Technical Report

UCAM-CL-TR-267
ISSN 1476-2986

Number 267

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Untyped strictness analysis

Christine Ernoult, Alan Mycroft

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Untyped strictness analysis

Christine Ernoult and Alan Mycroft
Computer Laboratory, Cambridge University
New Museums Site, Pembroke Street
Cambridge CB2 3QG
United Kingdom
E-mail: ce@cl.cam.ac.uk, am@cl.cam.ac.uk

Abstract

We re-express Hudak and Young's higher-order strictness analysis for the untyped λ-calculus in a conceptually simpler and more semantically-based manner. We show our analysis to be a sound abstraction of Hudak and Young's which is also complete in a sense we make precise.

Background

Untyped strictness analysis is currently a little out of vogue. There are two reasons for this. One is that the standard reference [3] is presentationally hard to read and, as we show, is complicated by spurious domain elements. The other is that most of the functional programming world uses some form of typed (typically simple polymorphic) λ-calculus. Strictness analysis for such languages benefits from the simple exposition of the Imperial College stable and various finiteness properties seemingly associated by the decidability of type inference.

However, some properties of (*e.g.*) the second-order polymorphic λ-calculus are best proved by appeal to untyped results and, as yet, we know of no polymorphic invariance properties which allow lifting of results for simple types.

It is with this interest in such strictness analysis that we give a more fundamental explanation of the ideas in the untyped λ-calculus which both better explains the theory and encourages its use as a basis for such extended analyses.

We discuss the treatment of domain errors which influence strictness. In particular, it is common to wish errors to give non-$\perp$ values (exceptions) in an untyped language, but when we see the untyped language used as an underlying implementation of a typed language such as the 2nd-order λ-calculus then (unobtainable) domain errors should be treated as $\perp$ to ease strictness analysis.

1 Introduction

Strictness analysis was originated by Mycroft [8] for the first-order case over flat domains, using a formalism based on abstraction and concretisation functions.

Temporarily, suppose that D is a flat cpo. Let 2 stand for the the set $\{0,1\}$ ordered by $0 < 1$. Recall that $f : D^n \to D$ is *strict* in its k-th argument if $(\forall \overline{x} \in D^n) f(x_1, \ldots, x_{k-1}, \bot, x_{k+1}, \ldots, x_n) = \bot$. Mycroft developed a strictness theory for first order functions on flat domains which gave a standard interpretation of a program user-defined function symbol (say `f`) as a function f as above and also non-standard interpretation $f^\sharp : 2^n \to 2$. Such $f^\sharp$ satisfy a correctness property with respect to f along the lines of $f^\sharp(1, \ldots, 1, 0, 1, \ldots, 1) = 0 \Rightarrow f$ is strict in its k-th argument. This property is respected by composition and fixpoint extraction and so lifts from base functions to user-defined functions.

Burn, Hankin and Abramsky [1] showed that the *Hoare* (or *relational*) powerdomain could be used to generate a theory of strictness analysis for the *simply typed* λ-calculus. (Their system abstracts functions between concrete domains with functions between abstract domains).

Around the same time Hudak and Young [3] gave a definition of *strictness pairs* which enabled them to analyse the *untyped* λ-calculus. They observed that an expression has not only a "direct strictness" (the set of variables which are evaluated when it is), but also a "delayed strictness" (the set of variables which are evaluated when the expression is applied). They suggested that the strictness property should perhaps be captured by an object, the domain of strictness pairs *Sp* defined by:

$$Sp = \mathcal{P}(V) \times (Sp \to Sp)$$

where V is the set of variable names and $\mathcal{P}(V)$ is ordered by reverse inclusion $\supseteq$. With every expression e in a strictness environment *senv*, they associated a strictness pair that provides properties of e both as an 'isolated value' and as a 'function to be applied':

$$\mathcal{S}[\![e]\!]senv = \langle sv, sf \rangle$$

This work was less semantically based than Burn, Hankin and Abramsky's because its use of power-sets of variable names in the 'strictness pair' domain introduced syntactic objects into a semantic construction. In retrospect, it was both over-syntactic and unnecessary in the sense that $\mathcal{P}(V)$ can be replaced by 2 with no loss of expressive power as we show in section 3, where we use the notation $\mathcal{E}_{HY}[\![\,\cdot\,]\!]$ instead of $\mathcal{S}[\![\,\cdot\,]\!]$

This work is structured in the following manner. Section 2 explains notation and the syntax and standard semantics for the untyped λ-calculus. It also describes the problem of domain error. Section 3 gives strictness interpretations which formalise Hudak and Young's and also our improvement. Section 4 sets up the relationship between the standard semantics, Hudak and Young's strictness and ours. Section 5 shows the correctness and completeness of our strictness interpretation relative to Hudak and Young's.

2 Notation and λ-calculus

Here we use the word *domain* to mean complete (pointed) partial order as usual. Let 2 stand for the domain $\{0,1\}$ ordered by $0 < 1$. Recursive domain definitions are as usual and $+, \oplus, \times, \rightarrow$ will mean respectively separated sum, coalesced sum, cartesian product and continuous function space.

2.1 Untyped λ-calculus

We consider the untyped λ-calculus with constants. Let C and V be sets of constants (including primitive functions) and variables ranged over by c and x respectively (z will also be used to range over integer constants). For the purposes of this paper we will assume C contains $\mathbb{Z}$ and Turing-sufficient arithmetic constants $\{\texttt{plus}, \texttt{minus}, \texttt{cond}\}$. (The first argument of `cond` is required to be an integer which is tested for zero/non-zero as in the 'C' programming language).

The set Λ of λ-calculus terms is then:

$$e \in \Lambda ::= c \mid x \mid \lambda x.e \mid e\ e'$$

The standard domain of interpretation is:

$$U = \mathbb{Z} + (U \rightarrow U) + \{wrong\} \quad [\equiv \mathbb{Z}_\perp \oplus (U \rightarrow U)_\perp \oplus \{wrong\}_\perp].$$

Injections into this sum will be written $in_z(\cdot)$, $in_f(\cdot)$, and $in_w(\cdot)$. We use `typewriter font` for syntactic objects and *italic font* for mathematical (meta-language) objects.

In the untyped world we need to inject functions (in $U \rightarrow U$) into U to represent them as values and outject them from U to $U \rightarrow U$ to apply them. This can be summarised by two functions *lam* and *app* respectively such as:

$$\begin{aligned} &lam\ x = in_f(x) \\ &app\ x\ y = case\ x\ of\ \ in_f(f) \Rightarrow f(y) \\ &\qquad\qquad\qquad\qquad\quad else\ err. \end{aligned}$$

Here '*err*' typically represents $\perp$ or $in_w(wrong)$, see below. Hudak and Young use the symbol '*error*' to represent such domain errors for constants — their treatment of these (and also for *app*) suggests they mean our $in_w(wrong)$. Milner [7] used a similar '*wrong*' value to handle domain errors.

2.2 Definition of an interpretation

An interpretation I is a tuple $(D_I; lam_I, app_I, num_I, plus_I, minus_I, cond_I, err_I)$ where D_I is a cpo and $lam_I : (D \rightarrow D) \rightarrow D$ and $app_I : D \rightarrow (D \rightarrow D)$ are continuous functions; $num_I : \mathbb{Z} \rightarrow D$ is a function and $plus_I, minus_I, cond_I, err_I \in D$. (We drop the subscripts when the context is clear.)

Given such an interpretation, I, we can define the notion of environment (over I) by

$$Env_I = V \rightarrow D$$

We use the letter ρ to range over environments. Such an interpretation, I, naturally defines an associated semantics

$$\mathcal{E}_I : \Lambda \rightarrow Env_I \rightarrow D$$

in the following manner:

$$\begin{aligned}
&\mathcal{E}_I[\![x]\!]\rho = \rho(x)\\
&\mathcal{E}_I[\![c]\!]\rho = \mathcal{K}_I[\![c]\!]\\
&\mathcal{E}_I[\![\lambda x.e]\!]\rho = lam_I(\lambda d \in D.\mathcal{E}_I[\![e]\!]\rho[d/x])\\
&\mathcal{E}_I[\![e\ e']\!]\rho = app_I(\mathcal{E}_I[\![e]\!]\rho)(\mathcal{E}_I[\![e']\!]\rho)
\end{aligned}$$

Here we use $\mathcal{K}_I$ for the meaning of constants — it is simply given by

$$\begin{aligned}
&\mathcal{K}_I[\![z]\!] = num_I(z)\\
&\mathcal{K}_I[\![\texttt{plus}]\!] = plus_I\\
&\mathcal{K}_I[\![\texttt{minus}]\!] = minus_I\\
&\mathcal{K}_I[\![\text{cond}]\!] = cond_I.
\end{aligned}$$

We write STD to refer to the standard interpretation given by U as domain and the constants as given below. Arithmetic constants have the usual meanings for arguments within $\mathbb{Z}$ in STD (including $num\ z = in_z z$) — we now consider their definition over the larger space D. The otherwise unused err_{STD} provides a convenient way of varying the error value in *plus*, *app etc.* used in the semantics for constants. (This is important as strictness depends on it.) Although this is rather an abuse of notation, given an interpretation, say STD above, we will write $STD[\perp/err]$ or $STD[in_w(wrong)/err]$ to represent an interpretation in which the error value *and all parts of the interpretation which use it* are altered.

2.3 Semantics of constants

2.3.1 Treatment of domain errors

We use the phrase "domain errors" to refer to situations such as $\texttt{plus}(\lambda\texttt{x.x})3$ or $3(2)$ in which an inappropriate value is used for an operand. To clarify this, let us consider an example, the function F defined by

$$\texttt{F} = \lambda\texttt{x}.\lambda\texttt{y.plus x y}$$

Is F strict in y? In the standard interpretation we obviously have

$$plus(in_z(m))(in_z(n)) = in_z(m+n)$$

but this does not define the other cases of $plus(in_f(f))$ and $plus(in_z(m))(in_f(g))$. If we define

$$plus(in_f(f)) = \perp$$

then F is strict in y, but if we define

$$plus(in_f(f))y = in_w(wrong)$$

then F is non-strict in y. Similarly $app(in_z(z))x$ and $app(in_w(wrong))x$ provide similar choices which affect strictness.

As Mishra noted in [5] some very specific choices are made in the denotational semantics regarding such issues as: domain errors due to primitive functions or whether all looping terms should be regarded as denoting the same value.

2.3.2 Subtlety of partial applications

Note that, even for a fixed choice of domain error value there is still a non-trivial choice for semantics of partially applied constants. Clarifying Hudak and Young's remark, there is a non-trivial choice of semantics of the (strict, curried) constants due to the lifting which occurs as a consequence of the above separated sum. (The problem arises from the non-isomorphism of $(A \times B \to C)_\perp$ and $(A \to (B \to C)_\perp)_\perp$ which causes η-equivalence to fail). For example, in the standard interpretation we can give

$$\begin{aligned}
\mathcal{K}[\![\texttt{plus}]\!] &= in_f \lambda x. in_f \lambda y. case\,(x,y)\ of\ \ \begin{array}[t]{l}(in_z(i), in_z(j)) \Rightarrow in_z(i+j)\\ else\ err\end{array}\\
\mathcal{K}[\![\texttt{cond}]\!] &= in_f \lambda x. in_f \lambda y. in_f \lambda z. case\, x\ of\ \ \begin{array}[t]{l}in_z(n) \Rightarrow (n \neq 0 \to y, z)\\ else\ err\end{array}
\end{aligned}$$

or we can give the following versions (which are more strict in the case of $err = \perp$)

$$\begin{aligned}
\mathcal{K}[\![\texttt{plus}]\!] &= in_f \lambda x. case\, x\ of\ \ \begin{array}[t]{l}in_z(i) \Rightarrow in_f \lambda y. case\, y\ of\ \ \begin{array}[t]{l}in_z(j) \Rightarrow in_z(i+j)\\ else\ err\end{array}\\ else\ err\end{array}\\
\mathcal{K}[\![\texttt{cond}]\!] &= in_f \lambda x. case\, x\ of\ \ \begin{array}[t]{l}in_z(n) \Rightarrow (n \neq 0 \to (in_f \lambda y. in_f \lambda z. y), (in_f \lambda y. in_f \lambda z. z))\\ else\ err\end{array}
\end{aligned}$$

Such differences are important for the precise details of the abstract strictness interpretation given in section 3.

To reproduce as closely as possible Hudak and Young's world, we adopt the former definitions and $err_{STD} = in_w(wrong)$.

3 Untyped strictness

In this section we give a simpler and more semantically oriented framework for the strictness analysis of Hudak and Young [3]. Section 2 gave the syntax and standard interpretation of our λ-calculus which yields the standard value domain

$$U = \mathbb{Z} + (U \to U) + \{wrong\} \quad [\equiv \mathbb{Z}_\perp \oplus (U \to U)_\perp \oplus \{wrong\}_\perp]$$

Now, since the abstract domain for $\mathbb{Z}_\perp$ is to be 2 as in the first order case, it might appear that the cpo

$$S = \{1\} + (S \to S)$$

is a suitable domain of strictness properties (the separated sum adds a $\perp$ element corresponding to 0. However, the untyped nature of functions like $\lambda\texttt{x.cond x}\ 7\ (\lambda\texttt{y}.42+\texttt{y})$ means that we need more least upper bounds to exist. Recalling the natural isomorphism of $\mathcal{P}(A+B)$ and $\mathcal{P}(A) \times \mathcal{P}(B)$ and the similarly of uncertainty induced by imprecise knowledge and non-determinism leads us to consider the larger cpo given by

$$S = 2 \times (S \to S)$$

which can now be viewed as a simpler formulation of Hudak and Young's strictness pairs. We adopt the name strictness pairs and their notation: elements $s \in S$ are written $\langle v, f\rangle$ with s_v, s_f standing for the components of s.

3.1 Strictness in the presence of domain errors

Note that the treatment of domain errors affects strictness. In the $STD[in_w(wrong)/err]$ interpretation above, we have that λ**x.cond** (λy.y) **x x** is not strict in **x** and hence neither is λx.λy.**cond y x x**. Oddly, Hudak and Young's original analysis incorrectly gives these as strict.

3.2 Strictness semantic interpretation

We take

$$S = 2 \times (S \to S)$$

as above for the domain part of the interpretation. Then the interpretation is completed by:

$$\begin{aligned}
&lam\ x = \langle 1, x\rangle \\
&app\ x\ y\ = \langle x_v \sqcap (x_f y)_v, (x_f y)_f\rangle \\
&\qquad\quad = \langle x_v, \top_{S\to S}\rangle \sqcap (x_f y) \\
&err = \langle 1, \lambda s.err\rangle \\
&\quad\ \ = \top_S \\
&num\ z = \langle 1, \lambda s.err\rangle \\
&plus = minus = \langle 1, \lambda x.\langle 1, \lambda y.\langle x_v \sqcap y_v, \lambda s.err\rangle\rangle\rangle \\
&cond\ = \langle 1, \lambda x.\langle 1, \lambda y.\langle 1, \lambda z.\langle x_v \sqcap (y_v \sqcup z_v), y_f \sqcup z_f\rangle\rangle\rangle\rangle \\
&\qquad = \langle 1, \lambda x.\langle 1, \lambda y.\langle 1, \lambda z.\langle x_v, \top_{S\to S}\rangle \sqcap (y \sqcup z)\rangle\rangle\rangle
\end{aligned}$$

The strictness interpretation of *cond* above is for the first choice (*i.e.* Hudak and Young's) of standard semantics of **plus** and **cond** given in section 2.3.2, *i.e.* when $cond\ \bot \neq \bot$. For the case of $cond\ \bot = \bot$ we would have the better (enabling more strictness inferences) interpretation as

$$\begin{aligned}
&plus = \langle 1, \lambda x.\langle x_v, \lambda y.\langle x_v \sqcap y_v, \lambda s.err\rangle\rangle\rangle \\
&cond = \langle 1, \lambda x.\langle x_v, (x_v = 0) \to \lambda s.err,\ \ \lambda y.\langle 1, \lambda z.(y \sqcup z)\rangle\rangle\rangle.
\end{aligned}$$

We will refer to this interpretation as EM and use 'EM' subscripts on its components when the context requires.

3.3 Hudak and Young's strictness interpretation

Let us call HY-strictness the strictness interpretation HY defined by

$$(S_{HY}; lam_{HY}, app_{HY}, num_{HY}, plus_{HY}, minus_{HY}, cond_{HY}, err_{HY})$$

satisfying the definitions below. These are taken from the strictness semantics of Hudak and Young, save that we use the $\sqcup$ symbol to denote the least upper bound on $S_{HY} \rightarrow S_{HY}$ but inexplicably they use $\sqcap$ "for clarity". Similarly, to make the semantic basis clearer, we have used the $\sqcup$ symbol instead of the synonymous $\cap$ on $(\mathcal{P}(V), \supseteq)$ and similarly $\sqcap$ for $\cup$.) We also have no need for "hatted" variables $\hat{x}$ to range over sets of variables which they used because of their mix of syntax and semantics. HY is given, dropping subscripts, by:

$$
\begin{aligned}
&S = (\mathcal{P}(V), \supseteq) \times (S \rightarrow S)\\
&lam\ x = \langle \{\}, x \rangle\\
&app\ x\ y\ = \langle x_v \sqcap (x_f y)_v, (x_f y)_f \rangle\\
&\qquad\quad = \langle x_v, \top_{S \rightarrow S} \rangle \sqcap (x_f y)\\
&err = \langle \{\}, \lambda s.err \rangle\\
&\quad\ \ = \top_S\\
&num\ z = \langle \{\}, \lambda s.err \rangle\\
&plus = minus = \langle \{\}, \lambda x.\langle \{\}, \lambda y.\langle x_v \sqcap y_v, \lambda s.err \rangle \rangle \rangle\\
&cond\ = \langle \{\}, \lambda x.\langle \{\}, \lambda y.\langle \{\}, \lambda z.\langle x_v \sqcap (y_v \sqcup z_v), y_f \sqcup z_f \rangle \rangle \rangle \rangle\\
&\qquad = \langle \{\}, \lambda x.\langle \{\}, \lambda y.\langle \{\}, \lambda z.\langle x_v, \top_{S \rightarrow S} \rangle \sqcap (y \sqcup z) \rangle \rangle \rangle
\end{aligned}
$$

It appears that merely re-phrasing Hudak and Young's formulation as an interpretation helps to separate syntax and semantics.

3.3.1 Warning

As we noted in section 3.1 the definition of $cond_{HY}$ is only correct with respect to $err_{STD} = \perp$ not $err_{STD} = in_w(wrong)$. Accordingly, to ensure the correctness of the following theorem from now on we take

$$
\mathcal{K}_{STD}[\![\mathsf{cond}]\!] = in_f \lambda x.in_f \lambda y.in_f \lambda z.case\ x\ of\ \begin{array}{l} in_z(n) \Rightarrow (n \neq 0 \rightarrow y, z) \\ else\ \perp \end{array}
$$

instead of that given in section 2.3.2.

4 Relationship between various interpretations

We claim the following results.

1. (From Hudak and Young) HY $(= HY[\top_{S_{HY}}/err])$ is a correct abstraction of STD $(= STD[in_w(wrong)/err])$
2. $HY[\perp/err]$ is a correct abstraction of $STD[\perp/err]$
3. EM is a correct abstraction of HY
4. EM is complete for HY
5. $EM[\perp/err]$ is a correct abstraction of $HY[\perp/err]$
6. $EM[\perp/err]$ is complete for $HY[\perp/err]$

The correctness relations between STD and EM hold by transitivity.

The next section sets about proving that results 3 and 4, *i.e.* that EM is a correct abstraction of HY which is also complete.

5 Relationship to Hudak and Young's strictness

We now set up a relationship between between HY-strictness HY and EM-strictness EM from sections 3.2 and 3.3. This relationship is then shown to induce an *abstraction* of HY-strictness into EM-strictness. Moreover, the abstraction is *complete* in that all properties exploited by Hudak and Young are derivable *via* our strictness interpretation.

For notational reasons in this section we will use A for S_{EM} and B for S_{HY}.

Both A and B are given as recursive function spaces, *viz*

$$\begin{aligned} A &= 2 \times (A \to A) \\ B &= (\mathcal{P}(V), \supseteq) \times (B \to B) \end{aligned}$$

Let us define $\gamma_1 : 2 \to \mathcal{P}(V)$ by

$$\begin{aligned} \gamma_1(0) &= V \\ \gamma_1(1) &= \{\}. \end{aligned}$$

Now, the relation we seek to define should satisfy

$$\begin{aligned} \sim \ &\subseteq \ A \times B \\ (x, f) \sim (y, g) \ &\Leftrightarrow \ y = \gamma_1(x) \ \wedge \\ &\quad\ (\forall a \in A, b \in B) \ a \sim b \Rightarrow f(a) \sim g(b) \end{aligned}$$

but it is unclear whether this is a well-definition. To prove the unique existence and various properties of $\sim$ we define it simultaneously with the inverse limit construction for A and B.

Recall that domain equations like that for A above are solved by the inverse limit construction — we put $A_0 = \{\perp\}$, the trivial domain, and then put $A_{k+1} = 2 \times (A_k \to A_k)$. There are embedding $i_k : A_k \to A_{k+1}$ and projection $p_k : A_{k+1} \to A_k$ maps between A_k and A_{k+1}. A is obtained as the limit

$$A_\infty = \{(a_0, a_1, \ldots) \in \prod_k A_k \mid a_k = p_{k+1}(a_{k+1})\}$$

The isomorphism of A and $2 \times (A \to A)$ is obtained pointwise from the p_k and i_k. The construction for B is identical.

We can define approximants of $\sim$ in the following manner

$$\begin{aligned} \sim_k \ &\subseteq \ A_k \times B_k \\ a \sim_0 b \ &\triangleq \ \text{true} \\ (x, f) \sim_{k+1} (y, g) \ &\triangleq \ y = \gamma_1(x) \ \wedge \\ &\quad\ (\forall a \in A_k, b \in B_k) \ a \sim_k b \Rightarrow f(a) \sim_k g(b) \end{aligned}$$

and hence properly define

$$\begin{aligned} \sim \ &\subseteq \ A \times B \\ (a_0, a_1, \ldots) \sim (b_0, b_1, \ldots) \ &\Leftrightarrow \ (\forall k) \ a_k \sim_k b_k. \end{aligned}$$

It is convenient to write

$$\begin{aligned}
\overset{1}{\sim} &\subseteq 2 \times \mathcal{P}(V) \\
\overset{2}{\sim}_{k+1} &\subseteq (A_k \to A_k) \times (B_k \to B_k) \\
x \overset{1}{\sim} y &\triangleq y = \gamma_1(x) \\
f \overset{2}{\sim}_{k+1} g &\triangleq (\forall a \in A_k, b \in B_k)\ a \sim_k b \Rightarrow f(a) \sim_k g(b)
\end{aligned}$$

so that

$$(x, f) \sim_{k+1} (y, g) \quad \Leftrightarrow \quad x \overset{1}{\sim} y \wedge f \overset{2}{\sim}_{k+1} g.$$

It is also convenient to define here the type-induced ('logical') relations from $\sim$. Allowing t to range over meta-language types given by $t ::= D \mid t \to t$ we define

$$\begin{aligned}
a \sim^D b &\Leftrightarrow a \sim b \\
f \sim^{t \to t'} g &\Leftrightarrow ((\forall x, y)\ x \sim^t y \Rightarrow fx \sim^{t'} gy)
\end{aligned}$$

The limit relation $\overset{2}{\sim}$ now coincides with $\sim^{D \to D}$

We now have distributivity lemma for $\sim$:
Lemma: $\sim$ preserves arbitrary LUBs and GLBs (including $\perp$ and $\top$) in that, given possible empty sequences $a^i \in A, b^i \in B$, we have

$$((\forall i)\ a^i \sim b^i)(\bigsqcup_i a^i \sim \bigsqcup_i b^i \quad \wedge \quad \sqcap_i a^i \sim \sqcap_i b^i)$$

5.1 Proposition: relatedness

For all λ-terms $e \in \Lambda$ we have that

$$(\forall \eta \in Env_{EM}, \rho \in Env_{HY})\ \eta \sim \rho \Rightarrow \mathcal{E}_{EM}[\![e]\!]\eta \sim \mathcal{E}_{HY}[\![e]\!]\rho$$

where $\eta \sim \rho \Leftrightarrow (\forall x \in V)\ \eta(x) \sim \rho(x)$. It turns out that this abstraction relation is both correct and complete and we study these aspects after a proof sketch.

5.2 Proof

We the above proposition by structural induction on the (object) term e. But first we need some lemmas, *viz*

- $app_{EM} \sim^{D \to (D \to D)} app_{HY}$
- $lam_{EM} \sim^{(D \to D) \to D} lam_{HY}$.
- $(\forall z \in \mathbb{Z}) num_{EM}(z) \sim num_{HY}(z)$
- $plus_{EM} \sim plus_{HY}$
- $minus_{EM} \sim minus_{HY}$

- $cond_{EM} \sim cond_{HY}$
- $err_{EM} \sim err_{HY}$

Given these lemmas, proved below, the theorem is a trivial structural induction. We give two cases:

- case $e = x$: trivial.
- case $e = \lambda x.e'$: By inductive hypothesis, supposing also $a \sim b$ then $\mathcal{E}_{EM}[\![e']\!]\eta[a/x] \sim \mathcal{E}_{HY}[\![e']\!]\rho[b/x]$. Hence by the lemma $lam_{EM}\lambda a.\mathcal{E}_{EM}[\![e']\!]\eta[a/x] \sim lam_{EM}\lambda b.\mathcal{E}_{HY}[\![e']\!]\rho[b/x]$.

Proof of lemmas

We give the representative cases for *app* and *cond.*

- $app_{EM} \sim^{D\to(D\to D)} app_{HY}$: Assume $a \sim b$ and $a' \sim b'$ then, expanding the definitions of app_{HY} and app_{EM}, it is equivalent to prove
$$\langle a_v, \top_{A\to A}\rangle \sqcap (a_f a') \sim \langle b_v, \top_{B\to B}\rangle \sqcap (b_f b').$$
This holds since $a \sim b \Leftrightarrow a_v \overset{1}{\sim} b_v \wedge a_f \overset{2}{\sim} b_f$ and the lemma for $\sim$-preservation of $\sqcup$ and $\sqcap$.

- $cond_{HY} \sim cond_{EM}$: We need to prove
$$\langle 1, \lambda a.\langle 1, \lambda a'.\langle 1, \lambda a''.\langle a_v, \top_{A\to A}\rangle \sqcap (a' \sqcup a'')\rangle\rangle\rangle \sim$$
$$\langle \{\}, \lambda b.\langle \{\}, \lambda b'.\langle \{\}, \lambda b''.\langle b_v, \top_{B\to B}\rangle \sqcap (b' \sqcup b'')\rangle\rangle\rangle.$$
Assume $a \sim b$, $a' \sim b'$ and $a'' \sim b''$ then, using the recursive definition of $\sim$ and recalling that $1 = \top_A$ and $\{\} = \top_B$, this is equivalent to
$$\top_A \overset{1}{\sim} \top_B \quad \wedge \quad \langle a_v, \top_{A\to A}\rangle \sqcap (a' \sqcup a'') \sim \langle b_v, \top_{B\to B}\rangle \sqcap (b' \sqcup b'').$$
The first conjunct holds by definition and by the lemma for $\sim$-preservation of $\sqcup$ and $\sqcap$ it suffices to show
$$\langle a_v, \top_{A\to A}\rangle \sim \langle b_v, \top_{B\to B}\rangle.$$
This holds since $a \sim b \Rightarrow a_v \overset{1}{\sim} b_v$ and
$$\top_{A\to A} = \lambda x \in A.\top_A \overset{2}{\sim} \lambda y \in B.\top_B = \top_{B\to B}.$$

5.3 Proposition: soundness

The relation $\sim$ restricts to a embedding-closure pair (an abstraction of $B = S_{HY}$ by $A = S_{EM}$). The concretisation and abstraction maps respectively are $\gamma : A \to B$ and $\alpha : B \to A$ given by

$$\begin{aligned}\gamma(a) &= \bigsqcup\{b \in B | a \sim b\}\\ \alpha(b) &= \sqcap\{a \in A | a \sim b\} = \sqcap\{a \in A \mid \gamma(b) \sqsubseteq a\}\end{aligned}$$

The α and γ form a galois connection as usual and correctness of the remainder of the interpretation interpretation (*i.e. lam, app, plus etc.*) with respect to (α, γ) follows from that the base lemmas above.

5.4 Proposition: completeness

Since the trivial abstract interpretation would be sound with respect to HY-strictness, we now show that EM-strictness can provide all the information that HY-strictness can. This is a completeness argument. Note that we cannot expect to have a natural completeness result of the form "EM-strictness of expressions determines their HY-strictness". Consider the term λx.x: this has HY-strictness of $(\{\}, \lambda x \in S_{HY}.x)$ and EM-strictness of $(0, \lambda x \in S_{EM}.x)$. It is unreasonable to expect some function of the latter, coarser-grained, interpretation to yield the former, finer, one.[1]

Accordingly, our completeness result relies on the observation that Hudak and Young's analysis makes strictness optimisations only on the basis of limited predicates (actually whether the first component of S_{HY} is empty or non-empty). The rest of the internal structure is non-observable. Accordingly, we wish to assert that our simpler internal structure gives rise to the precisely the same observable properties.

The key notion is that both the *EM* and *HY* interpretations are only used for strictness optimisations, *i.e.* early evaluation of an expression. Although it is rarely clearly stated, we implicitly have a predicate which whose result tells us when an abstract value permits strictness optimisations. Here, this predicate (subset of S_{HY} or S_{EM}) is given by

$$p(v, f) \Leftrightarrow v = \perp.$$

This is a sound predictor of when the standard interpretation gives $\perp$ for some prescribed assignments of values to free variables. We abuse notation by using the p for both S_{HY} and S_{EM}.

Our completeness result is that, for all meta-terms e,

$$(\forall \eta \in Env_{EM}, \rho \in Env_{HY})\ \eta \sim \rho \Rightarrow (p(\mathcal{E}_{EM}[\![e]\!]\eta) \Leftrightarrow p(\mathcal{E}_{HY}[\![e]\!]\rho))$$

Thus all optimisations permitted by the *HY* interpretation are also permitted by the *EM* interpretation. This forms the basis of our claim that the *HY* domain had spurious elements.

6 Problem of infinite chains

Hudak and Young mentioned in [3] that their higher-order analysis is not guaranteed to terminate. Indeed, this is the case when a strictness pair needs to be applied an infinite number of times. They gave the following example, $\mathtt{f} = \lambda\mathtt{x}.\mathtt{f\ x\ x}$ which leads to EM-strictness

$$s = \langle 1, \lambda x. \langle s_v \sqcap (s_f x)_v \sqcap ((s_v x)_f x)_v, ((s_f x)_f x)_f \rangle \rangle$$

or HY-strictness

$$s = \langle \{\}, \lambda x. \langle s_v \cup (s_f x)_v \cup ((s_v x)_f x)_v, ((s_f x)_f x)_f \rangle \rangle$$

[1] The general question of completeness in abstract interpretation is being developed in a companion paper.

There is a circularity which Hudak and Young suggest is due to the fact that "early" elements of $\mathcal{P}(V)$ in the nested pairs depend on "deeper" $S_{HY} \to S_{HY}$ elements. Their solution to this problem of infinite chains is merely suggesting "to impose a weak type discipline". The next paragraph shows how this could work for the simply typed λ-calculus and, although this is clearly not the best way to handle the simply typed λ-calculus, it points to how one might treat the 2nd order λ-calculus.

Further work

It would be desirable to consider whether certain finite-height lattices could represent strictness properties for the untyped λ-calculus instead of the infinite chains present in Hudak and Young. For example, if the given program in Λ can be corresponds to a (type-stripped) program in the simply typed λ-calculus (with (object) types ranged over by t) then we can use $\sum \mathcal{T}_{\mathbb{Z}_\perp}[\![t]\!]$ for the value domain (a retract of $D = \mathbb{Z} + (D \to D)$) and hence $\sum \mathcal{T}_2[\![t]\!]$ for the strictness domain (a retract of $S = 2 \times (S \to S)$) where

$$\begin{aligned} \mathcal{T}_X[\![int]\!] &= X \\ \mathcal{T}_X[\![t \to t']\!] &= \mathcal{T}_X[\![t]\!] \to \mathcal{T}_X[\![t']\!]. \end{aligned}$$

This exhibits [1] within our model and the key point is that $\sum \mathcal{T}_2[\![t]\!]$ has no infinite ascending chains. The key question is to whether there exists finite height models for another subset of Λ, those programs corresponding to second-order polymorphically typable terms — this would enable us to conclude Hudak and Young's suggestion of modelling list operators as λ-terms and thereby inheriting a sensible strictness theory.

Acknowledgments

Thanks to Andy Pitts for explaining subtleties of recursive function spaces. We are indebted to the referees for their careful reading and apposite comments of the draft version of [2] which included sketched versions of these results. This research was supported by SERC grant GR/H14465.

References

[1] Burn, G., Hankin, C. and Abramsky, S. The theory and practice of strictness analysis for higher order functions. In [4].

[2] Ernoult, C. and Mycroft A. Uniform ideals and strictness analysis. Lecture Notes in Computer Science: Proc. 18th ICALP, vol. 510, Springer-Verlag, 1991.

[3] Hudak, P. and Young, J. Higher order strictness analysis in untyped lambda calculus. Proc. 13th ACM symp. on Principles of Programming Languages, 1986.

[4] Jones, N.D. and Ganzinger, H. (eds.) Programs as Data Objects. Lecture Notes in Computer Science: Proc. of a Workshop, Copenhagen, vol. 215, Springer-Verlag, 1985.

[5] Kuo, T.-M. and Mishra, P. Strictness analysis: a new perspective based on type inference. ACM-IFIP, Proc. of the functional programming and computer architecture conference, 1989.

[6] MacQueen, D., Plotkin, G.D. and Sethi, R. An ideal model for recursive polymorphic types. Proc. 11th ACM symp. on Principles of Programming Languages, 1984.

[7] Milner, R. A theory of type polymorphism in programming. JCSS 1978.

[8] Mycroft, A. Abstract interpretation and optimising transformations of applicative programs. Ph.D. thesis, Edinburgh University, 1981. Available as computer science report CST-15-81.

[9] Mycroft, A. and Jones, N.D. A relational framework for abstract interpretation. In [4].