

Strictness Logic and Polymorphic Invariance

P. N. Benton*

University of Cambridge

Abstract

We describe a logic for reasoning about higher-order strictness properties of typed lambda terms. The logic arises from axiomatising the inclusion order on certain closed subsets of domains. The axiomatisation of the lattice of strictness properties is shown to be sound and complete, and we then give a program logic for assigning properties to terms. This places work on strictness analysis via type inference on a firm theoretical foundation. We then use proof theoretic techniques to show how the derivable strictness properties of different instances of polymorphically typed terms are related.

1 Introduction

Strictness analysis for non-strict functional languages is the problem of trying to work out at compile time if a function is *strict*, ie. if $f\perp = \perp$. The information gained by a strictness analyser can be used to transform call by need into call by value, to spawn tasks in a parallel machine or to validate certain source-level program transformations.

Strictness analysis was first studied in Mycroft's thesis [11], which applied the framework of abstract interpretation to various analyses of functional programs. Since then, strictness analysis by abstract interpretation has been an active field of research. From the point of view of the present paper, the most important reference is Burn, Hankin and Abramsky's work on extending Mycroft's original analysis to higher order-functions [5].

Several people have noticed that type inference can be characterised as an abstract interpretation [12]. In [9], Kuo and Mishra turned this around by describing a way to perform strictness analysis by non-standard type inference. That paper is mainly concerned with getting a practical inference algorithm, and does not discuss correctness. It is also not clear whether they are really analysing typed or untyped programs.

*Author's address: University of Cambridge, Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, United Kingdom. Internet: Nick.Benton@cl.cam.ac.uk. Research supported by a SERC Research Studentship.

The work described here came from trying to understand what was actually going on semantically in their type system.

There seemed to be a fundamental shift from denotational analysis techniques to logical ones, so the obvious place to look for insight was work on domain logics [2]. These arise from Stone-type dualities between (topological) spaces and logics [8] – one can either view points as primary and then consider a property to be a set of points, or take properties as primary and then consider a point to be determined by the properties it satisfies.

Abramsky’s work is concerned with the logic of *observable properties* – things we can observe by looking at finite bits of output. These correspond to open sets in the Scott topology. Strictness is non-observable, and the strictness properties which we shall consider correspond to closed sets. The logic presented here is nevertheless essentially a fragment of the open set logic in [2], although the interpretations of propositions are very different.

It should be noted that the same system was arrived at independently in [7]. That work is complementary to this, however, in that Jensen considers the relation between conventional abstract interpretation and the logic whereas this paper relates the logic directly to the standard semantics. Jensen does not consider polymorphic invariance. For reasons of space, all the proofs in this paper have been either omitted or sketched briefly. More detailed proofs will appear in [4].

2 The Language Λ_T

This section briefly describes the language with which we shall be working. Types, ranged over by σ, τ , are formed from a single base type A , which we interpret as the natural numbers, pairs and function spaces.

$$\sigma ::= A \mid \sigma \rightarrow \tau \mid \sigma \times \tau$$

We assume an infinite set $\{x^\sigma\}$ of distinct typed variables for each type σ . Terms of Λ_T , ranged over with s, t and so on, are then formed as follows

$$t ::= \underline{n}^A \mid x^\sigma \mid (t^{\sigma \rightarrow \tau} s^\sigma)^\tau \mid (\lambda x^\sigma. t^\tau)^{\sigma \rightarrow \tau} \mid (s^\sigma, t^\tau)^{\sigma \times \tau} \mid \text{cond}^{A \rightarrow \sigma \rightarrow \sigma \rightarrow \sigma} \mid \text{fix}^{(\sigma \rightarrow \sigma) \rightarrow \sigma} \mid \text{plus}^{A \rightarrow A \rightarrow A} \mid \pi_1^{\sigma \times \tau \rightarrow \sigma} \mid \pi_2^{\sigma \times \tau \rightarrow \tau}$$

Adding other base types, such as booleans, or other strict primitive functions makes no essential difference to the work considered here, so we omit them in the interests of brevity.

Λ_T may be given a call by name denotational semantics in the usual way. We define the set $\{D_\sigma\}$ of domains inductively by $D_A = \mathcal{N}_\perp$, $D_{\sigma \rightarrow \tau} = [D_\sigma \rightarrow D_\tau]$, $D_{\sigma \times \tau} = D_\sigma \times D_\tau$. Next we define an environment $\rho \in Env$ to be a type-respecting partial function from variables to the disjoint union of all the D_σ and we then define $\llbracket t^\sigma \rrbracket : Env \rightarrow D_\sigma$ by induction on the structure of t^σ .

Formation rules	
$\mathbf{t}, \mathbf{f} \in L_\sigma$	$\frac{\phi, \psi \in L_\sigma}{\phi \wedge \psi \in L_\sigma}$
$\frac{\phi \in L_\sigma \quad \psi \in L_\tau}{(\phi \rightarrow \psi) \in L_{\sigma \rightarrow \tau}}$	$\frac{\phi \in L_\sigma \quad \psi \in L_\tau}{(\phi \times \psi) \in L_{\sigma \times \tau}}$
Inference rules	
$\phi \leq \phi$ [refl]	$\phi \leq \mathbf{t}$ [t] $\mathbf{f} \leq \phi$ [f]
$\frac{\phi \wedge \psi \leq \phi \quad \psi \leq \chi}{\phi \leq \chi}$ [trans]	$\frac{\phi \wedge \psi \leq \psi \quad \phi \leq \psi_1 \quad \psi \leq \psi_2}{\phi \leq \psi_1 \wedge \psi_2}$ [\wedge]
$\frac{\phi \leq \phi' \quad \psi \leq \psi'}{(\phi \times \psi) \leq (\phi' \times \psi')}$ [\times]	
$\mathbf{t}_{\sigma \times \tau} \leq \mathbf{t}_\sigma \times \mathbf{t}_\tau$ [$\mathbf{t} \times$]	$\mathbf{f}_\sigma \times \mathbf{f}_\tau \leq \mathbf{f}_{\sigma \times \tau}$ [$\mathbf{f} \times$]
$(\phi \times \psi) \wedge (\phi' \times \psi') \leq (\phi \wedge \phi') \times (\psi \wedge \psi')$ [$\times \wedge$]	
$\frac{\phi' \leq \phi \quad \psi \leq \psi'}{(\phi \rightarrow \psi) \leq (\phi' \rightarrow \psi')}$ [\rightarrow]	
$\mathbf{t}_{\sigma \rightarrow \tau} \leq \mathbf{t}_\sigma \rightarrow \mathbf{t}_\tau$ [$\mathbf{t} \rightarrow$]	$\mathbf{t}_\sigma \rightarrow \mathbf{f}_\tau \leq \mathbf{f}_{\sigma \rightarrow \tau}$ [$\mathbf{f} \rightarrow$]
$(\phi \rightarrow \psi_1) \wedge (\phi \rightarrow \psi_2) \leq (\phi \rightarrow \psi_1 \wedge \psi_2)$ [$\rightarrow \wedge$]	

Figure 1: Formation and inference rules for \mathcal{L}_σ

3 The Logic of Strictness Properties

The properties in which we are interested will correspond to certain subsets of domains in the standard denotational semantics. More specifically, an *ideal* of a domain is a non-empty subset of the domain which is down-closed and closed under sups of chains. We will be interested in certain special ideals. Note that an ideal is the same thing as a non-empty Scott-closed set, or an element of the Hoare powerdomain.

We define a type-indexed family of propositional theories $\mathcal{L}_\sigma = (L_\sigma, \leq_\sigma)$, where L_σ is a set of propositions and \leq_σ is the entailment relation, as shown in figure 1. We define $\phi_\sigma = \psi_\sigma$ to mean $\phi_\sigma \leq \psi_\sigma$ and $\psi_\sigma \leq \phi_\sigma$.

Note that the axioms [$\rightarrow \wedge$] and [$\mathbf{t} \rightarrow$], which describe how the type structure interacts with the logical structure, could both have been captured in the one axiom $\bigwedge_{i \in I} (\phi \rightarrow \psi_i) = (\phi \rightarrow \bigwedge_{i \in I} \psi_i)$, if we allow the indexing set to be empty. It is probably also worth pointing out that [$\mathbf{t} \rightarrow$] and [$\mathbf{f} \rightarrow$] explicitly identify \perp and $\lambda x. \perp$.

Now at each type σ we have a domain D_σ , and a propositional theory \mathcal{L}_σ . The next step is to relate them by giving an interpretation map $\llbracket \cdot \rrbracket$ which takes each proposition

$\phi \in \mathcal{L}_\sigma$ to the set of elements of D_σ which satisfy it (ie. its *extent*):

$$\begin{aligned} \llbracket \mathbf{t} \rrbracket_\sigma &= D_\sigma \\ \llbracket \mathbf{f} \rrbracket_\sigma &= \{\perp\} \\ \llbracket \phi \wedge \psi \rrbracket_\sigma &= \llbracket \phi \rrbracket_\sigma \cap \llbracket \psi \rrbracket_\sigma \\ \llbracket \phi \times \psi \rrbracket_{\sigma \times \tau} &= \{(x, y) \in D_{\sigma \times \tau} \mid x \in \llbracket \psi \rrbracket_\sigma \text{ and } y \in \llbracket \phi \rrbracket_\tau\} \\ \llbracket \phi \rightarrow \psi \rrbracket_{\sigma \rightarrow \tau} &= \{f \in D_{\sigma \rightarrow \tau} \mid f \llbracket \phi \rrbracket_\sigma \subseteq \llbracket \psi \rrbracket_\tau\} \end{aligned}$$

Proposition 3.1 For any $\phi \in \mathcal{L}_\sigma$, $\llbracket \phi \rrbracket_\sigma$ is an ideal of D_σ . □

Example Some examples of the sort of properties which we can express in this system:

- A function $f : A \rightarrow A$ is strict iff $f \in \llbracket \mathbf{f}_A \rightarrow \mathbf{f}_A \rrbracket$.
- A function $f : A \rightarrow A$ is the constant \perp function iff $f \in \llbracket \mathbf{t}_A \rightarrow \mathbf{f}_A \rrbracket$.
- A function $h : A \times A \rightarrow A$ is strict in both its arguments iff $h \in \llbracket (\mathbf{f}_A \times \mathbf{t}_A \rightarrow \mathbf{f}_A) \wedge (\mathbf{t}_A \times \mathbf{f}_A \rightarrow \mathbf{f}_A) \rrbracket$.
- A function $g : (A \rightarrow A) \rightarrow (A \rightarrow A)$ maps strict functions to strict functions iff $g \in \llbracket (\mathbf{f}_A \rightarrow \mathbf{f}_A) \rightarrow (\mathbf{f}_A \rightarrow \mathbf{f}_A) \rrbracket$

Proposition 3.2 (Soundness) If $\vdash \phi \leq \psi$ then $\llbracket \phi \rrbracket \subseteq \llbracket \psi \rrbracket$. □

At first sight, it does not seem likely that the small set of rules which we have just given is going to be complete. In fact it turns out that it is, and the reason for this is that at each type we are restricting attention to a very well-behaved subset of the set of all ideals of the domain.

For the next bit we really want to work not in \mathcal{L}_σ , but in its Lindenbaum algebra (poset reflection) $\mathcal{L}\mathcal{A}_\sigma = (\mathcal{L}_\sigma / =_\sigma)$, which is obviously a finite lattice with all meets (and hence all joins). However, we will abuse notation and write ϕ_σ when we really mean its equivalence class $[\phi]_{=_\sigma}$. We will also write $\phi \leq \psi$ for $\vdash \phi \leq \psi$, $x \models \phi$ for $x \in \llbracket \phi \rrbracket$ and omit type subscripts all over the place.

Let $\alpha_\sigma : D_\sigma \rightarrow \mathcal{L}\mathcal{A}_\sigma$ be defined by $\alpha_\sigma(x) = \bigwedge \{\phi \mid x \models \phi\}$, so α takes a domain element to the conjunction of all the propositions which it satisfies. The conjunction is finite as the lattice is.

Completeness will essentially come from the fact that α is surjective. The way we are going to show that is to construct an explicit map $\gamma_\sigma : \mathcal{L}\mathcal{A}_\sigma \rightarrow D_\sigma$ such that $\alpha_\sigma \circ \gamma_\sigma = \text{id}_{\mathcal{L}\mathcal{A}_\sigma}$. This is not quite analagous to the concretisation map appearing in traditional abstract interpretation, since we are only picking a *representative* of the extent of each equivalence class of propositions.

The definition of the γ_σ proceeds by recursion on the type structure. The base case is easy, just let $\gamma_A(\mathbf{f}) = \perp_A$ and $\gamma_A(\mathbf{t}) = 42$ (say). For product types we define $\gamma_{\sigma \times \tau}(\phi \times \psi) = (\gamma_\sigma(\phi), \gamma_\tau(\psi))$. We do not need explicitly to consider conjunctions of propositions at product types, since the rules of the logic allow us to remove them: $(\phi \times \psi) \wedge (\phi' \times \psi') = ((\phi \wedge \phi') \times (\psi \wedge \psi'))$.

For function types, we need an auxiliary definition. For $\phi \in \mathcal{L}\mathcal{A}_\sigma$ and $\psi \in \mathcal{L}\mathcal{A}_\tau$, define the *co-step function* $[\phi, \psi] : \mathcal{L}\mathcal{A}_\sigma \rightarrow \mathcal{L}\mathcal{A}_\tau$ by

$$[\phi, \psi](\chi) = \begin{cases} \psi & \text{if } \chi \leq \phi \\ \mathbf{t} & \text{otherwise} \end{cases}$$

Note that this is monotonic. A typical element of $\mathcal{L}\mathcal{A}_{\sigma \rightarrow \tau}$ looks like $\bigwedge_{i \in I} \phi_i \rightarrow \psi_i$, so we can define $\gamma_{\sigma \rightarrow \tau}$ inductively by

$$\gamma_{\sigma \rightarrow \tau}(\bigwedge_{i \in I} \phi_i \rightarrow \psi_i) = \gamma_\tau \circ \prod_i [\phi_i, \psi_i] \circ \alpha_\sigma$$

Where $\prod_i [\phi_i, \psi_i](\chi) = \bigwedge \{\psi_i \mid \chi \leq \phi_i\}$. It is easy to show that γ is well-defined on equivalence classes of propositions.

The next two results follow by induction on types and amount to showing that $\alpha \circ \gamma = \text{id}$.

Proposition 3.3 $\gamma(\phi) \models \phi$. □

Proposition 3.4 If $\gamma(\phi) \models \psi$ then $\phi \leq \psi$. □

Corollary 3.5 (Completeness) If $\llbracket \phi \rrbracket \subseteq \llbracket \psi \rrbracket$ then $\vdash \phi \leq \psi$.

Proof. $\llbracket \phi \rrbracket \subseteq \llbracket \psi \rrbracket \Rightarrow \gamma(\phi) \models \psi \Rightarrow \phi \leq \psi$. □

Corollary 3.6 (Disjunction Property) $\llbracket \phi \rrbracket \subseteq \llbracket \psi_1 \rrbracket \cup \llbracket \psi_2 \rrbracket$ implies either $\llbracket \phi \rrbracket \subseteq \llbracket \psi_1 \rrbracket$ or $\llbracket \phi \rrbracket \subseteq \llbracket \psi_2 \rrbracket$.

Proof. WLOG $\gamma(\phi) \models \psi_1$ and then $\phi \leq \psi_1$ so $\llbracket \phi \rrbracket \subseteq \llbracket \psi_1 \rrbracket$ by soundness. □

It is an interesting observation that in some sense we only have completeness because of the disjunction property. If we had $\llbracket \phi \rrbracket \subseteq \llbracket \psi_1 \rrbracket \cup \llbracket \psi_2 \rrbracket$ without $\llbracket \phi \rrbracket \subseteq \llbracket \psi_1 \rrbracket$ or $\llbracket \phi \rrbracket \subseteq \llbracket \psi_2 \rrbracket$ then we would certainly also have $\llbracket \psi_1 \rightarrow \chi \wedge \psi_2 \rightarrow \chi \rrbracket \subseteq \llbracket \phi \rightarrow \chi \rrbracket$ but no way of proving it without adding disjunction to the logic.

The proof of Proposition 3.4 also leads to the following useful result:

Lemma 3.7 (Entailment decomposition)

$$\bigwedge_{i \in I} (\phi_i \rightarrow \psi_i) \leq (\phi' \rightarrow \psi') \quad \text{iff} \quad \bigwedge \{\psi_i \mid \phi' \leq \phi_i\} \leq \psi'$$

□

$$\begin{array}{c}
\frac{\Gamma, x^\sigma : \phi_\sigma \vdash x^\sigma : \phi_\sigma [\text{var}] \quad \Gamma \vdash t^\sigma : \mathbf{t}_\sigma [\text{top}]}{\Gamma \vdash t^{\sigma \rightarrow \tau} : \phi_\sigma \rightarrow \psi_\tau \quad \Gamma \vdash s^\sigma : \phi_\sigma} [\text{app}] \\
\frac{\Gamma, x^\sigma : \mathbf{t}_\sigma \vdash t^\tau : \mathbf{f}_\tau}{\Gamma \vdash (\mathbf{t}^{\sigma \rightarrow \tau} s^\sigma)^\tau : \psi_\tau} [\text{absbot}] \quad \frac{\Gamma, x^\sigma : \phi_\sigma \vdash t^\tau : \psi_\tau}{\Gamma \vdash (\lambda x^\sigma. t^\tau)^{\sigma \rightarrow \tau} : \phi_\sigma \rightarrow \psi_\tau} [\text{abs}] \\
\frac{\Gamma \vdash (\lambda x^\sigma. t^\tau)^{\sigma \rightarrow \tau} : \mathbf{f}_{\sigma \rightarrow \tau} \quad \Gamma \vdash t^\sigma : \psi_\sigma}{\Gamma \vdash t^\sigma : (\phi \wedge \psi)_\sigma} [\text{conj}] \quad \frac{\Gamma \vdash (\lambda x^\sigma. t^\tau)^{\sigma \rightarrow \tau} : \phi_\sigma \rightarrow \psi_\tau \quad \Gamma \vdash t^\sigma : \phi_\sigma \quad \phi_\sigma \leq \psi_\sigma}{\Gamma \vdash t^\sigma : \psi_\sigma} [\text{sub}] \\
\frac{\Gamma \vdash t^\sigma : (\phi \wedge \psi)_\sigma \quad \Gamma \vdash s^\sigma : \phi_\sigma \quad \Gamma \vdash t^\tau : \psi_\tau}{\Gamma \vdash (s^\sigma, t^\tau)^{\sigma \times \tau} : \phi_\sigma \times \psi_\tau} [\text{pair}] \\
\Gamma \vdash \text{cond}^{A \rightarrow \sigma \rightarrow \sigma \rightarrow \sigma} : \mathbf{f}_A \rightarrow \mathbf{t}_\sigma \rightarrow \mathbf{t}_\sigma \rightarrow \mathbf{f}_\sigma [\text{cond1}] \\
\Gamma \vdash \text{cond}^{A \rightarrow \sigma \rightarrow \sigma \rightarrow \sigma} : \mathbf{t}_A \rightarrow \phi_\sigma \rightarrow \phi_\sigma \rightarrow \phi_\sigma [\text{cond2}] \\
\Gamma \vdash \text{plus}^{A \rightarrow A \rightarrow A} : (\mathbf{t}_A \rightarrow \mathbf{f}_A \rightarrow \mathbf{f}_A) \wedge (\mathbf{f}_A \rightarrow \mathbf{t}_A \rightarrow \mathbf{f}_A) [\text{plus}] \\
\Gamma \vdash \text{fix}^{(\sigma \rightarrow \sigma) \rightarrow \sigma} : (\phi_\sigma \rightarrow \phi_\sigma) \rightarrow \phi_\sigma [\text{fix}] \\
\Gamma \vdash \pi_1^{\sigma \times \tau \rightarrow \sigma} : \phi_\sigma \times \mathbf{t}_\tau \rightarrow \phi_\sigma [\pi_1] \quad \Gamma \vdash \pi_2^{\sigma \times \tau \rightarrow \tau} : \mathbf{t}_\sigma \times \phi_\tau \rightarrow \phi_\tau [\pi_2]
\end{array}$$

Figure 2: The program logic.

4 The Program Logic

Now we know how the strictness properties behave, we need a program logic which allows us to derive properties of terms in Λ_T . Since we want our system to be decidable, we know that the program logic cannot be complete for elementary computability reasons.

The inference rules for the program logic appear in figure 2. The notation is pretty standard – a context Γ is a finite set of assumptions of the form $x^\sigma : \phi_\sigma$ and we write $\Gamma, x^\sigma : \phi_\sigma$ to mean the context Γ with any existing binding for x^σ removed and the binding $x^\sigma : \phi_\sigma$ added. We restrict attention to *well-formed* judgements $\Gamma \vdash s^\sigma : \psi_\sigma$, in which all the free variables of s^σ appear in Γ .

To be able to talk about soundness, we have to extend the notion of satisfaction to terms in context. Write $\rho \models \Gamma$ if for all x^σ in $\text{dom}(\rho)$, $\Gamma = \Gamma', x^\sigma : \phi_\sigma$ and $\rho(x^\sigma) \models \phi_\sigma$. Then define $\Gamma \models s^\sigma : \phi_\sigma$ to mean that for all environments ρ such that $\rho \models \Gamma$, $\llbracket s^\sigma \rrbracket \rho \models \phi_\sigma$.

Proposition 4.1 (Soundness of the program logic)

If $\Gamma \vdash s^\sigma : \phi_\sigma$ then $\Gamma \models s^\sigma : \phi_\sigma$

□

The next proposition is useful in proving proof-theoretic properties of the program logic and would also be important in the design of an inference algorithm. It asserts that any derivation may be transformed into one which is ‘almost’ syntax-directed. Note that this is the reason for the inclusion in the program logic of the (apparently redundant) rule [absbot]. Without it, the result would be false.

Proposition 4.2 (Normal derivations) *If $\Gamma \vdash s^\sigma : \phi_\sigma$ is derivable in the program logic then there is a derivation in which the only uses of the rule [sub] occur immediately below axioms.*

Proof. This is essentially a cut-elimination argument. We first do an induction on the length of derivations with exactly one non-normal use of [sub], to show that such uses may be ‘floated up’ the derivation until they either disappear, hit an axiom or are absorbed into a normal use of [sub] by [trans]. A second induction on the number of non-normal uses of [sub] in a derivation completes the proof. \square

Proposition 4.3 (Subject β -conversion) *Derivable strictness properties are preserved by both β -reduction and β -expansion:*

$$\Gamma \vdash (\lambda x^\sigma. t^\tau)^{\sigma \rightarrow \tau} s^\sigma : \phi_\tau \quad \text{iff} \quad \Gamma \vdash t^\tau[s^\sigma/x^\sigma] : \phi_\tau$$

Proof. Both directions follow by induction on the structure of t^τ and consideration of the possible forms of normal derivations in the logic. \square

5 Polymorphic Invariance

We now consider moving from monomorphically typed terms to terms with explicit first-order polymorphic types. We show that the derivable strictness properties of different instances of a polymorphically typed term are related in a very pleasant way. As well as being an interesting theoretical result, this is important for any implementation, as it allows us to analyse polymorphic functions without analysing all their monotyped instances separately.

Polymorphic invariance of strictness analysis was first studied in [1]. A complication in that work was that the characterisation of strictness was denotational (semantic) whereas polymorphism came from the syntactic rules for type assignment. To reconcile these two viewpoints, Abramsky had to go via an operational semantics (which is inherently syntactic) for the abstract functions. Hughes has approached the problem semantically in [6], but the techniques used there are restricted to first-order functions. More recently, a better understanding of the semantics of polymorphism allowed Abramsky and Jensen to give a much neater semantic proof of invariance [3]. Here we show that our logical formulation of strictness analysis allows an enlightening and purely syntactic proof of invariance.

It should be noted that there are really two notions of polymorphic invariance – we can either show that strictness *properties* are polymorphic invariants (eg. if one instance of a function is strict, all instances are), or we can show that our *analysis method* detects a property at all types if it detects it at one. It is the second form which we consider here, and which Abramsky’s work addresses. He observes that this is the more important notion from the point of view of an implementation.

In fact, the situation is not as simple as the above might suggest. For example, the identity function specialised to arguments of type $A \rightarrow A$ has the property of taking strict functions to strict functions, but this cannot be a property of the version of the identity which is specialised to arguments of type A . What we actually show is that a strictness property holds at a particular instance of a term iff it holds at all instances for which that property makes sense (in a sense to be made precise).

We now extend our definition of Λ_T types to include type variables, which we range over with α, β

$$\sigma ::= \alpha \mid A \mid \sigma \rightarrow \tau \mid \sigma \times \tau$$

Terms are formed as before. In particular, they are still explicitly typed – we are merely allowing the types to contain type variables. We do not need to consider how such typed terms can be deduced from untyped terms. The propositional theory associated with a type variable α just contains \mathbf{t}_α and \mathbf{f}_α , so $\mathcal{L}_\alpha \cong \mathcal{L}_A$. The program logic remains unchanged and our previous proof-theoretic results go through as before.

We formalise the intuition that a property makes sense at any more complicated type as follows. We define the preorder \triangleleft to be the smallest relation on types which is closed under the following rules

$$\begin{array}{ccc} \sigma \triangleleft \sigma & A \triangleleft \sigma & \alpha \triangleleft \sigma \\ \frac{\sigma \triangleleft \sigma' \quad \tau \triangleleft \tau'}{\sigma \rightarrow \tau \triangleleft \sigma' \rightarrow \tau'} & & \frac{\sigma \triangleleft \sigma' \quad \tau \triangleleft \tau'}{\sigma \times \tau \triangleleft \sigma' \times \tau'} \end{array}$$

If we define a *substitution* to be a map from type variables to types (which extends in the obvious way to types), then it is plain that for any type σ and substitution S , $\sigma \triangleleft S(\sigma)$. We can also define the action of substitutions on terms, and it is clear that if s^σ is a well-formed term in our language, so is $S(s^\sigma)$.

We can now define the maps which embed one theory in a ‘more complicated’ one. For $\sigma \triangleleft \tau, \sigma' \triangleleft \tau'$, define $\mathcal{E}_{\sigma, \tau} : \mathcal{L}_\sigma \rightarrow \mathcal{L}_\tau$ by

$$\begin{aligned} \mathcal{E}_{\sigma, \tau}(\mathbf{t}_\sigma) &= \mathbf{t}_\tau & \mathcal{E}_{\sigma, \tau}(\mathbf{f}_\sigma) &= \mathbf{f}_\tau \\ \mathcal{E}_{\sigma, \tau}(\phi \wedge \psi) &= \mathcal{E}_{\sigma, \tau}(\phi) \wedge \mathcal{E}_{\sigma, \tau}(\psi) \\ \mathcal{E}_{\sigma \rightarrow \sigma', \tau \rightarrow \tau'}(\phi_\sigma \rightarrow \psi_{\sigma'}) &= \mathcal{E}_{\sigma, \tau}(\phi_\sigma) \rightarrow \mathcal{E}_{\sigma', \tau'}(\psi_{\sigma'}) \\ \mathcal{E}_{\sigma \times \sigma', \tau \times \tau'}(\phi_\sigma \times \psi_{\sigma'}) &= \mathcal{E}_{\sigma, \tau}(\phi_\sigma) \times \mathcal{E}_{\sigma', \tau'}(\psi_{\sigma'}) \end{aligned}$$

Proposition 5.1 *If $\sigma \triangleleft \tau$ and $\phi_\sigma \leq \psi_\sigma$ then $\mathcal{E}_{\sigma, \tau}(\phi_\sigma) \leq \mathcal{E}_{\sigma, \tau}(\psi_\sigma)$.* □

The above result says that entailments between propositions at simple types are still valid when we interpret the propositions at more complicated types, and hence that our embedding maps respect equivalence classes of propositions. We shall also need to know that the converse holds – if an inclusion holds at a complicated type between two propositions that make sense at a simpler type, then the inclusion holds at the simple type too.

Proposition 5.2 *If $\sigma \triangleleft \tau$ and $\mathcal{E}_{\sigma,\tau}(\phi_\sigma) \leq \mathcal{E}_{\sigma,\tau}(\psi_\sigma)$ then $\phi_\sigma \leq \psi_\sigma$.*

Proof. It is slightly tricky to show this directly from the logic. This is because of the [trans] rule (which is essentially a form of cut). Instead we note that Proposition 5.1 shows that $\mathcal{E}_{\sigma,\tau}$ induces $\bar{\mathcal{E}}_{\sigma,\tau} : \mathcal{L}\mathcal{A}_\sigma \rightarrow \mathcal{L}\mathcal{A}_\tau$. Since $\mathcal{L}\mathcal{A}_\sigma$ has, and $\bar{\mathcal{E}}_{\sigma,\tau}$ preserves, all meets, we know that $\bar{\mathcal{E}}_{\sigma,\tau}$ has a left adjoint $\bar{\mathcal{K}}_{\sigma,\tau} : \mathcal{L}\mathcal{A}_\tau \rightarrow \mathcal{L}\mathcal{A}_\sigma$ given by $\bar{\mathcal{K}}_{\sigma,\tau}(\phi_\tau) = \bigwedge \{\psi_\sigma \mid \bar{\mathcal{E}}_{\sigma,\tau}(\psi_\sigma) \geq \phi_\tau\}$. This, together with Lemma 3.7, allows one to show by induction on types that $\bar{\mathcal{K}}_{\sigma,\tau} \circ \bar{\mathcal{E}}_{\sigma,\tau}$ is the identity on $\mathcal{L}\mathcal{A}_\sigma$ from which the result follows. \square

We now turn to the relationship between the derivable strictness properties of differently typed instances of the same term. Define the action of a substitution on a strictness context by $S(\{x^\sigma : \phi_\sigma\}) = \{x^{S\sigma} : \mathcal{E}_{\sigma,S\sigma}(\phi_\sigma)\}$. Then if $\Gamma \vdash s^\sigma : \phi_\sigma$ is a well-formed strictness judgement, so is $S\Gamma \vdash S(s^\sigma) : \mathcal{E}_{\sigma,S\sigma}(\phi_\sigma)$.

Proposition 5.3 *If $\Gamma \vdash s^\sigma : \phi_\sigma$ then $S\Gamma \vdash S(s^\sigma) : \mathcal{E}_{\sigma,S\sigma}(\phi_\sigma)$* \square

This result is established by induction on derivations and can be seen as a soundness result for our system with the extended definition of type, as it says anything we can deduce at a type with type variables in can be deduced at all its ground instances and (by the previous soundness result) is therefore semantically valid at all of them.

The result which we really want is the converse to Proposition 5.3. However, the presence (even in normal derivations) of the [app] rule, which has the cut-like property of introducing a formula in the premises which does not appear in the conclusion, prevents a naive induction from working. Instead, we use a (unary) logical relation on terms in the style of Tait's proof of strong normalisation (see, for example, [10]). The details are slightly fiddly, but the argument is sketched below.

For a term t^σ , we define $\mathcal{P}(t^\sigma)$ to mean that for all Γ, S, ϕ_σ we have

$$S\Gamma \vdash S(t^\sigma) : \mathcal{E}_{\sigma,S\sigma}(\phi_\sigma) \quad \Rightarrow \quad \Gamma \vdash t^\sigma : \phi_\sigma$$

We then define a type-indexed family $\mathcal{G} = \{\mathcal{G}^\sigma\}$ of predicates on terms by

$$t^A \in \mathcal{G}^A \text{ iff } \mathcal{P}(t^A)$$

$$t^\alpha \in \mathcal{G}^\alpha \text{ iff } \mathcal{P}(t^\alpha)$$

$$t^{\sigma \rightarrow \tau} \in \mathcal{G}^{\sigma \rightarrow \tau} \text{ iff } \forall s^\sigma \in \mathcal{G}^\sigma. (t^{\sigma \rightarrow \tau} s^\sigma)^\tau \in \mathcal{G}^\tau$$

$$t^{\sigma \times \tau} \in \mathcal{G}^{\sigma \times \tau} \text{ iff } (\pi_1^{\sigma \times \tau} t^{\sigma \times \tau})^\sigma \in \mathcal{G}^\sigma \text{ and } (\pi_2^{\sigma \times \tau} t^{\sigma \times \tau})^\tau \in \mathcal{G}^\tau$$

Lemma 5.4 (Admissibility) *If $s^\sigma \in \mathcal{G}^\sigma$ and $t^\tau[s^\sigma/x^\sigma] \in \mathcal{G}^\tau$ then $((\lambda x^\sigma.t^\tau)s^\sigma) \in \mathcal{G}^\tau$.* \square

Lemma 5.5 *1. If $\mathcal{P}(t_i^{\tau_i})$ then $(x^\sigma t_1^{\tau_1} \dots t_n^{\tau_n}) \in \mathcal{G}$;
2. If $t^\tau \in \mathcal{G}^\tau$ then $\mathcal{P}(t^\tau)$.* \square

The proof of Lemma 5.4 is by induction on types and makes use of subject conversion. The two parts of Lemma 5.5 are proved simultaneously by induction on types. Next we show the following by induction on terms, using the previous two lemmas:

Lemma 5.6 (Basic Lemma) *If $s_i^{\sigma_i} \in \mathcal{G}^{\sigma_i}$ then for any t^τ we have $t^\tau[s_i^{\sigma_i}/x^{\sigma_i}] \in \mathcal{G}^\tau$.* \square

Note that to prove the basic lemma we also have to prove that each constant is in the appropriate \mathcal{G}^σ . This is not completely trivial. Finally, we can put all the pieces together to get:

Proposition 5.7 (Polymorphic invariance) *If $S\Gamma \vdash S(s^\sigma) : \mathcal{E}_{\sigma,S\sigma}(\phi_\sigma)$ then $\Gamma \vdash s^\sigma : \phi_\sigma$.* \square

One might suspect that the above is all unnecessary: maybe the only properties which we can ever show a complicated instance of a polymorphic function to have are ones which make sense at the simplest instance. More formally, we might imagine that the following holds:

$$\Gamma \vdash S(t^\sigma) : \phi_{S\sigma} \quad \Rightarrow \quad \exists \Gamma', \phi'_\sigma \text{ st. } \Gamma \leq S\Gamma', \mathcal{E}_{\sigma,S\sigma}(\phi'_\sigma) \leq \phi_{S\sigma} \text{ and } \Gamma' \vdash t^\sigma : \phi'_\sigma$$

It is shown in [6] that this is indeed the case for first-order functions. For higher-order functions, however, it is unfortunately false. See [3] for a counterexample.

So to get the best information we *do* have to analyse some non-basic instances of polymorphic functions. Even if we were only directly interested in simple strictness, and we have shown that we can get this by analysing at the simplest instance, to do that analysis for a function f , we may have to analyse a higher order instance of some function g , which is called by f . What our result tells us is that we only have to analyse the simplest instance of g for which the property we are looking for makes sense, not the instance at which g is actually called. This is a considerable improvement over the approach of [1], which suggests reanalysing each instance of g .

6 Conclusions and Further Work

We have shown how strictness analysis can be performed in a logical, rather than denotational, way. The analysis has been proved correct relative to the standard semantics and we have shown how the syntactic nature of the proof system leads to a relatively straightforward proof that the analysis is polymorphically invariant. An improvement on previous work is that we consider polymorphic invariance of higher-order properties, rather than just of a first-order property (simple strictness) of higher-order functions.

Jensen has shown [7] that $\mathcal{L}\mathcal{A}_\sigma$ is isomorphic to the abstract domain associated with σ in the conventional abstract interpretation approach to strictness analysis. One can also see the logic as giving an operational semantics for the relational abstractions proposed in, for example, [12]. This view deserves further investigation.

The treatment of products described in the present paper is rather weak. To get a better abstraction of products, and to deal with sums, we are naturally led to add disjunction to the logic. There is then no longer a simple equivalence between the logic and traditional abstract interpretation; this shows the benefits of relating the logic directly to the standard, rather than the abstract, semantics.

Further work needs to be done on implementing this kind of inference system. In practice, we probably do not want to use a ‘principal types’ algorithm, since that would correspond to computing the whole of the abstract function, which can be extremely expensive. Our logical approach separates out the individual properties, which get a bit lost in abstract interpretation. There is some reason to hope that this may lead to analysis algorithms which have better complexity, at least on average. One idea is to investigate the use of equational or order-sorted unification to obtain an inference algorithm. We also need to consider how best to make practical use of the results of section 5.

The kind of polymorphism which we have described is very simple. It would be also be interesting to study polymorphic invariance of program properties in the framework of a more powerful system, such as the Girard/Reynolds polymorphic lambda calculus.

7 Acknowledgements

I have benefitted greatly from discussing this material with several people. I would especially like to thank Gavin Bierman, Sebastian Hunt, Thomas Jensen, Alan Mycroft, Wesley Phoa and Andy Pitts.

References

- [1] Samson Abramsky. Strictness analysis and polymorphic invariance (extended abstract). In H. Ganzinger and N. Jones, editors, *Proceedings of the Workshop on Programs as Data Objects, Copenhagen*, volume 217 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, October 1985.

- [2] Samson Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51(1–2):1–78, 1991.
- [3] Samson Abramsky and Thomas P. Jensen. A relational approach to strictness analysis of higher order polymorphic functions. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1991.
- [4] P. N. Benton. *Static Analyses and Optimising Transformations for Lazy Functional Programs*. PhD thesis, University of Cambridge, 1992. in preparation.
- [5] Geoffrey L. Burn, Chris L. Hankin, and Samson Abramsky. The theory of strictness analysis for higher order functions. In H. Ganzinger and N. Jones, editors, *Proceedings of the Workshop on Programs as Data Objects, Copenhagen*, volume 217 of *Lecture Notes in Computer Science*. Springer-Verlag, October 1985.
- [6] R.J.M. Hughes. Abstract interpretation of first-order polymorphic functions. In *Glasgow Workshop on Functional Programming*, August 1988.
- [7] Thomas P. Jensen. Strictness analysis in logical form. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, 1991.
- [8] P. T. Johnstone. *Stone Spaces*. Cambridge studies in advanced mathematics. Cambridge University Press, 1982.
- [9] T.-M. Kuo and P. Mishra. Strictness analysis: A new perspective based on type inference. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*. ACM, 1989.
- [10] John C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 8, pages 365–458. Elsevier Science Publishers, 1990.
- [11] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, Department of Computer Science, University of Edinburgh, December 1981.
- [12] Alan Mycroft and Neil D. Jones. A relational framework for abstract interpretation. In H. Ganzinger and N. Jones, editors, *Proceedings of the Workshop on Programs as Data Objects, Copenhagen*, volume 217 of *Lecture Notes in Computer Science*. Springer-Verlag, October 1985.