Type Theory and Projections for Higher-Order Static Analysis

Clement A. Baker-Finch
University of Canberra*
clem@ise.canberra.edu.au

Abstract

A system of *annotated types* is proposed as a means of describing and inferring static information, such as strictness and constancy, about functional programs. An abstract semantics is given in terms of *projections*. The paper then proceeds to demonstrate a close connection between annotated type assignment and projection analysis.

1 Introduction.

Modern implementations of functional programming languages rely heavily on optimisations made on the basis of compile-time analysis of programs. The main approaches fall into three broad classes: those based on abstract interpretation (e.g. [Myc81, AH87]), those based on projections (e.g. [WH87, Lau91, Dav93, Bur90b]), and those based on type systems (e.g. [KM89, WBF93, Wad91, JG91]). The interrelations of these methods has attracted some study, with Burn [Bur90a] comparing abstract interpretation with projection analysis, and Jensen [Jen91] and Benton [Ben93] investigating links between abstract interpretation and the typing approach of Kuo and Mishra. The main contribution of this paper is to demonstrate a close correspondence between the annotated types approach of Wright [Wri91, Wri93] and Baker-Finch [BF92, BF93, WBF93] and projection analysis. This also suggests a clean extension of projection analysis to the higher-order case. The treatment of data structures and recursive types in the type system is new and hence of interest in its own right.

The paper is structured as follows. After defining the example language $\Lambda_{\mathcal{T}}$, annotated types are introduced and an assignment system is given. We survey some basic definitions and results about projections and then use the concept to give a non-standard semantics for $\Lambda_{\mathcal{T}}$ over the annotated types. Projection analysis is described and its relation with annotated types is demonstrated. Finally, we briefly show how to treat recursive types by extending the type system to include list data structures. Throughout, we use 'strictness and absence' as our sample analysis. This is chosen to follow the seminal paper [WH87], but the results established here apply to other projection analyses.

*PO Box 1 Belconnen, ACT, 2601, Australia

2 Standard semantics.

The language $\Lambda_{\mathcal{T}}$ to be considered in this article is a simply-typed λ-calculus augmented with pairs as an example data structure.

The type language is built on a finite set of base types $\imath, \jmath \in \mathcal{T}_0$, including *bool* and *int*. The types are then given by:

$$\sigma, \tau \in \mathcal{T} ::= \imath \mid \sigma \circ \tau \mid \sigma \to \tau$$

The type constructor $\_ \circ \_$ represents products. The notation is chosen to reflect the relevant logic connective 'fusion' [AB75, Dun86] which corresponds to the type assignment rules below. For the present paper this is only a notational matter.

At each type σ there is a countable set of variables Var_σ and a set of constants Con_σ.

$$x, y \in Var = \bigcup_{\sigma \in \mathcal{T}} Var_\sigma$$

$$c \in Con = \bigcup_{\sigma \in \mathcal{T}} Con_\sigma$$

For σ and τ distinct types, Var_σ, Var_τ, Con_σ and Con_τ are disjoint. The syntax of $\Lambda_{\mathcal{T}}$ terms is as follows:

$$\begin{aligned} e \in \Lambda_{\mathcal{T}} ::= \; & x \mid c \mid \lambda x.e \mid e_0 e_1 \\ & \mid \langle e_0, e_1 \rangle \mid \mathbf{pcase}\ e_0\ \mathbf{of}\ \langle x, y\rangle \Rightarrow e_1 \\ & \mid \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \mathbf{rec}(x)e \end{aligned}$$

Assume that the constants include: $\mathbf{n} : int$, for all $n \in \mathbb{Z}$; $\_ + \_ : int \to int \to int$; $\_ = \_ : int \to int \to bool$; and $\mathbf{true}, \mathbf{false} : bool$. The standard semantics of the language is straightforward and presented here as an interpretation I, distinguished from a non-standard semantics (interpretation S) given later in §5. We begin with a type-indexed family of domains $\{D^I_\sigma\}$ defined by induction on the structure of types in fig.1.

An *I-environment* ρ is a type-respecting partial function from variables to $\bigcup_{\sigma\in\mathcal{T}} D^I_\sigma$. The notation for environment update is $\rho[x \mapsto d]$. The standard semantic valuation function is $[\![\_]\!]^I : \Lambda_{\mathcal{T}} \to Env^I \to \bigcup_{\sigma\in\mathcal{T}} D^I_\sigma$ defined in fig.2.

The standard semantics of each constant c is given by value c^I (of the correct type). For example, $\mathbf{n}^I = n$, $\mathbf{true}^I = tt$ and $+^I$ is the addition function $D^I_{int} \to D^I_{int} \to D^I_{int}$.

$$[\![x]\!]^I\rho = \rho(x)$$

$$[\![c]\!]^I\rho = c^I$$

$$[\![\lambda x.e]\!]^I\rho = \lambda d \in D^I_\sigma.[\![e]\!]^I\rho[x \mapsto d] \quad \text{where } x \in Var_\sigma$$

$$[\![e_0 e_1]\!]^I\rho = ([\![e_0]\!]^I\rho)([\![e_1]\!]^I\rho)$$

$$[\![\langle e_0, e_1\rangle]\!]^I\rho = \langle [\![e_0]\!]^I\rho, [\![e_1]\!]^I\rho\rangle$$

$$[\![\mathbf{pcase}\ e_0\ \mathbf{of}\ \langle x,y\rangle \ \Rightarrow\ e_1]\!]^I\rho = [\![e_1]\!]^I\rho[x \mapsto d, y \mapsto d'] \quad \text{where } [\![e_0]\!]^I\rho = \langle d, d'\rangle$$

$$[\![\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]^I\rho = \begin{cases} [\![e_1]\!]^I\rho & \text{if } [\![e_0]\!]^I\rho = tt \\ [\![e_2]\!]^I\rho & \text{if } [\![e_0]\!]^I\rho = f\!f \\ \bot & \text{if } [\![e_0]\!]^I\rho = \bot \end{cases}$$

$$[\![\mathbf{rec}(x)e]\!]^I\rho = \bigsqcup_{i\in\omega} d_i \quad \text{where } d_0 = \bot,\ d_{i+1} = [\![e]\!]^I\rho[x \mapsto d_i]$$

Figure 2: Standard semantics for $\Lambda_{\mathcal{T}}$

$$D^I_{bool} = \{tt, f\!f\}_\bot$$

$$D^I_{int} = \mathbb{Z}_\bot$$

$$D^I_{\sigma\to\tau} = [D^I_\sigma \to D^I_\tau]$$

$$D^I_{\sigma\circ\tau} = D^I_\sigma \times D^I_\tau$$

Figure 1: I-domains

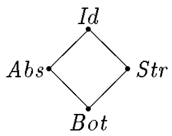

Figure 3: Strictness annotations

3 Annotated types.

The basic type language will now be extended to carry annotations which give information about the 'behaviour' or 'usage' of terms. This can be naturally reflected in a greater refinement of the semantic domains.

The sets of annotations will each have lattice ordering $\sqsubseteq$ here. In this paper we concentrate on annotations *Id*, *Str*, *Abs* and *Bot*, for strictness and absence analysis as in [WH87]. The ordering is $Bot \sqsubseteq Str \sqsubseteq Id$ and $Bot \sqsubseteq Abs \sqsubseteq Id$. We will write $\diamond$ for this lattice, resembling its Hasse diagram, shown in fig.3. The intuition behind these annotations is that *Str* (strict) indicates strictness, *Abs* (absent) indicates constancy, *Bot* (bottom) indicates divergence and *Id* (identity) gives no information. The $\sqsubseteq$ ordering also makes sense under this intuition. Semantically, these annotations represent *projections* (§4).

Some other possible lattices are $\mathbf{2} = Str \sqsubseteq Id$ for simple strictness analysis and $Bot \sqsubseteq Id$ (respectively representing *static* and *dynamic*) for binding-time analysis [Hun91, HS91]. Another system with application for sharing analysis has been variously presented by (at least) Hughes [Hug90], Bierman [Bie92], Baker-Finch [BF93] and Sestoft [Ses91] and is based on distinguishing *0*, *1* or *Many* 'uses'. However, given its essentially operational nature it does not seem feasible to describe the semantics of such a system in the style used in this paper.

The annotations are used to decorate the simple types as follows, for annotations α, β:

$$\sigma, \tau \in \mathcal{T} ::= \imath \mid \sigma\ {}_\alpha{\circ}_\beta \tau \mid \sigma \xrightarrow{\alpha} \tau$$

Intuitively, an annotation like $\sigma \xrightarrow{Str} \tau$ indicates that functions of this type are strict, while $\sigma \xrightarrow{Bot} \tau$ indicates a divergent function, $\sigma \xrightarrow{Abs} \tau$ indicates a constant function and $\sigma \xrightarrow{Id} \tau$ give no more information than the basic type $\sigma \to \tau$. A pair value with type $\sigma\ {}_\alpha{\circ}_\beta \tau$ will have its first and second components evaluated according to α and β respectively, when the pair itself is evaluated. For example, a function to return the first element of a pair will have a type $\sigma\ {}_{Str}{\circ}_{Abs} \tau \xrightarrow{Str} \sigma$.

The type assignment system over $\diamond$ is given in fig.4. As usual, Γ represents an *assumption set* consisting of bindings of variables to (annotated) types. However, each binding also carries an annotation indicating the demand on that variable in the expression being assigned a type, e.g. $x^\alpha\sigma$. To indicate that the assumption sets carry such annotations, we write Γ^J. In particular, Γ^{Abs} denotes an assumption set wherein all the variable annotations are *Abs*. When Γ^J and Γ^K appear in the same rule, the intention is that the

$$\frac{}{\Gamma^{Abs}, x^{Str}\sigma \vdash x : \sigma}\ Axiom$$

$$\frac{\Gamma^{J}, x^{\alpha}\sigma \vdash e : \tau}{\Gamma^{J} \vdash \lambda x.e : \sigma \xrightarrow{\alpha} \tau} \rightarrow\mathcal{I} \qquad \frac{\Gamma^{J} \vdash e_0 : \sigma \xrightarrow{\alpha} \tau \quad \Gamma^{K} \vdash e_1 : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma^{J+\alpha\times K} \vdash e_0 e_1 : \tau} \rightarrow\mathcal{E}$$

$$\frac{\Gamma^{J} \vdash e_0 : \sigma \quad \Gamma^{K} \vdash e_1 : \tau}{\Gamma^{\alpha\times J+\beta\times K} \vdash \langle e_0, e_1\rangle : \sigma\, {}_{\alpha}\!\circ_{\beta} \tau} \circ\mathcal{I} \qquad \frac{\Gamma^{J} \vdash e_0 : \sigma\, {}_{\alpha}\!\circ_{\beta} \tau \quad \Gamma^{K}, x^{\alpha}\sigma, y^{\beta}\tau \vdash e_1 : \nu}{\Gamma^{J+K} \vdash \mathbf{pcase}\ e_0\ \mathbf{of}\ \langle x, y\rangle \Rightarrow e_1 : \nu} \circ\mathcal{E}$$

$$\frac{\Gamma^{J} \vdash e_0 : bool \quad \Gamma^{K} \vdash e_1 : \sigma \quad \Gamma^{L} \vdash e_2 : \sigma}{\Gamma^{J+(K\sqcup L)} \vdash \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \sigma}\ Cond \qquad \frac{\Gamma^{J}, x^{\alpha}\sigma \vdash e : \sigma}{\Gamma^{J} \vdash \mathbf{rec}\ (x)e : \sigma}\ Rec$$

where $\sqsubseteq$ is extended as follows:
$\sigma \sqsubseteq \sigma$
$\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$ and $\alpha \sqsubseteq \alpha' \implies (\sigma' \xrightarrow{\alpha} \tau) \sqsubseteq (\sigma \xrightarrow{\alpha'} \tau')$
$\sigma \sqsubseteq \sigma'$ and $\tau \sqsubseteq \tau'$ and $\alpha \sqsubseteq \alpha'$ and $\beta \sqsubseteq \beta' \implies (\sigma\, {}_{\alpha}\!\circ_{\beta} \tau) \sqsubseteq (\sigma'\, {}_{\alpha'}\!\circ_{\beta'} \tau')$

Figure 4: Annotated type assignment

type-bindings are the same but the annotations may differ. The reader should be aware that, even though the variable annotations appear to the left of the turnstile, they are in fact completely determined by the derivation.

The constants are assigned types in the obvious way: $\Gamma^{Abs} \vdash \mathbf{n} : int$ for all integers n; $\Gamma^{Abs} \vdash \_+\_ : int \xrightarrow{Str} int \xrightarrow{Str} int$, and so on, with the annotations in the assumption set being Abs to indicate that no free variables are used in the constants. The annotation ordering $\sqsubseteq$ has been extended in the natural way to type expressions. This ordering is used in the side condition to the $\rightarrow\mathcal{E}$ rule to allow the weakening of information in the type annotations. Without such an option there would be terms with a simple type but with no annotated type. For example, consider a term containing two occurences of its outermost bound variable, such as f in $\lambda f.\ldots(f\mathbf{I})\ldots(f(\mathbf{K}X))\ldots$. Now $\mathbf{I}$ must have type $\sigma \xrightarrow{Str} \sigma$ and $\mathbf{K}X$ must have type $\sigma \xrightarrow{Abs} \sigma$. To resolve this conflict we may choose f to have type $(\sigma \xrightarrow{Id} \sigma) \xrightarrow{\cdots} \tau$, thus allowing both $f\mathbf{I}$ and $f(\mathbf{K}X)$ to be typed.

The operations $+, \times$ and $\sqcup$ on the assumption set annotations are extensions of simple operations on $\diamond$. As may be expected $\sqcup$ is *join* over $\diamond$. $+$ and $\times$ are as defined below.

$+$	*Id*	*Str*	*Abs*	*Bot*
Id	*Id*	*Str*	*Id*	*Bot*
Str	*Str*	*Str*	*Str*	*Bot*
Abs	*Id*	*Str*	*Abs*	*Bot*
Bot	*Bot*	*Bot*	*Bot*	*Bot*

Table 1: $+$ over $\diamond$

Now $\Gamma^{\alpha\times J}$ indicates that each variable annotation in Γ^{J} is multiplied[1] by α. Γ^{J+K} adds the annotations of corresponding variables and similarly for $\Gamma^{J\sqcup K}$.

$\times$	*Id*	*Str*	*Abs*	*Bot*
Id	*Id*	*Id*	*Abs*	*Abs*
Str	*Id*	*Str*	*Abs*	*Bot*
Abs	*Abs*	*Abs*	*Abs*	*Abs*
Bot	*Bot*	*Bot*	*Bot*	*Bot*

Table 2: $\times$ over $\diamond$

The annotations on pair and arrow types have respectively different interpretations (to be formalised later). The annotations on pair types are like the annotations on bindings in the type assumption sets. For example,

$$\Gamma^{J}, x^{\alpha}\sigma\, {}_{\beta}\!\circ_{\gamma} \tau \vdash e : \nu$$

indicates that evaluating $e : \nu$ creates a demand of α on x, β on the first component of x and γ on the second component of x. Hence, $+, \times$ and $\sqcup$ applied to annotated type assumption sets should apply to pair annotations in the same way that they apply to the variable binding annotations.

On the other hand, the annotations on arrow types indicate some expected behaviour of terms with that type. For example,

$$\Gamma^{J}, x^{\alpha}\sigma \xrightarrow{\beta} \tau \vdash e : \nu$$

indicates that only terms with behaviour consistent with $\sigma \xrightarrow{\beta} \tau$ can be substituted for x if $e : \nu$ is to be a correct typing. Thus $+, \times$ and $\sqcup$ should *not* apply to arrow annotations (or other annotations within arrow types) since they indicate a behaviour rather than a demand within $e : \nu$. Furthermore, if x is bound to an arrow type in two assumption

[1] Note that $\times$ is not commutative. In particular, $Abs \times Bot \neq Bot \times Abs$ and $Id \times Bot \neq Bot \times Id$. This is in line with the correspondence with [WH87] established in §6.

sets Γ^J and Γ^K which are to be combined, then the annotations on the type of x must be the same in both. This is not the case for pair types.

Definition 3.1 The operations $+$ and $\sqcup$ are extended to annotated types and type assumption sets as follows (taking $\star$ to be either $+$ or $\sqcup$):

- $(\Gamma^J, x^\alpha\sigma) \star (\Gamma^K, x^\beta\sigma') = \Gamma^{J\star K}, x^{(\alpha\star\beta)}(\sigma \star \sigma')$
- $\imath \star \imath = \imath$, for all $\imath \in \mathcal{T}_0$
- $(\sigma \xrightarrow{\alpha} \tau) \star (\sigma \xrightarrow{\alpha} \tau) = (\sigma \xrightarrow{\alpha} \tau)$
- $(\sigma\ {}_\alpha{\circ}_\beta \tau) \star (\sigma'\ {}_{\alpha'}{\circ}_{\beta'} \tau') = (\sigma \star \sigma')\ {}_{(\alpha\star\alpha')}{\circ}_{(\beta\star\beta')} (\tau \star \tau')$

The operation $\times$ is extended to annotated types and type assumption sets as follows:

- $\alpha \times (\Gamma^J, x^\beta\sigma) = \Gamma^{\alpha\times J}, x^{(\alpha\times\beta)}(\alpha \times \sigma)$
- $\alpha \times \imath = \imath$, for all $\imath \in \mathcal{T}_0$
- $\alpha \times (\sigma \xrightarrow{\beta} \tau) = (\sigma \xrightarrow{\beta} \tau)$
- $\alpha \times (\sigma\ {}_\beta{\circ}_\gamma \tau) = (\alpha \times \sigma)\ {}_{(\alpha\times\beta)}{\circ}_{(\alpha\times\gamma)} (\alpha \times \tau)$

In the *Rec* rule it appears that the J, α and σ must be 'guessed', but they can in fact be calculated in the obvious way by taking $\lim_{n\to\infty}$ of $\Gamma^{J_n}, x^{\alpha_n}\sigma_n \vdash e : \sigma_{n+1}$, where σ_0 has all arrows labelled with the least annotation (that is *Bot* in the case of $\diamond$).

Example (Fibonacci). As an example of this 'iterative' approach, consider the accumulating parameter definition of the Fibonacci function in fig.5. The details of the typing process follow. (For notational brevity, *Str, Abs, Bot* and *Id* are contracted to S, A, B and I respectively.) The limiting type indicates that *fib2* is strict in r and n but not necessarily strict in t.

To demonstrate the behaviour with pairs, fig.6 treats the(partly) uncurried version of the same function. The final type indicates that *pfib* is strict in both p and n but only the first element of the pair p may be safely evaluated in a strict context.

4 Projections

This section will briefly survey *projections* as used by Wadler and Hughes [WH87]. In order to talk about strictness, they attempt to characterise the concept of the necessity of some evaluation (rather than just its sufficiency) by lifting domains with a new least element which they write as a 'lightning bolt' to distinguish it from the existing bottom element. All functions are made strict on this new element.

In this paper however, we use more standard domain constructions and operators, hoping for greater familiarity. The notation is that of Gunter and Scott [GS90].

Definition 4.1 If D is a domain, then *liftD* has elements $\{\langle 0, d\rangle \mid d \in D\} \cup \{\bot\}$ with ordering:

- $\bot \sqsubset d$ for all $d \in \mathit{liftD}$
- $\langle 0, d\rangle \sqsubseteq \langle 0, d'\rangle$ iff $d \sqsubseteq d'$

For notational perspicuity, we will write $\mathit{lift}\, d$ rather than $\langle 0, d\rangle$. When there is no risk of ambiguity, we may just write d rather that $\mathit{lift}\, d$.

If D and E are domains, let $[D \circ\!\!\to E]$ denote the domain of *strict* functions from D to E. There is a continuous surjection $\mathit{strict} : [D \to E] \to [D \circ\!\!\to E]$ defined as follows:

$$\mathit{strict}\, f\, x = \begin{cases} f x & \text{if } x \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

If D and E are domains, their *smash product* $D \otimes E$ is the set $\{\langle x, y\rangle \in D \times E \mid x \neq \bot \text{ and } y \neq \bot\} \cup \{\bot\}$ with the component-wise ordering inherited from $D \times E$ and $\bot \sqsubseteq d$ for all $d \in D \times E$. Essentially, all pairs with a $\bot$ component are identified with $\bot$, making the pair constructor strict, as required. There is a continuous surjection $\mathit{smash} : D \times E \to D \otimes E$ defined as follows:

$$\mathit{smash}\langle x, y\rangle = \begin{cases} \langle x, y\rangle & \text{if } x \neq \bot \text{ and } y \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Definition 4.2 A continuous function α is a *projection* if it is idempotent ($\alpha \circ \alpha = \alpha$) and less or equal to than the identity ($\alpha \sqsubseteq \mathit{Id}$).

- The greatest projection is *Id* where $\mathit{Id}\, d = d$, for all d
- The least projection is *Bot* where $\mathit{Bot}\, d = \bot$, for all d
- A projection α over *liftD* is *lift-strict* if $\alpha(\mathit{lift}\bot) = \bot$
- The greatest lift-strict projection is *Str* where, for all d,

$$\mathit{Str}\, d = \begin{cases} d & \text{if } \mathit{lift}\bot \sqsubset d \\ \bot & \text{otherwise} \end{cases}$$

- Finally define *Abs* over *liftD* as follows:

$$\mathit{Abs}\, d = \begin{cases} \mathit{lift}\bot & \text{if } \bot \sqsubset d \\ \bot & \text{otherwise} \end{cases}$$

The annotations on types are intended to correspond to these projections in the precise way described below.

Definition 4.3 A function f is *α-strict in a β-context* if $\beta \circ f \circ \alpha = \beta \circ f$. This is indicated by writing $f : \beta \Rightarrow \alpha$.

More details will be given in §6 but the aim of projection analysis is basically to find such an α for each β and each function defined in a $\Lambda_\mathcal{T}$ program. For example, if $f : \beta \Rightarrow \mathit{Str}$ then it is safe to translate the function as though it were strict (in its first argument). In essence, the analysis technique consists of a process for deriving *projection transformers* (i.e. maps from projections to projections) for $\Lambda_\mathcal{T}$ expressions. For example, given a definition of f, the aim is to derive t_f such that $f : \beta \Rightarrow (t_f\beta)$ for all β.

Wadler and Hughes [WH87] introduce the notion of a *guard* to simplify the process of calculating these projection transformers.

Definition 4.4 Given a set of projections $\{\alpha_1, \ldots \alpha_n\}$ over some domain, suppose that a particular projection α_G in the set is such that for all f, if we are given $f : \alpha_G \Rightarrow \beta$ then we can deduce $f : \alpha_i \Rightarrow \beta_i$ for all $i = 1 \ldots n$. Then *α_G has the guard property for $\{\alpha_1, \ldots \alpha_n\}$.*

$fib = \lambda n. fib2\ 1\ 1\ n$
$fib2 = \mathbf{rec}\ (f)\lambda rtn.\mathbf{if}\ n = 1\ or\ 2\ \mathbf{then}\ r\ \mathbf{else}\ f(r+t)r(n-1)$

$$\begin{array}{rcl}
f : int \overset{Bot}{\to} int \overset{Bot}{\to} int \overset{Bot}{\to} int & \vdash & fib2 : int \xrightarrow{A+(S\sqcup B)} int \xrightarrow{A+(A\sqcup B)} int \xrightarrow{S+(A\sqcup B)} int \\
 & \equiv & fib2 : int \overset{Str}{\to} int \overset{Abs}{\to} int \overset{Str}{\to} int \\
f : int \overset{Str}{\to} int \overset{Abs}{\to} int \overset{Str}{\to} int & \vdash & fib2 : int \xrightarrow{A+(S\sqcup S)} int \xrightarrow{A+(A\sqcup S)} int \xrightarrow{S+(A\sqcup S)} int \\
 & \equiv & fib2 : int \overset{Str}{\to} int \overset{Id}{\to} int \overset{Str}{\to} int
\end{array}$$

Figure 5: Fibonacci

$$\begin{array}{l}
pfib = \mathbf{rec}\ (f)\lambda pn.\ \mathbf{if}\ n = 1\ or\ 2 \\
\qquad\qquad\qquad \mathbf{then\ pcase}\ p\ \mathbf{of}\ \langle res, prev\rangle \ \Rightarrow\ res \\
\qquad\qquad\qquad \mathbf{else\ pcase}\ p\ \mathbf{of}\ \langle res, prev\rangle \ \Rightarrow\ f\langle res + prev, res\rangle(n-1)
\end{array}$$

$$\begin{array}{rcl}
f : int\ {}_{Bot}\circ_{Bot} int \overset{Bot}{\to} int \overset{Bot}{\to} int & \vdash & pfib : int\ {}_{S}\circ_{A} int \xrightarrow{A+(S\sqcup S)} int \xrightarrow{S+(A\sqcup B)} int \\
 & \equiv & pfib : int\ {}_{Str}\circ_{Abs} int \overset{Str}{\to} int \overset{Str}{\to} int \\
f : int\ {}_{Str}\circ_{Abs} int \overset{Str}{\to} int \overset{Str}{\to} int & \vdash & pfib : int\ {}_{S\sqcup S}\circ_{A\sqcup S} int \xrightarrow{A+(S\sqcup S)} int \xrightarrow{S+(A\sqcup S)} int \\
 & \equiv & pfib : int\ {}_{Str}\circ_{Id} int \overset{Str}{\to} int \overset{Str}{\to} int
\end{array}$$

Figure 6: Fibonacci (pairs)

If the guard property holds, the obvious advantage is that projection transformers can be completely defined simply by defining them for the single value α_G.

Proposition 4.5 *Str* has the guard property for $\diamond$.

Proof For all f, $f : Bot \Rightarrow Bot$, $f : Abs \Rightarrow Abs$, and $f : Id \Rightarrow Abs \sqcup \beta$ where $f : Str \Rightarrow \beta$. $\square$

Thus, if we know $f : Str \Rightarrow \beta$ then we can simply work out β' in $f : \alpha \Rightarrow \beta'$ for all α. The guard property plays an important part in the establishment of a correspondence between the type assignment system and projection-based context analysis (§6).

5 Semantics of annotated types

Intuitively, if some function f is given type $\sigma \overset{\alpha}{\to} \tau$, this can be interpreted as indicating that f is α-strict in a *Str* context (and by the guard property, this is sufficient to deduce its behaviour in all other contexts too). For example, $\sigma \overset{Str}{\to} \tau$ indicates a strict function while $\sigma \overset{Bot}{\to} \tau$ indicates a divergent function.

We begin by modifying the standard semantics in line with the ideas of the preceding section by giving an interpretation S (for *strictness*) of the language $\Lambda_{\mathcal{T}}$. First the semantic domains are all lifted with a new least element, such that all functions will be strict on this new bottom. The type-indexed family of domains $\{D^S_\sigma\}$ is defined inductively on the structure of types σ in fig.7.

$$D^S_\imath = liftD^I_\imath, \text{ for all } \imath \in \mathcal{T}_0$$

$$D^S_{\sigma\to\tau} = [D^S_\sigma \circ\!\!\to D^S_\tau]$$

$$D^S_{\sigma\circ\tau} = D^S_\sigma \otimes D^S_\tau$$

Figure 7: S-domains

Now assuming S-environments to be type-respecting as discussed above, the semantics of $\Lambda_{\mathcal{T}}$ over the D^S domains is given by a semantic valuation function similar to the standard interpretation I of §2.

Now we must account for the annotations on types. Recall that there are essentially two different applications of the annotations – those on *arrows* tell us about how functions of that type use their arguments, while those on *pair types* indicate how the pair elements will be used by their context. Hence we should expect there to be two different

$$[\![x]\!]^S\rho = \rho(x)$$

$$[\![c]\!]^S\rho = c^S$$

$$[\![\lambda x.e]\!]^S\rho = \mathit{strict}(\lambda d \in D^S_\sigma . [\![e]\!]^S\rho[x \mapsto d]) \quad \text{where } x \in \mathit{Var}_\sigma$$

$$[\![e_0 e_1]\!]^S\rho = ([\![e_0]\!]^S\rho)([\![e_1]\!]^S\rho)$$

$$[\![\langle e_0, e_1\rangle]\!]^S\rho = \mathit{smash}\langle [\![e_0]\!]^S\rho, [\![e_1]\!]^S\rho\rangle$$

$$[\![\mathbf{pcase}\ e_0\ \mathbf{of}\ \langle x,y\rangle \ \Rightarrow\ e_1]\!]^S\rho = \begin{cases} [\![e_1]\!]^S\rho[x \mapsto d, y \mapsto d'] & \text{if } [\![e_0]\!]^S\rho = \langle d, d'\rangle \\ \bot & \text{if } [\![e_0]\!]^S\rho = \bot \end{cases}$$

$$[\![\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]^S\rho = \begin{cases} \bot & \text{if } [\![e_0]\!]^S\rho = \bot \text{ or } [\![e_1]\!]^S\rho = \bot \text{ or } [\![e_2]\!]^S\rho = \bot \\ \text{otherwise:} & \\ [\![e_1]\!]^S\rho & \text{if } [\![e_0]\!]^S\rho = \mathit{tt} \\ [\![e_2]\!]^S\rho & \text{if } [\![e_0]\!]^S\rho = \mathit{ff} \\ \mathit{lift}\bot & \text{if } [\![e_0]\!]^S\rho = \mathit{lift}\bot \end{cases}$$

$$[\![\mathbf{rec}(x)e]\!]^S\rho = \bigsqcup_{i\in\omega} d_i \quad \text{where } d_0 = \mathit{lift}\bot,\ d_{i+1} = [\![e]\!]^S\rho[x \mapsto d_i]$$

Figure 8: Lifted semantics for $\Lambda_{\mathcal{T}}$

(but interacting) semantic interpretations. First, if an expression is given type $\sigma\ {}_\alpha{\circ}_\beta \tau$ then the context containing the expression places demand α on the first element of the pair and demand β on the second element of the pair. That is, projections α and β can safely be applied to the respective pair elements.

Definition 5.1 For each annotated type σ there is a continuous function $\_ \restriction \sigma : D^S_\sigma \to D^S_\sigma$ such that, for all $\langle d, d'\rangle \in D^S_{\sigma\circ\tau}$,

$$\langle d, d'\rangle \restriction (\sigma\ {}_\alpha{\circ}_\beta \tau) = \langle \alpha(d \restriction \sigma), \beta(d' \restriction \tau)\rangle$$

and for σ not a pair type, $\_ \restriction \sigma$ is the identity.

We have already signalled our intention that giving f a type $\sigma \xrightarrow{\alpha} \tau$ means that f is α-strict in a *Str* context, that is $\mathit{Str}(fd) = \mathit{Str}(f(\alpha d))$. This can be extended to deal with pairs as arguments and results and formalised by the following definition.

Definition 5.2 For each annotated type σ there is a boolean valued function $\_\,\mathbf{in}\,\sigma$ on D^S_σ where $f\,\mathbf{in}\,\sigma \xrightarrow{\alpha} \tau$ iff $f \in D^S_{\sigma\to\tau}$ and for all $d\,\mathbf{in}\,\sigma$,

$$\mathit{Str}(f(d \restriction \sigma) \restriction \tau) = \mathit{Str}(f(\alpha(d \restriction \sigma)) \restriction \tau)$$

and for σ not a function type, $d\,\mathbf{in}\,\sigma$ iff $d \in D^S_\sigma$.

We are now in a position to develop the necessary correctness result. That is, the annotated types that can be assigned to a term reflect the semantic interpretation of that term.

Definition 5.3 An S-environment *satisfies* Γ^J provided that if $x^\alpha\sigma \in \Gamma^J$ then $\rho x\,\mathbf{in}\,\sigma$. Write $\Gamma^J \models e : \tau$ iff for all ρ satisfying Γ^J,

- $[\![e]\!]^S\rho\,\mathbf{in}\,\tau$
- for all $x^\alpha\sigma \in \Gamma^J$, $[\![\lambda x.e]\!]^S\rho\,\mathbf{in}\,\sigma \xrightarrow{\alpha} \tau$

A more direct but less readable equivalent to the second clause is:

- for all $x^\alpha\sigma \in \Gamma^J$,
  $\mathit{Str}([\![e]\!]^S\rho \restriction \tau) = \mathit{Str}([\![e]\!]^S\rho[x \mapsto \alpha((\rho x) \restriction \sigma)]) \restriction \tau)$

The correctness of the type system is expressed by the following soundness result:

Proposition 5.4 If $\Gamma^J \vdash e : \tau$ then $\Gamma^J \models e : \tau$.

Proof By induction over the length of the derivation of $\Gamma^J \vdash e : \tau$. □

6 Projection analysis.

Wadler and Hughes use projections in [WH87] to formalise and clarify the so-called backwards strictness analysis proposed in earlier papers, e.g. [Hug87] (see also [Hug90]). Essentially, given a context (i.e. a projection) indicating the demand made on the result of a function application, the aim is to deduce the demand made on each argument. The term 'backwards' is suggested by the observation that information flows from functions to their arguments, rather than the opposite, as in abstract interpretation. The next section demonstrates and explores a correspondence between such projection analyses and the type assignment system presented in this paper, which goes beyond simply noting that projections provide a common semantic basis. Given that the two techniques were developed independently, this is a noteworthy observation. The presentation in [WH87] is for a first-order language, so we will begin with that restriction in place. With reference back to the definitions and results about projections in §4, we briefly survey the approach of [WH87] beginning with some notation:

$$e^x\alpha = \beta$$

where α, β are projections, indicates that if e is evaluated in context α then the demand on the free variable x is β. More formally,

$$e^x\alpha = \beta \implies \alpha([\![e]\!]^S\rho) = \alpha([\![e]\!]^S\rho[x \mapsto \beta(\rho x)])$$

Note the close correspondence with the 3rd clause of defn.5.3. Accordingly, we could write

$$[\![\lambda x.e]\!]^S\rho : \alpha \Rightarrow \beta \quad \text{for} \quad e^x\alpha = \beta$$

Assuming a function definition like $fx_1 \ldots x_n = e$, Wadler and Hughes write:

$$f^i\alpha = \beta$$

to indicate that if an application of f is evaluated in context α then the demand on the ith argument is β. In terms of the notation introduced above, the following serves as a definition of f^i:

$$f^i\alpha \;=\; e^{x_i}\alpha$$

If we restrict attention to first-order terms of $\Lambda_{\mathcal{T}}$ and ignore pairs for now, the projection analysis over $\Diamond$ according to [WH87] is given in fig.9, where α is assumed to have the guard property and the results for other contexts can be deduced according to the technique given in §4. (Thus, over $\Diamond$, we may replace α with Str in all of the clauses above.)

$$x^x\alpha = \alpha$$

$$y^x\alpha = Abs \quad \text{if } x \neq y$$

$$c^x\alpha = Abs \quad \text{for all constants } c$$

$$(fe_1 \ldots e_n)^x\alpha = e_1^x(f^1\alpha) \,\&\, \ldots \,\&\, e_n(f^n\alpha)$$

$$(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2)^x\alpha = e_0^x Str \,\&\, (e_1^x\alpha \sqcup e_2^x\alpha)$$

Figure 9: Simple projection analysis

Definition 6.1 Wadler and Hughes [WH87] give the following definition for the operator & used above to combine projections:

$$(\alpha \& \beta)d = \begin{cases} \bot & \text{if } \alpha d = \bot \text{ or } \beta d = \bot \\ \alpha d \sqcup \beta d & \text{otherwise} \end{cases}$$

While higher-order projection analyses have been developed [DW90], the first-order case is more widely known and straightforward. To simplify comparisons between our work and [WH87] we restrict the type system to first-order in the obvious way. That is, all variables must be of atomic type and abstractions must take place 'all at once'. This requires minor modification of the rules of fig.4, to derive the first-order system shown in fig.10.

Proposition 6.2 The restricted type system corresponds to projection analysis as follows:

$$\Gamma^J, x^\alpha\sigma \vdash e : \tau \quad \Longleftrightarrow \quad e^x Str = \alpha$$

and

$$\vdash f : \sigma_1 \overset{\alpha_1}{\to} \cdots \sigma_n \overset{\alpha_n}{\to} \tau \quad \Longleftrightarrow \quad f^i Str = \alpha_i \quad \text{for all } i$$

Note that, by the guard property, fixing on the projection Str does not constitute a restriction of this result.

Proof Straightforward induction on the length of derivations. □

Since the type language has usage information distributed through type expressions, the projections in the result above are 'atomic'. In contrast, [Hug90, WH87] construct more complex projections to describe not only the usage of values but, in the case where values are structured, also the usage of the components of such values. Here we will follow the earlier notation of Hughes [Hug87, Hug90]. Consider a pair value. We are interested in the demand on the pair itself and, if it is evaluated, the demand on the first and second elements of the pair. If the components are also pairs then similar information is desired and so on. This can be indicated as $\alpha_{\beta,\gamma}$ to say that the demand on the pair is α, the demand on the first component is β and the demand on the second component is γ. Since the components may also be structured, β and γ may also be subscripted in this fashion.

For example, the demand on the argument to a function which returns the first element of a pair can be given as follows:

$$(\textbf{pcase } p \textbf{ of } \langle x, y\rangle \;\Rightarrow\; x)^p Str = Str_{Str,Abs}$$

To complete the projection analysis described above, we add the following clauses:

$$\langle e_0, e_1\rangle^x\alpha_{\beta,\gamma} = \alpha(e_0^x\beta \,\&\, e_1^x\gamma)$$

$$(\textbf{pcase } e_0 \textbf{ of } \langle x, y\rangle \;\Rightarrow\; e_1)^z\alpha = e_0^z\alpha_{\beta,\gamma} \,\&\, e_1^z\alpha \quad \text{where } e_1^x\alpha = \beta \text{ and } e_1^y\alpha = \gamma$$

Figure 11: Projection analysis for pairs

The extension to prop.6.2 to take account of these structured projections must include consideration of equivalences like:

$$\Gamma^J, x^\alpha\sigma\,{}_\beta{\circ}_\gamma\sigma' \vdash e : \tau \quad \Longleftrightarrow \quad e^x Str = \alpha_{\beta,\gamma}$$

and

$$\Gamma^J, x^\alpha\sigma \vdash e : \tau\,{}_\beta{\circ}_\gamma\tau' \quad \Longleftrightarrow \quad e^x Str_{\beta,\gamma} = \alpha$$

In general the correspondence result can be restated as follows:

Proposition 6.3 The type system corresponds to projection analysis as follows:

$$\Gamma^J, x^\alpha\sigma \vdash e : \tau \quad \Longleftrightarrow \quad e^x Str_{|\tau|} = \alpha_{|\sigma|}$$

and, for all $i = 1 \ldots n$,

$$\vdash f : \sigma_1 \overset{\alpha_1}{\to} \cdots \sigma_n \overset{\alpha_n}{\to} \tau \quad \Longleftrightarrow \quad f^i Str_{|\tau|} = \alpha_{i_{|\sigma_i|}}$$

where $|\imath|$ and $|\sigma \to \tau|$ are null and $|\sigma\,{}_\alpha{\circ}_\beta\tau| = \alpha_{|\sigma|}, \beta_{|\tau|}$.

Proof Extend the inductive proof of prop.6.2 with cases for $\circ\mathcal{I}$ and $\circ\mathcal{E}$. □

$$\frac{x_1^{\alpha_1}\imath_1, \ldots, x_n^{\alpha_n}\imath_n \vdash e : \jmath \qquad f x_1 \ldots x_n = e}{\vdash f : \imath_1 \stackrel{\alpha_1}{\rightarrow} \ldots \imath_n \stackrel{\alpha_n}{\rightarrow} \jmath} \rightarrow\mathcal{I}$$

$$\frac{\vdash f : \imath_1 \stackrel{\alpha_1}{\rightarrow} \ldots \imath_n \stackrel{\alpha_n}{\rightarrow} \jmath \qquad \Gamma^{J_1} \vdash e_1 : \imath_1 \quad \ldots \quad \Gamma^{J_n} \vdash e_n : \imath_n}{\Gamma^{\alpha_1 \times J_1 + \cdots + \alpha_n \times J_n} \vdash f e_1 \ldots e_n : \jmath} \rightarrow\mathcal{E}$$

Figure 10: First-order annotated type assignment

This demonstrates that annotated type assignment and projection analysis produce the same results but in fact the correspondence goes further than that, involving the basic operations underlying the two techniques. Projection analysis is based on the composition of projection transformers and the operator &. The basic operations of the type assignment system are $\times$ and $+$ as defined in §3 and they come into play at the very places where composition and & respectively arise in projection analysis. In the remainder of this section we show that & and $+$ are equivalent and that, given the guard property, $\times$ is simply an alternative means of calculating the composition of projection transformers.

6.1 & and $+$.

For simple 'atomic' projections, it is merely a matter of calculation, using table 1 for $+$ in and the defn.6.1 for & to establish the following result.

Proposition 6.4 For all projections $\alpha, \beta \in \diamond$, $\alpha \& \beta = \alpha + \beta$. $\square$

But what of constructed projections like $\alpha_{\beta,\gamma}$ which appear in projection analysis but not in the type system? According to the cited definition of & (defn.6.1),

$$\alpha_{\beta,\gamma} \& \alpha'_{\beta',\gamma'} = (\alpha \& \alpha')_{(\beta \& \beta'),(\gamma \& \gamma')}$$

A projection like $\alpha_{\beta,\gamma}$ will correspond to a fusion type like $\_\,{}_{\beta}\!\circ_{\gamma}\_$, with α possibly appearing as an arrow annotation or in a type assumption like $x^{\alpha}\sigma\ {}_{\beta}\!\circ_{\gamma}\tau$. Now the only way of 'combining' types like $\sigma\ {}_{\beta}\!\circ_{\gamma}\tau$ and $\sigma\ {}_{\beta'}\!\circ_{\gamma'}\tau$ is if they appear bound to the same variable in type assumption sets Γ^J and Γ^K which are 'summed' in a derivation to give Γ^{J+K}. Definition 3.1 says that

$$\begin{aligned}&(\Gamma^J, x^{\alpha}\sigma\ {}_{\beta}\!\circ_{\gamma}\tau) + (\Gamma^K, x^{\alpha'}\sigma\ {}_{\beta'}\!\circ_{\gamma'}\tau)\\ &= (\Gamma^{J+K}, x^{\alpha+\alpha'}\sigma\ {}_{\beta+\beta'}\!\circ_{\gamma+\gamma'}\tau)\end{aligned}$$

which corresponds exactly to the effect of & on constructed projections. Hence, while $+$ is only defined over simple annotations, the effect in the type assignment system is the same as & in projection analysis.

6.2 Composition and $\times$.

In projection analysis, for a function application like fe we may wish to calculate $e^x(f^1\alpha)$ for α a projection and e^x and f^1 projection transformers appropriate to e and f. In comparison, and with some abuse of notation, the type assignment system calculates $e^x Str \times f^1 Str$. Now recall that by the guard property, if we know $e^x(f^1 Str)$ then we can find $e^x(f^1\alpha)$ for all α. Similarly, if we know $e^x Str$ then we know $e^x\beta$ for all projections β, in particular $e^x(f^1 Str)$. Hence knowing both $e^x Str$ and $f^1 Str$ is sufficient to find the composition $e^x(f^1 Str)$ and this is exactly the definition of $\times$ given in §3. In summary, $f^1 Str \times e^x Str = e^x(f^1 Str)$ and the guard property ensures that this is sufficient to completely determine the composition of e^x and f^1. The following result provides a more formal statement.

Proposition 6.5 Given some collection A of projections $\{\alpha_1, \ldots \alpha_n\}$ for which $\alpha_G \in A$ has the guard property. Let t_1, t_2 be projection transformers corresponding to some functions f_1, f_2 respectively (i.e. $f_i : \alpha \Rightarrow t_i\alpha$ for any α and $i = 1$ or 2). Then their composition $t_1 \circ t_2$ can be fully determined from $t_1\alpha_G$ and $t_2\alpha_G$.

Proof Trivial. If we know $t_1\alpha_G$ then the guard property gives us $t_1\alpha_i$ for all $\alpha_i \in A$. In particular it gives us $t_1(t_2\alpha_G)$. $\square$

In this paper we define $(t_2\alpha_G) \times (t_1\alpha_G) = t_1(t_2\alpha_G)$.

7 Analysing lists.

This section outlines how the type system can be smoothly extended to deal with recursive types. We demonstrate with the important case of lists but the technique generalises quite easily. Our notational style borrows from [Hug90].

Recursive types naturally lead to the possibility of 'infinite' annotations or projections. The usual decision is to insist that the annotations are *uniform*, in the sense that the usage of components down the recursive chain is described by a single annotation. Thus, for lists, annotations are given to describe the usage of the head of the list and the tail of the list. The annotated type language is extended to admit lists as follows:

$$[\tau]_{\alpha:\beta}$$

to indicate that the elements of lists of this type all have demand α and the structure of the list (i.e. all the tails) have demand β. Note that in fact the assumption of uniformity goes beyond α and β, in that all elements of the list are assumed to have the same type σ and in our scheme this may contain annotations. For example, a list of type $[\sigma \stackrel{Str}{\rightarrow} \tau]_{\alpha:\beta}$ must consist of elements which are all *strict* functions. In other words, the component type also indicates some demand or behaviour of the list elements. This new form of annotated type leads to an obvious extension of defn.5.1:

$$[] \upharpoonright [\sigma]_{\alpha:\beta} = []$$

$$\frac{}{\Gamma^{Abs} \vdash [] : [\sigma]_{\alpha:\beta}}\ Nil \qquad \text{for any } \alpha, \beta, \sigma$$

$$\frac{\Gamma^{J} \vdash e_0 : \sigma' \qquad \Gamma^{K} \vdash e_1 : [\sigma]_{\alpha:\beta} \qquad \sigma' \sqsubseteq \sigma}{\Gamma^{\alpha \times J + \beta \times K} \vdash (e_0 : e_1) : [\sigma]_{\alpha:\beta}}\ Cons$$

$$\frac{\Gamma^{J} \vdash e_0 : [\sigma]_{\alpha:\beta} \qquad \Gamma^{K} \vdash e_1 : \tau \qquad \Gamma^{L}, x^{\alpha'}\sigma, y^{\beta'}[\sigma]_{\alpha:\beta} \vdash e_2 : \tau \qquad \alpha' \sqsubseteq \alpha,\ \beta' \sqsubseteq \beta}{\Gamma^{J+(K \sqcup L)} \vdash \textbf{lcase}\ e_0\ \textbf{of}\ [] \Rightarrow\ e_1 \mid (x : y)\ \Rightarrow\ e_2 : \tau}\ Lcase$$

Figure 12: Annotated type assignment for lists

$$(x : xs) \upharpoonright [\sigma]_{\alpha:\beta} = (\alpha(x \upharpoonright \sigma)) : (\beta(xs \upharpoonright [\sigma]_{\alpha:\beta}))$$

The type assignment rules for this extension are shown in fig.12.

We conclude with some example type schemes demonstrating the kind of information that can be deduced using this approach.

$$\begin{aligned} head\ xs \quad &= \quad \textbf{lcase}\ xs\ \textbf{of}\ [] \Rightarrow\ \textbf{error} \\ &\qquad \mid (y : ys)\ \Rightarrow\ y \end{aligned}$$

$$head : [\sigma]_{Str:Abs} \overset{Str}{\to} \sigma$$

The type for *head* indicates that only the first element of the list argument need be evaluated and the rest may safely be discarded.

$$\begin{aligned} reverse \quad &= \quad \textbf{lcase}\ xs\ \textbf{of}\ [] \Rightarrow\ [] \\ &\qquad \mid (y : ys)\ \Rightarrow\ append(reverse\ ys)[y] \end{aligned}$$

$$reverse : [\sigma]_{(\alpha \times (Str \sqcup \beta)):Str} \overset{Str}{\to} [\sigma]_{\alpha:\beta}$$

The element annotation $\alpha \times (Str \sqcup \beta)$ tells us (at least) that *reverse* is only head strict if its result is to be evaluated in a context which is *both* head strict and tail strict.

The final example below, *compose*, confirms expectations that a function that forms the composition of a list of functions is only tail strict if all the functional list elements are strict. Furthermore, *compose* is only strict on its second argument under the same circumstances.

$$\begin{aligned} compose\ fs\ x \quad &= \quad \textbf{lcase}\ fs\ \textbf{of}\ [] \Rightarrow\ x \\ &\qquad \mid (g : gs)\ \Rightarrow\ g(compose\ gs\ x) \end{aligned}$$

$$compose : [\sigma \overset{\alpha}{\to} \sigma]_{Str:\alpha} \overset{Str}{\to} \sigma \xrightarrow{Str \sqcup \alpha} \sigma$$

8 Conclusion.

We have outlined a system of annotated types for the static analysis of functional programs and given a compositional semantics in terms of projections. Type assignment and projection analysis were shown to produce corresponding results and to employ essentially equivalent basic operations (that is, $+$ is equivalent to &, and $\times$ imitates composition).

One key difference is that only 'atomic' projections appear in the type system whereas projection analysis uses 'constructed' projection (e.g. head- and tail- strictness). This may offer advantages in terms of semantic simplicity and operational control. Of course, the practicality of the typing approach will only be proven by the implementation and testing of such systems and work is progressing on this front.

9 Acknowledgements.

Much of the work for this paper was done while visiting The University of Cambridge Computer Laboratory, during which time I had many interesting and congenial discussions with Nick Benton, Gavin Bierman and Alan Mycroft. Useful feedback was gathered during a brief visit to the University of Glasgow Functional Programming Group and in particular, Kei Davis has set me straight on a number of matters. Thanks to Pierre Jouvelot for posing the question of how to give a compositional semantics for annotated types. The anonymous referees made several helpful observations and suggestions.

References

[AB75] A. R. Anderson and N. D. Belnap, Jr. *Entailment: The Logic of Relevance and Necessity.* Princeton University Press, 1975.

[AH87] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[Ben93] P. N. Benton. *Strictness Analysis of Lazy Functional Programs.* PhD thesis, University of Cambridge, 1993.

[BF92] C. A. Baker-Finch. Relevant logic and strictness analysis. In *Workshop on Static Analysis, LaBRI, Bordeaux*, pages 221–228. Bigre 81–82, 1992.

[BF93] C. A. Baker-Finch. Relevance and contraction: A logical basis for strictness and sharing analysis. Technical Report ISE RR 34/94, University of Canberra, 1993.

[Bie92] G. Bierman. Type systems, linearity and functional languages. CLICS Workshop. Slides appear in Aarhus technical report PB-397-I, 1992.

[Bur90a] G. L. Burn. A relation between abstract interpretation and projection analysis. In *17th Annual ACM Symposium on Principles of Programming Languages*, pages 151–156, 1990.

[Bur90b] G. L. Burn. Using projection analysis in compiling lazy functional programs. In *ACM Conference on Lisp and Functional Programming*, pages 227–241, 1990.

[Dav93] K. Davis. Higher-order binding-time analysis. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, 1993.

[Dun86] J. M. Dunn. Relevance logic and entailment. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Vol. III*. D. Reidel, 1986.

[DW90] K. Davis and P. Wadler. Strictness analysis in 4D. In *Glasgow Workshop on Functional Programming*. Springer-Verlag, 1990.

[GS90] C. A. Gunter and D. A. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier Science Publishers, 1990.

[HS91] S. Hunt and D. Sands. Binding time analysis: A new PERspective. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, 1991.

[Hug87] R. J. M. Hughes. Backwards analysis of functional programs. Technical Report CSC/87/R3, Department of Computing Science, University of Glasgow, 1987.

[Hug90] R. J. M. Hughes. Compile-time analysis of functional programs. In D. A. Turner, editor, *Research Topics in Functional Programming*. Addison-Wesley, 1990.

[Hun91] S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice*. PhD thesis, Imperial College, 1991.

[Jen91] T. P. Jensen. Strictness analysis in logical form. In *Fifth International Conference on Functional Programming and Computer Architecture*, pages 352–366, 1991. LNCS 523.

[JG91] P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *18th Annual ACM Symposium on Principles of Programming Languages*, 1991.

[KM89] T-M. Kuo and P. Mishra. Strictness analysis: A new perspective based on type inference. In *Fourth International Conference on Functional Programming and Computer Architecture*, pages 260–272. ACM Press, 1989.

[Lau91] J. Launchbury. *Projection Factorisations in Partial Evaluation*. PhD thesis, Glasgow University, 1991.

[Myc81] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, University of Edinburgh, 1981.

[Ses91] Peter Sestoft. *Analysis and Efficient Implementation of Functional Programs*. PhD thesis, University of Copenhagen, 1991.

[Wad91] P. Wadler. Is there a use for linear logic? In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM/IFIP, 1991.

[WBF93] D. A. Wright and C. A. Baker-Finch. Usage analysis with natural reduction types. In *Third International Workshop on Static Analysis, Padova*, pages 254–266, 1993. LNCS 724.

[WH87] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture*, pages 385–407, 1987. LNCS 274.

[Wri91] D. A. Wright. A new technique for strictness analysis. In *Theory and Practice of Software Development*, 1991. LNCS 494.

[Wri93] D. A. Wright. *Reduction Types and Intensionality in Lambda Calculus*. PhD thesis, University of Tasmania, 1993.