A Semantic Model of Binding Times for Safe Partial Evaluation[*]

Fritz Henglein David Sands
University of Copenhagen[†]

June 30, 1995

Abstract

In program optimisation an analysis determines some information about a portion of a program, which is then used to justify certain transformations on the code. The correctness of the optimisation can be argued *monolithically* by considering the behaviour of the optimiser and a particular analysis in conjunction. Alternatively, correctness can be established by finding an interface, a *semantic property*, between the analysis and the transformation. The semantic property provides modularity by giving a specification for a systematic construction of the analysis, and the program transformations are justified via the semantic properties.

This paper considers the problem of partial evaluation. The safety of a partial evaluator ("it does not go wrong") has previously been argued in the monolithic style by considering the behaviour of a particular binding-time analysis and program specialiser in conjunction. In this paper we pursue the alternative approach of justifying the binding-time properties semantically. While several semantic models have been proposed for binding times, we are not aware of any application of these models in proving the safety of a partial evaluator. In this paper we:

- identify problems of existing models of binding-time properties based on projections and partial equivalence relations (PERs), which imply that they are not adequate to prove the safety of simple off-line partial evaluators;
- propose a new model for binding times that avoids the potential pitfalls of projections/PERs;
- specify binding-time annotations justified by a "collecting" semantics, and clarify the connection between extensional properties (local analysis) and program annotations (global analysis) necessary to support binding-time analysis;
- prove the safety of a simple but liberal class of monovariant partial evaluators for a higher-order functional language with recursive types, based on annotations justified by the model.

[*]To appear in Proc. Seventh International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP), Utrecht, The Netherlands, September 20-22, 1995, Springer-Verlag Lecture Notes in Computer Science (LNCS)

[†]DIKU, Universitetsparken 1, 2100 København Ø; {henglein,dave}@diku.dk

1 Introduction

1.1 Transformations supported by Program Analysis

Program optimisation usually takes the following form: an analysis determines some information about a portion of a program, and the information is then used to justify certain transformations on the code. We consider two basic methods for establishing the correctness of such a process, which we call *monolithic* and *model-based*, respectively:

Monolithic The monolithic view considers the correctness of the analysis and the transformation simultaneously. The pair of the analysis and the transformation is correct if the transformation "works."

Model-based The model-based approach associates some semantic property with the information domain of the analysis. The correctness of the analysis, and the correctness of the transformation are then considered independently, but relative to this semantic property.

The monolithic approach has attracted much interest in the last few years. Its advocates, *e.g.* Wand [Wan93], Amtoft [Amt93], and Steckler [Ste94], argue that considering the correctness of the algorithm and transformation together leads to a much simpler proof. The slogan is:

The analysis is correct because the transformation works!

It is notable that these kinds of proofs are greatly aided by the non-algorithmic specification of the analysis in terms of non-standard type systems, or constraint systems. The disadvantages of this approach are that: as the name suggests, variations in either the analysis or the transformation require that the proof must be re-established for each change or combination of analysers and transformers; there is currently no support for systematic design of correct analyses; similar analysis may be used in justifying quite different kinds of transformation, but there are no "reusable" components in the correctness proof.

In principle, the model-based approach addresses each of these deficiencies. By associating a semantic property with each piece of information from a static analysis, one obtains an intermediary between the analysis and the transformation. This, in turn, achieves a factorisation of correctness of the analysis and the transformation with respect to the semantic property. This means that independent changes to either the analysis or transformation can be justified independently. Furthermore, it enables utilisation of techniques for systematic design of correct analyses, namely *abstract interpretation* [CC79]. Finally, it facilitates reuse of analyses for different transformations which rely on a common semantic property.

The problem in practice is, to quote Wand [Wan93]:

> "While program analyses of various sorts have been studied intensively for many years, it has proven remarkably difficult to specify the correctness of an analysis in a way that actually justifies the resulting transformation."

In this paper we address this problem for a particular transformation, *off-line partial evaluation*, in the setting of higher-order functional programs. The associated analysis is called *binding-time analysis*, and the core of the correctness problem is to verify that a partial evaluator does not "go wrong" when following binding-time annotations.

While there are numerous proofs of correctness using the monolithic approach, and several candidate semantic models for binding-time properties, we know of no correctness proof for a partial evaluator based on a semantic model of binding-time properties. In this paper we:[1]

[1]Due to space limitations most proofs have been omitted. They are contained in a full report obtainable from the TOPPS (DIKU programming language group) archive at `http://www.diku.dk/research-groups/topps`.

- identify problems of existing models of binding-time properties based on projections and partial equivalence relations (PERs), which imply that they are not adequate to prove the correctness of even simple off-line partial evaluators;
- propose a new model for binding times which avoids the potential pitfalls of projections/PERs;
- clarify the connection between extensional properties (local analysis) and program annotations (global analysis) necessary to support binding-time analysis;
- prove the correctness of a simple but liberal class of partial evaluators based on the soundness of the annotations with respect to the model, and demonstrate the applicability to both off-line and on-line partial evaluation.

Model-based analysis is the hallmark of abstract interpretation. Transformations often require an abstraction not directly of the standard semantics of a language, but of its *collecting semantics*, however. Expressing the collecting semantics in denotational models has proved to be difficult. Cousot and Cousot show how powersets support this step at higher type and apply it to what they call *comportment analysis* [CC94].

2 Off-line Partial Evaluation: Related Work

An off-line partial evaluator determines which parts of a program to evaluate, and which parts to leave as residual code, by following annotations produced by a binding-time analysis.

Given a description of the parameters in a program that will be known at partial evaluation time, a binding-time analysis must determine which parts of the program are dependent solely on these known parts (and therefore also known at partial evaluation time). A binding-time analysis performed prior to the partial evaluation process can have several practical benefits (see [Jon88]), and plays an essential rôle in most approaches to the generation of efficient compilers from interpreters [BJMS88].

2.1 Approaches to Correctness

Monolithic The monolithic view for partial evaluation considers the correctness of the off-line partial evaluator and the binding-time analysis simultaneously. The annotations produced by the binding-time analysis are considered to be correct if the partial evaluator, whose actions are governed by the annotations, behaves in the intended manner, e.g. it does not "go wrong" by expecting to be able to produce a value from a program fragment dependent on an unbound variable. Examples of this approach are seen in the work of Gomard [Gom92], based on a denotationally-specified partial evaluator for a lambda calculus with constants ("λ-mix"), Wand [Wan93] and Palsberg [Pal93], based on the pure lambda calculus, Henglein and Mossin [HM94] for a typed functional language and a denotationally specified partial evaluator, Consel et al. [CJØ94] for a rewriting-based approach, and more recently [Hat95] who considers the mechanical verification of the correctness proof for a λ-mix style partial evaluator.

Model-based The model-based approach has its roots in Jones' definition of *congruence* [Jon88], which specifies correctness of binding-time analysis by focusing on semantic dependency between different parts of a program. Launchbury adapted this idea to a functional setting, using the idea of domain projections to model binding times of structured data in a first-order language [Lau88, Lau89]. This domain-based approach was subsequently adopted and extended by Mogensen [Mog89] and De Niel *et al.* [NBV91]. Hunt and Sands [HS91] showed that Launchbury's analysis could be smoothly extended to higher types using partial equivalence relations (PERs) as a model of binding times, following Hunt [Hun90]. Davis [Dav94] considers a closely related extension to higher types with general recursive types.

There is a third approach to proving correctness, closely related to the model-based view, but arguably different: an off-line partial evaluator is viewed as an abstraction of an *on-line* one. An on-line partial evaluation does not follow static binding-time annotations, but computes the necessary information on the fly. Off-line partial evaluation is then viewed as a *restriction* of the actions of the on-line version, since it makes decisions about what to partially evaluate based on annotations, rather than actual data. (As we mention later, it might also be considered an *optimisation*, since it removes the need for many tests on the nature of the data manipulated.) This approach has been considered by Consel and Khoo [CK92] and Bulyonkov [Bul93]. Consel and Khoo give an abstract denotational specification of the values encountered by an on-line specialiser for a first-order functional language, and show that a binding-time analysis correctly abstracts these values. Off-line partial evaluation is then obtained by the restriction of the actions of the on-line version. Their highly abstract and non-operational specification of an on-line specialiser resembles a collecting interpretation (a *static* semantics in the terminology of [CC79]).

3 The Problem with Projections and PERs

In this section we consider the existing proposals for modelling binding times, including partially-static structures, in functional languages. The principal technique uses domain projections [Lau88]. We will argue that this model has potential flaws from the point of view of proving the correctness of a partial evaluator. These problems carry over to the PER model [HS91], and motivate the introduction of a new model in the next section.

3.1 Uniform Congruence

In his "re-examination" paper, Jones [Jon88] defines a semantic-based condition, *congruence*, which specifies when a binding-time analysis is correct. The essence of the definition is that the parts of a program that are deemed to be static will only ever depend on the static values (and the other parts of the program which are deemed static). Launchbury adapted this idea to a functional setting, and derived what he considered to be a necessarily stronger condition, called *uniform congruence*, and expressed this condition with the help of domain projections.

In Launchbury's setting, a first-order language of recursion equations, the job of a binding-time analysis is to determine a *program division*. A division Δ is a mapping which assigns a binding time to each function symbol defined in the program. (It is therefore *monovariant* because it assigns just one binding time for each definition.)

A binding-time is identified with (modelled by) a domain projection. A projection is a continuous map $\alpha : D \to D$ on a cpo D, such that $\alpha \sqsubseteq \lambda x \in D.\, x$ and $\alpha \circ \alpha = \alpha$. An intuition behind the use of projections to describe binding times is that the parts of its argument that a projection discards (replaces by bottom) represent the parts about which no information is known, where "no information" is equated with "dynamic." This interpretation is used to define when a program division is safe, *i.e.* uniformly congruent.

A program division Δ is deemed to be uniformly congruent, if for each definition and call instance of the form

$$f\ x = \ldots (g\ e) \ldots$$

which occurs in the program, then

$$\Delta(g) \circ [\![\lambda v.e[v/x]]\!] \circ \Delta(f) = [\![\lambda v.e[v/x]]\!] \circ \Delta(f)$$

This means that if $\Delta(f)$ describes the static-ness of the argument to f, then $\Delta(g)$ correctly predicts the static-ness of the argument e in the call $g\ e$.

In Hunt and Sands' terminology [HS91], in which binding times $\Delta(f)$ and $\Delta(g)$ are interpreted as equivalence relations (and at higher-types, as *partial* equivalence relations), this property would be written equivalently as

$$(\lambda v.e[v/x]) : \Delta(f) \Rrightarrow \Delta(g)$$

which by definition means that for all v_1, v_2 in the semantic domain associated with parameter x,

$$v_1\ \Delta(f)\ v_2 \Rightarrow [\![\lambda v.e[v/x]]\!]v_1\ \Delta(g)\ [\![\lambda v.e[v/x]]\!]v_2.$$

The "uniformity" in Launchbury's definition refers to the fact that no contextual assumptions are made about the possible value of x in the expression e. This reflects a simple but aggressive view of partial evaluation, which assumes that we can begin specialising the call to $(g\ e)$ without using knowledge of either the context "..." or the range of possible values of x in that context. This uniformity requirement is a strengthening of Jones' condition, and so fewer program divisions are permitted.

3.2 The Problem with Uniform Congruence

Launchbury's specialiser (for present purposes, a specialiser is just a partial evaluator which produces specialised variants of program text) specialises function calls with respect to the static parts of their arguments.

Clearly then, from the point of view of the specialiser the binding-time analysis is correct if the parts of the arguments that are deemed static can indeed be evaluated, and their evaluation either terminates with a value, or the specialiser goes into a loop in the attempt—in any case it must not get "stuck" trying to evaluate something dynamic, such as a free variable. (A consequence of the fact that Launchbury's language is statically typed is that there are no run-time errors in the standard interpreter.)

The job of the semantic specification of uniform congruence is to give an analysis-independent specification of a correct program division. This can be used to justify binding-time analyses independently of a specific partial evaluator (just as Launchbury has done). But for this to be adequate, one must be able to argue the correctness of a partial evaluator with respect to a uniformly congruent program division (something Launchbury did not do).

We argue that the semantic condition of uniform congruence is not sufficient to guarantee the correctness of a simple "mix-style" partial evaluator. That is to say, there are uniformly congruent program divisions which can cause a specialiser to "go wrong." What is more, we claim that Launchbury's *own* specialiser will go wrong on an instance of this example.

Consider the following (abstract) program:

```
letrec
  g(v, w)   =   e
  f(x, y)   =   g(if y then Ω else Ω′, y)
in f(i, j)
```

where g is non-recursive, Ω and Ω' are any expressions not involving y, but which diverge (fail to terminate) for all values of x.

Now suppose we specify that i is static and j is dynamic. This property of the pair (i, j) can be represented by a projection $\mathit{fst} \overset{\text{def}}{=} \lambda\langle x, y\rangle.\langle x, \bot\rangle$. Based on this specification, the division

$$[f \mapsto \mathit{fst}, g \mapsto \mathit{fst}]$$

is uniformly congruent. To see intuitively why, first note that since Ω and Ω' do not contain y, and under the assumption that x is static any sub-expressions of Ω and Ω' are also static. The surprise is that g's first argument can be considered static. Intuitively, this is correct because the value of its only argument (the call instance in f's definition) does not depend on the value of y — since it is always undefined ($\perp$)! The potential problem with this uniformly congruent division is readily apparent. The term $\texttt{if}\, y\, \texttt{then}\, \Omega\, \texttt{else}\, \Omega'$ is deemed to be static even though y is dynamic. This means that a partial evaluator may begin to evaluate the conditional, and thereby "go wrong" by either:

1. attempting to evaluate y, or

2. by expecting that the expression $\texttt{if}\, y\, \texttt{then}\, \Omega\, \texttt{else}\, \Omega'$ can be compared with other static "values", for example in a pending list of specialised function calls.

For Launchbury's simple partial evaluator it is the latter case. We can realise an instance of this scheme in Launchbury's PEL language and show that his specialiser "goes wrong" when given the above safe (= uniformly congruent) program division.

In the binding-time model we present in the next section domains contain "extra" elements that force $\texttt{if}\, y\, \texttt{then}\, \Omega\, \texttt{else}\, \Omega'$ to be dynamic. More concretely, y may not only be bound to $\perp$, *true* or *false*, but also to δ, an "anonymous" dynamic value. Whereas the result of the first three bindings is $\perp$, in the latter case it is the special value δ. Thus, in the extended domain with δ, $\texttt{if}\, y\, \texttt{then}\, \Omega\, \texttt{else}\, \Omega'$ does depend on y; in particular, if y is dynamic then so is the whole expression.[2]

Note that we *do not* claim that Launchbury's *analysis* will produce this program division (it will not). The point is that the safety condition which specifies a correct analysis must be adequate to prove the correctness of the transformation. We conclude that this is not the case for Launchbury's projection-based safety condition. The problem is inherited by Hunt and Sands' PER-based extension to higher-order functions — in fact we came across this problem in a higher-order setting, but a superficially quite different context: attempting to prove the correctness of a λ-mix style partial evaluator [Gom92] using the PER-model of binding times.

4 An Ideal Model of Binding Times

An appealing property of the projection/PER model of binding times is that it is purely extensional, relying as it does on the standard semantics.[3] As we showed in the previous section, using "bottom" to represent absence of information at partial-evaluation time confuses termination properties with neededness properties. Confusion arises because the property "static" does not necessarily mean "terminating."[4] So bottom is overloaded to denote static computation (though nonterminating) and static nonavailability of dynamic data, making it impossible to distinguish nonterminating computations that depend on dynamic inputs from those that don't.

[2]Our extended interpretation of *bool* contains yet another value, a top element $\top$; $\top$, however, is only required for a *higher-order* language. Note that the problem with Launchbury's model of binding times already occurs for a first-order language. Addressing this problem in the same language context only requires the additional element δ, not $\top$.

[3]However, in order to model anything interesting about binding times the model for the language under consideration *must* be lazy. Furthermore, we claim that for this purpose the laziness must be taken to its logical conclusion — i.e. function spaces should be lifted (contrary to the model in [HS91]) as well as tuples (contrary to [Lau89][HS91][Dav94]).

[4]In principal we have no objection to an interpretation of "static" which implies "terminating"; but this is neither the interpretation used in practice in existing partial evaluators, nor the interpretation used in this paper.

Our solution is a natural one, once we accept that we are modelling properties that only have meaning at partial evaluation time. To be able to make finer distinctions than in the standard semantics our solution is to augment the domain constructors in the standard semantics to provide extra elements: δ and $\top$. Intuitively, δ stands for an "anonymous dynamic value", and $\top$ denotes an abortive error; that is, the result of encountering an error situation that leads to the abort of the whole program evaluation.

Partially static structures are handled by allowing data structures to contain δ-elements at component types without them being identified with δ (or $\top$) of the compound type. This model enlarges the domains, and so gives rise to a choice as to how the operators of the language should be extended. The choices reflect choices in the partial evaluation strategy — but at a much more abstract level than if we were to describe a particular partial evaluator.

In the remainder of this section we describe the language and the model of binding times and associated binding-time annotations.

4.1 A Higher-order Functional Programming Language

Our setting is a higher-order simply typed programming programming language with unit, sum, pair and recursive types.

Types The *types* are described by the closed expressions in the following grammar:

$$\tau ::= \alpha \mid \mathrm{unit} \mid \tau' \to \tau'' \mid \tau' \times \tau'' \mid \tau' + \tau'' \mid \mathit{rec}\,\alpha.\ \tau'$$

Recursive types must be *formally contractive*; that is, in $\mathit{rec}\,\alpha.\ \tau$ the type τ must not be a type variable.

Syntax and typing rules for expressions The syntax of our language, including its "static" semantics, is given by rather standard typing rules, not presented here. They are for typing judgements $A \vdash e : \tau$ where A is a *type environment* mapping *program variables* to types, e is an *expression*, and τ is a type.

Standard denotational semantics To give a standard semantics for this language we can interpret types by Scott domains and type constructors by domain constructors appropriate for a call-by-name (lazy) language [Gun92].

Specifically, we can interpret unit by the one-element domain 1; $\to$ by the domain constructor for continuous functions; $\times$ by lifting the result of the *Cartesian product* constructor $(\cdot \times \cdot)_\perp$; $+$ by the *separated sum* constructor; and $\mathit{rec}\,\alpha.\ \tau$ by the inverse-limit of the domain constructor denoted by τ (as a function of α).

Expressions are denoted by domain elements. If $\vdash e : \tau$ for closed expression e then $[\![e]\!]_{std} \in [\![\tau]\!]_{std}$, where $[\![e]\!]_{std}$ is the domain element denoted by e. This yields a denotational semantics that is *observationally adequate* for a call-by-name operational semantics relative to observing termination at all nontrivial first-order types (which excludes unit).

4.2 Extended Domains

Previous models of binding times have built upon the standard denotational semantics either by interpreting binding times as projections or PERs on the *standard* domains. As shown in Section 3 this leads to problems in that these models may be too aggressive about what they classify as static. The reason is that the standard domains do not have any "room" for intensional information that captures the essential control dependencies — neededness information — that a (simple) partial evaluator must respect. This problem is even more pronounced in a call-by-value language, where the PER and projection analyses become trivial (useless).

In this section we *extend* the standard domains by adding *extra elements* in a structural fashion: For every type τ possessing a destructor operation we add a *dynamic element* δ_τ, which, intuitively, represents *completely dynamic* values at τ. This new value lets us distinguish a dynamic value of type $\tau \times \tau'$, say a variable, from a *pair* of dynamic values. In the latter case we can perform a static (partial-evaluation time) decomposition of the pair, whereas in the former we cannot. It is the ability to distinguish between being able to perform a destructive operation (in this case π_1 or π_2) at partial evaluation time that necessitates and explains the role of the dynamic element.

Furthermore, for every type τ we add a *top element* $\top_\tau$, which represents an error at type τ. We call the resulting domains *topped domains (with δ)*, in order to distinguish them from the *standard domains* introduced earlier. The elements of the standard domain are then embedded "naturally" in the topped domains; in particular functions are extended to map the new elements δ and $\top$ to $\top$.

Structurally topped domain constructions Our binding time domains are Scott-domains with isolated δ- and $\top$-elements. In what follows we will define the topped interpretation for types and terms. We will write $[\![\cdot]\!]$ to denote these mappings and use the following domain constructions on topped domains D, D'. Let $\top_D, \top'_D$ be the top elements in D and D', respectively.

- The *co-strict function domain* $D \hookrightarrow D'$ consists of the continuous functions from D to D' mapping $\top_D$ to $\top_{D'}$, plus a new top element $\top_{D \hookrightarrow D'}$. The partial ordering on non-$\top$ elements is inherited point-wise from D'.

- The *co-strict product domain* $D \otimes D'$ consists of the pairs (d, d') with $\top_D \neq d \in D$ and $\top_{D'} \neq d' \in D'$, extended with new bottom and top elements $\bot_{D \otimes D'}$ and $\top_{D \otimes D'}$, respectively. The partial ordering on pairs is inherited componentwise from D and D'.

- The *co-strict sum domain* $D \oplus D'$ consists of the elements $\mathit{inl}\, d$ and $\mathit{inr}\, d'$ for all $\top_D \neq d \in D$ and $\top_{D'} \neq d' \in D'$, extended with new bottom and top elements $\bot_{D \oplus D'}$ and $\top_{D \oplus D'}$, respectively. The partial ordering on elements $\mathit{inl}\, d$ is inherited from D, and on elements $\mathit{inr}\, d'$ from D'; elements $\mathit{inl}\, d$ and $\mathit{inr}\, d'$ are incomparable.

- The *topped domain* $D^\top$ consists of all the elements from D, plus a new top element, the partial ordering on non-top elements being the same as in D.

- The *dynamic domain* D^δ_S is D, extended with an additional element δ. S must be a set of pairwise incomparable elements from D. The partial order relation on non-δ elements is inherited from D. It is extended to δ as follows: δ is "immediately below" $\top_D$; that is, $\delta \sqsubset \top_D$, and $\top_D$ is the only element greater than δ. Furthermore, the elements of S are immediately below δ. That is, $d \sqsubset \delta$ if and only if $d \sqsubseteq s$ for some $s \in S$. This defines the partial order relation on D^δ_S. Note that, if $\top_D$ is isolated in D then D^δ_S is a domain in which both δ and $\top_D$ are isolated.

Every domain interpreting a type τ below has distinguished δ- and $\top$-elements. We shall denote these by δ_τ and $\top_\tau$, respectively.

$$\begin{array}{rcll}
[\![\text{unit}]\!] & = & 1^\top & \\
[\![\tau \to \tau']\!] & = & ([\![\tau]\!] \hookrightarrow [\![\tau']\!])^\delta_{\{f\}} & \text{where } f = \lambda d \in [\![\tau]\!].\mathit{if}\ d \sqsubseteq \delta_\tau\ \mathit{then}\ \delta_{\tau'}\ \mathit{else}\ \top_{\tau'} \\
[\![\tau \times \tau']\!] & = & ([\![\tau]\!] \otimes [\![\tau']\!])^\delta_{\{d\}} & \text{where } d = (\delta_\tau, \delta_{\tau'}) \\
[\![\tau + \tau']\!] & = & ([\![\tau]\!] \oplus [\![\tau']\!])^\delta_{\{d_1,d_2\}} & \text{where } d_1 = \mathit{inl}\ \delta_\tau \text{ and } d_2 = \mathit{inr}\ \delta_{\tau'} \\
[\![\mathit{rec}\ \alpha.\ \tau]\!] & = & \lim_{i\in\omega} F^i(1^\top) & \text{where } F(D) = [\![\tau]\!][\alpha \mapsto D]
\end{array}$$

The last clause denotes the inverse-limit construction for topped domains with co-strict projection/embedding-pairs. We state without proof that this yields a domain with distinguished δ- and $\top$-elements.

Note that we add a new top element for every constructed domain. Furthermore for every type constructor with the exception of unit we add a new element δ. Recall that the possibility of distinguishing completely dynamic values from partial dynamic values in a destructive context motivated our introduction of a distinct element δ in the first place. There is no destructor for unit — and hence no need to add a distinct δ-element.

As an example, let us define *bool* = unit $\oplus$ unit. The standard interpretation of *bool* has the three elements *true* = *inl* (), *false* = *inr* () and the bottom element $\bot_{bool}$. In the above extended interpretation, $[\![bool]\!]$ is the five-element domain consisting of elements $\{\bot, \mathit{true}, \mathit{false}, \delta_{bool}, \top_{bool}\}$, ordered by $\bot \sqsubset x \sqsubset \delta \sqsubset \top$ for $x = \mathit{true}$ or $x = \mathit{false}$.

Extended interpretation of expressions We have extended the interpretations of types, so now we must extend the interpretation of terms over these new types.

The extension of the standard interpretations of terms essentially follows the strictness properties of the basic syntactic constructs, so that any *destructor* (*e.g.* $\pi_1\cdot$, *case* $\cdot$ *of* ...) maps the elements δ and $\top$ of the type being "destructed" to δ and $\top$, respectively, of the resulting domain. Without giving the full details, the essence of the extension is characterised by the following semantic equations for $\top$. They are completely analogous for δ.

$$\left.\begin{array}{rcl}
\left[\!\!\left[\begin{array}{l} \mathit{case}\ e\ \mathit{of} \\ \quad \mathit{inl}\ x \Rightarrow e' \mid\mid \\ \quad \mathit{inr}\ y \Rightarrow e'' \end{array}\right]\!\!\right]\rho & = & \top \\
[\![e\ e']\!]\rho & = & \top \\
[\![\pi_1 e]\!]\rho & = & \top \\
[\![\pi_2 e]\!]\rho & = & \top
\end{array}\right\} \text{ if } [\![e]\!]\rho = \top$$

4.3 Binding times as ideals

A *binding time* at type τ is modelled by a nonempty, nonfull *ideal (closed set, inclusive set)* in the Scott-topology of $[\![\tau]\!]$; that is, it is a subset of $[\![\tau]\!]$ which is:

1. neither empty nor full: it contains $\bot_\tau$, but *not* $\top_\tau$;
2. downwards closed: $x \sqsubseteq y \in I \Longrightarrow x \in I$; and
3. closed under ω-chains: if $\{x_i\}$ is an ascending ω-chain with $x_i \in I$ for all $i \in \omega$ then $\bigsqcup_{i\in\omega} x_i \in I$.

For each type τ a set I is said to be a *(semantic) binding time* at type τ if I is a nonempty, nonfull ideal over the domain $[\![\tau]\!]$. We will say that an element d (of some domain E) has binding time I (an ideal over E) whenever $d \in I$. Let I and I' be arbitrary binding times at types τ and τ', respectively. The binding times are closed under the following operations:

$$\begin{aligned}
I \to I' &\overset{\text{def}}{=} \{f \in [\![\tau \to \tau']\!] \mid f \neq \top, f \neq \delta \text{ and } (\forall d \in I)\, f(d) \in I'\} \\
I \times I' &\overset{\text{def}}{=} \{(d, d') \in [\![\tau \times \tau']\!] \mid d \in I, d' \in I'\} \cup \{\bot\} \\
I + I' &\overset{\text{def}}{=} \{\mathit{inl}\, d \in [\![\tau + \tau']\!] \mid d \in I\} \\
&\quad\ \cup \{\mathit{inr}\, d' \in [\![\tau + \tau']\!] \mid d' \in I'\} \cup \{\bot\}
\end{aligned}$$

Furthermore, the ideals of $[\![\tau[\mathit{rec}\,\alpha.\ \tau/\alpha]]\!]$ and $[\![\mathit{rec}\,\alpha.\ \tau]\!]$ are in a one-to-one correspondence.

Being Scott-closed sets, binding times at the same type are closed under finite unions $I_1 \cup \ldots \cup I_n$ and (finite or infinite) intersections $\bigcap_{k \in K} I_k$.

4.4 Binding-time statements

We say closed expression e has binding time I and write $\models e : I$ if $[\![e]\!] \in I$. For open expressions, let B be a mapping from program variables to binding times. We write $B \models e : I$ if for all environments ρ and variables x in the domain of B such that $\rho(x) \in B(x)$ we have $[\![e]\!]\rho \in I$.

"Dynamic" and "Static" At each non-trivial type τ we define an ideal Δ_τ which represents the property "completely dynamic" at τ, and Σ_τ, which represents "surface" static. We define Δ_τ to be the *downwards closure* $\downarrow\!\delta_\tau$ of δ_τ; that is, the least ideal containing δ_τ: $\downarrow\!\delta_\tau = \{d \in [\![\tau]\!] : d \sqsubseteq \delta_\tau\}$. Note in particular that Δ_τ contains δ_τ, but not $\top_\tau$. The binding time Σ_τ at τ denotes $\Delta_\tau - \{\delta_\tau\}$; that is, all of Δ_τ except for its maximal element δ_τ. (Note that δ is isolated in every domain, so this *is* an ideal.) Intuitively, the elements in Σ_τ are those that are "surface" static, in the sense that one can apply the corresponding destructor at partial evaluation time without getting an error. We usually write Δ and Σ without subscripts whenever the type is derivable from the context.

Taking a domain such as *bool* $\times$ *bool*, we can represent the property that the pair is statically known (intuitively, available to the partial evaluator to destruct) by the ideal $\Sigma_{\mathit{bool} \times \mathit{bool}}$, which is equal to $\Delta_{\mathit{bool}} \times \Delta_{\mathit{bool}}$. Figure 1 sketches part of the Hasse diagram for *bool* $\times$ *bool*, and indicates some example binding times.

5 Internalising Binding-time Properties: Semantics-based Annotations

In partial evaluation and other transformations it is important to know not only *what extensional* property a program has, but also *how* it is established; in particular, what properties have to hold internally, for the individual *parts* of the program, since it is this *intensional* information that is usually exploited in optimizing transformations.

In partial evaluation it is rather useless in itself to find out that an expression has binding time "dynamic." Indeed this is usually stipulated from the outset. What is desired is a *proof* whose structure captures what the binding-time properties of individual parts of the expression are and how they can be combined to yield the binding time of the overall expression.

What is required then is an *internalisation* of what it means for an expression e to have a certain binding time. That is, the subexpressions of e must have certain binding times if the whole expression e is to have its final, desired binding time.

Ideally one would hope for a *complete* internalisation, in essence a *sound and complete* logic for inferring binding times. This may be undesirable (on top of being difficult to

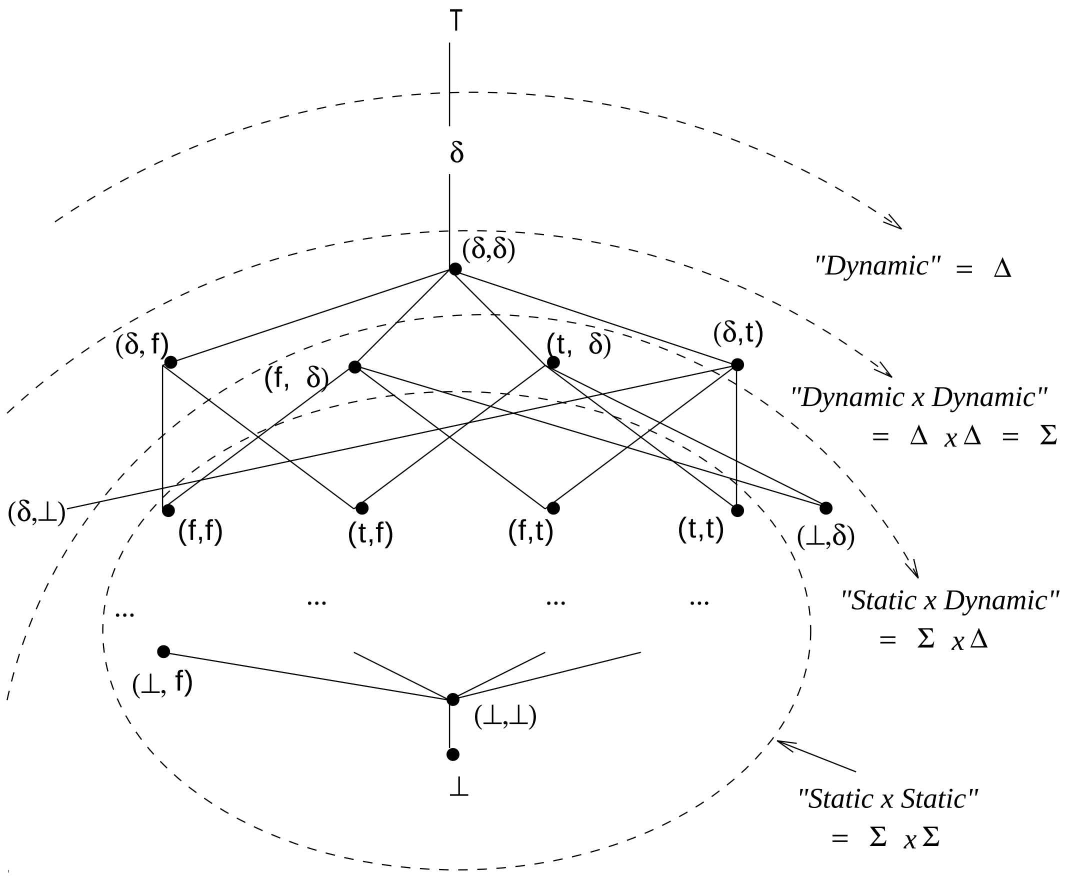

Figure 1: Example binding times in *bool* × *bool*

$$B\{x \mapsto I\} \rhd x : I \qquad \frac{B\{x \mapsto I\} \rhd e : I'}{B \rhd \lambda x.\, e : I \to I'} \qquad \frac{B \rhd e : I \to I' \quad B \rhd e' : I}{B \rhd e\, e' : I'}$$

$$\frac{B \rhd e : I \qquad B \rhd e' : I'}{B \rhd (e, e') : I \times I'} \qquad \frac{B \rhd e : I_1 \times I_2}{B \rhd \pi_i e : I_i} \qquad \frac{B \rhd e : \Delta}{B \rhd \pi_i e : \Delta} \quad (\text{i} = 1{,}2)$$

$$\frac{B \rhd e : I}{B \rhd \mathit{inl}\, e : I + \emptyset} \qquad \frac{B \rhd e : I}{B \rhd \mathit{inr}\, e : \emptyset + I}$$

$$\frac{\begin{array}{c} B \rhd e : I + \emptyset \\ B\{x \mapsto I\} \rhd e' : I'' \\ B\{x' \mapsto \Delta\} \rhd e'' : \Delta \end{array}}{B \rhd \left(\begin{array}{l} \mathit{case}\ e\ \mathit{of} \\ \quad \mathit{inl}\, x \Rightarrow e' \,\|\, \\ \quad \mathit{inr}\, x' \Rightarrow e'' \end{array}\right) : I''} \qquad \frac{\begin{array}{c} B \rhd e : \emptyset + I' \\ B\{x \mapsto \Delta\} \rhd e' : \Delta \\ B\{x' \mapsto I'\} \rhd e'' : I'' \end{array}}{B \rhd \left(\begin{array}{l} \mathit{case}\ e\ \mathit{of} \\ \quad \mathit{inl}\, x \Rightarrow e' \,\|\, \\ \quad \mathit{inr}\, x' \Rightarrow e'' \end{array}\right) : I''}$$

$$\frac{\begin{array}{c} B \rhd e : I + I' \\ B\{x \mapsto I\} \rhd e' : I'' \\ B\{x' \mapsto I'\} \rhd e'' : I'' \end{array}}{B \rhd \left(\begin{array}{l} \mathit{case}\ e\ \mathit{of} \\ \quad \mathit{inl}\, x \Rightarrow e' \,\|\, \\ \quad \mathit{inr}\, x' \Rightarrow e'' \end{array}\right) : I''} \qquad \frac{\begin{array}{c} B \rhd e : \Delta \\ B\{x \mapsto \Delta\} \rhd e' : \Delta \\ B\{x' \mapsto \Delta\} \rhd e'' : \Delta \end{array}}{B \rhd \left(\begin{array}{l} \mathit{case}\ e\ \mathit{of} \\ \quad \mathit{inl}\, x \Rightarrow e' \,\|\, \\ \quad \mathit{inr}\, x' \Rightarrow e'' \end{array}\right) : \Delta}$$

$$\frac{B\{f \mapsto I\} \rhd e : I}{B \rhd \mathit{fix}\ f.\, e : I}$$

Figure 2: Monovariantly annotated expressions, nonlogical rules

accomplish at all) since it expects the ensuing (automated) transformation process to exploit or at least to "understand" all proofs in the logic.

In this section we present an internalisation for *monovariantly well-annotated* expressions. Intuitively, monovariancy requires that the binding time of a bound variable be a fixed binding time (ideal), which must be "big enough" to contain the values the variable may ever be bound to. Similarly, the binding time of a subexpression must be a single ideal big enough to contain the value of the subexpression in every environment it is evaluated.

This is in contrast to *polyvariant binding-time analysis* where bound variables can be associated with several binding times (for different contexts) and where the binding times of (sub)expressions in the scope of bound variables may be *dependent* on (functions of) the binding times of the actual values.

We restrict our attention to monovariancy since monovariant binding-time analyses are currently better and easier understood in partial evaluation.[5]

5.1 Monovariant binding-time annotations

Monovariantly well-annotated expressions are defined by an inference system on *binding-time judgements* of the form $B \rhd e : I$, where I denotes a *semantic binding time*, e is a well-typed expression, and B is an environment associating variables with binding times.

[5]We believe the monovariant inference system given here can be extended to a polyvariant logic that is semantically complete, but leave this to future work.

$$\frac{B \rhd e : I}{B \rhd e : I'} \quad (I \subseteq I')$$

Figure 3: Logical rules

$$\frac{B \rhd e : I}{B \rhd e^b : I} \quad (b = \alpha(I))$$

Figure 4: Annotation rule

Figure 2 presents the *nonlogical* inference rules for inferring binding-time properties for the constructs of the language in a syntax-directed fashion. Figure 3 gives a *logical* rule that is applicable to *any* expression. It lets us *weaken* an ideal to any larger ideal. Finally, Figure 4 adds a rule for *annotating* an expression with an *abstraction* of an ideal.

Note that we have no *formal language* of binding times: the metavariables I, I' range over *semantic* binding times (ideals), Δ denotes binding time "dynamic" (at the relevant type) as defined in Section 4.4, and $\times, +$ and $\rightarrow$ are the ideal constructors from Section 4.3.

What is gained by this inference system, and what role does the annotation rule play in it? Consider a derivation of a binding-time property for an expression e *without* use of the annotation rule. This results in a binding-time judgement $A \rhd e : I$. Discarding the derivation and only retaining its conclusion we know the binding time of e itself, but nothing about the binding times of its subexpressions! The practical problem for a specialiser is that it needs to process program *parts* on the basis of *their* binding-time properties. Knowing only the binding-time property of the whole expression is *too little* information. One way of remedying this is to retain the whole derivation since it contains binding-time judgements for the subexpressions — this is, in essence, what the monolithic approach does: the result of a binding-time analysis is the (whole) derivation, and a specialiser does not operate on the program expression itself, but on a representation of a derivation for it. As we have remarked earlier, this ties the specialiser closely to a particular formalisation of a binding-time analysis. Furthermore, it may retain *too much* information, information that is not really relevant to partial evaluation or other transformations. Since there is a potentially large "semantic distance" between the original expression and the product of the analysis (the whole derivation), it furthermore complicates establishing the correctness of the specialiser since the specialiser is defined on derivations, whereas the standard semantics is defined on the original expression.

By using the annotation rule we can discard the derivation of a binding-time property for an expression and yet retain *relevant semantic information* about its subexpressions. This information is extensional in the sense that annotations only abstract the derived binding time for a subexpression, not a particular way the binding time can be established. We can think of the annotations as a "semantic trace" of how an expression has a certain binding-time property. The *abstraction function* α is a partial function mapping binding times to annotations. It expresses how much semantic information is retained from a binding time. The standard annotations we shall consider in the following section are $\mathcal{A} = \{S, D\}$ where S stands for "static" and D for "dynamic."

It is easy to check that $B \rhd e : I$ implies $B \models e : I$; that is, our monovariant internalisation is *sound*. For every (open) expression e and assumptions B mapping the free variables of e to binding-times contained in Δ there is at least one binding time I such that $B \rhd e : I$ is derivable using the rules of Figures 2 and 3. The internalisation is not *complete*, however:

$B \models e : I$ does not generally imply $B \rhd e : I$.

5.2 Preservation of Binding-Time Properties under Reduction

In this section we show that well-annotated expressions are closed under *reduction* in *any* context; that is, they have the *Subject Reduction Property.* As shown in the following section this is the crucial connection that lets us argue the safety of partial evaluation for expressions that are — via a translation into our annotation scheme — semantically well-annotated.

An *annotated expression* is a (well-typed) expression where subexpressions may carry arbitrary annotations from the range of α.

Definition 5.1 (Reduction rules) *The one-step reduction, $\rightarrow$, for annotated expressions is given by the following rules,*

$$(\lambda x.\, e_1)\; e \rightarrow e_1\{{}^{e}\!/\!_{x}\} \qquad \pi_1(e, e') \rightarrow e \qquad \pi_2(e, e') \rightarrow e'$$

$$\begin{pmatrix} \mathit{case\ inl}\,(e)\ \mathit{of} \\ \quad \mathit{inl}\, x \Rightarrow e_1 \parallel \\ \quad \mathit{inr}\, y \Rightarrow e_2 \end{pmatrix} \rightarrow e_1\{{}^{e}\!/\!_{x}\} \qquad \begin{pmatrix} \mathit{case\ inr}\,(e)\ \mathit{of} \\ \quad \mathit{inl}\, x \Rightarrow e_1 \parallel \\ \quad \mathit{inr}\, y \Rightarrow e_2 \end{pmatrix} \rightarrow e_2\{{}^{e}\!/\!_{y}\}$$

$$\mathit{fix}\ f.\, e_1 \rightarrow e_1\{{}^{\mathit{fix}\ f.\, e_1}\!/\!_{f}\} \qquad e^b \rightarrow e$$

The expressions to the left are called redexes *and those on the right the corresponding* reducts. *We close $\rightarrow$ under* arbitrary *contexts; that is, $e \rightarrow e'$ if $e = C[r]$ for some (single-hole) context $C[]$ and redex r with reduct r', where $e' = C[r']$. We write $e \rightarrow^+ e'$ if $e \rightarrow \ldots \rightarrow e'$ in one or more reduction steps; $e \rightarrow^* e'$ if $e = e'$ or $e \rightarrow^+ e'$.*

The main theorem of this section is that well-annotated terms are closed under reduction; that is, if $e \rightarrow^* e'$ and e is well-annotated then e' is also well-annotated. This is the critical connection between our semantic model of binding times and what one can do with this information.

Due to the annotations that (may) occur inside expressions the subject reduction property also has to hold *locally*; that is, intuitively, we must be able to establish that, if $e \rightarrow e'$ then $B \rhd e' : I$ can be obtained by "local" transformation of *any* proof of $B \rhd e : I$. For this to hold in the presence of the weakening rule of Figure 3, it is necessary and sufficient to establish that containments between constructed ideals hold only if suitable containment relations hold for the component ideals; that is,

$$\begin{aligned} I \times I' \subseteq J \times J' \quad &\Rightarrow \quad I \subseteq J \text{ and } I' \subseteq J' \\ I + I' \subseteq J + J' \quad &\Rightarrow \quad I \subseteq J \text{ and } I' \subseteq J' \\ I \rightarrow I' \subseteq J \rightarrow J' \quad &\Rightarrow \quad J \subseteq I \text{ and } I' \subseteq J' \end{aligned}$$

The technically crucial point is that the first two implications hold for *arbitrary* nonempty ideals, whereas the last implication does not: if J' is full (the whole domain) then $I \rightarrow I' \subseteq J \rightarrow J'$ for *any* choice of I, I', J!

This problem is the — sole — reason why our extended domains do not only contain an element δ representing dynamic values, but also an element $\top$, and why binding times, by definition, are nonfull ideals. The following lemma shows that the problematic implication *does* hold if J' is *not* full.

Lemma 5.2 *Let D be a domain with a top element $\top$. Let I, I', J, J' be nonempty ideals such that J' is a proper subset of D.*

Then $I \rightarrow I' \subseteq J \rightarrow J'$ if and only if $J \subseteq I$ and $I' \subseteq J'$.

Thus all three implications above hold for binding times, and we can establish our main theorem:

Theorem 5.3 (Subject Reduction Theorem) *If $B \rhd e : I$ and $e \rightarrow^* e'$ then $B \rhd e' : I$.*

6 Standard and Partial Evaluation

In this section we give a definition of off-line partial evaluation for our language, built by removing restrictions on reductions possible in the standard evaluation rules, and prove its safety from our semantic definition of a sound binding-time annotation.

6.1 Standard Call-by-name Evaluation

The operational semantics of the language is specified by reduction rules, which represent the basic computation steps, plus a description of the syntactic contexts in which the rules can be applied.

The standard operational semantics is built using the reduction rules of Definition 5.1, applying them to closed unannotated expressions. We need to specify the syntactic contexts in which we allow the reduction rules to be applied. We do not specify a deterministic reduction order, since it is sufficient (and more general) simply to constrain the reduction rules so that we: (i) do not reduce under a λ-abstraction, (ii) do not reduce under a fix-expression, (iii) do not reduce in the branches of a case expression, and (iv) do not reduce under a constructor (pairing, or sum injections). Let $\rightarrow_n$ be the resulting relation (call-by-*n*ame reduction). We state the following properties of $\rightarrow_n$ without proof: for all closed expressions $e : \tau$

- If $e \rightarrow_n e'$ then $e' : \tau$ and $[\![e]\!] = [\![e']\!]$
- If there is an infinite reduction sequence starting from e then all reduction sequences are infinite.
- For all e of first-order type, but excluding unit, we have that $[\![e]\!] = \bot$ if and only if there is an infinite reduction sequence starting from e.

Thus denotational semantics is sound with respect to a definition of observational equivalence which observes termination at any first-order type except unit.

6.2 Partial Evaluation

We define a class of partial evaluators by describing the possible reductions that a partial evaluator *may* perform. A particular partial evaluator could then be built by choosing some reduction strategy from these reductions.

We view partial evaluation as standard evaluation extended rather straightforwardly to handle "symbolic" values; that is, open expressions. "Dynamic" inputs are then just modelled by free variables.

Binding-time analysis is used to guide either the actions of a specialiser (so-called off-line partial evaluation) or to optimise its actions (on-line partial evaluation). In both cases it is important to guarantee that the specialiser can "trust" the information provided by the binding-time analysis in order to avoid costly checks of data at partial evaluation time (such as whether data are static or dynamic values). In this section we show how semantically consistent binding-time annotations ensure that a specialiser cannot "go wrong" as long as its actions (reduction steps) *respect* the (semantic) binding-time annotations.

Partial evaluation applies the same reduction rules as standard evaluation, but with fewer constraints. Firstly, we allow evaluation to be more eager, so evaluation of the components of pairs, or of the argument of the sum-injections, is now possible. Secondly, we draw upon the binding-time annotations to allow reductions to reach even deeper into an expression — namely, under lambda abstractions or in the branches of case-expressions. Due to this, and the presence of dynamic inputs (free variables), it must be guaranteed that a specialiser that executes a "static" reduction can be assured that it does not encounter dynamic data where it expects to find static data. In off-line partial evaluation only static reductions are performed. In on-line partial evaluation dynamic reductions may also be performed (see the next section), but require a check as the nature of the data (static or dynamic).

PE-annotations The definition of a monovariant binding-time annotation provided in the previous section is particularly simple because it only describes annotations on sub-expressions. The price of the simplicity of this definition is that there may not be an exact correspondence between the kind of annotations which are followed by a given partial evaluator, and the notion of a monovariant binding-time annotation. In particular, the partial evaluator we will describe operates on expressions with annotations on the binding occurrences of variables—and these are not proper sub-expressions. To avoid confusion we will call the annotations expected by our partial evaluator *PE-annotations.* When we come to prove the safety of the partial evaluator we will show how the PE-annotations can be interpreted as semantic annotations.

The PE-annotated expressions handled by our partial evaluator have possible annotations in two places: on all destructors (projections, case-expressions, applications), indicating a binding-time property of the expression in the "destructed" position, and on some binding occurrences of variables (lambda abstractions, fix-expressions and case expressions). Furthermore, the partial evaluator knows about only two annotations: dynamic (D) and static (S). Destructors annotated by S can be reduced (the partial evaluator expects that the destructed expression will be static), but dynamic destructors will not be reduced (since it cannot be trusted that the expression in the hole will be static).

Reduction now occurs in a more liberal class of contexts. In particular we can reduce under lambdas, or inside the branches of a case expression whenever the variable is annotated with dynamic.

The definition is divided into *static reductions* $\rightarrow_{pe}$, and *partial evaluation contexts* P. Let variables b, b_1, b_2 range over annotations $\{S, D\}$, and let $[D]$ denote either annotation D or "no annotation". The static reductions $\rightarrow_{pe}$ are given in Fig. 5 and the partial evaluation contexts $I\!\!P$ are given in Fig. 6.

Definition 6.1 *One step partial evaluation relation* $\mapsto_{pe}$ *is defined by closing the static reductions under partial evaluation contexts. In other words, for all annotated expressions* e, e', $e \mapsto_{pe} e'$ *iff* $e \equiv I\!\!P[e_1]$, *for some* $I\!\!P$, e_1 *such that* $e_1 \rightarrow_{pe} e_2$ *and* $I\!\!P[e_2] \equiv e'$.

Restricted to pure lambda-terms, the reductions permitted by our definition include those of Palsberg's definition of a *top-down partial evaluator* [Pal93].

6.3 Safety of the Partial evaluator

The definition of safety focuses on the statically annotated destructors. It is convenient to give a formal definition of these expressions:

$$\lambda x^{[D]}.\, e_1 \,@^S e \rightarrow_{pe} e_1\{e/x\} \qquad \pi_1^S(e,e') \rightarrow_{pe} e \qquad \pi_2^S(e,e') \rightarrow_{pe} e'$$

$$\begin{pmatrix} \mathit{case}^S\ \mathit{inl}\,(e)\ \mathit{of} \\ \quad \mathit{inl}\,x^{[b_1]} \Rightarrow e_1 \,\| \\ \quad \mathit{inr}\,y^{[b_2]} \Rightarrow e_2 \end{pmatrix} \rightarrow_{pe} e_1\{e/x\} \qquad \begin{pmatrix} \mathit{case}^S\ \mathit{inr}\,(e)\ \mathit{of} \\ \quad \mathit{inl}\,x^{[b_1]} \Rightarrow e_1 \,\| \\ \quad \mathit{inr}\,y^{[b_2]} \Rightarrow e_2 \end{pmatrix} \rightarrow_{pe} e_2\{e/y\}$$

$$\mathit{fix}\ x^S.\, e_1 \rightarrow_{pe} e_1\{\mathit{fix}\ x^S.\, e_1/x\}$$

Figure 5: Static Reduction Rules

$$\begin{array}{rcl} I\!\!P & ::= & [\,] \;\mid\; I\!\!P \,@^b e' \;\mid\; e \,@^b I\!\!P \;\mid\; \pi_1^b I\!\!P \;\mid\; \pi_2^b I\!\!P \\ & \mid & \begin{array}{l} \mathit{case}^b\ I\!\!P\ \mathit{of} \\ \quad \mathit{inl}\,x^{[D]} \Rightarrow e_1 \,\| \\ \quad \mathit{inr}\,y^{[D]} \Rightarrow e_2 \end{array} \;\mid\; \begin{array}{l} \mathit{case}^b\ e\ \mathit{of} \\ \quad \mathit{inl}\,x^{D} \Rightarrow I\!\!P \,\| \\ \quad \mathit{inr}\,y^{[D]} \Rightarrow e_2 \end{array} \;\mid\; \begin{array}{l} \mathit{case}^b\ e\ \mathit{of} \\ \quad \mathit{inl}\,x^{[D]} \Rightarrow e_1 \,\| \\ \quad \mathit{inr}\,y^{D} \Rightarrow I\!\!P \end{array} \\ & \mid & \lambda x^D.\, I\!\!P \;\mid\; \mathit{fix}\ x^D.\, I\!\!P \;\mid\; (I\!\!P, e') \;\mid\; (e, I\!\!P) \;\mid\; \mathit{inl}\ I\!\!P \;\mid\; \mathit{inr}\ I\!\!P \end{array}$$

Figure 6: Partial Evaluation Contexts

Definition 6.2 (Destructors) *Define the destructors $I\!\!D$ to be the following single-holed contexts:*

$$I\!\!D \;::=\; [\,]\,e \;\mid\; \pi_1[\,] \;\mid\; \pi_2[\,] \;\mid\; \begin{array}{l} \mathit{case}\,[\,]\ \mathit{of} \\ \quad \mathit{inl}\,x \Rightarrow e_1 \,\| \\ \quad \mathit{inr}\,y \Rightarrow e_2 \end{array}$$

All destructors occurring in a PE-annotated expression must carry an annotation (S or D). We call these expressions the PE-destructors. Let $I\!\!D^S$ and $I\!\!D^D$ respectively denote the static and dynamically annotated PE destructors. For example, if $I\!\!D$ is the destructor $\pi_1[\,]$, then $I\!\!D^D[x]$ is the expression $\pi_1^D x$. A static destructor tells the partial evaluator that it can expect that the expression in the hole can be evaluated to a constructor of the right type (or we loop in the attempt). What this means in an implementation (*e.g.* a partial evaluator like Similix [Bon91]) is that when partial evaluation of the expression in the destructor position has finished, it will be trusted that the result will be of the right kind to be "destructed". The partial evaluator "goes wrong" and reaches a possible *error state* if this is not the case. Before we give a syntactic characterisation of these error states, we note the following properties, where we assume that the standard semantics of an annotated expression is defined to be that of the corresponding unannotated version.

Proposition 6.3 *If $A \vdash e : \tau$ and $e \rightarrow_{pe} e'$ then $A \vdash e' : \tau$, and for all environments ρ matching type environment A, $[\![e]\!]\rho = [\![e']\!]\rho$*

So partial evaluation preserves the type and denotation of an expression. Note that if we wish to consider the underlying language to be call-by-value, then this partial evaluator increases termination properties (so $[\![e]\!]_{val} \sqsubseteq_{val} [\![e']\!]_{val}$) in the manner of lambda-mix [GJ91]. However, this is not the aspect of safety that concerns binding-time analysis.

The fact that we always have a well-typed program leads to the conclusion that the error states are those for which a variable, or a dynamic destructor, appears in the hole of a static destructor (and that this occurs in some partial evaluation context). The following proposition helps characterise the error states:

Proposition 6.4 *If $\mathbb{D}[e]$ is a well-typed expression, and $\mathbb{D}^S[e]$ is not a partial evaluation redex, then either $e \equiv x$ or $e \equiv \mathbb{D}'[e']$ for some destructor $\mathbb{D}'$.*

Definition 6.5 (Error states) *A PE-annotated expression e is in an* error state *if either*

1. $e \equiv \mathbb{P}[\,\mathbb{D}_1^S[x]\,]$ *for some $\mathbb{P}$, $\mathbb{D}_1$, x, or*
2. $e \equiv \mathbb{P}[\,\mathbb{D}_1^S[\mathbb{D}_2^D[e']]\,]$ *for some $\mathbb{P}$, $\mathbb{D}_1$, $\mathbb{D}_2$, e'.*

Our goal is to show that applying the reduction rules of the partial evaluator on a semantically well-annotated program can never lead to an error state. To do this we must give a definition of "a well-annotated expression" by interpreting PE-annotated expressions as semantic annotations.

Definition 6.6 *We define a mapping, $\widehat{\cdot}$, from PE-annotated expressions to (ordinary) annotated expressions by induction on the syntax:*

$$\widehat{x} = x \qquad \widehat{(e_1, e_2)} = (\widehat{e_1}, \widehat{e_2}) \qquad \widehat{inl\, e} = inl\, \widehat{e} \qquad \widehat{inr\, e} = inr\, \widehat{e}$$

$$\widehat{\lambda x^{[D]}.\, e} = \lambda x.\, (\widehat{e}\{x^{[D]}/_x\}) \qquad \widehat{e_1\ @^b e_2} = (\widehat{e_1})^b\ \widehat{e_2} \qquad \widehat{\pi_1^b e} = \pi_1(\widehat{e}^{\,b}) \qquad \widehat{\pi_2^b e} = \pi_2(\widehat{e}^{\,b})$$

$$\left(\begin{array}{l} \widehat{case^b\ e\ of} \\ \quad inl\ x_1^{[D]} \Rightarrow e_1 \,\| \\ \quad inr\ x_2^{[D]} \Rightarrow e_2 \end{array}\right) = \begin{array}{l} case\ (\widehat{e}^{\,b})\ of \\ \quad inl\ x_1 \Rightarrow (\widehat{e_1})\{x_1^{[D]}/_{x_1}\} \,\| \\ \quad inr\ x_2 \Rightarrow (\widehat{e_2})\{x_2^{[D]}/_{x_2}\} \end{array}$$

So, for example, if e is the PE-annotated expression $\lambda x^D.\, (x, x)$, then $\widehat{e} = \lambda x.\, (x^D, x^D)$.

Next we must give an interpretation of the annotations $\{S, D\}$ as abstractions of ideals. In what follows we assume the following definition for the abstraction map α:

$$\alpha(I_\tau) = \begin{cases} D, & \text{if } I = \Delta_\tau \\ S, & \text{if } I \subseteq \Sigma_\tau \\ \text{undefined}, & \text{otherwise} \end{cases}$$

Now we can define when a PE-annotated expression is semantically well-annotated.

Definition 6.7 *A PE-annotated open expression e, such that $A \vdash e : \tau$ for some type environment A, is* well-annotated *if there exists an I such that $B_0 \rhd \widehat{e} : I$, where $B_0 = \{x \mapsto \Delta_{A(x)} \mid x$ in the domain of $A\}$.*

The structure of the proof of safety of the partial evaluator is as follows: first we show that anything appearing in a partial evaluation context is well-annotated, providing that the whole expression is well-annotated. We use this to argue that the error states are not well-annotated, and hence that the partial evaluator never starts out in an error state. The proof is completed by using the subject reduction property, which states that well-annotatedness is preserved by partial evaluation steps.

Theorem 6.8 *If e is well-annotated and $e \mapsto_{pe}^* e'$ then e' is not in an error state.*

7 On-line Partial Evaluation

Off-line partial evaluation is *defined* to be partial evaluation which uses a binding-time analysis. Conversely, the term *on-line* is used for partial evaluators which do not. This means that before attempting to do a reduction, an on-line partial evaluator must always check to see if the object being destructed is of the appropriate kind. It is generally able to perform more reductions than an off-line evaluator, but is potentially less efficient (and less simple in structure) because of the extra checking necessary.

In this section we show that the safety condition—that we never reach an error state—also holds for a form of on-line partial evaluation. This result is significant because it shows that an on-line partial evaluator could be optimised by using a binding-time analysis, since it removes partial-evaluation-time checks on the argument to a static destructor.

Definition 7.1 *Define an on-line reduction relation $\rightarrow_{on}$ on PE-annotated terms by extending the partial evaluation reductions $\mapsto_{pe}$ to include the following rewrite:*
If $\mathbb{D}^S[e] \rightarrow_{pe} e'$ then $\mathbb{P}[\mathbb{D}^D[e]] \rightarrow_{on} \mathbb{P}[e']$

Note then that we still do not permit reduction under non-dynamically annotated binding operators (not a severe restriction, since because they are static they are likely to be eliminated by reduction anyway). But now if a dynamic-annotated destructor is a redex, then it can be reduced.

Theorem 7.2 *If e is well annotated and $e \rightarrow^*_{on} e'$ then e' is not in an error state.*

8 Correctness of binding-time analyses

We have focused on proving safety (correctness) of partial evaluation from a semantic model. A reasonable question is whether it is possible to design analyses and show that they are sound *w.r.t.* this semantic model. In this section we briefly claim that this is so, by outlining how existing monovariant binding-time analyses can be justified in our model and monovariantly internalised.

8.1 λ-mix

Gomard and Jones describe a simple, but illustrative off-line partial evaluator for the kernel of an untyped higher-order programming language [GJ91, Gom92]. We shall only consider the (pure) lambda calculus subset of the language. It is untyped, but can be understood to be typed by giving every expression the type $rec\,\alpha.\ \alpha \rightarrow \alpha$. Our extended domain interpretation maps this type to the "smallest" domain D_∞ such that $D_\infty \cong (D_\infty \hookrightarrow D_\infty)^{\delta}_{\{f\}}$ via top-strict continuous isomorphism $\Psi : (D_\infty \hookrightarrow D_\infty)^{\delta}_{\{f\}} \rightarrow D_\infty$, where $f = \lambda d \in D_\infty.\mathit{if}\ d \sqsubseteq \delta_{D_\infty}\ \mathit{then}\ \delta_{D_\infty}\ \mathit{else}\ \top_{D_\infty}$.

As before, the semantic ideal Δ modelling "dynamic" (D) is $\downarrow\delta_{D_\infty}$, whereas "(surface) static" (S) is $\Sigma = \Delta - \{\delta_{D_\infty}\} = \Psi(\Delta \rightarrow \Delta)$.

The binding-time analysis of Gomard and Jones can be described by an inference system consisting of rules that are derivable in our monovariant internalisation. We give the rules using the ordinary annotated expressions. These correspond, via a translation as in Definition 6.6, to PE-annotated terms where not only destructors, but also constructors are annotated by either S or D.

This shows that the binding-time analysis of Gomard and Jones is *sound* with respect to our model of binding times and our monovariant internalisation. An immediate consequence of the Subject Reduction Theorem is that no partial evaluator, in particular λ-mix, whose actions can be modelled by the reductions of Section 5 can reach an error state.

$$A\{x \mapsto I\} \vdash x : I$$

$$\frac{A\{x \mapsto I\} \vdash e : I'}{A \vdash (\lambda x.\, e)^S : I \to I'} \qquad \frac{A\{x \mapsto \Delta\} \vdash e : \Delta}{A \vdash (\lambda x.\, e)^D : \Delta}$$

$$\frac{A \vdash e : I \to I' \quad A \vdash e' : I}{A \vdash (e^S)\ e' : I'} \qquad \frac{A \vdash e : \Delta \quad A \vdash e' : \Delta}{A \vdash (e^D)\ e' : \Delta}$$

Figure 7: Binding-time analysis a la Gomard and Jones

8.2 Other analyses

Mogensen Mogensen extends the binding times in Gomard and Jones' analysis with recursively specified binding times [Mog92]. His off-line partial evaluator has been shown correct relative to the analysis by Wand [Wan93]. Mogensen's analysis can also be justified by the rules of Figure 7. The safety of his partial evaluator follows from our Subject Reduction Theorem.

Palsberg/Schwartzbach The binding-time annotations of Palsberg and Schwartzbach [Pal93] are the same as for Gomard/Jones and Mogensen. Their binding-time analysis, however, cannot be shown correct relative to our monovariant internalisation since our inference system lacks the ability of propagating *disjunctive* properties. Adding rules for *unions* of ideals, in the style of Jensen [Jen92], to our internalisation, seems to provide a — still monovariant — internalisation of binding-time properties that subsumes their analysis. This extension promises interesting applications to constructor specialisation and closure analysis.

Launchbury and Hunt/Sands The binding-time models of Launchbury [Lau89] and Hunt and Sands [HS91] can be expressed in our model in the sense that their (syntactic descriptions of) binding times can be interpreted as ideals in our extended domains, giving valid binding-time statements for expressions. The analysis of Hunt and Sands is polyvariant, however, and thus cannot be expressed in our monovariant internalisation. We conjecture that Launchbury's analysis is expressible in our monovariant internalisation.

9 Conclusions and Further Work

In this paper we have considered a model-based approach to the safety of off-line partial evaluation. We have motivated a new model for structured binding times in higher-order functional languages, illustrating problems with the previous models using projections and PERs. The model is based on an extension of the standard domains with "extra" elements δ (anonymous dynamic value) and $\top$ (error value) at each type, reflecting the finer distinctions that we are able to make between programs at partial evaluation time than we are able to make during normal execution. We tackle the problem of program annotation by showing that semantic properties can be expressed in a structural syntax-directed style. This is essentially a collecting interpretation [CC79], but avoids the cumbersome details of an explicit "sticky" semantics mapping properties to program points (cf. [Nie85])

The model is able to represent partially static data structures in the manner of [Mog88] and [Lau88], as well as properties of higher-order functions. Furthermore, we are not dependent on any assumption of lazy data structures and non-strict evaluation in the underlying language.

We have shown that the model is adequate to prove safety for a class of partial evaluators for this language. This class of partial evaluators is similar in spirit to Palsberg's definition of top-down partial evaluators for the pure lambda-calculus. We believe this is the first proof of its kind—based on a semantic specification of a safe binding time annotation, rather than on a particular analysis. We have also shown that sound binding-time annotations are preserved by dynamic reductions, a fact which has implications for the optimisation of on-line partial evaluators. Finally, we have argued that existing analyses can be shown to be sound with respect to the model given here.

9.1 Limitations and Further Work

There are some fundamental limitations in the definition of a sound annotation which are necessary in order to prove the correctness of simple-minded partial evaluators. One such limitation is the "uniformity" assumption [Lau89], which is implicit in the structural nature of our conditions for a safe annotation[6]. This restriction is fundamental in the sense that it would actually be *unsafe* to perform, for example, constant propagation or relational analyses between variables unless the partial evaluator where to employ exactly the same flow analysis "on-line" (as in Turchin's *driving* [Tur86]). This also means that we cannot account for partial evaluators which perform arbitrary algebraic manipulations (*e.g.* code propagation across dynamic conditionals [Bon92]), unless they can be factored out in a pre-processing stage (*e.g.* [CD91]).

Polyvariant binding-time analysis is intimately connected to (finitary or infinitary) conjunctive properties of functions. This can be modelled by taking intersections of ideals. Finitary conjunctive properties of functions capture the polyvariant binding-time analyses of Gengler and Rytz [GR92] and Consel [Con93]. Infinitary conjunctive properties can be expressed as binding-time functions [HS91, CJØ94] or polymorphic types [HM94]. We plan on extending the monovariant internalisation of this paper to a sound and complete polyvariant internalisation using infinitary conjunction. This, we hope, will enable us to justify both polyvariant analyses as well as the safety of partial evaluators driven by polyvariant analyses.

Acknowledgements

Thanks to the referees for numerous useful suggestions for improvements. This work was partly funded by the Danish Science Council (SNF), project "DART".

References

[Amt93] T. Amtoft. Minimal thunkification. In *Proceedings of the 3rd International Symposium on Static Analysis*, number 724 in LNCS. Springer-Verlag, 1993.

[BEJ88] D. Bjørner, Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop, Gammel Avernæs, Denmark, October 1987.* North-Holland, 1988. 625 pages.

[BJMS88] Anders Bondorf, Neil D. Jones, Torben Mogensen, and Peter Sestoft. Binding time analysis and the taming of self-application. Draft, 18 pages, DIKU, University of Copenhagen, August 1988.

[6]This does *not* imply that the analysis must give a uniform treatment to the components of recursive types (in the sense of e.g. [EM91]). The definition of a safe annotation permits, for example, properties such as "the first ten elements of the list are static."

[Bon91] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, December 1991. Selected papers of ESOP '90, the 3rd European Symposium on Programming.

[Bon92] Anders Bondorf. Improving binding times without explicit cps-conversion. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California*, pages 1–10, June 1992.

[Bul93] Mikhail Bulyonkov. Extracting polyvariant binding time analysis from polyvariant specializer. In *Proc. ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), Copenhagen, Denmark*, pages 59–65. ACM, ACM Press, June 1993.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *ACM 5th Symposium on Principles of Programming Languages*, 1979.

[CC94] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *Proc. 1994 Int'l Conf. on Computer Languages, Toulouse, France*, pages 95–112. IEEE Computer Society Press, May 1994.

[CD91] C. Consel and O. Danvy. For a better support of static data flow. In J. Hughes, editor, *Proc. 5th ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA), Cambridge, Massachusetts*, number 523 in Lecture Notes in Computer Science, pages 496–519. Springer-Verlag, Aug. 1991.

[CJØ94] Charles Consel, Pierre Jouvelot, and Peter Ørbæk. Separate polyvariant binding time reconstruction. CRI Report A/261, Ecole des Mines, Oct. 1994.

[CK92] Charles Consel and Siau Cheng Khoo. On-line & off-line partial evaluation: Semantic specifications and correctness proofs. Technical Report YALEU/DCS/RR-912, Yale University Department of Computer Science, June 1992.

[Con93] Charles Consel. Polyvariant binding-time analysis for applicative languages. In *Proc. Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), Copenhagen, Denmark*, pages 66–77, June 1993.

[Dav94] K. Davis. Pers from projections for binding-time analysis. In *Proceedings of the ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 1994. (Proceedings available as Tech Report 94/9, University of Melbourne).

[EM91] C. Ernoult and A. Mycroft. Uniform ideals and strictness analysis. In *Proc. 18th Int'l Coll. on Automata, Languages and Programming (ICALP), Madrid, Spain*, number 510 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[GJ91] C. Gomard and N. Jones. A partial evaluator for the untyped lambda calculus. *J. Functional Programming*, 1(1):21–69, 1991.

[Gom92] C. Gomard. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, 1992.

[GR92] Marc Gengler and Bernhard Rytz. A polyvariant binding time analysis handling partially known values. In *Proc. Workshop on Static Analysis (WSA), Bordeaux, France*, pages 322–330, Sept. 1992.

[Gun92] Carl Gunter. *Semantics of Programming Languages — Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[Hat95] J. Hatcliff. A mechanised proof of correctness of off-line partial evaluation. In *These proceedings*, 1995.

[HM94] Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In Donald Sannella, editor, *Proceedings of European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, April 1994.

[HS91] S. Hunt and D. Sands. Binding Time Analysis: A New PERspective. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, pages 154–164, September 1991. ACM SIGPLAN Notices 26(9).

[Hun90] S. Hunt. PERs generalise projections for strictness analysis. In *Proceedings of the Third Glasgow Functional Programming Workshop*, Ullapool, 1990. Springer Workshops Series.

[Jen92] T. Jensen. *Abstract interpretation in logical form.* PhD thesis, Department of Computing, Imperial College, November 1992. (Available as DIKU tec. report 93/11).

[Jon88] N.D. Jones. Automatic program specialization: A re-examination from basic principles. In *[BEJ88]*, 1988.

[Lau88] J. Launchbury. Projections for specialisation. In *[BEJ88]*, 1988.

[Lau89] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD thesis, Department of Computing, University of Glasgow, 1989.

[Mog88] T. Mogensen. Partially static structures in a self-applicable partial evaluator. In *[BEJ88]*, 1988.

[Mog89] T. Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Diaz and F. Orejas, editors, *TAPSOFT '89 (LNCS 352)*, pages 298–312. Springer-Verlag, 1989.

[Mog92] Torben Æ. Mogensen. Self-applicable partial evaluation for pure lambda calculus. In Charles Consel, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 116–121. ACM, Yale University, 1992.

[NBV91] A. De Niel, E. Bevers, and K. De Vlamnick. Program bifurcation for polymorphically typed functional languages. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, September 1991. ACM SIGPLAN Notices 26(9).

[Nie85] F. Nielson. Program transformation in a denotational setting. *ACM Transactions on Programming Languages and Systems*, 7(3):359–379, 1985.

[Pal93] J. Palsberg. Correctness of binding time analysis. *Journal of Functional Programming*, 3(3):347–363, 1993.

[Ste94] P. Steckler. *Correct Higher-Order Program Transformations.* PhD thesis, College of Computer Science, Northeastern University, Boston, 1994. Tech Report NU-CCS-94-15.

[Tur86] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8:292–325, July 1986.

[Wan93] Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, July 1993. preliminary version appeared in *Conf. Rec. 20th ACM Symp. on Principles of Prog. Lang.* (1993), 137–143.