

A Semantic Model of Binding Times for Safe Partial Evaluation

Fritz Henglein David Sands
University of Copenhagen*

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.

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

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

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 eg. [Wan93] [Amt93] [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;
- utilisation of techniques for systematic design of correct analyses, namely *abstract interpretation* [CC79];
- 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. identify problems 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 simple off-line partial evaluators;
2. propose a new model for binding times which avoids the potential pitfalls of projections/PERs;
3. clarify the connection between extensional properties (local analysis) and program annotations (global analysis) necessary to support binding time analysis;
4. 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.

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 in [Mog89][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 [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 $\alpha : D \rightarrow D$ 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, ie. uniformly congruent.

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

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

which occurs in the program, then

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

Which 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

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

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

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

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:

$$\begin{array}{l} \text{letrec} \\ \quad g(v, w) = e \\ \quad f(x, y) = g(\text{if } y \text{ then } \Omega \text{ else } \Omega', y) \\ \text{in } f(i, j) \end{array}$$

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 $fst \stackrel{\text{def}}{=} \lambda \langle x, y \rangle. \langle x, \perp \rangle$. Based on this specification, the division

$$[f \mapsto fst, g \mapsto fst]$$

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 `if y then Ω else Ω'` 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 `if y then Ω else Ω'` 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.¹

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—and 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.² 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.”³

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 a top element. The rôle of the top elements is somewhat like an error element, but modelling errors that can occur exclusively at partial evaluation time. Partially static structures are handled by allowing data structures to contain top elements without them being identified with top. The top-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.

¹At the time of writing we have only checked this claim by inspecting the code reproduced in [Lau89], and not by executing it.

²However, 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]).

³In 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.

4.1 A Higher-order Functional Programming Language

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

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

$$\tau ::= \alpha \mid \text{unit} \mid \tau' \rightarrow \tau'' \mid \tau' \times \tau'' \mid \tau' + \tau'' \mid \text{rec } \alpha. \tau'$$

Recursive types must be *formally contractive*; that is, in $\text{rec } \alpha. \tau$ 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. The rules are included in the appendix. They are specified by inference rules 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; \rightarrow 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 $\text{rec } \alpha. \tau$ 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 $\llbracket e \rrbracket_{std} \in \llbracket \tau \rrbracket_{std}$, where $\llbracket e \rrbracket_{std}$ 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 *top elements* in a structural fashion: For every first-order type possessing a destructor operation we add a new “top” element, which, intuitively, represents *completely dynamic* values at the resulting type. 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 this 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 top element. We call the resulting domains *topped domains*, in order to distinguish them from the *standard domains* introduced earlier. The elements of the standard domain are then embedded in the topped domains as *ideals* and represent the *completely static* values of a type. Note, in particular, that every closed expression is mapped to a completely static value.

Structurally topped domain constructions Our binding time domains are Scott-domains with an isolated top element. A domain E plus an additional element \top_E , such that $x \sqsubseteq \top_E$ for all $x \in E$, is denoted by E^\top .

In what follows we will define the topped interpretation for types and terms. We will use $\llbracket \cdot \rrbracket$ to denote these mappings. Types are interpreted as follows:

$$\begin{aligned} \llbracket \text{unit} \rrbracket &= 1 \\ \llbracket \tau \rightarrow \tau' \rrbracket &= (\llbracket \tau \rrbracket \rightarrow \llbracket \tau' \rrbracket)^\top \\ \llbracket \tau \times \tau' \rrbracket &= (\llbracket \tau \rrbracket \times \llbracket \tau' \rrbracket)^\perp \\ \llbracket \tau + \tau' \rrbracket &= (\llbracket \tau \rrbracket + \llbracket \tau' \rrbracket)^\top \\ \llbracket \text{rec } \alpha. \tau \rrbracket &= \lim_{i \in \omega} F^i(1) \quad \text{where } F(D) = \llbracket \tau \rrbracket \text{ for } \llbracket \alpha \rrbracket = D \end{aligned}$$

Note that we add a new top element for every constructed domain with the exception of unit. The fact that unit is not topped reflects the fact that there is no destructor for it—and hence no notion of a partial evaluation error.

As an example, let us define $\text{bool} = \text{unit} + \text{unit}$. The standard interpretation of bool has the three elements $\text{true} = \text{inl}()$, $\text{false} = \text{inr}()$ and the bottom element \perp_{bool} . In the above extended interpretation, $\llbracket \text{bool} \rrbracket$ is the four-element “diamond” domain with elements $\{\perp, \text{true}, \text{false}, \top\}$, and ordered by $\perp \sqsubseteq x \sqsubseteq \top$ for all x in $\llbracket \text{bool} \rrbracket$.

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* (eg. $\pi_1 \cdot$, $\text{case } \cdot \text{ of } \dots$) maps the top element of the type being “destroyed” to top element of the resulting domain. Without giving the full details, the essence of the extension is characterised by the following semantic equations:

$$\left. \begin{aligned} \left[\begin{array}{l} \text{case } e \text{ of} \\ \text{inl } x \Rightarrow e' \\ \text{inr } y \Rightarrow e'' \end{array} \right] \rho &= \top \\ \llbracket e \ e' \rrbracket \rho &= \top \\ \llbracket \pi_1 e \rrbracket \rho &= \top \\ \llbracket \pi_2 e \rrbracket \rho &= \top \end{aligned} \right\} \text{if } \llbracket e \rrbracket \rho = \top$$

4.3 Binding times as ideals

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

1. downwards closed: $x \sqsubseteq y \in I \implies x \in I$; and
2. 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 an ideal I is said to be a binding time at type τ if I is an ideal over the domain $\llbracket \tau \rrbracket$. 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:

$$\begin{aligned} I \rightarrow I' &\stackrel{\text{def}}{=} \{f \in \llbracket \tau \rightarrow \tau' \rrbracket \mid f \neq \top \text{ and } (\forall d \in I) f(d) \in I'\} \\ I \times I' &\stackrel{\text{def}}{=} \{(d, d') \in \llbracket \tau \times \tau' \rrbracket \mid d \in I, d' \in I'\} \cup \{\perp\} \\ I + I' &\stackrel{\text{def}}{=} \{\text{inl } d \in \llbracket \tau + \tau' \rrbracket \mid d \in I\} \\ &\quad \cup \{\text{inr } d' \in \llbracket \tau + \tau' \rrbracket \mid d' \in I'\} \cup \{\perp\} \end{aligned}$$

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

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

4.4 Binding-time statements

We say expression e has binding time I and write $\models e : I$ if $\llbracket e \rrbracket \in I$. Let A be a mapping from program variables to binding times. We write $A \models e : I$ if for all environments ρ and variables x in the domain of A such that $\rho(x) \in A(x)$ we have $\llbracket e \rrbracket \rho \in I$.

“Dynamic” and “Static” At each non-trivial type τ we define an ideal Δ which represents the property “completely dynamic” at that type, and Σ , which represents “surface” static. We define Δ to be simply the whole domain $\llbracket \tau \rrbracket$, which, in particular, includes the top element $\top_{\llbracket \tau \rrbracket}$.

The binding time Σ at τ denotes $\llbracket \tau \rrbracket - \{\top_{\llbracket \tau \rrbracket}\}$; that is, all of $\llbracket \tau \rrbracket$ except for its top element. (Note that \top is isolated in every domain, so this *is* an ideal.) Intuitively, the elements in Σ are those that are “surface” static, in the sense that one can apply the corresponding destructor without getting \top as the result.

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

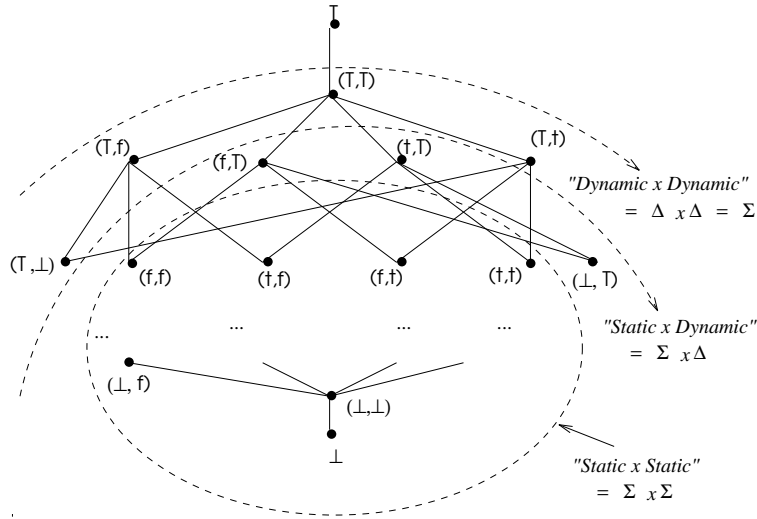


Figure 1: Example binding times in $\text{bool} \times \text{bool}$

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 and, 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 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, monovariance requires that the binding time of a bound variable and any subexpression be a fixed ideal, which must be “big enough” to capture the binding times of all the values the variables may be bound to and the subexpressions evaluate to.

This is in contrast to *polyvariant binding-time analysis* where, intuitively, 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 monovariance since monovariant binding-time analyses are currently better and easier understood in partial evaluation.⁴

5.1 Monovariant binding-time annotations

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. It is easy to check that $A \vdash e : I$ implies $A \models e : I$. Finally, Figure 4 adds a rule for *annotating* an expression with an *abstraction* of an ideal.

The annotations can be used to guide or influence the actions of a specialiser. In this sense they reify parts of the internalisation of inferring binding times. How much of the semantic information of an ideal is abstracted is expressed in the *abstraction function* α . Given a domain \mathcal{A} of *annotations* α is a monotonic function from the (power)domain of ideals to \mathcal{A} . The standard annotations we shall consider in the following section are $\mathcal{A}_f = \{S, D\}$ ordered by $S < D$.

Given an (open) expression e and any assumptions A on the binding times of free variables in e , there is a binding-time I such that $A \vdash e : I$ is derivable using the rules of Figures 2 and 3. Using the annotation rule in Figure 4 we can “add” annotations to subexpressions.

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.

⁴We believe the monovariant inference system given here can be extended to a polyvariant logic that is semantically complete, but leave this to future work.

$$\begin{array}{c}
A\{x \mapsto I\} \vdash x : I' \quad \frac{A\{x \mapsto I\} \vdash e : I'}{A \vdash \lambda x. e : I \rightarrow I'} \quad \frac{A \vdash e : I \rightarrow I' \quad A \vdash e' : I}{A \vdash e e' : I} \\
\\
\frac{A \vdash e : I \quad A \vdash e' : I'}{A \vdash (e, e') : I \times I'} \quad \frac{A \vdash e : I_1 \times I_2}{A \vdash \pi_i e : I_i} \quad \frac{A \vdash e : \Delta}{A \vdash \pi_i e : \Delta} \quad (i = 1, 2) \\
\\
\frac{A \vdash e : I}{A \vdash \text{inl } e : I + \emptyset} \quad \frac{A \vdash e : I}{A \vdash \text{inr } e : \emptyset + I} \\
\\
\frac{A \vdash e : I + \emptyset \quad A\{x \mapsto I\} \vdash e' : I'' \quad A\{x' \mapsto \Delta\} \vdash e'' : \Delta}{A \vdash \left(\begin{array}{c} \text{case } e \text{ of} \\ \text{inl } x \Rightarrow e' \parallel \\ \text{inr } x' \Rightarrow e'' \end{array} \right) : I''} \quad \frac{A \vdash e : \emptyset + I' \quad A\{x \mapsto \Delta\} \vdash e' : \Delta \quad A\{x' \mapsto I'\} \vdash e'' : I''}{A \vdash \left(\begin{array}{c} \text{case } e \text{ of} \\ \text{inl } x \Rightarrow e' \parallel \\ \text{inr } x' \Rightarrow e'' \end{array} \right) : I''} \\
\\
\frac{A \vdash e : I + I' \quad A\{x \mapsto I\} \vdash e' : I'' \quad A\{x' \mapsto I'\} \vdash e'' : I''}{A \vdash \left(\begin{array}{c} \text{case } e \text{ of} \\ \text{inl } x \Rightarrow e' \parallel \\ \text{inr } x' \Rightarrow e'' \end{array} \right) : I''} \quad \frac{A \vdash e : \Delta \quad A\{x \mapsto \Delta\} \vdash e' : \Delta \quad A\{x' \mapsto \Delta\} \vdash e'' : \Delta}{A \vdash \left(\begin{array}{c} \text{case } e \text{ of} \\ \text{inl } x \Rightarrow e' \parallel \\ \text{inr } x' \Rightarrow e'' \end{array} \right) : \Delta} \\
\\
\frac{A\{f \mapsto I\} \vdash e : I}{A \vdash \text{fix } f. e : I}
\end{array}$$

Figure 2: Monovariantly annotated expressions, nonlogical rules

$$\frac{A \vdash e : I}{A \vdash e : I'} \quad (I \subseteq I')$$

Figure 3: Logical rules

$$\frac{A \vdash e : I}{A \vdash e^b : I} \quad (b = \alpha(I))$$

Figure 4: Annotation rule

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

$$\begin{aligned}
(\lambda x. e_1) e &\rightarrow e_1\{e/x\} & \pi_1(e, e') &\rightarrow e & \pi_2(e, e') &\rightarrow e' \\
\left(\begin{array}{l} \text{case } \text{inl} (e) \text{ of} \\ \text{inl } x \Rightarrow e_1 \parallel \\ \text{inr } y \Rightarrow e_2 \end{array} \right) &\rightarrow e_1\{e/x\} & \left(\begin{array}{l} \text{case } \text{inr} (e) \text{ of} \\ \text{inl } x \Rightarrow e_1 \parallel \\ \text{inr } y \Rightarrow e_2 \end{array} \right) &\rightarrow e_2\{e/y\} \\
\text{fix } f. e_1 &\rightarrow e_1\{\text{fix } f. e_1/f\} & e^b &\rightarrow e
\end{aligned}$$

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 $e \rightarrow \dots \rightarrow e'$ in zero, one or more reduction steps.*

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. It requires two lemmas, the first of which explains the role of the “double-topped” range domains in the domains for function types.

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 $A \vdash e' : I$ can be obtained by “local” transformation of *any* proof of $A \vdash e : I$.

It is this latter requirement that accounts for the somewhat curious mapping of function types to function spaces with “double-topped” range domain. For first-order types we can identify binding time “dynamic” with the full underlying domain. This is not possible for higher-order types if we want to establish subject reduction for function applications — which is critical for arguing the safety of partial evaluation (see following section). The reason is that we would like to reason that $I \rightarrow I' \subseteq J \rightarrow J'$ only if $J \subseteq I$ and $I' \subseteq J'$. But this is not true if J' is *full*; i.e., the complete underlying domain. For full J' we have $I \rightarrow I' \subseteq J \rightarrow J'$ for *any* choice of $I, I', J!$

The following lemma shows, however, that our reasoning *is* admissible as long as J' is *not* full, at least if the underlying domain has a top element.

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 $I' \subseteq J'$.

PROOF. See appendix ◇

Lemma 5.2 explains the role of the *second* top in the ranges $D^{\top\top}$ of the function domains for function types. Dynamic values at function type are modelled by the function that yields the first (“lower”) top element of $D^{\top\top}$, instead of by a single element \top as in first-order types. The second (“upper”) top adds an extra element to the range of the function domain to make sure that we can reason “backwards” from containments between function ideals to containment relations between the domain and range ideals.

Lemma 5.3 (Substitution Lemma) *If $A\{x \mapsto I''\} \vdash e : I'''$ and $A \vdash e' : I'$ with $I' \subseteq I''$ and $I''' \subseteq I$ then $A \vdash e\{e'/x\} : I$.*

PROOF. By straightforward induction on e . ◇

Theorem 5.4 (Subject Reduction Theorem) *If $A \vdash e : I$ and $e \rightarrow^* e'$ then $A \vdash e' : I$.*

PROOF. The proof is in three parts. First we show that the theorem holds for a single reduction step where e is a redex. In the second part we show that it holds for $C[e]$ where e is a redex and $C[]$ is an arbitrary context. The final step is an induction on the length of reductions. In the appendix we elaborate the first two steps (the last one being straightforward). \diamond

6 Standard and Partial Evaluation

In this section we give a definition of off-line partial evaluation for this language, built by removing restrictions on reductions possible in the standard evaluation rules, and prove its safety from the 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 Def. 5.1, applying them to closed (unannotated) terms. 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 lambda-abstraction (ii) do not reduce under a fix-expression, (iii) do not reduce in the branches of a case expression. (iv) do not reduce under a constructor (pairing, or sum injections). Let \rightarrow_n be the resulting relation (call-by-name reductions). 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 $\llbracket e \rrbracket = \llbracket e' \rrbracket$
- 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 $\llbracket e \rrbracket = \perp$ 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 terms. “Dynamic” inputs are then just modelled by free variables.

Binding-time analysis is used to guide either the actions of a specializer (so-called off-line partial evaluation) or to optimize its actions (on-line partial evaluation). In both cases it is important to guarantee that the specializer 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 specialized 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 a term—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 specialized 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 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 terms 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 terms 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 “deconstructed” 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 term will be static), but dynamic destructors will not be reduced (since it cannot be trusted that the term 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* \mathcal{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 \mathcal{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 $e \equiv \mathcal{P}[e_1]$, for some \mathcal{P}, e_1 such that $e_1 \rightarrow_{pe} e_2$ and $\mathcal{P}[e_2] \equiv e'$.*

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 terms:

$$\begin{array}{c}
\lambda x^{[D]}. e_1 @^S e \rightarrow_{pe} e_1 \{e/x\} \quad \pi_1^S(e, e') \rightarrow_{pe} e \quad \pi_2^S(e, e') \rightarrow_{pe} e' \\
\left(\begin{array}{l} \text{case}^S \text{ inl } (e) \text{ of} \\ \text{inl } x^{[b_1]} \Rightarrow e_1 \parallel \\ \text{inr } y^{[b_2]} \Rightarrow e_2 \end{array} \right) \rightarrow_{pe} e_1 \{e'/x\} \quad \left(\begin{array}{l} \text{case}^S \text{ inr } (e) \text{ of} \\ \text{inl } x^{[b_1]} \Rightarrow e_1 \parallel \\ \text{inr } y^{[b_2]} \Rightarrow e_2 \end{array} \right) \rightarrow_{pe} e_2 \{e'/y\} \\
\text{fix } x^S. e_1 \rightarrow_{pe} e_1 \{\text{fix } x^S. e_1/x\}
\end{array}$$

Figure 5: Static Reduction Rules

$$\begin{array}{l}
\mathbb{P} ::= [] \mid \mathbb{P} @^b e' \mid e @^b \mathbb{P} \mid \pi_1^b \mathbb{P} \mid \pi_2^b \mathbb{P} \\
\quad \text{case}^b \mathbb{P} \text{ of} \quad \text{case}^b e \text{ of} \quad \text{case}^b e \text{ of} \\
\quad \text{inl } x^{[D]} \Rightarrow e_1 \parallel \quad \text{inl } x^D \Rightarrow \mathbb{P} \parallel \quad \text{inl } x^{[D]} \Rightarrow e_1 \parallel \\
\quad \text{inr } y^{[D]} \Rightarrow e_2 \quad \text{inr } y^{[D]} \Rightarrow e_2 \quad \text{inr } y^D \Rightarrow \mathbb{P} \\
\mid \lambda x^D. \mathbb{P} \mid \text{fix } x^D. \mathbb{P} \mid (\mathbb{P}, e') \mid (e, \mathbb{P}) \mid \text{inl } \mathbb{P} \mid \text{inr } \mathbb{P}
\end{array}$$

Figure 6: Partial Evaluation Contexts

Definition 6.2 (Destructors) *Define the destructors \mathbb{D} to be the following single-holed contexts:*

$$\mathbb{D} ::= [] e \mid \pi_1[] \mid \pi_2[] \mid \begin{array}{l} \text{case } [] \text{ of} \\ \text{inl } x \Rightarrow e_1 \parallel \\ \text{inr } y \Rightarrow e_2 \end{array}$$

All destructors occurring in a PE-annotated term must carry an annotation (S or D). We call these terms the PE-destructors. Let \mathbb{D}^S and \mathbb{D}^D respectively denote the static and dynamically annotated PE destructors. For example, if \mathbb{D} is the destructor $\pi_1^D[]$, then $\mathbb{D}^D[x]$ is the term $\pi_1^D x$. A static destructor tells the partial evaluator that it can expect that the term 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 (eg. a partial evaluator like Similix [Bon91]) is that when partial evaluation of the term in the destructor position has finished, it will be trusted that the result will be of the right kind to be “destroyed”. 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 are 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 , $\llbracket e \rrbracket \rho = \llbracket e' \rrbracket \rho$*

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 $\llbracket e \rrbracket_{val} \sqsubseteq_{val} \llbracket e' \rrbracket_{val}$) (in the manner of lambda-mix [GJ91]). However, this is not the aspect of safety that concerns the 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 $\mathcal{D}[e]$ is a well-typed term, and $\mathcal{D}^S[e]$ is not a partial evaluation redex, then either $e \equiv x$ or $e \equiv \mathcal{D}'[e']$ for some destructor \mathcal{D}' .*

PROOF. Simple case analysis on the possible types of e for each possible destructor \mathcal{D} . \diamond

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

1. $e \equiv \mathcal{P}[\mathcal{D}_1^S[x]]$ for some \mathcal{P} , \mathcal{D}_1 , x , or
2. $e \equiv \mathcal{P}[\mathcal{D}_1^S[\mathcal{D}_2^D[e']]]$ for some \mathcal{P} , \mathcal{D}_1 , \mathcal{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 term” by interpreting PE-annotated expressions as semantic annotations.

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

$$\begin{aligned} \widehat{x} &= x & \widehat{(e_1, e_2)} &= (\widehat{e_1}, \widehat{e_2}) & \widehat{\text{inl } e} &= \text{inl } \widehat{e} & \widehat{\text{inr } e} &= \text{inr } \widehat{e} \\ \widehat{\lambda x^{[D]}. e} &= \lambda x. (\widehat{e}\{x^{[D]}/x\}) & \widehat{e_1 @^b e_2} &= (\widehat{e_1})^b \widehat{e_2} & \widehat{\pi_1^b e} &= \pi_1(\widehat{e})^b & \widehat{\pi_2^b e} &= \pi_2(\widehat{e})^b \\ \widehat{\left(\begin{array}{l} \text{case}^b e \text{ of} \\ \text{inl } x_1^{[D]} \Rightarrow e_1 \parallel \\ \text{inr } x_2^{[D]} \Rightarrow e_2 \end{array} \right)} &= \text{case } (\widehat{e})^b \text{ of} \\ & \quad \text{inl } x_1 \Rightarrow (\widehat{e_1})\{x_1^{[D]}/x_1\} \parallel \\ & \quad \text{inr } x_2 \Rightarrow (\widehat{e_2})\{x_2^{[D]}/x_2\} \end{aligned}$$

So, for example, if e is the PE-annotated term $\lambda x^D. (x, x)$, 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 :

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

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

Definition 6.7 *An 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 $A_0 \vdash \widehat{e} : I$, where*

$$A_0 = \{x \mapsto \Delta_{A(x)} \mid x \text{ in the domain of } A\}.$$

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 term 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. First we need a couple of technical lemmas:

Lemma 6.8 *For all $A, I, I', A\{x \mapsto I'\} \vdash e\{x^D/x\} : I \iff A\{x \mapsto \Delta\} \vdash e : I$*

Lemma 6.9 *Extend $\widehat{\cdot}$ to operate on contexts, by specifying that $\widehat{[]} = []$. For all partial evaluation contexts \mathcal{P} and PE-annotated expressions e , if $\{x_1 \dots x_n\}$ are the variables captured by \mathcal{P} , then*

$$\widehat{\mathcal{P}[e]} = \widehat{\mathcal{P}}[\widehat{e}\{x_1^D \dots x_n^D/x_1 \dots x_n\}]$$

Proposition 6.10 *If e is well annotated and $e \equiv \mathbb{P}[e']$ then e' is well annotated.*

PROOF. Suppose e is well annotated. Then by definition $A_0 \vdash \widehat{\mathbb{P}[e']} : I$ for some I and some A_0 mapping free variables to Δ . Assume that variables captured by \mathbb{P} are $\{x_1 \dots x_n\}$. By Lemma 6.9 $A_0 \vdash \widehat{\mathbb{P}[\widehat{e'}\{x_1^D \dots x_n^D/x_1 \dots x_n\}]} : I$. By a straightforward induction on \mathbb{P} we can show that we must have

$$A_0[x_1 \mapsto I_1 \dots x_n \mapsto I_n] \vdash \widehat{e'}\{x_1^D \dots x_n^D/x_1 \dots x_n\} : I'$$

for some $I', I_1 \dots I_n$. Now by Lemma 6.8 we can conclude that $A_0\{x_1 \mapsto \Delta\} \dots \{x_n \mapsto \Delta\} \vdash \widehat{e'} : I'$ and hence that e' is well annotated. \diamond

Lemma 6.11 *A well annotated expression cannot be in an error state.*

PROOF. Suppose that e is in an error state. Then by definition either $e \equiv \mathbb{P}[\mathcal{D}_1^S[x]]$ or $e \equiv \mathbb{P}[\mathcal{D}_1^S[\mathcal{D}_2^D[e']]]$. Consider the second case (the first case is similar). Now suppose, towards a contradiction, that e is well annotated. By Prop. 6.10 it follows that $\mathcal{D}_1^S[\mathcal{D}_2^D[e']]$ is also well annotated. From the definition we can calculate that $\mathcal{D}_1^S[\widehat{\mathcal{D}_2^D[e']}] = \widehat{\mathcal{D}_1}[(\widehat{\mathcal{D}_2}[(e')^D])^S]$. Finally, by straightforward case analysis from the rules we conclude that there can be no inference of the form $A_0 \vdash \widehat{\mathcal{D}_1}[(\widehat{\mathcal{D}_2}[(e')^D])^S] : I$ for any A_0 (not containing unfeasible assumptions $x \mapsto \emptyset$), thus contradicting the assumption that e is well annotated. \diamond

Lemma 6.12 *If $e \mapsto_{pe} e'$ then $\widehat{e} \rightarrow^+ \widehat{e}'$*

Theorem 6.13 *If e is well annotated and $e \mapsto_{pe}^* e'$ then e' is not in an error state.*

PROOF. Assume e is well annotated – ie. that $A_0 \vdash \widehat{e} : I$ for some I . By Lemma 6.12 we have that $\widehat{e} \rightarrow^+ \widehat{e}'$, and so by Subject Reduction (Theorem 5.4) $A_0 \vdash \widehat{e}' : I$. Hence e' is well annotated, and so by Lemma 6.11 e' cannot be in an error state. \diamond

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:*

$$\text{If } \mathcal{D}^S[e] \mapsto_{pe} e' \text{ then } \mathbb{P}[\mathcal{D}^D[e]] \rightarrow_{on} \mathbb{P}[e']$$

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.

$$\begin{array}{c}
A\{x \mapsto I\} \vdash x : I \\
\\
\frac{A\{x \mapsto I\} \vdash e : I'}{A \vdash (\lambda x. e)^S : I \rightarrow I'} \quad \frac{A\{x \mapsto \Delta\} \vdash e : \Delta}{A \vdash (\lambda x. e)^D : \Delta} \\
\\
\frac{A \vdash e : I \rightarrow I' \quad A \vdash e' : I}{A \vdash (e^S) e' : I'} \quad \frac{A \vdash e : \Delta \quad A \vdash e' : \Delta}{A \vdash (e^D) e' : \Delta}
\end{array}$$

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

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

PROOF. Easy adaptation of the proof for the off-line case, since the analogue of Lemma 6.12 holds (because annotations can be deleted), and so the result follows directly from the Subject Reduction Theorem. \diamond

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 wrt. 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 $D_\infty \cong (D_\infty \rightarrow D_\infty)^\top$. As before, the semantic ideal Δ modelling “dynamic” (D) is the whole domain, whereas “static” (“ S ”) is interpreted as $D_\infty \rightarrow D_\infty$.

The binding-time analysis of Gomard and Jones can be described by an inference system consisting of rules that are derived 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.

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. Even so, it uses a “tagged”

representation for expressions at partial evaluation time and checks at specialization time whether the expressions indeed have the binding-time predicted by the binding-time analysis — which, indeed, they always have!

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. Since this extension promises interesting applications to constructor specialization and closure analysis, this appears to be a very promising avenue for further work.

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 a top element 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] [Lau88], and also properties of higher-order function. 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 annotation⁵. 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 were 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 (eg. code propagation across dynamic conditionals [Bon92]), unless they can be factored out in a pre-processing stage (eg.[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 internalization of this paper to a sound and complete polyvariant internalization using infinitary conjunction. This, we hope, should enable us to justify the above 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.
- [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.
- [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.

⁵This 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.”

- [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. Ernout 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 *Proceedings of PLILP'95*, Lecture Notes in Computer Science. Springer-Verlag, 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.

$$\begin{array}{c}
A \vdash x : A(x) \quad A \vdash () : \text{unit} \\
\\
\frac{A\{x \mapsto \tau\} \vdash e : \tau'}{A \vdash \lambda x : \tau. e : \tau \rightarrow \tau'} \quad \frac{A \vdash e : \tau \rightarrow \tau' \quad A \vdash e' : \tau}{A \vdash e e' : \tau'} \\
\\
\frac{A \vdash e : \tau \quad A \vdash e' : \tau'}{A \vdash (e, e') : \tau \times \tau'} \quad \frac{A \vdash e : \tau \times \tau'}{A \vdash \pi_1 e : \tau} \quad \frac{A \vdash e : \tau \times \tau'}{A \vdash \pi_2 e : \tau'} \\
\\
\frac{A \vdash e : \tau}{A \vdash \text{inl } e : \tau + \tau'} \quad \frac{A \vdash e : \tau'}{A \vdash \text{inr } e : \tau + \tau'} \\
\\
\frac{A \vdash e : \tau + \tau' \quad A\{x \mapsto \tau\} \vdash e' : \tau'' \quad A\{x' \mapsto \tau'\} \vdash e'' : \tau''}{A \vdash \left(\begin{array}{l} \text{case } e \text{ of} \\ \text{inl } x \Rightarrow e' \parallel \\ \text{inr } x' \Rightarrow e'' \end{array} \right) : \tau''} \\
\\
\frac{A\{f \mapsto \tau\} \vdash e : \tau}{A \vdash \text{fix } f. e : \tau}
\end{array}$$

Figure 8: Definition of simply-typed functional programming language

- [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.
- [Nie90] F. Nielson. Two-level semantics and abstract interpretation — fundamental studies. *Theoretical Computer Science*, (69):117–242, 90.
- [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.

A Appendix

In this section we provide some details omitted from the main text. Figure 8 gives the static semantics of the language.

Lemma A.1 (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 not full in D ; i.e., it is a proper subset of D .*

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

PROOF. **if:** This is the well-known contravariant containment rule for ideals.

only if: We prove this result by contradiction. Assume it doesn't hold that $J \subseteq I$ and $I' \subseteq J'$ whenever $I \rightarrow I' \subseteq J \rightarrow J'$. Then for some ideals I, I', J, J' either $J \subseteq I$ fails to hold or, if it does hold, $I' \subseteq J'$ fails. Let us consider these two cases in turn.

1. Assume that $J \not\subseteq I$; that is, there exists $d_J \in J$ such that $d_J \notin I$. Take a function $f \in I \rightarrow I'$. (Such an f exists since ideals are nonempty.) Let us define a function f' as follows:

$$f'(x) \stackrel{\text{def}}{=} \begin{cases} f(x), & \text{if } x \in I \\ \top, & \text{otherwise} \end{cases}$$

It can be checked that f' is continuous.

Now, $f' \in I \rightarrow I'$ since $f'(I) = f(I) \subseteq I'$, but $f' \notin J \rightarrow J'$. To wit, $d_J \in J$, but $f'(d_J) = \top \notin J'$ since J' cannot contain \top as it, otherwise, would have to be full. This shows that $I \rightarrow I' \not\subseteq J \rightarrow J'$, which is in contradiction to our assumption.

2. Assume that $I' \not\subseteq J'$; that is, there exists $d_{I'} \in I'$ such that $d_{I'} \notin J'$. Consider the constant function g defined by $g(x) \stackrel{\text{def}}{=} d_{I'}$. Clearly it is continuous and an element of $I \rightarrow I'$. It is not in $J \rightarrow J'$, however, since there is $x \in J$ such that $g(x) = d_{I'} \notin J'$. Thus $g \in I \rightarrow I' \not\subseteq J \rightarrow J'$, which contradicts our assumption.

This concludes the proof. ◇

Theorem A.2 (Subject Reduction Theorem) *If $A \vdash e : I$ and $e \rightarrow^* e'$ then $A \vdash e' : I$.*

PROOF. 1. Assume that e is a redex. We shall only consider the case where $e = (\lambda x. e_1) e_2$ as this is the most difficult case, which requires Lemma 5.2. The other cases are similar in principle, but easier. Without loss of generality we may assume that every application of a non-logical rule is followed by exactly one application of the weakening rule.

We have to show that $A \vdash (\lambda x. e_1) e_2 : \bar{I}$ implies $A \vdash e_1\{e_2/x\} : \bar{I}$. Assume $A \vdash (\lambda x. e_1) e_2 : \bar{I}$. This must have been derived by application of the rule for application yielding $A \vdash (\lambda x. e_1) e_2 : I$, followed by (a single) application of the weakening rule. Clearly it is sufficient to show that $A \vdash e_1\{e_2/x\} : I$

Since $A \vdash (\lambda x. e_1) e_2 : I$ is derived using the application rule there must be an ideal I' such that $A \vdash \lambda x. e_1 : I' \rightarrow I$ and $A \vdash e_2 : I'$. Furthermore, I must not contain all elements of the underlying domain.

The former judgement must be derivable from $A\{x \mapsto I''\} \vdash e_1 : I'''$ for some ideals I'', I''' , which gives $A \vdash \lambda x. e_1 : I'' \rightarrow I''' \subseteq I' \rightarrow I$. Since I is not full it follows by Lemma 5.2 that $I' \subseteq I''$ and $I''' \subseteq I$.

Since $A\{x \mapsto I''\} \vdash e_1 : I'''$ and $A \vdash e_2 : I'$ with $I' \subseteq I'', I''' \subseteq I$ it follows by the Substitution Lemma (Lemma 5.3) that $A \vdash e_1\{e_2/x\} : I$, which is what we had to show.

2. Let us assume now that $e = C[r]$ where $C[\]$ is a context and r is a redex with reduct r' . We want to show that if $A \vdash e : I$ then also $A \vdash e' : I$ for $e' = C[r']$.

Any proof of $A \vdash C[r] : I$ contains a judgement $A' \vdash r : I'$ for the occurrence of e in the context $C[\]$ since the proof rules are syntax-directed. In the first step we have shown that this implies $A' \vdash r' : I'$. Since the proof rules are syntax-directed we can build a proof of $A \vdash C[r'] : I$. (Formally this is done by induction on $C[\]$.)

This concludes the proof. ◇