PERs from Projections for Binding-time Analysis

Kei Davis
Computing Science Department
University of Glasgow
Glasgow G12 8QQ, UK
kei@dcs.glasgow.ac.uk.

Abstract

First-order projection-based binding-time analysis has proven genuinely useful in partial evaluation [Lau91a, Lau91c]. There have been three notable generalisations of projection-based analysis to higher order. The first lacked a formal basis [Mog89]; the second used structures strictly more general that projections, namely *partial equivalence relations* (PERs) [HS91]; the third involved a complex construction that gave rise to impractically large abstract domains [Dav93]. This paper presents a technique free of these shortcomings: it is simple, entirely projection-based, satisfies a formal correctness condition, and gives rise to reasonably small abstract domains. Though the technique is cast in terms of projections, there is also an interpretation in terms of PERs. The principal limitation of the technique is the restriction to *monomorphic* typing.

1 Introduction and Background

We take as given that binding-time analysis is essential for good partial evaluation, and we do not address the issue of annotating programs according to the results of analysis. Numerous binding-time analysis techniques have been proposed and implemented; we greatly narrow the field of discussion by restricting attention to those for which there is a formally stated notion of correctness that the technique has been proven to satisfy.

Analysis techniques can usually be identified as being based on either a non-standard denotational semantics or a non-standard typing. Examples in the latter category include those of Gomard [Go92], Jensen [Jen92], the Nielsons [NN88], Schmidt [Sch88], and Henglein and Mossin [HM94]. Our focus is on those techniques based on non-standard interpretation, in particular, those using projections or partial equivalence relations (PERs) as the basic abstract values.

A domain *projection* is a continuous idempotent function that approximates the identity. Launchbury [Lau88] hit upon the idea of using projections to encode degrees of staticness of data. The basic idea is that a projection maps to $\perp$ that part of a data structure that is dynamic (possibly not determined), and acts as the identity on that part which is static (definitely determined). Examples are the identity ID, the greatest projection, which specifies that values are entirely static; the constant $\perp$ function BOT, the least projection, which specifies that values are entirely dynamic; and projections FST and SND on product domains, defined by

$$FST\ (x,y) = (x,\perp)\ , \qquad SND\ (x,y) = (\perp,y)\ ,$$

specifying staticness in the first and second components of pairs, respectively. The nominal goal of analysis is, given function f denoted by some programming-language expression, and projection δ encoding the staticness of the argument of f, to determine γ satisfying the *safety condition* $\gamma \circ f \sqsubseteq f \circ \delta$. For example, taking γ to be SND satisfies $\gamma \circ swap \sqsubseteq swap \circ FST$ for $swap$ defined by $swap\ (x,y) = (y,x)$. Taking γ to be BOT always satisfies the safety condition but tells nothing; greater γ is more informative. Launchbury [Lau91a] showed that this safety condition satisfies, and in a sense which he formalises, is equivalent to the correctness condition for binding-time analysis in the general framework of Jones [Jon88]. Using projection-based analysis, Launchbury implemented both monomorphic and polymorphic versions of a partial evaluator for a first-order language. His work culminated in the first strongly-typed partial evaluator [Lau91c]—strong evidence for the value of the projection-based approach.

There have been three notable attempts to generalise Launchbury's techniques to higher order. The first was Mogensen's generalisation of the polymorphic technique [Mog89]. Though successfully implemented, there is no formal statement of what it means for the analysis to be correct; even if such a statement were made, proving correctness would likely be difficult because of the highly intensional nature of the analysis: the non-standard values associated with expressions are strongly dependent on their syntactic structure, and projections are encoded symbolically as *abstract closures*, with approximation performed algebraically 'on-the-fly' according to time and space considerations. Nonetheless, the experiment provided evidence for the *practicality* of the projection-based approach at higher order.

The second generalisation was Hunt and Sands', of the monomorphic technique to higher order [HS91]. Their observation was that a projection, regarded as a set of domain-range pairs, is an *equivalence relation*: given γ, values u and v are in the same equivalence class if $\gamma\ u = \gamma\ v$, and the canonical elements of the equivalence classes are the set of fixed points (range) of γ. Hunt showed that the safety condition $\gamma \circ f \sqsubseteq f \circ \delta$ holds iff f is related to itself by $\delta \to \gamma$

where $\rightarrow$ is the standard operation on binary relations, so

$$\begin{aligned} &(\gamma \circ f \sqsubseteq f \circ \delta) \\ \Leftrightarrow\ &(\forall\ u, v\ .\ \delta\ u = \delta\ v\ \Rightarrow\ \gamma\ (f\ u) = \gamma\ (f\ v))\ . \end{aligned}$$

Then, for example, $(BOT \rightarrow ID)\ (f, f)$ asserts that f maps dynamic arguments to static results. In general $\delta \rightarrow \gamma$ is not an equivalence relation, but it is always a *partial* equivalence relation: it is symmetric and transitive but not necessarily reflexive. Unlike projections regarded as relations, PERs are closed under $\rightarrow$; the result, as Hunt and Sands show, is that 'scaling up' to higher-order analysis is reasonably straightforward. One disadvantage of their method is that PER spaces are considerably larger than the projection spaces on the same domains, and it is not clear which PERs to choose for (finite) abstract domains. Here they borrowed heavily from the projection world, using standard abstract projection domains at ground types. Further, its practicality has not been demonstrated by implementation, and, because of the unfamiliar territory, a promising route to a polymorphic generalisation is obscure.

The third generalisation was ours, to an entirely projection-based, monomorphic, higher-order technique [Dav93]. One observation motivating the approach is that there is no meaningful abstraction of values to projections, only of functions f to projection transformers τ (functions from projections to projections) satisfying $(\tau\ \delta) \circ f \sqsubseteq f \circ \delta$ for all δ. To make this abstraction possible a semantics intermediate between the standard and analysis semantics was introduced. Moving from the standard to intermediate semantics involved a translation of each ground type `T` to a function type `E -> T` (for a fixed type `E`), the values of which, being functions, could then be abstracted. The result, while proven correct with respect to a formal safety condition, is probably not practicable because of the growth in the sizes of (usefully rich) abstract domains induced by the type translation.

This paper presents a technique far simpler technique than our previous one. No intermediate semantics is required, and the correctness condition and proof are much simpler. Because the translation of ground types to function types is avoided the abstract domains are much smaller. Though entirely projection-based, we show that there is a reading of the results in terms of PERs, intimating a close relationship with the PER-based technique.

2 Language and Standard Semantics

The source language is a simple, strongly typed, monomorphic, non-strict functional language. The grammar for the language of types and type definitions is given in Figure 1. Nullary product corresponds to the so-called *unit* type. A unary product `(T)` will always have the same semantics as `T`. The types used in the examples are defined as follows.

```
FunList =   nil ()
          + cons (Int -> Int, FunList) ,

FunTree =   leaf (Int -> Int)
          + branch (FunTree, FunTree) .
```

The grammar for expressions is given in Figure 2. Addition for integers is provided as typical of operations on flat data types in this setting. A unary tuple `(e)` will always have the same semantics as `e`. The (monomorphic) typing of expressions is entirely standard and is omitted.

$\mathtt{T}$ ::=	$\mathtt{A}$	[Type Name]
|	`Int`	[Integer]
|	$(\mathtt{T}_1, \ldots, \mathtt{T}_n)$	[Product, $n \geq 0$]
|	$\mathtt{c}_1\ \mathtt{T}_1$ + ... + $\mathtt{c}_n\ \mathtt{T}_n$	[Sum, $n \geq 1$]
|	$\mathtt{T}_1$ -> $\mathtt{T}_2$	[Function]
$\mathtt{D}$ ::=	$\mathtt{A}_1 = \mathtt{T}_1$; ... ; $\mathtt{A}_n = \mathtt{T}_n$	[Type Definitions]

Figure 1: Types and type definitions.

$\mathtt{e}$ ::=	`x`	[Variable]
|	$\mathtt{n}_i$	[Numeral]
|	$\mathtt{e}_1$ + $\mathtt{e}_2$	[Integer addition]
|	$\mathtt{e}_1\ \mathtt{e}_2$	[Function application]
|	$(\mathtt{e}_1, \ldots, \mathtt{e}_n)$	[Tuple construction]
|	let $(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{e}_0$ in $\mathtt{e}_1$	[Tuple decomposition]
|	$\mathtt{c}_i$ e	[Sum construction]
|	case $\mathtt{e}_0$ of $\{\mathtt{c}_i\ \mathtt{x}_i$ -> $\mathtt{e}_i\}$	[Sum decomposition]
|	`\x:T.e`	[Lambda abstraction]
|	$\mathtt{e}_1\ \mathtt{e}_2$	[Function application]
|	`fix e`	[Fixed point]

Figure 2: Expressions.

2.1 Expression semantics

Since two different expression semantics will be given, following Abramsky [Abr90] we define a semantics $\mathcal{E}$ parameterised by a set of *defining constants*. The semantics $\mathcal{E}$ is defined in Figure 3; the defining constants are *plus*, *sel*$_i$, *tuple*, *inc*$_i$, *outc*$_i$, *choose*, *mkfun*, *apply*, and *fix*. The two instances of $\mathcal{E}$ are distinguished by a superscript: $\mathcal{E}^{\mathrm{S}}$ for the standard semantics and $\mathcal{E}^{\mathrm{P}}$ for the non-standard semantics. The corresponding type semantics have the same superscripts, as do the defining constants.

It is useful to regard the free-variable environment of each expression as having some tuple type $(\mathtt{T}_1, \ldots, \mathtt{T}_n)$, and environment lookup as indexing (as in a categorical semantics, or De Bruijn indexing); variables are indexed implicitly or explicitly by their index in the free variable environment. Then for both versions of the semantics and all expressions `e` of type `T` with environment of type $(\mathtt{T}_1, \ldots, \mathtt{T}_n)$,

$$\mathcal{E}[\![\mathtt{e}]\!] \ \in\ \mathcal{T}[\![(\mathtt{T}_1, \ldots, \mathtt{T}_n)]\!] \ \rightarrow\ \mathcal{T}[\![\mathtt{T}]\!]\ .$$

Noting that $\rho[\![\mathtt{x}_i]\!]$ is short for *sel*$_i$ ρ, environment update

$$\mathcal{E}[\![\,\mathtt{x}_i\,]\!]\ \rho \;=\; \rho[\![\,\mathtt{x}_i\,]\!] \;=\; \mathit{sel}_i\ \rho$$

$$\mathcal{E}[\![\,\mathtt{e}_1\ \mathtt{+}\ \mathtt{e}_2\,]\!]\ \rho \;=\; \mathit{plus}\ (\mathcal{E}[\![\,\mathtt{e}_1\,]\!]\ \rho,\ \mathcal{E}[\![\,\mathtt{e}_2\,]\!]\ \rho)$$

$$\begin{aligned}&\mathcal{E}[\![\,\mathtt{(e_1,\ldots,e}_n\mathtt{)}\,]\!]\ \rho\\ &\quad= \mathit{tuple}\ (\mathcal{E}[\![\,\mathtt{e}_1\,]\!]\ \rho,\ \ldots,\ \mathcal{E}[\![\,\mathtt{e}_n\,]\!]\ \rho)\end{aligned}$$

$$\begin{aligned}&\mathcal{E}[\![\,\mathtt{let\ (x_1,\ldots,x}_n\mathtt{)\ =\ e_0\ in\ e_1}\,]\!]\ \rho\\ &\quad= \mathcal{E}[\![\,\mathtt{e}_1\,]\!]\ \rho[\mathtt{x}_i \mapsto \mathit{sel}_i\ (\mathcal{E}[\![\,\mathtt{e}_0\,]\!]\ \rho)\mid 1\le i\le n]\end{aligned}$$

$$\mathcal{E}[\![\,\mathtt{c}_i\ \mathtt{e}\,]\!]\ \rho \;=\; \mathit{inc}_i\ (\mathcal{E}[\![\,\mathtt{e}\,]\!]\ \rho)$$

$$\begin{aligned}&\mathcal{E}[\![\,\mathtt{case\ e_0\ of\ c_1\ x_1\ \text{->}\ e_1;\ \ldots;\ c}_n\ \mathtt{x}_n\ \mathtt{\text{->}\ e}_n\,]\!]\ \rho\\ &\quad= \mathit{choose}\ (\ \mathcal{E}[\![\,\mathtt{e}_0\,]\!]\ \rho,\\ &\qquad\qquad \mathcal{E}[\![\,\mathtt{e}_1\,]\!]\ \rho[\mathtt{x}_1\mapsto \mathit{outc}_1\ (\mathcal{E}[\![\,\mathtt{e}_0\,]\!]\ \rho)],\\ &\qquad\qquad \vdots\\ &\qquad\qquad \mathcal{E}[\![\,\mathtt{e}_n\,]\!]\ \rho[\mathtt{x}_n\mapsto \mathit{outc}_n\ (\mathcal{E}[\![\,\mathtt{e}_0\,]\!]\ \rho)])\end{aligned}$$

$$\mathcal{E}[\![\,\mathtt{\backslash x.e}\,]\!]\ \rho \;=\; \mathit{mkfun}\ (\lambda x.\mathcal{E}[\![\,\mathtt{e}\,]\!]\ \rho[\mathtt{x}\mapsto x])$$

$$\mathcal{E}[\![\,\mathtt{e}_1\ \mathtt{e}_2\,]\!]\ \rho \;=\; \mathit{apply}\ (\mathcal{E}[\![\,\mathtt{e}_1\,]\!]\ \rho)\ (\mathcal{E}[\![\,\mathtt{e}_2\,]\!]\ \rho)$$

$$\mathcal{E}[\![\,\mathtt{fix\ e}\,]\!]\ \rho \;=\; (\mathit{fix}\circ\mathit{apply})\ (\mathcal{E}[\![\,\mathtt{e}\,]\!]\ \rho)$$

Figure 3: Parameterised semantics.

$\rho[\mathtt{x}_i \mapsto v]$ is defined by

$$\begin{aligned}&\mathit{tuple}\ (\mathit{sel}_1\ \rho,\ \ldots,\ \mathit{sel}_{i-1}\ \rho,\\ &\qquad v,\\ &\qquad \mathit{sel}_{i+1}\ \rho,\ \ldots,\ \mathit{sel}_n\ \rho)\ .\end{aligned}$$

It is convenient to regard `Int` as being defined as the infinite sum

$$\mathtt{Int}\ =\ \ldots\ +\ \mathtt{n}_{-1}\ \mathtt{()}\ +\ \mathtt{n}_0\ \mathtt{()}\ +\ \mathtt{n}_1\ \mathtt{()}\ +\ \ldots\ ,$$

where in practice we write $\mathtt{n}_i$ as short for $\mathtt{n}_i$ `()`.

2.2 Standard semantics

The standard S type and expression semantics are defined in Figure 4. Function types give rise to lifted function spaces as in Abramsky's lazy lambda calculus [Abr89], and the semantics distinguishes those expressions of function type that have WHNF (have value *lift* f for some f) and those that do not (have value $\perp$). Products are unlifted; a unary sum-of-products gives a lifted product. Domain $+$ is separated sum. Recursive type definitions give rise to recursive domain specifications which have the usual least-fixed-point solutions.

3 Domain Factorisation

A key observation of [Dav93] was that since there is no concept of staticness of the body of a lambda expression, there

$$\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{Int}\,]\!] \;=\; \mathit{Int} \;=\; \mathbf{Z}_\perp$$

$$\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{(T_1,\ldots,T}_n\mathtt{)}\,]\!] \;=\; \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_1\,]\!]\ \times\ \ldots\ \times\ \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_n\,]\!]$$

$$\begin{aligned}&\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{c_1\ T_1\ +\ \ldots\ +\ c}_n\ \mathtt{T}_n\,]\!]\\ &\quad= \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_1\,]\!]\ +\ \ldots\ +\ \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_n\,]\!]\end{aligned}$$

$$\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T_1\ \text{->}\ T_2}\,]\!] \;=\; (\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_1\,]\!]\ \rightarrow\ \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_2\,]\!])_\perp$$

$$\mathit{plus}^{\mathsf{S}}\ (x,y) \;=\; x+y$$

$$\mathit{tuple}^{\mathsf{S}}\ (x_1,\ldots,x_n) \;=\; (x_1,\ldots,x_n)$$

$$\mathit{sel}_i^{\mathsf{S}}\ (x_1,\ldots,x_n) \;=\; x_i$$

$$\mathit{inc}_i^{\mathsf{S}} \;=\; \mathit{in}_i\circ\mathit{lift}$$

$$\mathit{outc}_i^{\mathsf{S}}\ x \;=\; \mathit{drop}\circ\mathit{out}_i$$

$$\begin{aligned}\mathit{choose}^{\mathsf{S}}\ (\perp,\quad &x_1,\ldots,x_n) \;=\; \perp\\ \mathit{choose}^{\mathsf{S}}\ (\mathit{in}_i\ v,\ \ &x_1,\ldots,x_n) \;=\; x_i\end{aligned}$$

$$\mathit{mkfun}^{\mathsf{S}} \;=\; \mathit{lift}$$

$$\mathit{apply}^{\mathsf{S}} \;=\; \mathit{drop}$$

$$\mathit{fix}^{\mathsf{S}} \;=\; \mathit{lfp}\quad \text{[Least fixed point]}$$

Figure 4: Standard type and expression semantics.

is no point in having projections on function spaces. Hence domains are factored into their evaluable or *data* parts, and their unevaluable but applicable *forward* parts. For example, for $(T\rightarrow U)_\perp = \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T\ \text{->}\ U}\,]\!]$ we need only distinguish two degrees of definedness—between $\perp$ and values of the form *lift* f. This may be encoded by $\mathbf{1}_\perp$—the data part of $(T\rightarrow U)_\perp$—on which there are precisely two projections, namely ID and BOT. Here the forward part is $T\rightarrow U$.

The *data domain* corresponding to type `T` is $\mathcal{D}[\![\,\mathtt{T}\,]\!]$, where $\mathcal{D}$ is defined exactly like $\mathcal{T}^{\mathsf{S}}$ except that function spaces are replaced by the one-point domain $\mathbf{1} = \{()\}$, that is, $\mathcal{D}[\![\,\mathtt{T_1\ \text{->}\ T_2}\,]\!] = \mathbf{1}_\perp$. The function $\mathit{data}_{\mathtt{T}}$ from values in $\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!]$ to their data parts in $\mathcal{D}[\![\,\mathtt{T}\,]\!]$ is a projection: it is like the identity except that values from function spaces are mapped into $\mathbf{1}$. The projection $\mathit{data}_{\mathtt{T}}$ is defined in terms of the structure of `T` as follows.

$$\begin{aligned}\mathit{data}_{\mathtt{Int}} &= \mathit{id}_{\mathtt{Int}}\ ,\\ \mathit{data}_{(\mathtt{T}_1,\ldots,\mathtt{T}_n)} &= \mathit{data}_{\mathtt{T}_1}\ \times\ \ldots\ \times\ \mathit{data}_{\mathtt{T}_n}\ ,\\ \mathit{data}_{\mathtt{c}_1\ \mathtt{T}_1\ +\ \ldots\ +\ \mathtt{c}_n\ \mathtt{T}_n} &= \mathit{data}_{\mathtt{T}_1}\ +\ \ldots\ +\ \mathit{data}_{\mathtt{T}_n}\ ,\\ \mathit{data}_{\mathtt{T}_1\text{->}\mathtt{T}_2} &= (\lambda x.())_\perp\ .\end{aligned}$$

The last definition uses function lifting, defined by $f_\perp\ \perp = \perp$ and $f_\perp\ (\mathit{lift}\ x) = \mathit{lift}\ (f\ x)$. Recursive type definitions give rise to recursive function specifications with only one solution.

In the same style as $\mathcal{D}[\![\,\mathtt{T}\,]\!]$ and $data_{\mathtt{T}}$ we define $\mathcal{T}^{\mathsf{S}}_{\ddagger}[\![\,\mathtt{T}\,]\!]$ to give the forward domain for `T`, and $fun_{\mathtt{T}}$ to be the function mapping values in $\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!]$ to their forward parts in $\mathcal{T}^{\mathsf{S}}_{\ddagger}[\![\,\mathtt{T}\,]\!]$, as follows. Roughly, $\mathcal{T}^{\mathsf{S}}_{\ddagger}$ is like $\mathcal{T}^{\mathsf{S}}$ with all lifting removed and sum replaced by product.

$$\begin{aligned}
&\mathcal{T}^{\mathsf{S}}_{\ddagger}[\![\,\mathtt{Int}\,]\!] \;=\; \mathbf{1}\;,\\
&\mathcal{T}^{\mathsf{S}}_{\ddagger}[\![\,(\mathtt{T}_1,\ldots,\mathtt{T}_n)\,]\!] \;=\; \mathcal{T}^{\mathsf{S}}_{\ddagger}[\![\,\mathtt{T}_1\,]\!] \;\times\;\ldots\;\times\; \mathcal{T}^{\mathsf{S}}_{\ddagger}[\![\,\mathtt{T}_n\,]\!]\;,\\
&\mathcal{T}^{\mathsf{S}}_{\ddagger}[\![\,\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n\,]\!]\\
&\qquad = \mathcal{T}^{\mathsf{S}}_{\ddagger}[\![\,\mathtt{T}_1\,]\!] \;\times\;\ldots\;\times\; \mathcal{T}^{\mathsf{S}}_{\ddagger}[\![\,\mathtt{T}_n\,]\!]\;,\\
&\mathcal{T}^{\mathsf{S}}_{\ddagger}[\![\,\mathtt{T}_1\ \mathtt{->}\ \mathtt{T}_2\,]\!] \;=\; \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_1\,]\!] \;\to\; \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}_2\,]\!]\;.
\end{aligned}$$

The mapping from standard values to their forward parts is defined by

$$\begin{aligned}
&fun_{\mathtt{Int}} \;=\; \lambda x.()\;,\\
&fun_{(\mathtt{T}_1,\ldots,\mathtt{T}_n)} \;=\; fun_{\mathtt{T}_1} \;\times\;\ldots\;\times\; fun_{\mathtt{T}_n}\;,\\
&fun_{\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n}\ \bot \qquad = \;\bot\\
&fun_{\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n}\ (in_i\ v)\\
&\qquad = \;(\bot,\ldots,\bot,v,\bot,\ldots,\bot) \quad [v \text{ in } i^{th} \text{ position}]\;,\\
&fun_{\mathtt{T}_1\mathtt{->}\mathtt{T}_2} \;=\; drop\;.
\end{aligned}$$

We write $fac_{\mathtt{T}}$ for $\lambda x.(data_{\mathtt{T}}\ x,\ fun_{\mathtt{T}}\ x)$. Then $\mathcal{D}[\![\,\mathtt{T}\,]\!]\times\mathcal{T}^{\mathsf{S}}_{\ddagger}[\![\,\mathtt{T}\,]\!]$ is a factorisation of $\mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!]$, and

$$fac_{\mathtt{T}} \;\in\; \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!] \;\to\; (\mathcal{D}[\![\,\mathtt{T}\,]\!] \;\times\; \mathcal{T}^{\mathsf{S}}_{\ddagger}[\![\,\mathtt{T}\,]\!])$$

is an embedding which *determines* the corresponding projection $unfac_{\mathtt{T}}$ (since they are related by $unfac_{\mathtt{T}} \circ fac_{\mathtt{T}} = id$ and $fac_{\mathtt{T}} \circ unfac_{\mathtt{T}} \sqsubseteq id$). Rather than give an explicit definition of $unfac_{\mathtt{T}}$ we give some examples. The factorisation of *Int* is $Int \times \mathbf{1}$; more generally, the factorisation of any domain D corresponding to a type not containing `->` is just $D \times \mathbf{1}$. If $T = \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!]$ and $U = \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{U}\,]\!]$ then the factorisation of $(T \to U)_\bot \;=\; \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\ \mathtt{->}\ \mathtt{U}\,]\!]$ is $\mathbf{1}_\bot \times (T \to U)$—note that factorisation 'stops' at function-space constructors. Now $fac_{\mathtt{T->U}}\ \bot = (\bot,\bot)$, and $fac_{\mathtt{T->U}}\ (lift\ f) = (lift\ (),\ f)$ for all f. In the other direction $unfac_{\mathtt{T->U}}\ (\bot, f) = \bot$ and $unfac_{\mathtt{T->U}}\ (lift\ (),\ f) = (lift\ f)$; this must be since in general $unfac_{\mathtt{T}} \circ fac_{\mathtt{T}}$ is the identity.

4 Projection Semantics

Standard values v and v' are related by nonstandard value (α,κ) when their data parts are related by α, and their forward parts are logically related by κ. At each type `T` the relation is $\mathcal{R}[\![\,\mathtt{T}\,]\!]\ ((\alpha,\kappa),\cdot,\cdot)$, defined by

$$\begin{aligned}
&\mathcal{R}[\![\,\mathtt{T}\,]\!]\ ((\alpha,\kappa),v,v')\\
&\quad = (\alpha\ d = \alpha\ d') \;\wedge\; \mathcal{R}_{\ddagger}[\![\,\mathtt{T}\,]\!]\ (\kappa,f,f')\\
&\qquad \text{where}\\
&\qquad\quad (d,f) \;\;=\; fac_{\mathtt{T}}\ v\\
&\qquad\quad (d',f') \;=\; fac_{\mathtt{T}}\ v'\;,
\end{aligned}$$

$$\begin{aligned}
&plus^{\mathsf{P}}\ ((\alpha,()),(\beta,()))\\
&\quad = \begin{cases} (ID,()), & \text{if } \alpha = ID \text{ and } \beta = ID\\ (BOT,()), & \text{otherwise}\end{cases}\\
&tuple^{\mathsf{P}}\ ((\alpha_1,\kappa_1),\ldots,(\alpha_n,\kappa_n))\\
&\quad = ((\alpha_1\times\ldots\times\alpha_n),(\kappa_1,\ldots,\kappa_n))\\
&sel_i^{\mathsf{P}}\ ((\alpha_1\times\ldots\times\alpha_n),(\kappa_1,\ldots,\kappa_n)) \;=\; (\alpha_i,\kappa_i)\\
&inc_i^{\mathsf{P}}\ (\alpha,\kappa) \;=\; (C_i\ \alpha,\ (\top,\ldots,\top,\kappa,\top,\ldots,\top))\\
&outc_i^{\mathsf{P}}\ (\alpha,(\kappa_1,\ldots,\kappa_n)) \;=\; (OUTC_i\ \alpha,\ \kappa_i)\\
&choose^{\mathsf{P}}\ ((\alpha,\kappa),x_1,\ldots,x_n)\\
&\quad = \begin{cases} \bot, & \text{if } \alpha \not\sqsupseteq \sqcap_{1\le i\le n}\ C_i\ BOT\\ x_1 \sqcap\ldots\sqcap x_n, & \text{otherwise}\end{cases}\\
&mkfun^{\mathsf{P}}\ f \;=\; (ID,f)\\
&apply^{\mathsf{P}}\ (\alpha,f) \;=\; \begin{cases} \bot, & \text{if } \alpha = BOT\\ f, & \text{if } \alpha = ID\end{cases}\\
&fix^{\mathsf{P}} \;=\; gfp \quad \text{[Greatest fixed point]}
\end{aligned}$$

Figure 5: Projection semantics.

where $\mathcal{R}_{\ddagger}[\![\,\mathtt{T}\,]\!]$ is defined by

$$\begin{aligned}
&\mathcal{R}_{\ddagger}[\![\,\mathtt{Int}\,]\!]\ ((),(),()) \;=\; True\;,\\
&\mathcal{R}_{\ddagger}[\![\,(\mathtt{T}_1,\ldots,\mathtt{T}_n)\,]\!] \;=\; \mathcal{R}_{\ddagger}[\![\,\mathtt{T}_1\,]\!] \;\times\;\ldots\;\times\; \mathcal{R}_{\ddagger}[\![\,\mathtt{T}_n\,]\!]\;,\\
&\mathcal{R}_{\ddagger}[\![\,\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n\,]\!]\\
&\qquad = \mathcal{R}_{\ddagger}[\![\,\mathtt{T}_1\,]\!] \;\times\;\ldots\;\times\; \mathcal{R}_{\ddagger}[\![\,\mathtt{T}_n\,]\!]\;,\\
&\mathcal{R}_{\ddagger}[\![\,\mathtt{T}_1\ \mathtt{->}\ \mathtt{T}_2\,]\!] \;=\; \mathcal{R}[\![\,\mathtt{T}_1\,]\!] \;\to\; \mathcal{R}[\![\,\mathtt{T}_2\,]\!]\;.
\end{aligned}$$

Here $\times$ and $\to$ are the standard operations on (ternary) relations: for relations R and S we have $(R \times S)((x,y),(x',y'),(x'',y''))$ iff $R(x,x',x'')$ and $S(y,y',y'')$, and $(R \to S)(f,g,h)$ iff for all x, y, and z such that $R(x,y,z)$ we have $S(f\ x,\ g\ y,\ h\ z)$. Here, recursive type definitions give recursive relation specifications, which have inclusive least-fixed-point solutions. (A relation is inclusive if, when it holds for each element of an ascending chain, it also holds at the limit. Such relations are sometimes called *admissible* or *chain complete*. Some work is required to show inclusivity of these recursively-defined relations.) Thus the P type semantics $\mathcal{T}^{\mathsf{P}}$ must be

$$\mathcal{T}^{\mathsf{P}}[\![\,\mathtt{T}\,]\!] \;=\; |\,\mathcal{D}[\![\,\mathtt{T}\,]\!]\,| \;\times\; \mathcal{T}^{\mathsf{P}}_{\ddagger}[\![\,\mathtt{T}\,]\!]\;,$$

where $|\,\mathcal{D}[\![\,\mathtt{T}\,]\!]\,|$ is the lattice of projections on domain $\mathcal{D}[\![\,\mathtt{T}\,]\!]$, and $\mathcal{T}^{\mathsf{P}}_{\ddagger}$ is defined exactly like $\mathcal{T}^{\mathsf{S}}_{\ddagger}$ with superscript P everywhere replacing superscript S.

The defining constants for the projection semantics $\mathcal{E}^{\mathsf{P}}$ are given in Figure 5. The following notation is used for specifying projections on the data domains. For sum type $\mathtt{c}_1\ \mathtt{T}_1\ \mathtt{+}\ \ldots\ \mathtt{+}\ \mathtt{c}_n\ \mathtt{T}_n$ with data domain $D_1 + \ldots + D_n$ define $C_i\ \alpha$ to be $ID + \cdots + ID + \alpha + ID + \cdots + ID$ where α is the i^{th}

summand, $OUTC_i\ (\gamma_1 + \cdots + \gamma_n)$ to be γ_i, and $OUTC_i\ BOT$ to be BOT.

The Central Result. For expression `e` of type `T` with free-variable environment of type `E`, the functions $\mathcal{E}^{\mathsf{P}}[\![\,\mathtt{e}\,]\!]$, $\mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}\,]\!]$, and $\mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}\,]\!]$ are logically related by $\mathcal{R}$, that is

$$(\mathcal{R}[\![\,\mathtt{E}\,]\!] \to \mathcal{R}[\![\,\mathtt{T}\,]\!])\ (\mathcal{E}^{\mathsf{P}}[\![\,\mathtt{e}\,]\!],\ \mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}\,]\!],\ \mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}\,]\!])\ .$$

Further, $\mathcal{R}[\![\,\mathtt{T}\,]\!]\ ((\alpha,\kappa),\cdot,\cdot)$ is a PER for all α and κ. $\Box$

The bulk of proof is omitted; it consists of showing that the defining constants are similarly related, and a simple induction on the structure of expressions showing that if the defining constants are logically related, then so are the semantic functions. By way of example we consider the expression form `e`$_1$ `+ e`$_2$, and the relevant defining constant *plus*. To show that $plus^{\mathsf{S}}$ and $plus^{\mathsf{P}}$ are correctly related we need to show that

$$((\mathcal{R}[\![\,\mathtt{Int}\,]\!] \times \mathcal{R}[\![\,\mathtt{Int}\,]\!]) \to \mathcal{R}[\![\,\mathtt{Int}\,]\!])\ (plus^{\mathsf{P}},\ plus^{\mathsf{S}},\ plus^{\mathsf{S}})\ ,$$

that is, for all α_1, α_2, d_1, d_2, d_1', and d_2' that

$$\begin{aligned}&(ID\ d_1 = ID\ d_1')\ \wedge\ (ID\ d_2 = ID\ d_2')\\ \Rightarrow\ &ID\ (d_1 + d_2) = ID\ (d_1' + d_2')\end{aligned}$$

and for $\alpha_1 \neq ID$ or $\alpha_2 \neq ID$ that

$$\begin{aligned}&(\alpha_1\ d_1 = \alpha_1\ d_1')\ \wedge\ (\alpha_2\ d_2 = \alpha_2\ d_2')\\ \Rightarrow\ &BOT\ (d_1 + d_2) = BOT\ (d_1' + d_2')\ .\end{aligned}$$

For the inductive case `e`$_1$ `+ e`$_2$ with environment type `E` we need to show that if

$$(\mathcal{R}[\![\,\mathtt{E}\,]\!] \to \mathcal{R}[\![\,\mathtt{Int}\,]\!])\ (\mathcal{E}^{\mathsf{P}}[\![\,\mathtt{e}_1\,]\!],\ \mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}_1\,]\!],\ \mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}_1\,]\!])$$

and

$$(\mathcal{R}[\![\,\mathtt{E}\,]\!] \to \mathcal{R}[\![\,\mathtt{Int}\,]\!])\ (\mathcal{E}^{\mathsf{P}}[\![\,\mathtt{e}_2\,]\!],\ \mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}_2\,]\!],\ \mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}_2\,]\!])$$

then

$$(\mathcal{R}[\![\,\mathtt{E}\,]\!] \to \mathcal{R}[\![\,\mathtt{Int}\,]\!])\ (\mathcal{E}^{\mathsf{P}}[\![\,\mathtt{e}_1\ \texttt{+}\ \mathtt{e}_2\,]\!],\ \mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}_1\ \texttt{+}\ \mathtt{e}_2\,]\!],\ \mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}_1\ \texttt{+}\ \mathtt{e}_2\,]\!])\ .$$

Suppose $\mathcal{R}[\![\,\mathtt{E}\,]\!]\ (\rho^{\mathsf{P}}, \rho^{\mathsf{S}}, \rho'^{\mathsf{S}})$. By the induction hypothesis we have $\mathcal{R}[\![\,\mathtt{Int}\,]\!]\ (\alpha_i, v_i, v_i')$, where $(\alpha_i, ()) = \mathcal{E}^{\mathsf{P}}[\![\,\mathtt{e}_i\,]\!]\ \rho^{\mathsf{P}}$, $(v_i, ()) = \mathcal{E}^{\mathsf{P}}[\![\,\mathtt{e}_i\,]\!]\ \rho^{\mathsf{S}}$, and $(v_i', ()) = \mathcal{E}^{\mathsf{P}}[\![\,\mathtt{e}_i\,]\!]\ \rho'^{\mathsf{S}}$, for $i = 1, 2$. Now

$$\mathcal{R}[\![\,\mathtt{Int}\,]\!]\ (\mathcal{E}^{\mathsf{P}}[\![\,\mathtt{e}_1\ \texttt{+}\ \mathtt{e}_2\,]\!]\ \rho^{\mathsf{P}},\ \mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}_1\ \texttt{+}\ \mathtt{e}_2\,]\!]\ \rho^{\mathsf{S}},\ \mathcal{E}^{\mathsf{S}}[\![\,\mathtt{e}_1\ \texttt{+}\ \mathtt{e}_2\,]\!]\ \rho'^{\mathsf{S}})$$

iff

$$\mathcal{R}[\![\,\mathtt{Int}\,]\!]\ (plus^{\mathsf{P}}\ ((\alpha_1, ()), (\alpha_2, ())),\ plus^{\mathsf{S}}\ ((v_1, ()), (v_2, ())),\ plus^{\mathsf{S}}\ ((v_1', ()), (v_2', ()))) \ ,$$

which holds since $plus^{\mathsf{P}}$ and $plus^{\mathsf{P}}$, and their various arguments, are correctly related.

$$BOT\ \mathbf{proj}\ \mathtt{Int} \qquad ID\ \mathbf{proj}\ \mathtt{Int}$$

$$BOT\ \mathbf{proj}\ \mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2 \qquad ID\ \mathbf{proj}\ \mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2$$

$$BOT\ \mathbf{proj}\ \mathtt{c}_1\ \mathtt{T}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \mathtt{c}_n\ \mathtt{T}_n$$

$$\frac{\gamma_1\ \mathbf{proj}\ \mathtt{T}_1 \quad \cdots \quad \gamma_n\ \mathbf{proj}\ \mathtt{T}_n}{(C_1\ \gamma_1) \sqcap \cdots \sqcap (C_n\ \gamma_n)\ \mathbf{proj}\ \mathtt{c}_1\ \mathtt{T}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \mathtt{c}_n\ \mathtt{T}_n}$$

$$\frac{\gamma_1\ \mathbf{proj}\ \mathtt{T}_1 \quad \cdots \quad \gamma_n\ \mathbf{proj}\ \mathtt{T}_n}{\gamma_1 \times \ldots \times \gamma_n\ \mathbf{proj}\ (\mathtt{T}_1, \ldots, \mathtt{T}_n)}$$

$$\frac{\begin{array}{c}\gamma_1\ \mathbf{proj}\ \mathtt{A}_1 \quad \cdots \quad \gamma_n\ \mathbf{proj}\ \mathtt{A}_n\\ \vdash P(\gamma_1, \ldots, \gamma_n)\ \mathbf{proj}\ \mathtt{T}_i(\mathtt{A}_1, \ldots, \mathtt{A}_n)\end{array}}{\mu(\gamma_1, \ldots, \gamma_n).P(\gamma_1, \ldots, \gamma_n)\ \mathbf{proj}\ \mathtt{A}_i}$$

$$[\text{where } \mathtt{A}_i\texttt{=}\mathtt{T}_i(\mathtt{A}_1, \ldots, \mathtt{A}_n)]$$

Figure 6: Inference rules for finite projection domains.

5 Abstract Domains

At each type `T` we require a finite abstraction of $\mathcal{T}^{\mathsf{P}}[\![\,\mathtt{T}\,]\!]$. This abstract domain is $FProj_{\mathtt{T}} \times FFor_{\mathtt{T}}$, where $FProj_{\mathtt{T}}$ is a finite abstraction of the lattice $|\,\mathcal{D}[\![\,\mathtt{T}\,]\!]\,|$, and $FFor_{\mathtt{T}}$ is a finite abstraction of $\mathcal{T}_{\ddagger}^{\mathsf{P}}[\![\,\mathtt{T}\,]\!]$. The definition of $FProj_{\mathtt{T}}$ is based on that in [Lau91a]. A projection γ is in $FProj_{\mathtt{T}}$ if γ **proj** `T` can be inferred from the rules given in Figure 6. For recursively-defined types the rules yield only those projections that act on each recursive instance of a data structure in the same way. Thus $FProj_{\mathtt{FunList}}$ comprises BOT and $SPINE\ \alpha$ for α ranging over BOT and ID, where

$$SPINE\ \alpha\ =\ \mu\gamma.(NIL\ ID)\ \sqcap\ (CONS\ (\alpha\ \times\ \gamma))\ ,$$

so $SPINE\ BOT$ specifies static spines and dynamic elements, and $SPINE\ ID$ is the identity. More generally, the abstract list constructor is isomorphic to lifting. Similarly, $FProj_{\mathtt{FunTree}}$ comprises BOT and $LBR\ \alpha$ for α ranging over BOT and ID, where

$$LBR\ \alpha\ =\ \mu\gamma.(LEAF\ \alpha)\ \sqcap\ (BRANCH\ (\gamma\ \times\ \gamma))\ ,$$

so $LBR\ \alpha$ specifies static branches and leaves, and α of the leaf nodes. Again, the abstract tree constructor is isomorphic to lifting. This compares favourably with BHA strictness analysis, for which the corresponding abstract constructors are typically *double* lifting [Wad87, Sew93].

Value κ is in $FFor_{\mathtt{T}}$ if κ **fabsf** `T` can be inferred from the following.

There is only one forward value at type `Int`.

$$()\ \mathbf{fabsf}\ \mathtt{Int}\ .$$

For products and sums,

$$\frac{\kappa_1\ \mathbf{fabsf}\ \mathtt{T}_1 \quad \cdots \quad \kappa_n\ \mathbf{fabsf}\ \mathtt{T}_n}{(\kappa_1, \ldots, \kappa_n)\ \mathbf{fabsf}\ (\mathtt{T}_1, \ldots, \mathtt{T}_n)}\ ,$$

$$\frac{\kappa_1\ \textbf{fabsf}\ \texttt{T}_1 \quad \cdots \quad \kappa_n\ \textbf{fabsf}\ \texttt{T}_n}{(\kappa_1,\ldots,\kappa_n)\ \ \textbf{fabsf}\ \ \texttt{c}_1\ \texttt{T}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \texttt{c}_n\ \texttt{T}_n}\ .$$

Function spaces consist of a set of step functions closed under lub.

$$\frac{\begin{array}{ll}\tau_1 \in FTran_{\texttt{T}_1} & \kappa_1\ \textbf{fabsf}\ \texttt{T}_1 \\ \tau_2 \in FTran_{\texttt{T}_2} & \kappa_2\ \textbf{fabsf}\ \texttt{T}_2\end{array}}{step\ ((\tau_1,\kappa_1),(\tau_2,\kappa_2))\ \ \textbf{fabsf}\ \ (\texttt{T}_1\ \texttt{->}\ \texttt{T}_2)}\ ,$$

where

$$\begin{aligned} step\ (v_1,v_2)\ x &= v_2, \text{ if } v_1 \sqsubseteq x \\ step\ (v_1,v_2)\ x &= \bot, \text{ otherwise }, \end{aligned}$$

and

$$\frac{\kappa_1\ \ \textbf{fabsf}\ \ (\texttt{T}_1\ \texttt{->}\ \texttt{T}_2) \qquad \kappa_2\ \ \textbf{fabsf}\ \ (\texttt{T}_1\ \texttt{->}\ \texttt{T}_2)}{(\kappa_1 \sqcup \kappa_2)\ \ \textbf{fabsf}\ \ (\texttt{T}_1\ \texttt{->}\ \texttt{T}_2)}\ .$$

This gives the full space of monotonic functions on the abstract domains.

For recursively-defined types, roughly speaking, we choose those forward values that represent each component of the same type by the same value. Given type definitions $\texttt{A}_1\ \texttt{=}\ \texttt{T}_1\texttt{;}\ \ \ldots\texttt{;}\ \texttt{A}_n\ \texttt{=}\ \texttt{T}_n$, which we will write $\texttt{A}_i\texttt{=}\texttt{T}_i\texttt{(A}_1\texttt{,}\ldots\texttt{,A}_n\texttt{)}$, $1 \le i \le n$, if by assuming κ_i **fabsf** $\texttt{A}_i$ for $1 \le i \le n$ we may deduce $P_i(\kappa_1,\ldots,\kappa_n)$ **fabsf** $\texttt{T}_i(\texttt{A}_1\ldots\texttt{A}_n)$ for $1 \le i \le n$, then

$$\mu(\kappa_1,\ldots,\kappa_n).(P_1(\kappa_1,\ldots,\kappa_n),\ldots,P_n(\kappa_1,\ldots,\kappa_n))$$

is a tuple $(\kappa_1,\ldots,\kappa_n)$ of values such that κ_i **fabsf** $\texttt{A}_i$ for $1 \le i \le n$.

For all T the set $FProj_{\texttt{T}} \times FFor_{\texttt{T}}$ is a finite lattice containing the top and bottom elements of $\mathcal{T}^{\mathsf{P}}[\![\texttt{T}]\!]$.

For T not containing `->` the domain $\mathcal{T}^{\mathsf{P}}_{\ddagger}[\![\texttt{T}]\!]$ is isomorphic to $\mathbf{1}$, so $FFor_{\texttt{Int}}$ is $\mathbf{1}$. For `Int -> Int` we have $\mathcal{T}^{\mathsf{P}}_{\ddagger}[\![\texttt{Int -> Int}]\!] = (|\mathcal{D}[\![\texttt{Int}]\!]| \times \mathbf{1}) \to (|\mathcal{D}[\![\texttt{Int}]\!]| \times \mathbf{1})$, so $FFor_{\texttt{Int->Int}}$ is $(FProj_{\texttt{Int}} \times \mathbf{1}) \to (FProj_{\texttt{Int}} \times \mathbf{1})$. A data structure of recursive type `A=T(A)` may be thought of as some (possibly infinite) number of elements of `T(())`. For example, the value *cons* $(f,\ cons\ (g,\ nil\ ()))$ in the standard domain for `FunList` decomposes into *cons* $(f,())$, *cons* $(g,())$, and *nil* $()$. The (implicit) abstraction function maps such a data structure to the greatest lower bound of these elements, giving a safe abstraction of the nonstandard values of each element. Thus $FFor_{\texttt{A=T(A)}} = FFor_{\texttt{T(())}}$, so $FFor_{\texttt{FunList}} \cong FFor_{\texttt{FunTree}} \cong FFor_{\texttt{Int->Int}}$.

6 Examples of Analysis

For all closed expressions e the abstract value $\mathcal{E}^{\mathsf{P}}[\![\texttt{e}]\!]\ []$ of e is of the form (ID,κ), showing that closed expressions are always entirely static. For expressions of function type the abstract forward value κ is a function from abstract arguments of e to abstract results.

First we consider functions on lists. Let expression `length` denote usual length function:

```
fix (\length .
  \xs . case xs of
          nil u  -> 0
          cons p -> let (z,zs) = p in
                      1 + length zs) .
```

The abstract forward value of `length` is of the form $\lambda(\alpha,\kappa).(\tau\ \alpha,())$, where τ maps *SPINE BOT* and *SPINE ID* to *ID*, and *BOT* to *BOT*. This reveals that the result of `length` is independent of the values of list elements, and gives a static result when the argument has a static spine.

Let `append` stand for the expression denoting the usual function for appending two lists:

```
fix (\append .
  \xs . \ys .
    case xs of
      nil u  -> ys
      cons p -> let (z,zs) = p in
                  cons (z, append zs ys)) .
```

The abstract value of `append` is

$$\begin{array}{l}(ID,\ \lambda(\alpha_{\texttt{xs}},\kappa_{\texttt{xs}}). \\ (ID,\ \lambda(\alpha_{\texttt{ys}},\kappa_{\texttt{ys}}). \\ (\alpha_{\texttt{xs}} \sqcap \alpha_{\texttt{ys}},\ \kappa_{\texttt{xs}} \sqcap \kappa_{\texttt{ys}})))\ .\end{array}$$

This reveals that partial applications of `append` are static up to WHNF, and the abstract value of the result is the greatest lower bound of the two arguments. In general, the abstract value of a closed expression of the form $\texttt{\textbackslash x}_1\texttt{.\textbackslash x}_2\texttt{.}\ldots\texttt{.e}$ will reveal that all partial applications of the expression are static up to WHNF.

Let `reverse1` stand for the expression denoting the naive reverse function:

```
fix (\reverse1 .
  \xs . case xs of
          nil u ->
            nil ()
          cons p ->
            let (z,zs) = p in
              append (reverse zs)
                     (cons (z, nil ()))) .
```

The abstract forward value of `reverse1` is the identity, so the abstraction of the elements of a list doesn't change by reversing the list.

Let `compose` stand for `\f.\g.\x.f (g x)`. The abstract value of `compose` is

$$\begin{array}{l}(ID,\ \lambda(\alpha_{\texttt{f}},\kappa_{\texttt{f}}). \\ (ID,\ \lambda(\alpha_{\texttt{g}},\kappa_{\texttt{g}}). \\ (ID,\ \left\{\begin{array}{ll}\bot, & \text{if } \alpha_{\texttt{f}} = BOT \text{ or } \alpha_{\texttt{g}} = BOT \\ \kappa_{\texttt{f}} \circ \kappa_{\texttt{g}}, & \text{otherwise}\end{array}\right\})))\ .\end{array}$$

Thus, the result of the application of the composition of two functions is dynamic, and maps all values to dynamic values, if either function is dynamic; otherwise, the result is given by the application of the composition of the abstract forward values to the abstract argument.

Let `listcomp` stand for the expression denoting the function that composes lists of functions:

```
fix (\listcomp .
  \fs . case fs of
          nil u ->
            \x.x
          cons p ->
            let (g,gs) = p in
              compose g (listcomp gs)) .
```

The abstract value of `listcomp` is

$$(ID,\ \lambda(\alpha_{\mathtt{fs}}, \kappa_{\mathtt{fs}}).\\ (ID,\ \begin{cases} \bot, & \text{if } \alpha_{\mathtt{fs}} \neq \mathit{SPINE\ ID} \\ \sqcap_{i \geq 0} (\kappa_{\mathtt{fs}})^i, & \text{otherwise} \end{cases})) \ .$$

Since the abstract values of lists contain no information about the length of the lists of which they are abstractions, the abstract value of the composition of list elements is the glb of the composition over all lengths.

Let `flatten` stand for the expression denoting the function that flattens trees into lists:

```
fix (\flatten .
  \t . case t of
         leaf l   -> cons (l, nil ())
         branch p -> let (t1,t2) = p in
                       append (flatten t1)
                              (flatten t2)) .
```

The abstract value of `flatten` is

$$(ID,\ \lambda(\alpha_{\mathtt{fs}}, \kappa_{\mathtt{fs}}).\\ (\tau\ \alpha,\ \begin{cases} \bot, & \text{if } \tau\ \alpha_{\mathtt{fs}} \neq \mathit{LBR\ ID} \\ \kappa_{\mathtt{fs}}, & \text{otherwise} \end{cases})) \ ,$$

where τ maps BOT to BOT, and $LBR\,\alpha$ to $SPINE\,\alpha$ for α ranging over ID and BOT.

The function denoted by `compose listcomp flatten` that composes trees of functions has abstract value

$$(ID,\ \lambda(\alpha_{\mathtt{fs}}, \kappa_{\mathtt{fs}}).\\ (ID,\ \begin{cases} \bot, & \text{if } \alpha_{\mathtt{fs}} \neq \mathit{LBR\ ID} \\ \sqcap_{i \geq 0} (\kappa_{\mathtt{fs}})^i, & \text{otherwise} \end{cases})) \ .$$

Similarly to the case for lists, the abstract values of trees contain no information about the structure of the trees of which they are abstractions, so the abstract value of the composition of the values of the leaves is the glb of the composition over all tree structures.

7 More on Abstract Domains

The sizes of the abstract domains and the representations of the abstract values can be considerably optimised. The non-standard semantics of application—embodied by $\mathit{apply}^{\mathsf{P}}$—guarantees that the abstract values (BOT, f) and (BOT, f') from $\mathcal{T}^{\mathsf{P}}[\![\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2]\!]$, for all f and f', are effectively the same: $\mathit{apply}^{\mathsf{P}}\ (BOT, f) = \bot$ for all f. A practical analyser would take advantage of this fact, identifying (BOT, f) over all f. More generally, for function types embedded within data structures, e.g. $(\mathtt{Int},\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2)$, abstract values $((\alpha, ()), (BOT, f))$ would be identified over all f.

In the following we restrict attention to *denotable* values: at each type $\mathtt{T}$ those values that can be expressed as $\mathcal{E}[\![\,\mathtt{e}\,]\!]\ []$ for some $\mathtt{e}$. For every value $v \in \mathcal{T}^{\mathsf{S}}[\![\,\mathtt{T}\,]\!]$ there is a best abstraction—a greatest value $(\alpha, \kappa) \in \mathcal{T}^{\mathsf{P}}[\![\,\mathtt{T}\,]\!]$ such that $\mathcal{R}[\![\,\mathtt{T}\,]\!]((\alpha,\kappa), v, v)$. For types $\mathtt{T}_1$ and $\mathtt{T}_2$ not containing `->` we have $\mathcal{T}^{\mathsf{P}}_{\ddagger}[\![\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2]\!] \cong (|\,\mathcal{D}[\![\mathtt{T}_1]\!]\,| \times 1) \to (|\,\mathcal{D}[\![\mathtt{T}_2]\!]\,| \times 1)$, and if $\kappa \in \mathcal{T}^{\mathsf{P}}_{\ddagger}[\![\,\mathtt{T}\,]\!]$ is greatest such that $\mathcal{R}_{\ddagger}[\![\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2]\!]\ (\kappa, f, f)$ for denotable f, then κ maps $(ID, ())$ to $(ID, ())$ and distributes over $\sqcap$. The subset of functions from $\mathcal{T}^{\mathsf{P}}_{\ddagger}[\![\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2]\!]$ that map $(ID, ())$ to $(ID, ())$ and distribute over $\sqcap$ forms a complete lattice, hence attention may be restricted to this subset. Not only does this reduce the number of abstract functions, it also reduces their representation: such functions are determined by their behaviour on the $\sqcap$-*basis* of the lattice $|\,\mathcal{D}[\![\mathtt{T}_1]\!]\,|$—the set B of values such that every element of $|\,\mathcal{D}[\![\mathtt{T}_1]\!]\,|$ is the glb of some (possibly empty) subset of B, and no element of B can be expressed as the glb of some subset not containing that element. This optimisation can be generalised to higher order: $\mathcal{T}^{\mathsf{P}}_{\ddagger}[\![\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2]\!]$ may be restricted to functions that map $\top$ to $\top$ and distribute over $\sqcap$, for all function types $\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2$. (These results follow from those shown in [Dav94].)

8 Related Work

Consel [Con90] describes a binding-time analysis for higher-order untyped languages. As in our analysis abstract values have two parts, the first describing the static/dynamic properties of values, and the second describing how (for function types) abstract arguments are mapped to abstract results; there appears to be an implicit domain factorisation similar to ours, based on implicit type information collected by analysis semantics. In this respect there are many superficial similarities between the two techniques. No formal relation to the standard semantics is given, making a formal comparison with our technique problematic.

9 Conclusion

We have successfully generalised Launchbury's monomorphic projection-based binding-time analysis to higher-order, using abstract domains smaller than those typically used in BHA strictness analysis.

The next step would be to generalise to handle Hindley-Milner polymorphism. This has been done with good results at first order for binding-time analysis [Lau91a], and for BHA strictness analysis at higher order [Bar91, Bar93]. We anticipate that the combined use of these theories will give a reasonably straightforward generalisation to polymorphism. In addition to making the analysis more widely applicable, this should also greatly reduce the run-time cost of analysis.

Though not developed here, using the same approach it is possible to give a strictness analysis technique, again closely related to Hunt's PER-based strictness analysis technique [Hun91].

10 Acknowledgements

Thanks to the referees for their constructive comments on the first version of this paper.

References

[Abr89] S. Abramsky. "The lazy lambda calculus." *Research Topics in Functional Programming.* David Turner, ed., Addison-Wesley 1989.

[Abr90] S. Abramsky. "Abstract interpretation, logical relations and Kan extensions." *Journal of Logic and Computation*, 1, 1990.

[Bar91] G. Baraki. "A note on abstract interpretation of polymorphic functions." In [Hug91].

[Bar93] G. Baraki. *Abstract Interpretation of Polymorphic Higher-Order Functions.* Ph.D. thesis, Research report FP-1993-7, Department of Computing Science, University of Glasgow.

[BEJ88] D. Bjørner, A.P. Ershov, and N.D. Jones, eds. *Partial Evaluation and Mixed Computation, Proceedings IFIP TC2 Workshop, Gammel Avernæs*, Denmark, October 1987. North-Holland, 1988.

[Con90] C. Consel. "Binding Time Analysis for Higher Order Untyped Functional Languages." Proceedings of the 1990 ACM Conference on LISP and Functional Programming, pp264-272.

[Dav93] K. Davis. "Higher-order Binding-time Analysis." *Proceedings of the 1993 ACM on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '93)*, ACM Press, 1993.

[Dav94] K. Davis. *Projection-based Program Analysis.* Thesis submitted for degree of Ph.D., Computing Science Department, University of Glasgow, 1994.

[Go92] C.K. Gomard. "A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics." ACM TOPLAS, Vol 14, No. 2, April 1992.

[HM94] F. Henglein and C. Mossin. "Polymorphic binding-time analysis." European Symposium on Programming (ESOP '94).

[Hug91] J. Hughes, ed. *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture (FPCA '91)*, Cambridge, Sept 1991. LNCS 523, Springer Verlag, 1991.

[Hun91] S. Hunt. "PERs generalise projections for strictness analysis (extended abstract)." *Proceedings of the 1990 Glasgow Workshop on Functional Programming.* Simon L. Peyton Jones *et al.*, eds. Springer Workshops in Computing. Springer-Verlag, 1991.

[HS91] S. Hunt and D. Sands. "Binding time analysis: a new PERspective." *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices Vol. 26, No. 9, 1991.

[Jen92] T. Jensen. *Abstract Interpretation in Logical Form.* Ph.D. thesis, Report 93/11, Department of Computer Science, University of Copenhagen, 1992.

[Jon88] N.D. Jones. "Automatic program specialization: A re-examination from basic principles." In [BEJ88].

[Lau88] J. Launchbury. "Projections for specialisation." In [BEJ88].

[Lau91a] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD Thesis, Glasgow University, Nov 89. Distinguished Dissertation in Computer Science, Vol 1, CUP, 1991.

[Lau91c] J. Launchbury. "A strongly-typed, self-applicable partial evaluator." In [Hug91].

[Mog89] T. Mogensen. "Binding-time analysis for polymorphically typed higher order languages." *International Joint Conference on Theory and Practice of Software Development.* LNCS 352. Springer-Verlag 1989.

[NN88] H.R. Nielson and F. Nielson. "Automatic binding-time analysis for a typed λ-calculus." *Science of Computer Programming 10*, North Holland, 1988. Also in POPL '88.

[Sch88] D.A. Schmidt. "Static properties of partial reduction." In [BEJ88].

[Sew93] J. Seward. "Polymorphic strictness analysis using frontiers." *Proceedings of the 1993 ACM on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '93)*, ACM Press, 1993.

[Wad87] P. Wadler. "Strictness analysis on non-flat domains by abstract interpretation over finite domains. " S. Abramsky, C. Hankin, eds. *Abstract Interpretation of Declarative Languages.* Ellis-Horwood, 1987.