Higher-order Binding-time Analysis

Kei Davis*

Abstract

The partial evaluation process requires a *binding-time analysis*. Binding-time analysis seeks to determine which parts of a program's result is determined when some part of the input is known. Domain *projections* provide a very general way to encode a description of which parts of a data structure are *static* (known), and which are *dynamic* (not static). For *first-order* functional languages Launchbury [Lau91a] has developed an abstract interpretation technique for binding-time analysis in which the basic abstract value is a projection. Unfortunately this technique does not generalise easily to *higher-order* languages. This paper develops such a generalisation: a projection-based abstract interpretation suitable for higher-order binding-time analysis.

Launchbury [Lau91b] has shown that binding-time analysis and strictness analysis are equivalent problems at first order, and for projection-based analyses have exactly the same safety condition. We argue that the same is true at higher order, and hence our development also leads to a higher-order projection-based strictness analysis.

The principal limitation of our technique is the restriction to *monomorphic* typing.

1 Introduction

For first-order functional languages there exist both forward and backward abstract interpretation techniques in which the basic abstract value is a projection. Backward techniques [WH87, DW90] are useful for strictness analysis. Forward techniques can also give strictness information [Dav92], but here we are interested in their use for binding-time analysis in partial evaluation. Launchbury [Lau91a] describes a forward technique which he implemented as part of the first self-applicable, strongly-typed partial evaluator [Lau91c]. Unfortunately, neither the forward nor backward techniques generalise easily to *higher-order* languages. A higher-order projection-based backward strictness analysis has been described [DW91]; this paper develops a higher-order forward technique suitable for higher-order binding-time analysis, and briefly describes how to obtain the backward technique.

A domain *projection* is a continuous idempotent function that approximates the identity. The projections on a domain form a complete lattice, with the identity ID as the greatest element and the constant bottom function BOT as the least. A projection may be used to encode which components of values (data structures) are *static* (certainly available) and which are *dynamic* (possibly unavailable), by acting as the identity on static components and mapping dynamic components to $\perp$. With this interpretation, underestimation (in the usual function ordering) of projections is safe: static objects may be safely regarded as dynamic, but not vice versa.

As a concrete example, let $FST\ (x, y) = (x, \perp)$. Then FST encodes the information that the first component of a pair is static and the second dynamic. If $SND\ (x, y) = (\perp, y)$ and $swap\ (x, y) = (y, x)$ then the equation $SND \circ swap = swap \circ FST$ shows that if the first component of the argument to $swap$ is static and the second dynamic, then first component of the result is dynamic and the second static. Bearing in mind that the desired direction of information flow is from function argument to result (*forward*), and that underestimation is safe, the essence of the problem is, given function f and projection δ, to find projection γ such that $\gamma \circ f \sqsubseteq f \circ \delta$. More generally, given f we seek to find a function τ from projections to projections such that $(\tau\ \delta) \circ f \sqsubseteq f \circ \delta$ for all δ. Such a τ will be called a *projection abstraction* of f. The constant BOT function is always a projection abstraction of f, but greater τ are more informative; in fact, every continuous function has a greatest (most informative) projection abstraction [Dav92].

In Launchbury's development [Lau91a], an abstract semantics of a first-order non-strict functional language is given such that the abstract value of an expression $\mathtt{e}$ is a projection abstraction of $\lambda\rho.\mathcal{E}_\phi[\![\,\mathtt{e}\,]\!]\ \rho$, or just $\mathcal{E}_\phi[\![\,\mathtt{e}\,]\!]$—the standard evaluation function mapping variable environments ρ to values (here ϕ is an environment of first-order functions). In that development the abstract value of a first-order function definition is a projection abstraction of its standard value, so that abstract application is composition: in general if τ_f and τ_g are projection abstractions of f and g respectively then $\tau_f \circ \tau_g$ is a projection abstraction of $f \circ g$, so the abstract value of $\mathcal{E}_\phi[\![\,\mathtt{f\ e}\,]\!] = \phi[\![\,\mathtt{f}\,]\!] \circ \mathcal{E}_\phi[\![\,\mathtt{e}\,]\!]$ is $\tau_f \circ \tau_e$ where τ_f and τ_e are projection abstractions of $\phi[\![\,\mathtt{f}\,]\!]$ and $\mathcal{E}_\phi[\![\,\mathtt{e}\,]\!]$ respectively.

We take the central problem to be to find a projection abstraction of $\mathcal{E}[\![\,\mathtt{e}\,]\!]$ via a compositional abstract semantics. Some observations motivate our approach. Firstly, taking

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

a projection abstraction of $\mathcal{E}[\![\,\mathtt{e}\,]\!]$ for every e does not give a compositional semantics: the types just don't work out. This approach works at first order because there first-order functions are not first-class; in particular, they do not depend on any environment. Other 'obvious' schemes for inducing types at higher order similarly fail.

Secondly, for the purpose of binding-time analysis, the domain of projections on a function space is far richer than necessary: only ID and BOT are practically useful since a function is simply either static or dynamic. This suggests that the abstraction of an expression of function type $E \to (V \to W)$ by an element of $|\,E\,| \to |\,V \to W\,|$, where E is the domain of environments and $|\,U\,|$ denotes the domain of projections on domain U, is inappropriate; we require only a function from $|\,E\,|$ to $\{ID, BOT\}$. Further, since we will use the framework of Abramsky's *lazy lambda calculus* [Abr89] in which function types denote lifted function domains, and recognise that ID and BOT are precisely the projections guaranteed to be distinct on an arbitrary lifted domain $D_\perp$, we only need (in a sense made precise later) projections on the data-structure (or *data*) part—here the lifting—and no projections on function domains at all. This is a key observation; the idea that will be made precise and that guides the development is that, given a function, the projection abstraction that we require is not of the 'whole' function, but just the mapping from the data part of its argument to the data part of its result. In general this contains so little of the original function information that the rest of the information must be retained in some form; this form will be designed to allow a compositional abstract semantics.

Thus, as it turns out, abstract values will consist of two components: a projection abstraction part and a *forward* part—a more usual function from abstract values (corresponding to concrete arguments) to abstract values (corresponding to concrete results). Returning to the first-order case, the argument and result of every function (whether of the form $\mathcal{E}_\phi[\![\,\mathtt{e}\,]\!]$ or $\phi[\![\,\mathtt{f}\,]\!]$) are entirely data and the corresponding projection abstraction is of the whole function; hence the forward part is trivial and in the special case of first-order analysis simply does not appear.

The development proceeds as follows. We first define a language of types and expressions and their standard semantics. Next comes the key step: an intermediate non-standard semantics that isolates the dependency in the standard semantics of the data part of the result on the data part of the environment. Projection abstraction of the data-dependency components leads to the desired abstract semantics. Finally we show that reversing the direction of projection abstraction gives the complementary backward strictness analysis.

2 Language and Standard Semantics

The source language is a simple, strongly typed, monomorphic, non-strict functional language. A program in this language consists of a sequence of type definitions followed by an expression. The grammar for types and type definitions is as follows.

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

Nullary product corresponds to the so-called *unit* type. A unary product (T) will always have the same semantics as just T. Sums may be unary or multiary. Two type definitions that will be used later are for Bool, the booleans, and FunList, lists of functions from integers to integers:

```
Bool = True () + False () ,

FunList = Nil + Cons (Int -> Int, FunList) .
```

The grammar for expressions is

$$
\begin{array}{rl}
\mathtt{e} ::= & \mathtt{x} \\
| & \mathtt{n} \\
| & (\mathtt{e}_1 \ \mathtt{+} \ \mathtt{e}_2) \\
| & (\backslash\mathtt{x.e}) \\
| & (\mathtt{e}_1 \ \mathtt{e}_2) \\
| & (\mathtt{e}_1, \ldots, \mathtt{e}_n) \\
| & (\mathtt{case}\ \mathtt{e}_0\ \mathtt{of}\ (\mathtt{x}_1, \ldots, \mathtt{x}_n)\ \mathtt{->}\ \mathtt{e}_1) \\
| & (\mathtt{c}_i\ \mathtt{e}) \\
| & (\mathtt{case}\ \mathtt{e}_0\ \mathtt{of}\ \mathtt{c}_1\ \mathtt{x}_1\ \mathtt{->}\ \mathtt{e}_1; \ldots; \mathtt{c}_n\ \mathtt{x}_n\ \mathtt{->}\ \mathtt{e}_n) \\
| & (\mathtt{fix}\ \mathtt{e})
\end{array}
$$

The expression part e of a program need not be closed; for example e might have free variables such as input, a 'standard' input list of characters. Free variables are assumed to be bound by a global environment. This concept is important to our development: it allows every expression to be treated in the same way—closed expressions are not special. Addition for integers is provided as typical of arithmetic operations in this setting.

The (monomorphic) typing of expressions is entirely standard and is omitted.

2.1 Expression semantics

Since three different expression semantics will be given, it is convenient to express all of the semantics by a *generic* semantics $\mathcal{E}$ that is parameterised by a set of constants defined for each particular semantics. A particular instance of $\mathcal{E}$ will be indicated by a superscript, e.g. $\mathcal{E}^S$ for the standard semantics. The corresponding type semantics will have the same superscript, e.g. $\mathcal{T}^S$. 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). Then for all versions of the semantics and expressions e of type T with environment of type $(\mathtt{T}_1, \ldots, \mathtt{T}_n)$,

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

The generic semantics is defined as follows.

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

$$\mathcal{E}[\![\,\mathtt{n}\,]\!]\ \rho = \mathit{mkint}_{\mathcal{N}[\![\mathtt{n}]\!]}$$

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

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

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

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

$$\mathcal{E}[\![\,(\mathtt{e}_1, \ldots, \mathtt{e}_n)\,]\!]\ \rho = \mathit{tuple}\ (\mathcal{E}[\![\,\mathtt{e}_1\,]\!]\ \rho)\ \ldots\ (\mathcal{E}[\![\,\mathtt{e}_n\,]\!]\ \rho)$$

$$\mathcal{E}[\![\texttt{case } \texttt{e}_0 \texttt{ of } (\texttt{x}_1,\ldots,\texttt{x}_n) \texttt{ -> } \texttt{e}_1]\!]\,\rho = \mathcal{E}[\![\texttt{e}_1]\!]\,\rho[\texttt{x}_i \mapsto sel_i\ (\mathcal{E}[\![\texttt{e}_0]\!]\,\rho) \mid 1 \leq i \leq n]$$

$$\mathcal{E}[\![\texttt{c}_i\ \texttt{e}]\!]\,\rho = inc_i\ (\mathcal{E}[\![\texttt{e}]\!]\,\rho)$$

$$\begin{aligned}&\mathcal{E}[\![\texttt{case } \texttt{e}_0 \texttt{ of } \texttt{c}_1\ \texttt{x}_1 \texttt{ -> } \texttt{e}_1;\ldots;\ \texttt{c}_n\ \texttt{x}_n \texttt{ -> } \texttt{e}_n]\!]\\ &\quad = choose\ (\mathcal{E}[\![\texttt{e}_0]\!]\,\rho)\\ &\qquad\qquad (\mathcal{E}[\![\texttt{e}_1]\!]\,\rho[\texttt{x}_1 \mapsto outc_1\ (\mathcal{E}[\![\texttt{e}_0]\!]\,\rho)])\\ &\qquad\qquad \vdots\\ &\qquad\qquad (\mathcal{E}[\![\texttt{e}_n]\!]\,\rho[\texttt{x}_n \mapsto outc_n\ (\mathcal{E}[\![\texttt{e}_0]\!]\,\rho)])\end{aligned}$$

The function $\mathcal{N}$ maps a numeral `n` to the corresponding integer in the domain $Int = \mathbf{Z}_\perp$. Recalling that $\rho[\![\texttt{x}_i]\!]$ is short for $sel_i\ \rho$, environment update is defined by

$$\begin{aligned}&\rho[\texttt{x}_i \mapsto v]\\ &\quad = tuple\ (sel_1\ \rho)\ \ldots\ (sel_{i-1}\ \rho)\\ &\qquad\qquad v\\ &\qquad\qquad (sel_{i+1}\ \rho)\ \ldots\ (sel_n\ \rho)\ .\end{aligned}$$

Then for each expression semantics we need only define the constants sel_i, $mkint_n$, $plus$, $mkfun$, $apply$, $tuple$, inc_i, $outc_i$, $choose$, and fix. Like the evaluation function $\mathcal{E}$ each of these constants has functionality parameterised by some set of types and the particular type semantics $\mathcal{T}$, for example,

$$mkint_n \in \mathcal{T}[\![\texttt{Int}]\!]\ ,$$

$$mkfun \in (\mathcal{T}[\![\texttt{T}_1]\!] \rightarrow \mathcal{T}[\![\texttt{T}_2]\!]) \rightarrow \mathcal{T}[\![\texttt{T}_1 \texttt{ -> } \texttt{T}_2]\!]\ ,$$

$$apply \in \mathcal{T}[\![\texttt{T}_1 \texttt{ -> } \texttt{T}_2]\!] \rightarrow (\mathcal{T}[\![\texttt{T}_1]\!] \rightarrow \mathcal{T}[\![\texttt{T}_2]\!])\ ,$$

and so on. With one noted exception there will be no need to make parameterisation by type explicit.

2.2 Standard semantics

The standard type and expression semantics defined following are essentially that of Abramsky's lazy lambda calculus, in which function types denote lifted function spaces, allowing the semantics to differentiate between expressions of function type with no weak head normal form (WHNF), which denote $\perp$, and those with WHNF, which denote $lift\ f$ for some f. In effect this gives the function constructor `\` the same semantic status as a sum-type constructor `c`.

$$\mathcal{T}^S[\![\texttt{Int}]\!] = Int\ ,$$

$$\mathcal{T}^S[\![(\texttt{T}_1,\ldots,\texttt{T}_n)]\!] = \mathcal{T}^S[\![\texttt{T}_1]\!] \times \ldots \times \mathcal{T}^S[\![\texttt{T}_n]\!]\ ,$$

$$\mathcal{T}^S[\![\texttt{c}_1\ \texttt{T}_1 \texttt{ + } \ldots \texttt{ + } \texttt{c}_n\ \texttt{T}_n]\!] = \mathcal{T}^S[\![\texttt{T}_1]\!] + \ldots + \mathcal{T}^S[\![\texttt{T}_n]\!]\ ,$$

$$\mathcal{T}^S[\![\texttt{T}_1 \texttt{ -> } \texttt{T}_2]\!] = (\mathcal{T}^S[\![\texttt{T}_1]\!] \rightarrow \mathcal{T}^S[\![\texttt{T}_2]\!])_\perp\ .$$

Note that products are unlifted; a unary sum-of-products gives a lifted product. Domain $+$ is separated sum, e.g. $U + V = \{\perp\} \cup \{in_1\ u \mid u \in U\} \cup \{in_2\ v \mid v \in V\}$, and $out_i\ \perp = \perp$, $out_i\ (in_j\ v) = \perp$ for $i \neq j$, and $out_i\ (in_i\ v) = v$. Recursive type definitions give rise to recursive domain specifications which have the usual least fixed point solutions.

The constants for the standard expression semantics are defined as follows.

$$mkint_n^S = n$$

$$plus^S\ x\ y = x + y$$

$$mkfun^S = lift$$

$$apply^S = drop$$

$$fix^S = lfp$$

$$tuple^S\ x_1\ \ldots\ x_n = (x_1,\ldots,x_n)$$

$$sel_i^S = \pi_i$$

$$inc_i^S = in_i$$

$$outc_i^S\ x = out_i$$

$$choose^S\ x_0\ x_1\ \ldots\ x_n = case\ x_0\ of\ \{in_i\ v \rightarrow x_i\}$$

Throughout, π_i selects the i^{th} component of a tuple, lfp denotes least fixed point, and $case$ is always strict in its first argument.

3 Data-dependency semantics

The data-dependency, or D, semantics is intended to capture the dependency of the data part of $\mathcal{E}^S[\![\texttt{e}]\!]\,\rho$ on the data part of ρ. Two ideas guide the development of this semantics: the separation of values into their data and forward parts, and a method for determining the dependency of the value of a *sub*-expression of expression `e` on the environment of `e`. We address these issues in turn, then combine them to give the D semantics.

3.1 Abstracting dependency on the environment

Let `e` be a 'top-level' expression, that is, one that is not a subexpression of some other expression, and call the environment in which it is evaluated the *global* environment. The function defining the dependency of the value of `e` on the global environment is precisely $\mathcal{E}^S[\![\texttt{e}]\!]$. However, the value of subexpression `e'` of `e` depends on the value of a *local* environment which in general differs from the global environment, in particular, it may contain bindings for lambda-bound variables. Still, the local environment must be function of the global environment, so the value of `e'` is, if indirectly, a function of the global environment.

We wish to make explicit the value of `e'` on the global environment; following is a brief sketch of how this may be done. In this non-standard semantics the value of every expression (in its local environment) is function of the global environment. Let σ range over global environments (which we think of as tuples) and ρ range over local environments (which we'll think of as mapping variables to values, to avoid confusion). Then the local environment for the top-level expression maps each variable $\texttt{x}_i$ to the selector function π_i, which just picks out the appropriate value from the global environment. To make the idea clear we give the corresponding semantics of a few other expression forms. The value of a numeral is a constant function of the global environment, that is, $\mathcal{E}^M[\![\,\mathtt{n}\,]\!]\ \rho = \lambda\sigma.n$, where n is the standard meaning of n, and superscript M indicates this non-standard semantics. Then $\mathcal{E}^M[\![\,\mathtt{x}_1\mathtt{+}\mathtt{x}_2\,]\!]\ [\mathtt{x}_1 \mapsto \pi_1, \mathtt{x}_2 \mapsto \pi_2]$ is $\lambda\sigma.\pi_1\ \sigma + \pi_2\ \sigma$; applying this function to the global environment gives the usual result. More generally,

$$\begin{aligned}&\mathcal{E}^M[\![\,\mathtt{e}_1\ \mathtt{+}\ \mathtt{e}_2\,]\!]\ \rho\\ &\quad = \lambda\sigma.(\mathcal{E}^M[\![\,\mathtt{e}_1\,]\!]\ \rho\ \sigma) + (\mathcal{E}^M[\![\,\mathtt{e}_2\,]\!]\ \rho\ \sigma)\ .\end{aligned}$$

Products are similar, for example,

$$\begin{aligned}&\mathcal{E}^M[\![\,(\mathtt{e}_1,\mathtt{e}_2)\,]\!]\ \rho\\ &\quad = \lambda\sigma.(\mathcal{E}^M[\![\,\mathtt{e}_1\,]\!]\ \rho\ \sigma,\ \mathcal{E}^M[\![\,\mathtt{e}_2\,]\!]\ \rho\ \sigma)\ .\end{aligned}$$

For lambda abstraction we have

$$\mathcal{E}^M[\![\mathtt{\backslash x.e}]\!]\ \rho\ =\ \lambda\sigma.\mathit{lift}\ (\lambda x.\mathcal{E}^M[\![\,\mathtt{e}\,]\!]\ \rho[\mathtt{x} \to x])\ ,$$

which comes from a domain of the form

$$E \to ((E \to U) \to (E \to V))_\perp\ ,$$

where E is the domain of global environments. This makes clear that expressions of function type, like all other expressions, are functions of the global environment. Then for application we have

$$\begin{aligned}&\mathcal{E}^M[\![\,\mathtt{e}_1\ \mathtt{e}_2\,]\!]\ \rho\\ &\quad = \lambda\sigma.\mathit{drop}\ (\mathcal{E}^M[\![\,\mathtt{e}_1\,]\!]\ \rho\ \sigma)\ (\mathcal{E}^M[\![\,\mathtt{e}_2\,]\!]\ \rho)\ \sigma\ .\end{aligned}$$

Thus, to evaluate an application the first expression is evaluated, applied to the global environment to yield a function, which is turn applied to the value of the second expression; the result is again a function of the global environment. A slightly more complicated example is

```
case b of True x -> \y.y; False x -> \y.1 .
```

Evaluation in environment ρ gives

$$\begin{aligned}&\lambda\sigma.\mathit{case}\ (\rho[\![\mathtt{b}]\!]\ \sigma)\ \mathit{of}\\ &\qquad \mathit{in}_1\ x\ \to\ \mathit{lift}\ (\lambda y.y)\\ &\qquad \mathit{in}_2\ x\ \to\ \mathit{lift}\ (\lambda y.\lambda\sigma.1)\ .\end{aligned}$$

(What we have described is a variant of the call-by-name translation of the monad of state readers into the lambda calculus [Wad90]; here global environments play the role of states.)

3.2 An informal explanation of the development

This section is intended to give an intuitive feeling for the subsequent development by considering a simple example.

Consider the expression `\x.b`, where `x` is of type `Int` and `b` is of type `Bool`. Then

$$BOT \circ \mathcal{E}^S[\![\mathtt{\backslash x.b}]\!]\ \sqsubseteq\ \mathcal{E}^S[\![\mathtt{\backslash x.b}]\!] \circ BOT\ .$$

What's more, given BOT on the right-hand side, BOT is the greatest projection on the left-hand side that satisfies the inequality. Yet for the purposes of *evaluation* this is not want we want—it is too strong. Even if the value of the free variable `b` is not known, the value of the lambda expression is known insofar as can be observed, that is, it has WHNF. Since for binding-time analysis this is useful information, it would be desirable to somehow obtain ID given BOT on the right-hand side. Following we give an indication of how this problem is solved. In the M semantics, the domain corresponding to the type of this expression is $E \to ((E \to U) \to (E \to V))_\perp$, where E and V are $\mathcal{T}^S[\![\mathtt{Bool}]\!]$ and U is *Int*. We may factor $((E \to U) \to (E \to V))_\perp$ into $\mathbf{1}_\perp \times ((E \to U) \to (E \to V))$; the embedding is defined by

$$\begin{aligned}\mathit{emb}\ \perp\ &=\ (\perp,\perp)\ ,\\ \mathit{emb}\ (\mathit{lift}\ f)\ &=\ (\mathit{lift}\ (),\ f)\ .\end{aligned}$$

Here $\mathbf{1}_\perp$ encodes the data or observable part of the value. Then the M function domain has an embedding into $E \to (\mathbf{1}_\perp \times ((E \to U) \to (E \to V)))$, which is isomorphic to $(E \to \mathbf{1}_\perp) \times (E \to ((E \to U) \to (E \to V)))$. Under the embedding and implied isomorphism, the first component of $\mathcal{E}^S[\![\mathtt{\backslash x.b}]\!]$ is $\lambda\rho.\mathit{lift}\ ()$—isolating the information that for any environment `\x.b` has WHNF. The greatest projection abstraction of this function maps BOT to ID, as desired.

The other question is how to deal with the second component. Recall application $\mathit{apply}^M\ f\ x$ in M is $\lambda\sigma.\mathit{drop}(f\ \sigma)\ x\ \sigma$. This makes clear that when applied, the same value of σ is used by both the function and its result. Hence given a value g from $E \to ((E \to U) \to (E \to V))$ we may safely 'defer' the dependence of g on the value from E by applying h, where $h\ g = \lambda x.\lambda\sigma.g\ \sigma\ x\ \sigma$, to yield a value in $(E \to U) \to (E \to V)$, which makes clear that this component does not depend on the environment. To summarise, in the factored form of the M semantics we expect the value of `\x.b` to come from a domain of the form

$$(E \to \mathbf{1}_\perp)\ \times\ ((E \to U) \to (E \to V))\ .$$

An argument of `\x.b` will come from domain $E \to U$, so abstract application (as it turns out) is ordinary application of the second component of the abstract function to the abstract argument, since the second component also contains the information in the first component (recall the action of h above).

Just as projection abstraction maps from $E \to \mathbf{1}_\perp$ to $|\,E\,| \to |\,\mathbf{1}_\perp\,|$ so it maps from $E \to U$ and $E \to V$ to $|\,E\,| \to |\,U\,|$ and $|\,E\,| \to |\,V\,|$ respectively. Hence there is a map from

$$(E \to \mathbf{1}_\perp)\ \times\ ((E \to U) \to (E \to V))$$

to

$$(|\,E\,| \to |\,\mathbf{1}_\perp\,|)\ \times\ ((|\,E\,| \to |\,V\,|) \to (|\,E\,| \to |\,V\,|))\ .$$

Again, the values from the first component are the desired projection abstractions; and abstract application will be ordinary application of the second component of the abstract function to the abstract argument.

Necessarily, this section has been very informal. The domain factorisation and implied semantics has not been made precise; the formal relation between the implied semantics and the standard semantics has not been given, and we have not considered that the environment, argument, and result types may also be higher order and the corresponding domains require (recursively) similar factorisation and abstraction.

3.3 Domain factorisation

Given type `T` with corresponding domain T in the standard semantics, we wish to factor values in T into their data and forward parts. The *data domain* corresponding to type `T` is $\mathcal{D}[\![\mathtt{T}]\!]$, defined as follows. Roughly, $\mathcal{D}$ is like $\mathcal{T}^S$ except that function spaces are replaced by the one-point domain $\mathbf{1} = \{()\}$.

$$\mathcal{D}[\![\,\mathtt{Int}\,]\!]\ =\ \mathit{Int}$$

$$\mathcal{D}[\![\, \texttt{(T}_1\texttt{, ..., T}_n\texttt{)} \,]\!] = \mathcal{D}[\![\, \texttt{T}_1 \,]\!] \times \ldots \times \mathcal{D}[\![\, \texttt{T}_n \,]\!]$$

$$\mathcal{D}[\![\, \texttt{c}_1 \; \texttt{T}_1 \; \texttt{+} \ldots \texttt{+} \; \texttt{c}_n \; \texttt{T}_n \,]\!] = \mathcal{D}[\![\, \texttt{T}_1 \,]\!] + \ldots + \mathcal{D}[\![\, \texttt{T}_n \,]\!]$$

$$\mathcal{D}[\![\, \texttt{T}_1 \texttt{ -> } \texttt{T}_2 \,]\!] = \mathbf{1}_\perp$$

Next we define $data_{\texttt{T}}$, the function from values in $\mathcal{T}^S[\![\,\texttt{T}\,]\!]$ to their data parts in $\mathcal{D}[\![\,\texttt{T}\,]\!]$.

$$data_{\texttt{Int}} = id_{Int}$$

$$data_{(\texttt{T}_1,\ldots,\texttt{T}_n)} = data_{\texttt{T}_1} \times \ldots \times data_{\texttt{T}_n}$$

$$data_{\texttt{c}_1\ \texttt{T}_1\ +\ \ldots\ +\ \texttt{c}_n\ \texttt{T}_n} = data_{\texttt{T}_1} + \ldots + data_{\texttt{T}_n}$$

$$data_{\texttt{T}_1\texttt{->}\texttt{T}_2} = (\lambda x.())_\perp$$

The last definition uses function lifting, defined by $f_\perp\ \perp = \perp$ and $f_\perp\ (lift\ x) = lift\ (f\ x)$. Recursive type definitions give rise to recursive function specifications, with the usual least fixed point solutions.

In the same style as $\mathcal{D}[\![\,\texttt{T}\,]\!]$ and $data_{\texttt{T}}$ we define $\mathcal{F}^S[\![\,\texttt{T}\,]\!]$ and $for_{\texttt{T}}$ to give the forward domain for T and the function from $\mathcal{T}^S[\![\,\texttt{T}\,]\!]$ to their forward parts in $\mathcal{F}^S[\![\,\texttt{T}\,]\!]$, respectively. Roughly, $\mathcal{F}^S$ is like $\mathcal{T}^S$ with all lifting removed and sum replaced by product. (The treatment of `Int` throughout is consistent with `Int` being defined as an infinite sum of nullary products.)

$$\mathcal{F}^S[\![\, \texttt{Int} \,]\!] = \mathbf{1}$$

$$\mathcal{F}^S[\![\, \texttt{(T}_1\texttt{, ..., T}_n\texttt{)} \,]\!] = \mathcal{F}^S[\![\, \texttt{T}_1 \,]\!] \times \ldots \times \mathcal{F}^S[\![\, \texttt{T}_n \,]\!]$$

$$\mathcal{F}^S[\![\, \texttt{c}_1 \; \texttt{T}_1 \; \texttt{+} \ldots \texttt{+} \; \texttt{c}_n \; \texttt{T}_n \,]\!] = \mathcal{F}^S[\![\, \texttt{T}_1 \,]\!] \times \ldots \times \mathcal{F}^S[\![\, \texttt{T}_n \,]\!]$$

$$\mathcal{F}^S[\![\, \texttt{T}_1 \texttt{ -> } \texttt{T}_2 \,]\!] = \mathcal{T}^S[\![\, \texttt{T}_1 \,]\!] \rightarrow \mathcal{T}^S[\![\, \texttt{T}_2 \,]\!]$$

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

$$fun_{\texttt{Int}} = \lambda x.()$$

$$fun_{(\texttt{T}_1,\ldots,\texttt{T}_n)} = fun_{\texttt{T}_1} \times \ldots \times fun_{\texttt{T}_n}$$

$$fun_{\texttt{c}_1\ \texttt{T}_1\ +\ \ldots\ +\ \texttt{c}_n\ \texttt{T}_n} = \lambda x.case\ x\ of\ \{in_i\ y \rightarrow (\perp,\ldots,\perp,y,\perp,\ldots,\perp)\}$$

$$fun_{\texttt{T}_1\texttt{->}\texttt{T}_2} = drop$$

We will write $fac_{\texttt{T}}$ for $\lambda x.(data_{\texttt{T}}\ x,\ for_{\texttt{T}}\ x)$. Then

$$\mathcal{D}[\![\,\texttt{T}\,]\!] \times \mathcal{F}^S[\![\,\texttt{T}\,]\!]$$

is a factorisation of $\mathcal{T}^S[\![\,\texttt{T}\,]\!]$, and

$$fac_{\texttt{T}} \in \mathcal{T}^S[\![\,\texttt{T}\,]\!] \rightarrow (\mathcal{D}[\![\,\texttt{T}\,]\!] \times \mathcal{F}^S[\![\,\texttt{T}\,]\!])$$

is an embedding which *determines* a corresponding projection $unfac_{\texttt{T}}$. Rather than give an explicit definition of $unfac_{\texttt{T}}$ 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 $T = \mathcal{T}^S[\![\,\texttt{T}\,]\!]$ and $U = \mathcal{T}^S[\![\,\texttt{U}\,]\!]$ then the factorisation of $(T \rightarrow U)_\perp = \mathcal{T}^S[\![\,\texttt{T->U}\,]\!]$ is $\mathbf{1}_\perp \times (T \rightarrow U)$. Notice that factorisation 'stops' at the function space constructor. Now $fac_{\texttt{T->U}}\ \perp = (\perp,\perp)$ and $fac_{\texttt{T->U}}\ (lift\ f) = (lift\ (),\ f)$ for all f. In the other direction $unfac_{\texttt{T->U}}\ (\perp,\perp) = \perp$ and $unfac_{\texttt{T->U}}\ (lift\ (),\ f) = (lift\ f)$; this must be since in general $unfac_{\texttt{T}} \circ fac_{\texttt{T}}$ is the identity. However, $unfac_{\texttt{T->U}}\ (\perp, f) = \perp$. Roughly, loss of the data part loses subsidiary forward parts. This has a natural operation interpretation: if we have a data structure containing functions and lose (part of) a data structure, e.g. throw away the pointer to it, then the functions (forward parts) are effectively undefined since they become inaccessable. Further, for all data values $\sigma \in \mathcal{D}[\![\,\texttt{T}\,]\!]$ and all forward values $f \in \mathcal{F}^S[\![\,\texttt{T}\,]\!]$ it is a fact that $data_{\texttt{T}}\ (unfac_{\texttt{T}}\ (\sigma, f)) = \sigma$. This shows that the data part of a value is unaffected by changes in the forward values; intuitively, changing the functions at the leaves of a data structure doesn't change the value of the data structure anywhere else.

3.4 Type semantics

Let E be the type of global environments, that is, the tuple type of the environment of the top-level expression to be analysed. Then the data-dependency type semantics $\mathcal{T}^D$ is defined as

$$\mathcal{T}^D[\![\,\texttt{T}\,]\!] = (\mathcal{D}[\![\,\texttt{E}\,]\!] \rightarrow \mathcal{D}[\![\,\texttt{T}\,]\!]) \times \mathcal{F}^D[\![\,\texttt{T}\,]\!]$$

with $\mathcal{F}^D$ is defined exactly like $\mathcal{F}^S$, with superscript D everywhere replacing superscript S. Thus, for example,

$$\mathcal{F}^D[\![\, \texttt{T}_1 \texttt{ -> } \texttt{T}_2 \,]\!] = \mathcal{T}^D[\![\, \texttt{T}_1 \,]\!] \rightarrow \mathcal{T}^D[\![\, \texttt{T}_2 \,]\!] .$$

The intention is that an appropriate value from $\mathcal{T}^D[\![\,\texttt{T}\,]\!]$ will embody the dependency (in the standard semantics) of the data part of the result (from $\mathcal{D}[\![\,\texttt{T}\,]\!]$) on the data part of the argument (from $\mathcal{D}[\![\,\texttt{E}\,]\!]$). The second component is the forward part: in the case of function types as shown, a function from abstract argument to abstract results. Intuitively, the formulation of the corresponding expression semantics acts like the M semantics for data parts, and like the standard semantics for forward parts. The type definition makes clear that the values of lambda-expression bodies—the forward parts—do not depend on the value of the global environment.

3.5 Relation between S and D semantics

At each type T we define a relation $rel^{SD}_{\texttt{T},\sigma} \in \mathcal{T}^S[\![\,\texttt{T}\,]\!] \leftrightarrow \mathcal{T}^D[\![\,\texttt{T}\,]\!]$, parameterised by a data value $\sigma \in \mathcal{D}[\![\,\texttt{E}\,]\!]$, as follows (to alleviate clutter superscript SD will be omitted).

$$rel_{\texttt{T},\sigma}\ (v,\ (g,f)) = (data_{\texttt{T}}\ v = g\ \sigma) \wedge (frel_{\texttt{T}}\ (for_{\texttt{T}}\ v,\ f))$$

where $frel_{\texttt{T}} \in \mathcal{F}^S[\![\,\texttt{T}\,]\!] \leftrightarrow \mathcal{F}^D[\![\,\texttt{T}\,]\!]$ 'logically' relates forward values as follows.

$$frel_{\texttt{Int}}\ (x,y) = True$$

$$frel_{(\texttt{T}_1,\ldots,\texttt{T}_n)} = frel_{\texttt{T}_1} \times \ldots \times frel_{\texttt{T}_n}$$

$$frel_{\texttt{c}_1\ \texttt{T}_1\ +\ \ldots\ +\ \texttt{c}_n\ \texttt{T}_n} = frel_{\texttt{T}_1} \times \ldots \times frel_{\texttt{T}_n}$$

$$frel_{\texttt{T}_1\texttt{->}\texttt{T}_2} = rel_{\texttt{T}_1,\sigma} \rightarrow rel_{\texttt{T}_2,\sigma}$$

Here $\times$ and $\rightarrow$ are defined on relations in the standard way: for relations R and S we have $(R \times S)((x,y),(x',y'))$ iff $R(x,x')$ and $S(y,y')$, and $(R \rightarrow S)(f,g)$ iff for all x and y such that $R(x,y)$ we have $S(f\ x,\ g\ y)$. 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.)

3.6 Semantics of expressions

For all $\mathtt{e} : \mathtt{T}$ we require that $\mathcal{E}^S[\![\,\mathtt{e}\,]\!]$ and $\mathcal{E}^D[\![\,\mathtt{e}\,]\!]$ be logically related. Recalling that $\mathcal{E}[\![\,\mathtt{e}\,]\!] \in \mathcal{T}[\![\,\mathtt{E}\,]\!] \rightarrow \mathcal{T}[\![\,\mathtt{T}\,]\!]$ for $\mathtt{e} : \mathtt{T}$, the required relation is $rel_{\mathtt{E},\sigma} \rightarrow rel_{\mathtt{T},\sigma}$ for all σ. It is not hard to show that if the constants defining the S and D semantics are similarly related, then so are the expression semantics. For example, the generic functionality of *mkfun* is $(\mathcal{T}[\![\,\mathtt{T}_1\,]\!] \rightarrow \mathcal{T}[\![\,\mathtt{T}_2\,]\!]) \rightarrow \mathcal{T}[\![\,\mathtt{T}_1\ \mathtt{->}\ \mathtt{T}_2\,]\!]$ and we require $mkfun^S$ to be related to $mkfun^D$ by

$$(rel_{\mathtt{T}_1,\sigma} \rightarrow rel_{\mathtt{T}_2,\sigma}) \rightarrow rel_{\mathtt{T}_1\mathtt{->}\mathtt{T}_2,\sigma}\ .$$

The following definitions give correctly related constants.

$$mkint^D_n \;=\; (\lambda\sigma.n,\ ())\ ,$$

$$plus^D\ (g_1,\ f_1)\ (g_2,\ f_2) \;=\; (\lambda\sigma.(g_1\ \sigma) + (g_2\ \sigma),\ ())$$

$$mkfun^D\ f \;=\; (\lambda\sigma.lift\ (),\ f)$$

$$apply^D\ (g,\ f)\ x \;=\; f\ x$$

$$fix^D \;=\; lfp$$

$$\begin{aligned} &tuple^D\ (g_1,f_1)\ \ldots\ (g_n,f_n)\\ &\quad= (\lambda\sigma.(g_1\ \sigma,\ldots,g_n\ \sigma),\ (f_1,\ldots,f_n)) \end{aligned}$$

$$sel^D_i\ (g,\ f) \;=\; (\pi_i \circ g,\ \pi_i\ f)$$

$$inc^D_i\ (g,\ f) \;=\; (in_i \circ g,\ (\bot,\ldots,\bot,f,\bot,\ldots,\bot))$$

$$outc^D_i\ (g,\ f) \;=\; (out_i \circ g,\ \pi_i\ f)$$

For inc^D_i the symbol f appears in the i^{th} position in the tuple on the right-hand side.

For $choose^D$ we must make explicit the dependence on the type of its result so we define $choose^D_{\mathtt{T}}$ for each result type $\mathtt{T}$.

$$\begin{aligned} &choose^D_{\mathtt{T}}\ \rho\ (g_0,f_0)\ \ldots\ (g_m,f_m)\\ &\quad= (\lambda\sigma.case\ (g_0\ \sigma)\ of\ \{in_i\ x\ \rightarrow\ g_i\ \sigma\},\\ &\qquad forpart_{\mathtt{T}}\ f_1\ \ldots\ f_m)\\ &\quad where\\ &\qquad forpart_{\mathtt{Int}}\ f_1\ \ldots\ f_m \;=\; ()\\ &\qquad forpart_{(\mathtt{T}_1,\ldots,\mathtt{T}_n)}\ f_1\ \ldots\ f_m\\ &\qquad\quad= (forpart_{\mathtt{T}_1}\ (\pi_1\ f_1)\ \ldots\ (\pi_1\ f_m),\\ &\qquad\qquad \vdots\\ &\qquad\qquad forpart_{\mathtt{T}_n}\ (\pi_n\ f_1)\ \ldots\ (\pi_n\ f_m))\\ &\qquad forpart_{\mathtt{c}_1\ \mathtt{T}_1\ +\ \ldots\ +\ \mathtt{c}_n\ \mathtt{T}_n}\ f_1\ \ldots\ f_m\\ &\qquad\quad= (forpart_{\mathtt{T}_1}\ (\pi_1\ f_1)\ \ldots\ (\pi_1\ f_m),\\ &\qquad\qquad \vdots\\ &\qquad\qquad forpart_{\mathtt{T}_n}\ (\pi_n\ f_1)\ \ldots\ (\pi_n\ f_m))\\ &\qquad forpart_{\mathtt{T}_1\mathtt{->}\mathtt{T}_2}\ f_1\ \ldots\ f_m\\ &\qquad\quad= \lambda x\ .\ choose^D_{\mathtt{T}_2}\ (g_0,f_0)\ (f_1\ x)\ \ldots\ (f_m\ x) \end{aligned}$$

3.7 Using the relation

Fact. For every forward value $f \in \mathcal{F}^S[\![\,\mathtt{T}\,]\!]$ there is a D value $d_f \in \mathcal{T}^D[\![\,\mathtt{T}\,]\!]$ such that for all data values $\sigma \in \mathcal{D}[\![\,\mathtt{T}\,]\!]$ the values $unfac_{\mathtt{T}}\ (\sigma,f)$ and d_f are related by $rel_{\mathtt{T},\sigma}$.

It is clear from the definition of the relation that for any type and any f the first component of d_f must be the identity. Rather than give the mapping from f to the second component of d_f we just give some examples. For any type $\mathtt{T}$ not containing `->` and hence $f = ()$, the corresponding d_f is $(id, ())$. For $\mathtt{T} = \mathtt{Int->Int}$ and f any function from integers to integers, the corresponding d_f is

$$(id,\ \lambda(q,r).(\lambda\sigma.case\ (q\ \sigma)\ of\ lift\ () \rightarrow f\ (q\ \sigma), ()))\ .$$

Recall that $\mathcal{E}^S[\![\,\mathtt{e}\,]\!]$ is related to $\mathcal{E}^D[\![\,\mathtt{e}\,]\!]$ by $rel_{\mathtt{E},\sigma} \rightarrow rel_{\mathtt{T},\sigma}$ for all σ and $\mathtt{e}:\mathtt{T}$ with environment of type $\mathtt{E}$. Let $f \in \mathcal{F}^S[\![\,\mathtt{E}\,]\!]$ be the forward part of a standard environment, and $\rho_f \in \mathcal{T}^D[\![\,\mathtt{E}\,]\!]$ be a D environment related to f as just described. Then for all data parts of standard environments σ the value $\mathcal{E}^S[\![\,\mathtt{e}\,]\!]\ (unfac_{\mathtt{T}}\ (\sigma,f))$ is related to $\mathcal{E}^D[\![\,\mathtt{e}\,]\!]\ \rho_f$ by $rel_{\mathtt{T},\sigma}$. This means that if $(q,r) = \mathcal{E}^D[\![\,\mathtt{e}\,]\!]\ \rho_f$, then for all σ it must be that $data_{\mathtt{T}}\ (\mathcal{E}^S[\![\,\mathtt{e}\,]\!]\ (unfac_{\mathtt{T}}\ (\sigma,f))) = q\ \sigma$. It is in this sense that the D semantics captures the dependency of the data part of $\mathcal{E}^S[\![\,\mathtt{e}\,]\!]\ \rho$ on the data part of ρ. Obviously for this to work the (nominal) forward part f of ρ must be known. Later we argue that this is not a practical restriction.

Now we introduce projections. Every projection γ on the data domain for any type $\mathtt{T}$ defines a projection γ' on the standard domain for $\mathtt{T}$ defined by

$$\gamma'\ v \;=\; unfac_{\mathtt{T}}\ (\gamma\ (data_{\mathtt{T}}\ v),\ for_{\mathtt{T}}\ v)\ .$$

Roughly speaking, γ' is the greatest projection that acts like γ on the data part of its argument. Then for $f \in \mathcal{F}^S[\![\,\mathtt{E}\,]\!]$ and $\rho_f \in \mathcal{T}^D[\![\,\mathtt{E}\,]\!]$ with $(q,r) = \mathcal{E}^D[\![\,\mathtt{e}\,]\!]\ \rho_f$, function τ a projection abstraction of q, and for all ρ such that $for_{\mathtt{T}}\ \rho = f$, we have

$$\forall\ \gamma\ .\ (\tau\ \gamma) \circ data_{\mathtt{T}} \circ \mathcal{E}^S[\![\,\mathtt{e}\,]\!] \;\sqsubseteq\; data_{\mathtt{T}} \circ \mathcal{E}^S[\![\,\mathtt{e}\,]\!] \circ \gamma'\ .$$

The requirement that some information be known about the forward part of the standard environment in order to get good results is seemingly severe. Typically, however, whole programs to not take values of function type as input and so the input has a trivial forward part and so no information need be supplied; for higher-order functions inside the program the semantics *generates* the appropriate forward information, hence we claim that this is no real restriction at all. Here, if $\mathtt{T}$ does not contain `->`, then $data_{\mathtt{T}}$ is the identity function. When both environment and result types do not contain `->` this condition simplifies to the standard safety condition

$$\forall\ \gamma\ .\ (\tau\ \gamma) \circ \mathcal{E}^S[\![\,\mathtt{e}\,]\!] \;\sqsubseteq\; \mathcal{E}^S[\![\,\mathtt{e}\,]\!] \circ \gamma\ ,$$

hence our claim that this is a strict generalisation of Launchbury's technique.

4 Projection Semantics

We intend that a value $(q, r) \in \mathcal{T}^D[\![\mathtt{T}]\!]$ be related to $(\tau, \kappa) \in \mathcal{T}^P[\![\mathtt{T}]\!]$ in the projection (P) semantics when τ is a projection abstraction of q and r is 'logically' related to κ. This dictates that the P type semantics be

$$\mathcal{T}^P[\![\mathtt{T}]\!] \;=\; (|\,\mathcal{D}[\![\mathtt{E}]\!]\,| \;\to\; |\,\mathcal{D}[\![\mathtt{T}]\!]\,|) \;\times\; \mathcal{F}^P[\![\mathtt{T}]\!]$$

where, as with $\mathcal{F}^D$, the domain $\mathcal{F}^P$ is defined just like $\mathcal{F}^S$, with superscript P everywhere replacing superscript S. The precise relation $rel^{DP}_{\mathtt{T}} \in \mathcal{T}^D[\![\mathtt{T}]\!] \leftrightarrow \mathcal{T}^P[\![\mathtt{T}]\!]$ between D and P values is

$$\begin{aligned} &rel^{DP}_{\mathtt{T}}\ ((q, r),\ (\tau, \kappa)) \\ &\quad = (\tau \text{ is a projection abstraction of } q) \wedge \\ &\qquad (frel^{DP}_{\mathtt{T}}\ (r,\ \kappa)) \end{aligned}$$

where $frel^{DP}_{\mathtt{T}}$ is defined exactly like $frel^{SD}_{\mathtt{T}}$, with superscript SD everywhere replaced by superscript DP, and σ omitted.

In principle, we can 'read off' the appropriate definition for each P-semantics constant from the definition of the corresponding D-semantics constant: the first component of the former may be any projection abstraction of the first component of the latter, then the second component is correctly induced: its definition is textually the same. For example,

$$mkfun^D\ f \;=\; (\lambda\sigma.lift\ (),\ f)\ ,$$

so

$$mkfun^P\ f \;=\; (\lambda\alpha.ID,\ f)\ ,$$

since $\lambda\alpha.ID$ is a (in fact, the greatest) projection abstraction of $\lambda\sigma.lift\ ()$. Ideally we would always choose the greatest such projection abstraction for each constant, but for $choose^D$ it is not clear how to avoid some approximation. The most obvious approximation corresponds exactly to the approximation resulting from abstraction to our particular choice of finite abstract projection domains, so, rather than give an approximate semantics for the full projection domains and then approximate again, we will simply give the final abstract P-semantics constants.

We need some notation for specifying projections on the data domains. For products define $(\gamma_1, \ldots, \gamma_n) = \gamma_1 \times \ldots \times \gamma_n$ and $PI_i\ (\gamma_1, \ldots, \gamma_n) = \gamma_i$. For sum type $\mathtt{c}_1\ \mathtt{T}_1$ `+ ... +` $\mathtt{c}_n\ \mathtt{T}_n$ with data domain $T_1 + \ldots + T_n$ define $C_i\ \alpha = ID + \cdots + ID + \alpha + ID + \cdots + ID$ (where α is the i^{th} summand) and $OUTC_i\ (\gamma_1 + \cdots + \gamma_n) = \gamma_i$ and $OUTC_i\ BOT = BOT$.

The abstract type semantics is

$$\mathcal{T}^A[\![\mathtt{T}]\!] \;=\; (FinProj_{\mathtt{E}} \;\to\; FinProj_{\mathtt{T}}) \;\times\; FinFor_{\mathtt{T}}$$

where $FinProj_{\mathtt{T}}$ and $FinFor_{\mathtt{T}}$ are defined following.

The definition of $FinProj_{\mathtt{T}}$ is based on that in [Lau91a]. A projection $\gamma \in \mathcal{D}[\![\mathtt{T}]\!]$ is in $FinProj_{\mathtt{T}}$ if γ **proj** `T` can be inferred from the following rules.

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

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

$$BOT\ \textbf{proj}\ \mathtt{c}_1\ \mathtt{T}_1\ \texttt{+ ... +}\ \mathtt{c}_n\ \mathtt{T}_n$$

$$\frac{P_1\ \textbf{proj}\ \mathtt{T}_1\ \cdots\ P_n\ \textbf{proj}\ \mathtt{T}_n}{C_1\ P_1 \sqcap \cdots \sqcap C_n\ P_n\ \textbf{proj}\ \mathtt{c}_1\ \mathtt{T}_1\ \texttt{+ ... +}\ \mathtt{c}_n\ \mathtt{T}_n}$$

$$\frac{P_1\ \textbf{proj}\ \mathtt{T}_1\ \cdots\ P_n\ \textbf{proj}\ \mathtt{T}_n}{(P_1, \ldots, P_n)\ \textbf{proj}\ \texttt{(}\mathtt{T}_1\texttt{, ... ,}\mathtt{T}_n\texttt{)}}$$

Then $()$ is the sole projection on the one-point domain corresponding to the nullary tuple type. For recursive types we consider the simplest non-trivial case `A=T(A)`. Assuming γ **proj** `A`,

$$\frac{P(\gamma)\ \textbf{proj}\ \mathtt{T(A)}}{\mu\gamma.P(\gamma)\ \textbf{proj}\ \mathtt{A{=}T(A)}}\ .$$

The generalisation to multiple interdependent type definitions is straightforward.

We give two examples. The projections in $FinProj_{\mathtt{Bool}}$ are BOT and $True\ () \sqcap\ False\ ()$, which is just ID. (Similarly, treating `Int` as though it were an infinite sum would give exactly the projections ID and BOT.) In fact, for every `T` the projections ID and BOT are in $FinProj_{\mathtt{T}}$. For `FunList`, besides ID and BOT there is a projection

$$\mu\gamma.Nil\ () \sqcap\ Cons\ (BOT,\ \gamma)$$

in $FinProj_{\mathtt{FunList}}$, which we will abbreviate $SPINE$, that leaves the spine of its list argument intact, but maps each element of the list to $\perp$.

The definition of $FinFor_{\mathtt{T}}$ is just like $\mathcal{F}^P[\![\mathtt{T}]\!]$ except at recursive types. A data structure of recursive type `A=T(A)` may be thought of as some (possibly infinite) number of elements of `T(())`. For example, writing nil for in_1 and $cons$ for in_2, the value $cons\ (f,\ cons\ (g,\ nil\ ()))$ in the standard domain for `FunList` decomposes into $cons\ (f,\ ())$, $cons\ (g,\ ())$, and $nil\ ()$. Our (implicit) abstraction function maps such a data structure to the greatest lower bound of these elements—this is safe in the context of projection abstraction. Thus $FinFor_{\mathtt{A=T(A)}} = FinFor_{\mathtt{T(())}}$. Then $FinFor_{\mathtt{FunList}}$ is

$$\begin{aligned} \mathbf{1} \times ((\quad &((FinProj_{\mathtt{E}} \to FinProj_{\mathtt{Int}}) \times \mathbf{1}) \\ &\to ((FinProj_{\mathtt{E}} \to FinProj_{\mathtt{Int}}) \times \mathbf{1})) \times \mathbf{1})\ . \end{aligned}$$

The abstract P constants are defined as follows.

$$mkint^P_n \;=\; (\lambda\alpha.ID,\ ())\ ,$$

$$plus^P\ (\tau_1,\ f_1)\ (\tau_2,\ f_2) \;=\; (\lambda\alpha.(\tau_1\ \alpha) \sqcap (\tau_2\ \alpha),\ ())$$

$$mkfun^P\ f \;=\; (\lambda\alpha.ID,\ f)$$

$$apply^P\ (\tau,\ \kappa)\ x \;=\; \kappa\ x$$

$$fix^P \;=\; gfp$$

$$\begin{aligned} &tuple^P\ (\tau_1, \kappa_1)\ \ldots\ (\tau_n, \kappa_n) \\ &\quad = (\lambda\alpha.(\tau_1\ \alpha,\ \ldots,\ \tau_n\ \alpha),\ (\kappa_1, \ldots, \kappa_n)) \end{aligned}$$

$$sel^P_i\ (\tau,\ \kappa) \;=\; (PI_i \circ \tau,\ \pi_i\ \kappa)$$

$$inc^P_i\ (\tau,\ \kappa) \;=\; (C_i \circ \tau,\ (\top, \ldots, \top, \kappa, \top, \ldots, \top))$$

$$outc^P_i\ (\tau,\ \kappa) \;=\; (OUTC_i \circ \tau,\ \pi_i\ \kappa)$$

Here gfp denotes *greatest* fixed point, and $\top$ the top element of a lattice. In fact, any fixed point and any value in place of each $\top$ is safe, but the greatest such values give the most informative results.

Like $choose^D$ we must make $choose^P$ an explicit function of the result type:

$$
\begin{aligned}
& choose^P_{\mathtt{T}}\ \rho\ (\tau_0, \kappa_0)\ \ldots\ (\tau_m, \kappa_m) \\
& \quad = (\lambda\alpha.case\ (\tau_0\ \alpha)\ of \\
& \qquad\qquad C_1\ \gamma_1\ \sqcap \cdots \sqcap C_n\ \gamma_n\ \rightarrow \\
& \qquad\qquad\quad (\sqcap_i\ (\tau_i\ \alpha),\ forpart_{\mathtt{T}}\ \kappa_1\ \ldots\ \kappa_m)\ .
\end{aligned}
$$

The local function *forpart* is defined exactly the same as in $choose^D$, with superscript P replacing superscript D.

Finally, we give examples of analysis, in which we employ the following pun: for ID and BOT in $|\,\mathcal{D}[\![\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2]\!]\,| = |\,\mathbf{1}_\perp|$, the corresponding projections BOT' and ID' in $|\,\mathcal{T}^S[\![\mathtt{T}_1\ \texttt{->}\ \mathtt{T}_2]\!]\,|$ are just BOT and ID, respectively. By similarly overloading the definitions of the C_i we may dispense with the superscript $'$ altogether. Let expression `e` denote the length function for lists of integers:

```
fix (\length.\xs.
  case xs of
    nil x -> 0
    cons ys -> case ys in
                 (z,zs) -> 1 + length zs) .
```

Since `e` has no free variables the abstract environment may be taken to be empty. Then $\mathcal{E}^P[\![\mathtt{e}]\!][]$ is (τ, κ), where $\tau = \lambda\alpha.ID$, indicating that the value of the expression is always static, and

$$
\begin{aligned}
\kappa = \lambda(\tau,\kappa).(\ \lambda\alpha.\quad & case\ \tau\ \alpha\ of \\
& \quad BOT \rightarrow BOT \\
& \quad SPINE \rightarrow ID \\
& \quad ID \rightarrow ID, \\
()\ ) \ .
\end{aligned}
$$

To get results in the form of Launchbury's at first order, such abstract values must be (abstractly) applied to the identity projection abstraction and the trivial forward part, that is $(\lambda\alpha.\alpha, ())$ (space does not permit a full explanation of this). The result is just

$$
\begin{aligned}
(\lambda\alpha.\quad & case\ \alpha\ of \\
& \quad BOT \rightarrow BOT \\
& \quad SPINE \rightarrow ID \\
& \quad ID \rightarrow ID, \\
())\ .
\end{aligned}
$$

This tells us that so long as the spine of the argument list is static, then so is the result. For `xs:FunList`, that is, lists of functions the abstract argument would be $(\lambda\alpha.\alpha, \kappa)$, where κ safely approximates the abstract value of every function in the list. In this example there is no dependency on the values of the list element and therefore no dependency on κ; the result would be the same for any κ. More generally, in the absence of any information about the elements of the list the value $\perp$ is always safe.

Now for a more complicated example. Let expression `e` be

```
(fix (\compose.\fs.\x.
  case fs of
    Nil z -> x
    Cons p ->
      case p in
        (g,gs) -> g (compose gs x))) hs y .
```

Here `hs:FunList` and `y:Int` are free variables; the value of the expression is the composition of the functions in `hs` applied to `y`. We take the type `E` of the environment to be `(FunList,Int)`, the first component corresponding to `hs` and the second to `y`. We consider three possible values for the standard environment. First, suppose that we know nothing about the functions in `hs`; we may only safely assume that they return dynamic results for all arguments. Then the appropriate value for the abstract environment ρ^A is

$$
\begin{aligned}
& (\lambda(\alpha_{hs}, \alpha_y).(\alpha_{hs}, \alpha_y), \\
& \ ((), (\lambda(\tau,\kappa).(\lambda\alpha.BOT, \perp), ())), ())\ .
\end{aligned}
$$

Then $\mathcal{E}^A[\![\mathtt{e}]\!]\ \rho^A = (\lambda(\alpha_{hs}, \alpha_y).BOT,\ ())$. This tells us that for any projection (α_{hs}, α_y) on the standard environment the result is dynamic. Now suppose that each function in `hs` maps static arguments to static results. Then the abstact environment could be

$$
(\lambda(\alpha_{hs}, \alpha_y).(\alpha_{hs}, \alpha_y),\ ((), (\lambda(\tau,\kappa).(\tau,\kappa), ())), ())\ ,
$$

which gives abstract result $(\tau,\ ())$, where

$$
\begin{aligned}
\tau\ (ID,\ ID) &= ID\ , \\
\tau\ (SPINE,\ ID) &= BOT\ , \\
\tau\ (BOT,\ ID) &= BOT\ , \\
\tau\ (\alpha,\ BOT) &= BOT\ .
\end{aligned}
$$

In other words, the result is static only if the environment is entirely static. Finally, if every function in `hs` always produces a static result, that is, is a constant function, then an appropriate abstract environment is

$$
(\lambda(\alpha_{hs}, \alpha_y).(\alpha_{hs}, \alpha_y),\ ((), (\lambda(\tau,\kappa).(\lambda\alpha.ID, \kappa), ())), ())
$$

which gives abstract result $(\tau,\ ())$, where for all projections β on `y`,

$$
\begin{aligned}
\tau\ (ID,\ \beta) &= ID\ , \\
\tau\ (SPINE,\ \beta) &= BOT\ , \\
\tau\ (BOT,\ \beta) &= BOT\ .
\end{aligned}
$$

That is, for entirely static `hs` the result is static.

5 Backward Strictness Analysis

Backward strictness analysis differs essentially from binding-time analysis only in that the direction of information flow is reversed: given f and δ we seek to find γ such that $\gamma \circ f \sqsubseteq f \circ \delta$. (In practice it is $f_\perp$ rather than f that is analysed, but this does not change the nature of the development.) Projection-based backward strictness analysis can detect such properties as *head strictness*, which is known [Kam92] to be undetectable in the BHA framework for abstract interpretation [BHA86].

Let a mapping τ of projections to projections such that for all γ we have $\gamma \circ f \sqsubseteq f \circ (\tau\ \gamma)$ be called a *backward projection abstraction* of f. Here smaller τ are more informative. The higher-order backward strictness semantics is obtained from the D semantics just as is the binding-time semantics except that (greatest) projection abstractions are replaced by (least) backward projection abstractions, *gfp* and $\top$ by *lfp* and $\perp$, respectively, again since lesser values are more informative, and composition of projection abstractions replaced by reverse composition of backward projection abstractions. The result is the analysis described in [DW91].

6 Related Work

Many approaches to binding-time analysis have been proposed and implemented; we will discuss only those most closely related to ours.

As stated, our analyses are a generalisation to higher order of Launchbury's monomorphic binding-time analysis and Wadler and Hughes' strictness analysis; our analyses reduce to theirs when restricted to first order (in fact, our strictness analysis is stronger, but it is essentially the same).

Hunt [Hun91] presented a strictness analysis technique, and with Sands [HS91] a binding-time analysis technique, both using *partial equivalence relations* (PERs) as the basic abstract values. The relation between these two techniques is roughly analogous to the relation between ours. Hunt showed that projection-based strictness analysis is an instance of PER-based analysis, and the same appears to hold for binding-time analysis. The PER-based analysis techniques have the advantage of extending smoothly to higher order, whereas the projection-based approaches required some work to so generalise. Of more practical interest is the nature of the finite abstract domains that arise naturally for each technique. In this respect the approaches appear to be closely comparable. For example, at base types such as `Int` the abstract PERs are $\{\mathrm{S}, \mathrm{D}\}$ with $\mathrm{S} \sqsubset \mathrm{D}$, where S denotes static (corresponding to ID) and D dynamic (corresponding to BOT—note the ordering is reversed). For function types the abstract PERs are defined by simple induction, for example for `Int->Int` the values are the monotonic maps from $\{\mathrm{S}, \mathrm{D}\}$ to $\{\mathrm{S}, \mathrm{D}\}$ ordered pointwise, giving $\lambda x.\mathrm{S}$, $\lambda x.x$, and $\lambda x.\mathrm{D}$ (compare $\lambda \alpha.BOT$, $\lambda \alpha.\alpha$, and $\lambda \alpha.ID$). For `IntList` the abstract PERs are D (BOT), $\mathrm{SPINE}(\mathrm{D})$ ($SPINE$) and $\mathrm{SPINE}(\mathrm{S})$ (ID); again there is a 1-1 correspondence. A notable difference in the methods is our use of the function-space lifting of the lazy lambda calculus, but there seems to be no reason that the PER approach cannot be similarly extended: for `Int->Int` the relevant PERs are $\mathrm{LIFT}(\lambda x.\mathrm{S})$, $\mathrm{LIFT}(\lambda x.x)$, and $\mathrm{LIFT}(\lambda x.\mathrm{D})$, and a new top element D (notice lifting corresponds to abstract 'topping'–adding a new top element), and abstract application becomes

$$\begin{aligned} apply^{\#}\ \mathrm{D}\ x &= \mathrm{D}\ , \\ apply^{\#}\ \mathrm{LIFT}(f)\ x &= f\ x\ . \end{aligned}$$

We have not yet made a careful comparison of the two methods, either from a theoretical or practical standpoint, but such a comparison should be possible since Hunt gave a precise relationship between the standard and abstract semantics (as we did).

Mogensen [Mog89] describes his technique as a higher-order generalisation of Launchbury's polymorphic binding-time analysis. Higher-order functions are represented by *abstract closures*—symbolic representations of functions which are manipulated algebraically. Approximation of recursively-defined abstract closures is performed 'on-the-fly' according to time and space considerations. The nature of these approximations is strongly dependent on the syntax of the corresponding function definitions, so non-standard values are not functions of standard values, making precise comparison with our method difficult. Unlike our approach, the abstract values of higher-order functions are their projection abstractions, where projections on functions are operations that map (parts of) abstract closures to $\perp$.

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. In this respect there are many superficial similarities to our analysis. No formal relation to the standard semantics is given, making a formal comparison with our analysis problematic.

7 Conclusion

We have successfully generalised Launchbury's monomorphic projection-based binding-time analysis to higher-order. Because the abstract domains are comparable in size to those of BHA strictness analysis [BHA86], the time complexity (as a function of type structure) of an implementation is likely to be problematic, though promising advances are begin made in this area, most notably using *frontiers* [Hun89] and *concrete data structures* [Fer92] to represent abstract functions only at the values at which they are actually required.

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 the analogous backward strictness analysis [HL92], though the results are slightly weaker than for monomorphic analysis. For BHA-style strictness analysis, an approach to handling polymorphism has been given by Abramsky [Abr85], though the results are rather weak. It might be possible to combine these approaches: since we only require projections on data we could use the corresponding polymorphic projections for the projection abstractions, and Abramsky's approach to polymorphism at higher-order, that is, the forward components. However, the result might be quite weak, for the same reasons as is Abramsky's.

References

[Abr85] Abramsky, S. "Strictness analysis and polymorphic invariance." In *Proceedings of the Workshop on Programs a Data Objects* (Copenhagen). H. Ganzinger and N. Jones, eds. LNCS 217. Springer-Verlag, Berlin, 1985.

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

[BHA86] Burn, G., Hankin, C., and Abramsky, S. "The theory of strictness analysis for higher-order functions." In *Proceedings of the Workshop on Programs as Data Objects* (Copenhagen). H. Ganzinger and N. Jones, eds. LNCS 217. Springer-Verlag, Berlin, 1986.

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

[Dav92] Davis, K. "A Note on the Choice of Domains for Projection-Based Program Analysis." In *Functional Programming: Proceedings of the 1991 Glasgow Workshop, 13-15 August 1991, Isle of Skye, Scotland.* P. Wadler *et al.*, eds. Springer Workshops in Computing. Springer-Verlag, 1992.

[DW90] Davis, K. and Wadler, P. "Strictness analysis: Proved and improved." In *Functional Programming, Glasgow 1989: Proceedings of the 1989 Glasgow Workshop on Functional Programming, 21-23 August 1989, Fraserburgh, Scotland.* K. Davis and J. Hughes, eds. Springer Workshops in Computing. Springer-Verlag, 1990.

[DW91] Davis, K. and Wadler, P. "Strictness analysis in 4D." In *Functional Programming, Glasgow 1990: Proceedings of the 1990 Glasgow Workshop on Functional Programming, 13-15 August 1990, Ullapool, Scotland.* Simon L. Peyton Jones *et al.*, eds. Springer Workshops in Computing. Springer-Verlag, 1991.

[Fer92] Ferguson, A. "Concrete Data Structures" In *Functional Programming: Proceedings of the 1992 Glasgow Workshop, 6-8 July 1992, Ayr, Scotland.* Springer Workshops in Computing. Springer-Verlag (to appear).

[HL92] Hughes, R.J.M. and Launchbury, J. *Projections for polymorphic first-order strictness analysis.* Math. Struct. in Comp. Science, vol. 2, pp. 301-326, CUP, 1992.

[Hun89] Hunt, S. "Frontiers and open sets in abstract interpretation." *Functional Programming and Computer Architecture.* (Imperial College, London, September 1989.) ACM, Addison-Wesley Publishing, Reading, MA, U.S.A. 1989.

[Hun91] Hunt, S. "PERs Generalise Projections for Strictness Analysis." *Functional Programming, Glasgow 1990: Proceedings of the 1990 Glasgow Workshop on Functional Programming, 13-15 August 1990, Ullapool, Scotland.* Simon L. Peyton Jones *et al.*, eds. Springer Workshops in Computing. Springer-Verlag, 1991.

[HS91] Hunt, S., and Sands, D. "Binding time analysis: a new PERspective." *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '91)*, SIGPLAN NOTICES Vol. 26, No. 9. ACM Press 1991.

[Kam92] Kamin, S. *Head strictness is not monotonic abstract property.* To appear in *Information Processing Letters.*

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

[Lau91b] Launchbury, J. *Strictness and Binding-time Analyses: Two for the Price of One*, Proc. SIGPLAN Programming Language Design and Implementation, Toronto, 1991.

[Lau91c] Launchbury, J. *A Strongly-typed, Self-applicable Partial Evaluator*, ACM Functional Programming and Computer Architecture, Boston, 1991.

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

[WH87] Wadler, P., and Hughes, J. "Projections for Strictness Analysis." In *Proceedings of Functional Programming Languages and Computer Architecture* (Portland, Oregon). G. Kahn, ed. LNCS 274. Springer-Verlag, Berlin, 1987.

[Wad90] Wadler, P. "Comprehending Monads." In *Proceedings of the ACM Conference on Lisp and Functional Programming* (Nice, June 1990).