the generalisation to usage information, as has been done for reduction types in this paper. However, their approach also seeks to capitalise on type inference technology in its implementation. In addition, Benton [7] and Jensen [22] provide a discussion of the relationship of the approach of Kuo and Mishra with that of the abstract interpretation method of Burn et al [8].

References

[1] A. R. Anderson and N. D. Belnap, Jr. *Entailment: The Logic of Relevance and Necessity.* Princeton University Press, 1975.

[2] A. Avron. The semantics and proof theory of linear logic. *Theoretical Computer Science*, 57:161–184, 1988.

[3] C. A. Baker-Finch. Relevant logic and strictness analysis. In *Workshop on Static Analysis, LaBRI, Bordeaux*, pages 221–228. Bigre 81–82, 1992.

[4] H.P. Barendregt. *The Lambda-Calculus: its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, second edition, 1984.

[5] H.P. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 48(4):931–940, December 1983.

[6] H.P. Barendregt, J.R. Kennaway, J.W. Klop, and M.R. Sleep. Needed Reduction and Spine Strategies for the lambda-calculus. Technical report, Centre for Mathematics and Computer Science, May 1986.

[7] P.N. Benton. Strictness Logic and Polymorphic Invariance. In *Logical Foundations of Computer Science*, pages 33–44, 20–24 July 1992.

[8] G.L. Burn, C.L. Hankin, and S. Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[9] A. Church. The weak theory of implication. In Menne, Wilhelmy, and Angsil, editors, *Kontrolliertes Denken, Untersuchungen zum Logikkalkül und der Logik der Einzelwissenschaften*, pages 22–37. Kommissions-verlag Karl Alber, 1951.

[10] M. Coppo. An Extended Polymorphic Type System for Applicative Languages. In *Mathematical Foundations of Computer Science*, number 88 in Lecture Notes in Computer Science. Springer-Verlag, September 1980.

[11] M. Coppo. Type Inference, Abstract Interpretation and Strictness Analysis. Draft manuscript, 1992.

[12] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[13] H.B. Curry and R. Feys. *Combinatory Logic, Volume 1.* Studies in Logic and the Foundations of Mathematics. North-Holland, 1958.

[14] J. M. Dunn. Relevance logic and entailment. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Vol. III.* D. Reidel, 1986.

[15] B. Goldberg. Detecting Sharing of Partial Applications in Functional Programs. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[16] G. Helman. Completeness of the normal typed fragment of the λ-system U. *Journal of Philosophical Logic*, 6:33–46, 1977.

[17] G. Helman. *Restricted Lambda Abstraction and the Interpretation of Some Non-Classical Logics*. PhD thesis, University of Pittsburgh, 1977.

[18] J.R. Hindley. The Completeness Theorem for Typing λ-terms. *Theoretical Computer Science*, 22:1–17, 1983.

[19] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatorial Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[20] P. Hudak and R.Young. Higher-order strictness analysis in untyped lambda calculus. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 97–109. ACM, 1986.

[21] Bart Jacobs. Semantics of weakening and contraction. Technical report, Department of Pure Mathematics, University of Cambridge, 1992.

[22] T.P. Jensen. Strictness Analysis in Logical Form, 1991.

[23] T.P. Jensen and T.A. Mogensen. A Backwards Analysis for Compile-time Garbage Collection. In N.D. Jones, editor, *European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[24] J.W. Klop. *Combinatory Reduction Systems*. PhD thesis, State University of Utrecht, 1980.

[25] T-M. Kuo and P. Mishra. Strictness Analysis: A New Perspective based on Type Inference. In *FPCA'89*, pages 260–272, London, United Kingdom, September 1989.

[26] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*, pages 269–281. Springer-Verlag, 1980.

[27] A. Mycroft and N. Jones. A relational framework for abstract interpretation. In *Workshop on Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 156–171. Springer-Verlag, 1985.

[28] G.D. Plotkin. A Semantics for Type Checking. In *Theoretical Aspects of Computer Science*, volume 526 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 1991.

[29] P. Sestoft. *Analysis and Efficient Implementation of Functional Programs*. PhD thesis, DIKU, University of Copenhagen, 1991.

[30] J.H. Siekmann. Unification Theory. *Journal of Symbolic Computation*, 7:207–274, 1989.

[31] P. Wadler. Is there a use for Linear Logic? In *Partial Evaluation and Program Manipulation*. ACM Press, 1991.

[32] D.A. Wright. Strictness Analysis Via (Type) Inference. Technical Report TR89-3, University of Tasmania, September 1989.

[33] D.A. Wright. A New Technique for Strictness Analysis. In *Theory and Practice of Software Development*, number 494 in Lecture Notes in Computer Science. Springer-Verlag, April 1991.

[34] D.A. Wright. *Reduction Types and Intensionality in the Lambda-Calculus*. PhD thesis, University of Tasmania, 1992.