Separate Abstract Interpretation for Control-Flow Analysis

Yan Mei Tang and Pierre Jouvelot

CRI, Ecole des Mines de Paris, France

Abstract. Effect systems and abstract interpretation are two methods to perform static analysis of programs. We present a new technique that builds upon the type and effect information of module signatures to extend abstract interpretation in the context of separate compilation. We use control-flow analysis as an application of this idea to support our claim.

Control-flow analysis strives to determine at compile time which functions, in a given call environment, may be called by a particular application expression. This static control-flow analysis can be expressed using either a type and effect system or abstract interpretation. The type and effect approach supports separate compilation but, being structural, collapses all call environments together, thus limiting the precision of control-flow information. By contrast, the abstract interpretation approach fails to support separate compilation but, because of its more operational nature, can distinguish between call environments, thus performing a more precise analysis.

We present a new static control-flow analysis that combines both techniques in a single framework. This *separate abstract interpretation* is as effective as the abstract interpretation approach on closed expressions, but is also able to tackle expressions with free variables, using their types to approximate their abstract values. We prove that this separate abstract interpretation analysis is a conservative extension of abstract interpretation.

1 Introduction

Effect systems [3, 4, 7, 8] and abstract interpretation [2, 6] are two methods for performing static analysis of programs. Abstract interpretation is based on the approximation of the fixed point nature of the language dynamic semantics while effect systems only rely on the local structure of program syntax. If the abstract interpretation approach performs more precise static analysis due to its more operational nature, effect systems support separate compilation more naturally via module signatures. We present a new technique, called *separate abstract interpretation*, to extend abstract interpretation in the context of separate compilation based on the type and effect information of module signatures. Types, enriched with effect information, are used to conservatively approximate abstract values of the free variables of programs, thus enabling abstract interpretation to be performed on non-closed expressions. We use control-flow analysis as a motivating example of this new idea.

Control-flow analysis strives to determine at compile time control- flow behaviors of expressions in given call environments. Its relative precision crucially depends on the accuracy of the static representation of the dynamic call environments. Shivers in [6] introduces the notion of *order* to indicate the number of pending calls remembered during the analysis of a given application expression.

Static control-flow analysis can be expressed using either a type and effect system [8] or an abstract interpretation [6]. The type and effect approach supports separate compilation but, being syntactic, collapses all call environments together, thus limiting the precision of control-flow information to 0th order analysis. By contrast, the abstract interpretation approach fails to support separate compilation but, because of its more operational nature, can distinguish between call environments, thus allowing more precise analysis. We present a new static control-flow analysis that combines both techniques in a single framework. It is as effective as the abstract interpretation approach on closed expressions, but is also able to tackle expressions with free variables, using their types to approximate their abstract values. Types describe the structure of values. In particular, from the latent definition d of a function type $t' \xrightarrow{d} t$, one can determine the set of functions this type may correspond to, together with their control-flow behavior. From such types, one can define conservative approximations of abstract values which are used to pursue the abstract interpretation. We have proved that this new static system conservatively extends the abstract interpretation system and retains all its properties.

In the remainder of the paper, we briefly discuss the related work (Section 2), give the syntax and semantics of our language (Section 3), recall both the abstract interpretation (Section 4) and the type and effect system (Section 5) for control-flow analysis, show how these two techniques can be merged together (Section 6) to perform separate abstract interpretation (Section 7), discuss optimizations for increasing its flexibity and accuracy (Section 8) before concluding (Section 9). Unless precised otherwise, all proofs are presented in the appendix.

2 Related Work

In Shivers′s thesis [6], control-flow analyses of arbitrary order (nCFA, where n is the order) on programs written in continuation-passing style (CPS) [1] are defined and performed by using an abstract interpretation approach. These control-flow analyses are able to distinguish different call environments but fail to support separate compilation, thus limiting their real-world application.

Effect systems extend type systems with effect information. Just as types describe the possible values of expressions, effects describe their possible evaluation behaviors. Our previous paper [8] presented a type and control-flow effect system where the inferred control-flow effects of expressions describe all control-flow traces possibly occurring during their evaluation. This analysis supports separate compilation but collapses call environments together, thus is less precise.

Here, we extend the abstract interpretation approach for 1CFA to support modularity, i.e. separate compilation, by approximating unknown value environ-

ments of expressions via their type environments. Thus our control-flow analysis performs 1CFA, and possibly nCFA, even in the presence of separate compilation.

3 Language

3.1 Syntax

A simple functional language suffices to present our ideas, although our analysis can be easily extended to more complicated languages.

$$\begin{array}{lll} \mathtt{e} ::= \mathtt{x} & & \textit{value identifier} \\ \quad (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) & & \textit{abstraction} \\ \quad (\mathtt{e}\ \mathtt{e}')_{\mathtt{l}} & & \textit{application} \end{array}$$

where function definitions and function calls are tagged with unique labels (`l` and `n`) that allow to uniquely distinguish them. How this labeling is actually performed is not important, as long as the uniqueness property is preserved. Note however that these labels will appear in types and, eventually, module type signatures, so they must be easily understandable by the user. Since abstraction expressions are uniquely identified by their function label, we sometimes only use labels where lambda expressions are expected.

$$\begin{array}{lll} \mathtt{n} \in \mathrm{LFun} = \mathrm{Label} & & \textit{label of functions} \\ \mathtt{l} \in \mathrm{LCall} = \mathrm{Label} & & \textit{label of function calls} \end{array}$$

Since Shivers's abstract interpretation approach uses CPS-transformed programs, we need to define an extended syntax for CPS programs. The main difference with the user language is the introduction of binary functions (to deal with continuation arguments) and the restriction of arguments to self-evaluating expressions.

$$\begin{array}{lll} \mathtt{a} ::= \mathtt{x} & & \textit{value identifier} \\ \quad (\lambda_{\mathtt{n}}\ (\mathtt{x}\ \mathtt{k})\ \mathtt{e}) & & \textit{user-defined function} \\ \quad (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) & & \textit{continuation function} \\ \\ \mathtt{e} ::= (\mathtt{a}\ \mathtt{a}'\ \mathtt{a}'')_{\mathtt{l}} & & \textit{function application} \\ \quad (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} & & \textit{continuation application} \end{array}$$

User-defined functions are always binary, while continuation functions are unary. In the sequel, without loss of generality, we only specify the semantics of unary functions and calls. By convention, we use `k` as identifiers of continuation functions.

3.2 Dynamic Semantics

The dynamic semantics not only defines the values of expressions, but also keeps track of control-flow information during evaluation. We restrict the presentation of the dynamic semantics to CPS expressions.

Following [6], we use the notion of contours to keep track of scoping information. A *contour* b is a list of labels of function calls describing the current call path. A *contour environment* (also called a *call environment*) β maps any variable to the call path that precedes its actual value binding. A *value* v is either an integer or a closure. A *closure* cl is composed of a lambda expression (including the function label, argument name and function body) and contour environment. A *binding environment* E is a finite map from pairs of identifiers and contours to values, recording the bindings of identifiers in a given contour.

$$
\begin{array}{lll}
b \in \text{Contour} & = \text{LCall}^* & \textit{contour} \\
\beta \in \text{ContourEnv} & = \text{Id} \mapsto \text{Contour} & \textit{contour environment} \\
v \in \text{Value} & = \text{Int} + \text{Closure} & \textit{value} \\
cl \in \text{Closure} & = \text{Fun} * \text{ContourEnv} & \textit{closure} \\
E \in \text{Binding} & = \text{Id} * \text{Contour} \mapsto \text{Value} & \textit{binding environment}
\end{array}
$$

The *control-flow information* records the set of functions called at a given call environment. It is defined as a set of tuples $\{(\mathtt{l}, \beta, s)\}$ where the functions in s are called at call site $\mathtt{l}$ and the call environment β; we write such tuples $\{(\mathtt{l}, \beta) \rightsquigarrow s\}$. In the dynamic semantics, this set is always a singleton. We use sets to be compatible with the subsequent non-standard semantics, where they usually have more than one element. The emptyset $\emptyset$ indicates the absence of control-flow information.

$$c \in \text{Control} = \mathcal{P}(\text{LCall} * \text{ContourEnv} * \mathcal{P}(\text{LFun})) \quad \textit{control-flow information}$$

The dynamic semantics is specified by a set of inference rules [5]. The usual value environment is split in Shivers' approach in two components: a contour environment β and a binding environment E. The purpose of this uncoupling is to separate the syntactic component of closures from their semantic aspect. This is of outmost importance when performing abstract interpretation where this syntactic component is furthermore restricted to finite expressions.

The inference rule $\beta, E \vdash \mathtt{a} \rightarrow v$ associates the argument $\mathtt{a}$ in the contour environment β and global binding environment E with the value v it evaluates to. In the (var) rule, the value of $\mathtt{x}$ is retrieved from the binding environment E according to the contour where it was bound (recorded by the current contour environment β). In the (abs) rule, the closure is built with its lambda expression and current contour environment. Note that we use the function label $\mathtt{n}$ instead of the lambda expression.

$$(var) : \beta, E \vdash \mathtt{x} \rightarrow E(\mathtt{x}, \beta(\mathtt{x}))$$

$$(abs) : \beta, E \vdash \mathtt{n} \rightarrow (\mathtt{n}, \beta)$$

The inference rule $b, \beta, E \vdash \mathtt{e} \to v, c$ associates the function application $\mathtt{e}$ in the current contour b, contour environment β and global binding environment E with (1) the value v it evaluates to and (2) the control-flow information c recording the control-flow traces during its evaluation. In the (*app*) rule, control reaches the function call site $\mathtt{l}$ in the contour environment β, binding environment E and contour b, where the function $\mathtt{n}$ is called. Then control enters the function body $\mathtt{e}$, whose control-flow information is c. Note that, for simplicity, the binding environment E is global to the whole expression evaluation.

$$(app): \frac{\begin{array}{l}\beta, E \vdash \mathtt{a} \to (\lambda_{\mathtt{n}}(\mathtt{x})\ \mathtt{e}, \beta') \\ \beta, E \vdash \mathtt{a}' \to v' \\ b' = b.\mathtt{l} \\ b', \beta'[\mathtt{x} \mapsto b'], E \vdash \mathtt{e} \to v, c \\ E(\mathtt{x}, b') = v'\end{array}}{b, \beta, E \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} \to v, c \cup \{(\mathtt{l}, \beta) \rightsquigarrow \{\mathtt{n}\}\}}$$

where $f[\mathtt{x} \mapsto v]$ is the extension of f with the property that $f[\mathtt{x} \mapsto v](\mathtt{x}) = v$ and $f[\mathtt{x} \mapsto v](\mathtt{y}) = f(\mathtt{y})$. $[\mathtt{x} \mapsto v]$ is shorthand for $[][\mathtt{x} \mapsto v]$ where $[]$ is the empty constant function.

4 Abstract Interpretation Semantics

In Shivers' thesis, first-order control-flow analysis (1CFA) is performed with an abstract interpretation. The contour of the dynamic semantic, which is a call path, is abstracted to a single call site, which is the last element of the call path. Shivers uses a denotational approach for specifying his analysis; we give here a new presentation of this technique using an operational framework which allows us to merge it nicely with the type and effect approach (see Section 6).

4.1 Definition

The abstract domains correspond to those in the dynamic semantics, except that, since control-flow analysis only deals with functions and ignores integers, values are abstracted to sets of abstract closures. The empty set $\emptyset$ represents any integer.

$\hat{b}$	$\in \widehat{\text{Contour}}$	$= \text{LCall}$	*contour*
$\hat{\beta}$	$\in \widehat{\text{ContourEnv}}$	$= \text{Id} \mapsto \widehat{\text{Contour}}$	*contour environment*
$\hat{v}$	$\in \widehat{\text{Value}}$	$= \mathcal{P}(\widehat{\text{Closure}})$	*abstract value*
$\widehat{cl}$	$\in \widehat{\text{Closure}}$	$= \text{Fun} * \widehat{\text{ContourEnv}}$	*closure*
$\hat{E}$	$\in \widehat{\text{Binding}}$	$= \text{Id} * \widehat{\text{Contour}} \mapsto \widehat{\text{Value}}$	*binding environment*
$\hat{c}$	$\in \widehat{\text{Control}}$	$= \mathcal{P}(\text{LCall} * \widehat{\text{ContourEnv}} * \mathcal{P}(\text{LFun}))$	*control-flow information*

The inference rule $\hat{\beta}, \hat{E} \vdash \mathtt{a} \to \hat{v}$ associates the argument $\mathtt{a}$ in the contour environment $\hat{\beta}$ and global binding environment $\hat{E}$ with the value $\hat{v}$ it evaluates to.

$$(var) : \hat{\beta}, \hat{E} \vdash \mathtt{x} \rightarrow \hat{E}(\mathtt{x}, \hat{\beta}(\mathtt{x}))$$

$$(abs) : \hat{\beta}, \hat{E} \vdash \mathtt{n} \rightarrow \{(\mathtt{n}, \hat{\beta})\}$$

The inference rule $\hat{\beta}, \hat{E} \vdash \mathtt{e} \rightarrow \hat{v}, \hat{c}$ associates the function application $\mathtt{e}$ in the contour environment $\hat{\beta}$ and global binding environment $\hat{E}$ with (1) the value $\hat{v}$ it evaluates to and (2) the control-flow information $\hat{c}$. In the (*app*) rule, when control reaches the function call site $\mathtt{l}$ in the contour environment $\hat{\beta}$ and binding environment $\hat{E}$, the function $\mathtt{a}$ is evaluated to a set of closures, while the actual argument $\mathtt{a}'$ is evaluated to its value $\hat{v}'$. Each function $\mathtt{n}_i$ is possibly called at $\mathtt{l}$ at the call environment $\hat{\beta}$ from which control transfers to its function body $\mathtt{e}_i$. Note that, compared to the dynamic semantics, the call path $\hat{b}'$ is limited to a single call site; so, calls to the same function but in different environments in the dynamic semantics may get merged together.

$$(app) : \frac{\begin{array}{l}\hat{\beta}, \hat{E} \vdash \mathtt{a} \rightarrow \{(\lambda_{\mathtt{n}_i}(\mathtt{x}_i)\ \mathtt{e}_i, \hat{\beta}_i') \mid i = 1 \ldots r\} \\ \hat{\beta}, \hat{E} \vdash \mathtt{a}' \rightarrow \hat{v}' \\ \hat{b}' = \mathtt{l} \\ \left.\begin{array}{l}\hat{\beta}_i'[\mathtt{x}_i \mapsto \hat{b}'], \hat{E} \vdash \mathtt{e}_i \rightarrow \hat{v}_i, \hat{c}_i \\ \hat{E}(\mathtt{x}_i, \hat{b}') = \hat{v}'\end{array}\right\} i = 1 \ldots r\end{array}}{\hat{\beta}, \hat{E} \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} \rightarrow \cup_{i=1}^{r} \hat{v}_i, \cup_{i=1}^{r} (\hat{c}_i \cup \{(\mathtt{l}, \hat{\beta}) \rightsquigarrow \{\mathtt{n}_i\}\})}$$

4.2 Correctness

Since the abstract interpretation semantics is defined on finite domains, it terminates. We prove it is well-formed and consistent w.r.t. the dynamic semantics.

Contour environments $\hat{\beta}$, global binding environments $\hat{E}$ and abstract values $\hat{v}$ are related. We define a well-formedness relation $\mathcal{WF}$ between them that ensures that free variables of abstract closures are appropriately bound:

Definition 1 (Well-Formedness).

$$\mathcal{WF}(\hat{v}, \hat{E}) \Leftrightarrow \forall (\mathtt{n}, \hat{\beta}) \in \hat{v},\ \ \mathcal{WF}(\hat{\beta}, \hat{E})$$
$$\mathcal{WF}(\hat{\beta}, \hat{E}) \Leftrightarrow \forall \mathtt{x} \in Dom(\hat{\beta}),\ \ (\mathtt{x}, \hat{\beta}(\mathtt{x})) \in Dom(\hat{E})\ \wedge\ \mathcal{WF}(\hat{E}(\mathtt{x}, \hat{\beta}(\mathtt{x})), \hat{E})$$

Using this definition, we prove that the abstract interpretation semantics is well-formed.

Lemma 2. *If* $\hat{\beta}, \hat{E} \vdash \mathtt{a} \rightarrow \hat{v}$ *and* $\mathcal{WF}(\hat{\beta}, \hat{E})$*, then* $\mathcal{WF}(\hat{v}, \hat{E})$*.*

Proof. By direct application of Definition 1.

Theorem 3 (Well-Formedness of Abstract Semantics). *If* $\hat{\beta}, \hat{E} \vdash \mathtt{e} \to \hat{v}, \hat{c}$ *and* $\mathcal{WF}(\hat{\beta}, \hat{E})$*, then*

- $(\hat{v}, \hat{E})$ *is well-formed.*
- *all* $(\hat{\beta}', \hat{E}')$ *used in the* $\to$ *derivation tree of* $\mathtt{e}$ *are well-formed.*

We define the $\leq$ relation as an approximation relation between abstract values: $(\hat{v}, \hat{E}) \leq (\hat{v}', \hat{E}')$ if $(\hat{v}', \hat{E}')$ is a conservative approximation of $(\hat{v}, \hat{E})$. This relation can be straightforwardly extended to compare exact and abstract values.

Definition 4 (Consistency of Abstract Values). For the well-formed $(\hat{v}, \hat{E})$, $(\hat{v}', \hat{E}')$, $(\hat{\beta}, \hat{E})$ and $(\hat{\beta}', \hat{E}')$,

$$(\hat{v}, \hat{E}) \leq (\hat{v}', \hat{E}') \Leftrightarrow \forall(\mathtt{n}, \hat{\beta}) \in \hat{v},\ \exists \hat{\beta}',\ s.t.\ (\mathtt{n}, \hat{\beta}') \in \hat{v}' \ \wedge\ (\hat{\beta}, \hat{E}) \leq (\hat{\beta}', \hat{E}')$$
$$(\hat{\beta}, \hat{E}) \leq (\hat{\beta}', \hat{E}') \Leftrightarrow \forall \mathtt{x} \in \mathit{Dom}(\hat{\beta}),\ \mathtt{x} \in \mathit{Dom}(\hat{\beta}') \wedge (\hat{E}(\mathtt{x}, \hat{\beta}(\mathtt{x})), \hat{E}) \leq (\hat{E}'(\mathtt{x}, \hat{\beta}'(\mathtt{x})), \hat{E}')$$

We next define the $\sqsubseteq$ relation as an approximation relation between abstract control-flow effects: $c \sqsubseteq c'$ if c' is a conservative approximation of c. In other words, c is a more precise control-flow information than c'.

Definition 5 (Accuracy of Abstract Effects).

$$c \sqsubseteq c' \ \Leftrightarrow\ \forall(\mathtt{l}, \hat{\beta}) \rightsquigarrow s \in c,\ \exists \hat{\beta}', s'\ \ s.t.\ (\mathtt{l}, \hat{\beta}') \rightsquigarrow s' \in c' \ \wedge\ s \subseteq s'$$

Using the previous definitions, we can express that the abstract semantics safely approximates the dynamic one for both arguments and expressions:

Lemma 6.

$$\left.\begin{array}{l} \beta, E \vdash \mathtt{a} \to v \\ \hat{\beta}, \hat{E} \vdash \mathtt{a} \to \hat{v} \\ (\beta, E) \leq (\hat{\beta}, \hat{E}) \end{array}\right\} \ \Rightarrow\ \ (v, E) \leq (\hat{v}, \hat{E})$$

Proof. By direct application of Definition 4.

Theorem 7 (Consistency of Abstract Semantics).

$$\left.\begin{array}{l} b, \beta, E \vdash \mathtt{e} \to v, c \\ \hat{\beta}, \hat{E} \vdash \mathtt{e} \to \hat{v}, \hat{c} \\ (\beta, E) \leq (\hat{\beta}, \hat{E}) \end{array}\right\} \ \Rightarrow\ \left\{\begin{array}{l} (v, E) \leq (\hat{v}, \hat{E}) \\ c \sqsubseteq \hat{c} \end{array}\right.$$

5 Effect System Semantics

We designed an effect system [8] to perform 0th-order control-flow analysis in which all call environments are collapsed together. We adapt below this system to CPS expressions.

5.1 Definition

A type t can either be the basic type int, a user-defined function type $(t' * t'') \xrightarrow{d} t$ or continuation function type $t' \xrightarrow{d} t$. The *latent definition* d is a set of possibly aliased functions $\mathtt{n}_i$ of the same data type, together with their control-flow effect $\bar{c}_i$. A type environment $\mathcal{E}$ is a finite map from identifiers to types.

$$\begin{array}{lll} d \in \text{Def} & = \{(\mathtt{n}, \bar{c})\} \mid d' \cup d & \textit{function definition} \\ t \in \text{Type} & = int \mid (t' * t'') \xrightarrow{d} t \mid t' \xrightarrow{d} t & \textit{type} \\ \mathcal{E} \in \text{TEnv} & = \text{Id} \mapsto \text{Type} & \textit{type environment} \\ \bar{c} \in \widehat{\text{Control}} & & \end{array}$$

The control-flow effect $\bar{c}$ of an expression records all the function calls that possibly occur during its evaluation. Since this type and effect semantics does not keep track of call environments, all contour environments that appear in the domain of control-flow effects are unknown, and thus denoted by the empty constant function $[]$.

The inference rule $\mathcal{E} \vdash \mathtt{a} : t$ associates the argument $\mathtt{a}$ in the type environment $\mathcal{E}$ with its type t. In the (abs) rule, the function label $\mathtt{n}$ paired with its control-flow effect $\bar{c}$ is added to the latent definition d of the function type. These rules use implicit subeffecting by adding more functions to d, thus allowing functions of the same data type to be gathered together. This can be used whenever a type mismatch occurs in an application.

$$(var) : \mathcal{E} \vdash \mathtt{x} : \mathcal{E}(\mathtt{x})$$

$$(abs) : \frac{\begin{array}{l} \mathcal{E}_{\mathtt{x}}[\mathtt{x} \mapsto t'] \vdash \mathtt{e} : t, \bar{c} \\ (\mathtt{n}, \bar{c}) \in d \end{array}}{\mathcal{E} \vdash (\lambda_{\mathtt{n}}\ (\mathtt{x})\ \mathtt{e}) : t' \xrightarrow{d} t}$$

The inference rule $\mathcal{E} \vdash \mathtt{e} : t, \bar{c}$ associates the function application $\mathtt{e}$ with its type t and control-flow effect $\bar{c}$. In the (app) rule, the latent definition of the function type is used to determine all the functions $\mathtt{n}$ possibly called at the call site $\mathtt{l}$ and their possible control-flow effect $\bar{c}$.

$$(app) : \frac{\begin{array}{l} \mathcal{E} \vdash \mathtt{a} : t' \xrightarrow{d} t \\ \mathcal{E} \vdash \mathtt{a}' : t' \end{array}}{\mathcal{E} \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} : t, \cup_{(\mathtt{n}, \bar{c}) \in d} (\bar{c} \cup \{(\mathtt{l}, []) \rightsquigarrow \{\mathtt{n}\}\})}$$

5.2 Correctness

We prove that the type and effect semantics is a conservative approximation of the abstract semantics, which means that the abstract interpretation performs more precise control-flow analysis than the effect system.

To define the consistency between the abstract interpretation and the effect system, we introduce the ":" relation between abstract values, abstract environments and types, noted as $(\hat{v}, \hat{E}) : t$. This can be easily extended to environments.

Definition 8 (Types of Abstract Values). For the well-formed $(\hat{v}, \hat{E})$,

$$
\begin{array}{l}
(\emptyset, \hat{E}) : \mathit{int} \\
(\hat{v}, \hat{E}) : t \quad \Leftrightarrow \forall (\mathtt{n}, \hat{\beta}) \in \hat{v},\ \exists \mathcal{E},\ s.t.\ (\hat{\beta}, \hat{E}) : \mathcal{E}\ \wedge\ \mathcal{E} \vdash \mathtt{n} : t \\
(\hat{\beta}, \hat{E}) : \mathcal{E} \quad \Leftrightarrow \forall \mathtt{x} \in \mathit{Dom}(\hat{\beta}),\ \mathtt{x} \in \mathit{Dom}(\mathcal{E})\ \wedge\ (\hat{E}(\mathtt{x}, \hat{\beta}(\mathtt{x})), \hat{E}) : \mathcal{E}(\mathtt{x})
\end{array}
$$

Using these definitions, we can express that the type semantic conservatively approximates the abstract semantics for both arguments and expressions.

Lemma 9.

$$
\left.
\begin{array}{l}
\mathcal{E} \vdash \mathtt{a} : t \\
\hat{\beta}, \hat{E} \vdash \mathtt{a} \rightarrow \hat{v} \\
(\hat{\beta}, \hat{E}) : \mathcal{E}
\end{array}
\right\} \quad \Rightarrow \quad (\hat{v}, \hat{E}) : t
$$

Proof. By direct application of Definition 8.

Theorem 10 (Types of Abstract Semantics).

$$
\left.
\begin{array}{l}
\mathcal{E} \vdash \mathtt{e} : t, \bar{c} \\
\hat{\beta}, \hat{E} \vdash \mathtt{e} \rightarrow \hat{v}, \hat{c} \\
(\hat{\beta}, \hat{E}) : \mathcal{E}
\end{array}
\right\} \quad \Rightarrow \quad
\left\{
\begin{array}{l}
(\hat{v}, \hat{E}) : t \\
\hat{c} \sqsubseteq \bar{c}
\end{array}
\right.
$$

6 Approximating Abstract Values From Types

As stated before, control-flow analysis by abstract interpretation is more precise than the one based on the type and effect inference system since it distinguishes between call environments. It however fails to support separate compilation because the value environments $\hat{\beta}$ and $\hat{E}$ are unknown for separately compiled expressions. Note that the type environments $\mathcal{E}$ would be available in this setting.

6.1 Approximation Function $\mathcal{A}$

The key idea is to determine a priori the unknown abstract value environment from the type environment, therefore extending the abstract interpretation technique to support separate compilation. The approximation function $\mathcal{A}$ takes a type t and returns its abstract value $\hat{v}$, along with a binding environment $\hat{E}$ that binds the free variables of $\hat{v}$. Abstract closures are thus either built from actual function definitions or approximated from function types.

The type *int* denotes integers; its abstract value is thus $\emptyset$ and its binding environment $[]$.

$$\mathcal{A}(\mathit{int}) = (\emptyset, [])$$

The function type $(t' * t_0) \xrightarrow{d} t_1$, where d is $\{(\mathtt{n}_i, \bar{c}_i) \mid i = 1 \ldots q\}$, describes a set of user-defined functions $\mathtt{n}_i$ with their control-flow effect $\bar{c}_i$ possibly occurring

when calling $\mathtt{n}_i$. Since the program is in CPS form, t_0 is a continuation type $t \xrightarrow{d'} t_1$ where t is the type of the result value passed to the final continuation. Thus the abstract value $\hat{v}'$ corresponding to the function type is a set of closures $\{(\lambda_{\mathtt{n}_i}(\mathtt{x}\ \mathtt{k}_i)\ \mathtt{e}_i), \hat{\beta}_i \mid i = 1 \ldots q\}$ in which the body $\mathtt{e}_i$ simulates the control-flow effect $\bar{c}_i$ and the contour environment $\hat{\beta}_i$ binds a fresh variable $\mathtt{x}_i$ to a fresh coutour $\mathtt{l}_i$. The binding environment $\hat{E}'$ corresponding to the function type maps the pair $\mathtt{x}_i$ and $\mathtt{l}_i$ to the abstract value $\hat{v}$ corresponding to the return type t. By binding $\mathtt{x}_i$ to $\hat{v}$ in $\hat{E}'$ and applying, in $\mathtt{e}_i$, the final continuation $\mathtt{k}_i$ to $\mathtt{x}_i$ (see below), the abstract value $\hat{v}$ of the result type is passed to its final continuation $\mathtt{k}_i$.

$$
\begin{array}{l}
\mathcal{A}(t' * t_0 \xrightarrow{d} t_1) = \mathit{let}\ \ \{\mathtt{x}_i\},\ \{\mathtt{l}_i\}\ \mathit{and}\ \mathtt{l}\ \mathit{fresh} \\
\qquad\qquad\qquad\qquad t_0 = t \xrightarrow{d'} t_1 \\
\qquad\qquad\qquad\qquad (\hat{v}, \hat{E}) = \mathcal{A}(t) \\
\qquad\qquad\qquad\qquad \{\mathtt{e}_i\} = \{\mathcal{S}(\bar{c}_i, \mathtt{k}_i, \mathtt{x}_i, \mathtt{l})\} \\
\qquad\qquad\qquad\qquad \hat{v}' = \{(\lambda_{\mathtt{n}_i}(\mathtt{x}\ \mathtt{k}_i)\ \mathtt{e}_i, [\mathtt{x}_i \mapsto \mathtt{l}_i]) \mid i = 1 \ldots q\} \\
\qquad\qquad\qquad\qquad \hat{E}' = \hat{E}[(\mathtt{x}_i, \mathtt{l}_i) \mapsto \hat{v} \mid i = 1 \ldots q] \\
\qquad\qquad\quad \mathit{in}\ (\hat{v}', \hat{E}') \\
\qquad\qquad\quad \mathit{where} \\
\qquad\qquad\quad d = \{(\mathtt{n}_i, \bar{c}_i) \mid i = 1 \ldots q\} \\
\qquad\qquad\quad \bar{c}_i = \{(\mathtt{l}_j, []) \rightsquigarrow \{\mathtt{n}_{j1} \ldots \mathtt{n}_{jr}\} \mid j = 1 \ldots s\}
\end{array}
$$

Each closure body $\mathtt{e}_i$ simulates the control-flow effect $\bar{c}_i$ where, for each call site $\mathtt{l}_j$, all $\mathtt{n}_{jk}$ functions may be called. The expression $\mathcal{S}(\bar{c}, \mathtt{k}, \mathtt{x}, \mathtt{l})$ simulates the control-flow effect $\bar{c}$ and, eventually, applies the continuation $\mathtt{k}$ to the result $\mathtt{x}$ at call site $\mathtt{l}$. It is defined by induction on control-flow effects as below:

$$
\begin{array}{ll}
\mathcal{S}([], \mathtt{k}, \mathtt{x}, \mathtt{l}) & = (\mathtt{k}\ \mathtt{x})_{\mathtt{l}} \\
\mathcal{S}(\bar{c}' \cup \{(\mathtt{l}', []) \rightsquigarrow \{\mathtt{n}_1 \ldots \mathtt{n}_r\}\}, \mathtt{k}, \mathtt{x}, \mathtt{l}) & = \mathcal{S}'(\{\mathtt{n}_1 \ldots \mathtt{n}_r\}, \bar{c}', \mathtt{l}', \mathtt{k}, \mathtt{x}, \mathtt{l}) \\
 & \\
\mathcal{S}'(\emptyset, \bar{c}', \mathtt{l}', \mathtt{k}, \mathtt{x}, \mathtt{l}) & = \mathcal{S}(\bar{c}', \mathtt{k}, \mathtt{x}, \mathtt{l}) \\
\mathcal{S}'(s' \cup \{\mathtt{n}'\}, \bar{c}', \mathtt{l}', \mathtt{k}, \mathtt{x}, \mathtt{l}) & = (\ (\lambda_{\mathtt{n}'}(\mathtt{k}')\ \mathcal{S}'(s', \bar{c}', \mathtt{l}', \mathtt{k}, \mathtt{x}, \mathtt{l}))\ \mathtt{k}\ )_{\mathtt{l}'} \\
 & \quad \mathit{where}\ \mathtt{k}'\ \mathit{is\ fresh}
\end{array}
$$

At each call site $\mathtt{l}'$ in $\bar{c}$, the function $\mathcal{S}$ recursively calls $\mathcal{S}'$ which is recursively defined on the set of functions $\{\mathtt{n}_1 \ldots \mathtt{n}_r\}$ possibly called at $\mathtt{l}'$. Simulating the behavior of $\bar{c}$ may require replicating call site labels; this is nonetheless acceptable here since this abstract value is automatically generated.

This general definition of $\mathcal{S}$ being somewhat notationally confusing, we give below an example of a closure body for the simple control-flow effect $\bar{c}$:

$$\bar{c} = \{(\mathtt{l}_1, []) \rightsquigarrow \{\mathtt{n}_1\}, (\mathtt{l}_2, []) \rightsquigarrow \{\mathtt{n}_2, \mathtt{n}_3\}\}$$

where the number of call sites is limited to two, and each call site can only call one or two functions. The corresponding closure body $\mathcal{S}(\bar{c}, \mathtt{k}_i, \mathtt{x}_i, \mathtt{l})$ is then:

$$
\begin{array}{l}
(\ (\lambda_{\mathtt{n}_1}(\mathtt{k}'_1) \\
\quad (\ (\lambda_{\mathtt{n}_2}\ (\mathtt{k}'_2) \\
\quad\quad (\ (\lambda_{\mathtt{n}_3}\ (\mathtt{k}'_3) \\
\quad\quad\quad\quad (\mathtt{k}_i\ \mathtt{x}_i)_{\mathtt{l}}) \\
\quad\quad\ \mathtt{k}_i\ )_{\mathtt{l}_2}) \\
\quad\ \mathtt{k}_i\ )_{\mathtt{l}_2}) \\
\mathtt{k}_i\ )_{\mathtt{l}_1}
\end{array}
$$

6.2 Correctness of $\mathcal{A}$

The approximation function $\mathcal{A}$ has the following properties :

Lemma 11 (Well-Formedness of $\mathcal{A}(t)$). *$\mathcal{A}(t)$ is well-formed.*

Note that the abstract values $\hat{v}'$ defined by $\mathcal{A}$ include simulated call environments whose domains contain only fresh variables. We thus extend the approximation relation $\leq$ to compare the abstract values and the approximated ones in the following way:

Definition 12 (Consistent Abstract Values and Environments). For the well-formed $(\hat{v}, \hat{E})$ and $(\hat{v}', \hat{E}')$, if $(\hat{v}', \hat{E}')$ is defined via $\mathcal{A}$, then

$$(\hat{v}, \hat{E}) \leq (\hat{v}', \hat{E}') \Leftrightarrow \forall(\mathtt{n}, \hat{\beta}) \in \hat{v},\ \exists \hat{\beta}',\ s.t.\ (\mathtt{n}, \hat{\beta}') \in \hat{v}'$$

Using this extended definition, we get:

Lemma 13 (Consistency of $\mathcal{A}(t)$). *If $(\hat{v}, \hat{E}) : t$, then $(\hat{v}, \hat{E}) \leq \mathcal{A}(t)$*

Proof. By direct application of Definition 12.

Since simulated call environments do not correspond to actual call environments, we define, for the purpose of comparing them, a function $\mathcal{D}$ to delete these simulated environments in the control-flow effects obtained by abstract interpretation.

$$
\begin{array}{ll}
\mathcal{D}([]) & = [] \\
\mathcal{D}(\hat{c} \cup \{(\mathtt{l}, \hat{\beta}) \rightsquigarrow s\}) & = \mathcal{D}(\hat{c}) \cup \{(\mathtt{l}, []) \rightsquigarrow s\}
\end{array}
$$

Using the initial identity continuation Id at a given call site $\mathtt{l}_k$, the abstract interpretation of any of the q expressions $\mathtt{e}_i$, built by the function $\mathcal{S}$ from the control-flow effect $\bar{c}_i$ given by the type semantics, yields a control-flow effect $\hat{c}_i$ which, modulo $\mathcal{D}$, is the *same* as $\bar{c}_i$.

Lemma 14 (Simulation). *For any $\hat{\beta}_1$ and $\hat{E}_1$, if*

$$\hat{\beta}_1[\mathtt{x}_i \mapsto \mathtt{l}_i][\mathtt{k}_i \mapsto \mathtt{l}_k], \hat{E}_1[(\mathtt{x}_i, \mathtt{l}_i) \mapsto \hat{v}][(\mathtt{k}_i, \mathtt{l}_k) \mapsto \{Id\}] \vdash \mathcal{S}(\bar{c}_i, \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) \rightarrow \hat{v}, \hat{c}_i$$

then $\mathcal{D}(\hat{c}_i) = \bar{c}_i \cup \{(\mathtt{l}, []) \rightsquigarrow \{Id\}\}$.

7 Separate Abstract Interpretation

Separate abstract interpretation uses types and effects to compute conservative approximations of abstract values of the free variables occurring in a separately compiled CPS expression `e`. These values are used to create initial environments in which the classical abstract interpretation is performed. These initial abstract value environments $\hat{\beta}_0$ and $\hat{E}_0$ are defined via the function $\mathcal{A}$, based on the type environment $\mathcal{E}$ of `e`.

Given a CPS expression `e`, its initial contour environment $\hat{\beta}_0$ maps its free variables to fresh call site labels, since their real binding call sites are unknown. Its initial binding environment $\hat{E}_0$ is defined not only on the free variables of `e`, but also on those introduced by $\mathcal{A}$; these additional identifiers are bound in the additional binding environments $\hat{E}$ given by $\mathcal{A}$.

$$\hat{\beta}_0 = [\mathtt{x} \mapsto \mathtt{l} \mid \mathtt{x} \in \mathit{Dom}(\mathcal{E}) \ \wedge \ \mathit{fresh}\ \mathtt{l}]$$

$$\hat{E}_0 = \cup_{\mathtt{x} \in \mathit{Dom}(\mathcal{E}) \ \wedge \ (\hat{v},\hat{E}) = \mathcal{A}(\mathcal{E}(\mathtt{x}))} \ \hat{E}[(\mathtt{x}, \hat{\beta}_0(\mathtt{x})) \mapsto \hat{v}]$$

where $\cup$ is the function union with the property that $(f \cup g)(\mathtt{x}) = f(\mathtt{x}) \cup g(\mathtt{x})$.

The approximated initial environments have the following properties, corresponding to those of the approximation function $\mathcal{A}$.

Lemma 15 (Well-Formedness of $(\hat{\beta}_0, \hat{E}_0)$). *$(\hat{\beta}_0, \hat{E}_0)$ is well-formed.*

Lemma 16 (Consistency of $(\hat{\beta}_0, \hat{E}_0)$). *If $(\hat{\beta}_0', \hat{E}_0') : \mathcal{E}$, then $(\hat{\beta}_0', \hat{E}_0') \leq (\hat{\beta}_0, \hat{E}_0)$*

Classical abstract interpretation can then simply be applied on `e` with these approximated initial environments:

$$\hat{\beta}_0, \hat{E}_0 \vdash \mathtt{e} \rightarrow \hat{v}, \hat{c}$$

to implement the notion of separate abstract interpretation. Thanks to these approximated environments, we extended the abstract interpretation approach to support separate compilation. This new interpretation enjoys all the properties of the abstract interpretation semantics presented above, i.e. it terminates and is well-formed. It is thus a conservative approximation of abstract interpretation.

Theorem 17 (Separate Abstract Interpretation). *Separate abstract interpretation is a conservative extension of abstract interpretation.*

8 Optimizations of Separate Abstract Interpretation

8.1 Flexibility of Abstract Semantics

The abstract interpretation semantics defined in Section 4 restricts a lambda expression `n` in the value environment $\hat{\beta}$, $\hat{E}$ to a singleton $\{(\mathtt{n}, \hat{\beta})\}$, which limits the number of programs derivable by the abstract semantics. To increase the

flexibility of the abstract semantics, we could adjust (abs) rule to (abs'), which allows a lambda expression to admit a larger abstract value as long as its type is perserved.

$$(abs'):\ \frac{\begin{array}{l}(\hat{\beta},\hat{E}):\mathcal{E}\\ \mathcal{E}\vdash \mathtt{n}:t\\ \mathcal{WF}(\hat{v},\hat{E})\\ (\hat{v},\hat{E}):t\end{array}}{\hat{\beta},\hat{E}\vdash \mathtt{n}\rightarrow\{(\mathtt{n},\hat{\beta})\}\cup\hat{v}}$$

By direct application of Definition 1 and Definition 8, we can see that (abs') rule perserves the properties of (abs), namely (1) well-formedness, i.e. if $\mathcal{WF}(\hat{\beta},\hat{E})$, then $\mathcal{WF}(\{(\mathtt{n},\hat{\beta})\}\cup\ \hat{v},\hat{E})$ and (2) typability, i.e $(\{(\mathtt{n},\hat{\beta})\}\cup\ \hat{v},\hat{E}):t$. Thus this new abstract semantics enjoys all of the properties (see Section 4 and Section 5) of the previous abstract semantics, but is more flexible. It terminates, is well-formed, is a conservative approximation of the dynamic semantics, and is more precise than the type semantics.

8.2 Local Control-Flow Effects

Even though the previously described approximation function $\mathcal{A}$ enables abstract interpretation to be applied in the presence of separate compilation, it has the major drawback of limiting its accuracy. Indeed, in the function types of CPS expressions, the control-flow effects in their latent definitions d represent not only the local control-flow effects of function bodies but also,via final continuation calls, those of the continuation of the program. Consequently, the accuracy of separate abstract interpretation is only as good as the one of the type and effect analysis.

To improve the analysis requires the use of $\mathcal{A}$ on types restricted to local control-flow effects. This can be achieved by computing the abstract values of the free variables on the basis of their direct, non-CPS type in the following way.

Using the previous notations, the continuation type $t_0 = t \xrightarrow{d'} t_1$, where $d' = \{(\mathtt{n}'_i,\bar{c}'_i)\mid i = 1\ldots p\}$, describes a set of continuation functions $\mathtt{n}'_i$ and their control-flow effect $\bar{c}'_i$. User-defined functions of type $(t' * t_0)\xrightarrow{d} t_1$, where $d=\{(\mathtt{n}_i,\bar{c}_i)\mid i=1\ldots q\}$, accept continuations of type t_0, beside the argument of type t'. The control-flow effects $\bar{c}_i$ of their bodies include the control-flow effects that correspond to applying the final continuation to their result. By subtracting this control-flow effect $\cup_{i=1}^{r}\bar{c}'_i$ from $\bar{c}_i$, the remaining effect only corresponds to the local control-flow effect of the body of the function $\mathtt{n}_i$. This is equivalent to the control-flow effect recorded in the corresponding *direct* function type, if one ignores all continuation calls in CPS types.

To summarize, given a non-closed expression $\mathtt{e}$, control-flow analysis using separate abstract interpretation is performed according to the following steps:

1. Apply type and effect inference to get the type environment $\mathcal{E}$ of $\mathtt{e}$.
2. Use the function $\mathcal{A}$ onto $\mathcal{E}$ to approximate the corresponding initial abstract value environment $(\hat{\beta}_0, \hat{E}_0)$.
3. Transform $\mathtt{e}$ to its CPS form $\mathtt{e}'$.
4. Apply the classical abstract interpretation algorithm to $\mathtt{e}'$, based on $(\hat{\beta}_0, \hat{E}_0)$, to get the control-flow information.

9 Conclusion

We presented a new static control-flow analysis that combines both abstract interpretation and type and effect systems in a single framework. It is as effective as the abstract interpretation approach on closed expressions, but is also able to tackle expressions with free variables, using their type to approximate their abstract value. We have thus extended abstract interpretation to support separate compilation by approximating unknown abstract value environments of expressions from their type environments.

We have proved that the control-flow information obtained by this new analysis is a conservative approximation of abstract interpretation and is more precise than the type and effect system.

References

1. Appel, A. W. Compiling with Continuations. Princeton University, 1992.
2. Cousot, P., and Cousot, R. Abstract Interpretation, a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*. 1977.
3. Jouvelot, P., and Gifford, D. K. Algebraic Reconstruction of Types and Effects. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*. ACM, New-York, 1991.
4. Lucassen, J. M., and Gifford, D. K. Polymorphic Effect Systems. In *Proceedings of the 1988 ACM Conference on Principles of Programming Languages*. ACM, New-York, 1988.
5. Plotkin, G. A structural approach to operational semantics. In *Technical report DAIMI-FN-19*. Aarhus University, 1981.
6. Shivers, O. Control-Flow Analysis of Higher-Order Languages. PhD thesis, CMU, May 1991.
7. Talpin, J. P., and Jouvelot, P. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming*, vol.2, no. 3, July 1992.
8. Tang, Y. M., and Jouvelot, P. Control-Flow Effects for Escape Analysis. *WSA'92, Bordeaux, France*, September 1992.

Appendix

Proof of Theorem 3 (Well-Formedness of Abstract Semantics)

Proof. By induction on the number of reduction steps of expressions.

– The hypotheses are
(1) $\hat{\beta}, \hat{E} \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} \to \cup_{i=1}^{r} \hat{v}_i, \cup_{i=1}^{r} (\hat{c}_i \cup \{(\mathtt{l}, \hat{\beta}) \rightsquigarrow \{\mathtt{n}_i\}\})$
(2) $\mathcal{WF}(\hat{\beta}, \hat{E})$

From hypothesis (1), by (*app*) in abstract semantics
(3) $\hat{\beta}, \hat{E} \vdash \mathtt{a} \to \{(\lambda_{\mathtt{n}_i}(\mathtt{x}_i)\ \mathtt{e}_i, \hat{\beta}_i^{'}) \mid i = 1 \ldots r\}$
(4) $\hat{\beta}, \hat{E} \vdash \mathtt{a}' \to \hat{v}'$
(5) $\hat{\beta}_i^{'}[\mathtt{x}_i \mapsto \hat{b}^{'}], \hat{E} \vdash \mathtt{e}_i' \to \hat{v}_i, \hat{c}_i$
(6) $\hat{E}(\mathtt{x}_i, \hat{b}^{'}) = \hat{v}'$
where $\hat{b}^{'} = \mathtt{l}$ and $i = 1 \ldots r$

From (2)(3)(4), by Lemma 2
(7) $\mathcal{WF}(\{(\lambda_{\mathtt{n}_i}(\mathtt{x}_i)\ \mathtt{e}_i, \hat{\beta}_i^{'}) \mid i = 1 \ldots r\}, \hat{E})$
(8) $\mathcal{WF}(\hat{v}', \hat{E})$

From (7), by Definition 1
(9) $\mathcal{WF}(\hat{\beta}_i^{'}, \hat{E})$

From (6)(8)(9), by Definition 1
(10) $\mathcal{WF}(\hat{\beta}_i^{'}[\mathtt{x}_i \mapsto \hat{b}^{'}], \hat{E})$

From (10), by induction
All $(\hat{\beta}^{'}, \hat{E}^{'})$ used in the $\to$ derivation tree of $\mathtt{e}$ are well-formed

From (5)(10), by induction
(11) $(\hat{v}_i, \hat{E})$ is well-formed

From (11), by Definition 1
$(\cup_{i=1}^{r} \hat{v}_i, \hat{E})$ is well-formed ♣

Proof of Theorem 7 (Consistency of Abstract Semantics)

Proof. By induction on the number of reduction steps of expressions.

– The hypotheses are
(1) $(\beta, E) \leq (\hat{\beta}, \hat{E})$
(2) $b, \beta, E \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} \to v, c \cup \{(\mathtt{l}, \beta) \rightsquigarrow \{\mathtt{n}\}\}$
(3) $\hat{\beta}, \hat{E} \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} \to \cup_{i=1}^{r} \hat{v}_i, \cup_{i=1}^{r} (\hat{c}_i \cup \{(\mathtt{l}, \hat{\beta}) \rightsquigarrow \{\mathtt{n}_i\}\})$

From hypothesis (2), by (app) in the dynamic semantics
(4) $\beta, E \vdash \mathtt{a} \to (\lambda_{\mathtt{n}}(\mathtt{x})\ \mathtt{e}, \beta')$
(5) $\beta, E \vdash \mathtt{a}' \to v'$
(6) $b', \beta'[\mathtt{x} \mapsto b'], E \vdash \mathtt{e} \to v, c$
(7) $E(\mathtt{x}, b') = v'$
where $b' = b.\mathtt{l}$

From hypothesis (3), by (app) in the abstract semantics
(8) $\hat{\beta}, \hat{E} \vdash \mathtt{a} \to \{(\lambda_{\mathtt{n}_i}(\mathtt{x}_i)\ \mathtt{e}_i, \hat{\beta}_i') \mid i = 1 \ldots r\}$
(9) $\hat{\beta}, \hat{E} \vdash \mathtt{a}' \to \hat{v}'$
(10) $\hat{\beta}_i'[\mathtt{x}_i \mapsto \hat{b}'], \hat{E} \vdash \mathtt{e}_i \to \hat{v}_i, \hat{c}_i$
(11) $\hat{E}(\mathtt{x}_i, \hat{b}') = \hat{v}'$
where $\hat{b}' = \mathtt{l}$ and $i = 1 \ldots r$

From (1)(4)(8) and (1)(5)(9), by Lemma 6
(12) $((\lambda_{\mathtt{n}}(\mathtt{x})\ \mathtt{e}, \beta'), E) \leq (\{(\lambda_{\mathtt{n}_i}(\mathtt{x}_i)\ \mathtt{e}_i, \hat{\beta}_i') \mid i = 1 \ldots r\}, \hat{E})$
(13) $(v', E) \leq (\hat{v}', \hat{E})$

From (12), by Definition 4, $\exists j\ (1 \leq j \leq r)$ *s.t.*
(14) $\lambda_{\mathtt{n}}(\mathtt{x})\ \mathtt{e} = \lambda_{\mathtt{n}_j}(\mathtt{x}_j)\ \mathtt{e}_j$
(15) $(\beta', E) \leq (\hat{\beta}_j', \hat{E})$

From (15)(7)(11)(13)(14), by Definition 4
(16) $(\beta'[\mathtt{x} \mapsto b'], E) \leq (\hat{\beta}_j'[\mathtt{x}_j \mapsto \hat{b}'], \hat{E})$

From (16)(6)(10), by induction
(17) $(v, E) \leq (\hat{v}_j, \hat{E})$
(17)$'$ $c \sqsubseteq \hat{c}_j$

From (17), by Definition 4
$(v, E) \leq (\cup_{i=1}^{r} \hat{v}_i, \hat{E})$

From (14)(17)$'$, by Definition 5
$c \cup \{(\mathtt{l}, \beta) \rightsquigarrow \{\mathtt{n}\}\} \sqsubseteq \cup_{i=1}^{r} (\hat{c}_i \cup \{(\mathtt{l}, \hat{\beta}) \rightsquigarrow \{\mathtt{n}_i\}\})$ ♣

Proof of Theorem 10 (Types of Abstract Semantics)

Proof. By induction on the number of reduction steps of expressions.

– The hypotheses are
(1) $(\hat{\beta}, \hat{E}) : \mathcal{E}$
(2) $\mathcal{E} \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} : t, \cup_{(\mathtt{n}, \bar{c}) \in d} (\bar{c} \cup \{(\mathtt{l}, []) \rightsquigarrow \{\mathtt{n}\}\})$
(3) $\hat{\beta}, \hat{E} \vdash (\mathtt{a}\ \mathtt{a}')_{\mathtt{l}} \to \cup_{i=1}^{r} \hat{v}_i, \cup_{i=1}^{r} (\hat{c}_i \cup \{(\mathtt{l}, \hat{\beta}) \rightsquigarrow \{\mathtt{n}_i\}\})$

From (2), by (app) in the type semantics
(4) $\mathcal{E} \vdash \mathtt{a} : t' \xrightarrow{d} t$
(5) $\mathcal{E} \vdash \mathtt{a}' : t'$

From (3), by (app) in the abstract semantics
(6) $\hat{\beta}, \hat{E} \vdash \mathtt{a} \rightarrow \{(\lambda_{\mathtt{n}_i}(\mathtt{x}_i)\ \mathtt{e}_i, \hat{\beta}_i') \mid i = 1 \ldots r\}$
(7) $\hat{\beta}, \hat{E} \vdash \mathtt{a}' \rightarrow \hat{v}'$
(8) $\hat{\beta}_i'[\mathtt{x}_i \mapsto \hat{b}'], \hat{E} \vdash \mathtt{e}_i \rightarrow \hat{v}_i, \hat{c}_i$
(9) $\hat{E}(\mathtt{x}_i, \hat{b}') = \hat{v}'$
where $\hat{b}' = \mathtt{l}$ and $i = 1 \ldots r$

From (1)(4)(6) and (1)(5)(7), by Lemma 9
(10) $(\{(\lambda_{\mathtt{n}_i}(\mathtt{x}_i)\ \mathtt{e}_i, \hat{\beta}_i') \mid i = 1 \ldots r\}, \hat{E}) : t' \xrightarrow{d} t$
(11) $(\hat{v}', \hat{E}) : t'$

From (10), by Definition 8, $\forall i\ (i = 1 \ldots r)$, $\exists \mathcal{E}_i'$ *s.t.*
(12) $(\hat{\beta}_i', \hat{E}) : \mathcal{E}_i'$
(13) $\mathcal{E}_i' \vdash (\lambda_{\mathtt{n}_i}\ (\mathtt{x}_i)\ \mathtt{e}_i) : t' \xrightarrow{d} t$

From (13), by (abs) in the type semantics, $\exists \bar{c}_i$
(14) $\mathcal{E}_{i\mathtt{x}_i}'[\mathtt{x}_i \mapsto t'] \vdash \mathtt{e}_i : t, \bar{c}_i$
(15) $(\mathtt{n}_i, \bar{c}_i) \in d$

From (12)(11)(9), by Definition 8
(16) $(\hat{\beta}_i'[\mathtt{x}_i \mapsto \hat{b}'], \hat{E}) : \mathcal{E}_{i\mathtt{x}_i}'[\mathtt{x}_i \mapsto t']$

From (16)(8)(14), by induction
(17) $(\hat{v}_i, \hat{E}) : t$
(17)′ $\hat{c}_i \sqsubseteq \bar{c}_i$

From (17), by Definition 8
$(\cup_{i=1}^{r} \hat{v}_i, \hat{E}) : t$

From (15)(17)′, by Definition 5
$\cup_{i=1}^{r}(\hat{c}_i \cup \{(\mathtt{l}, \hat{\beta}) \rightsquigarrow \{\mathtt{n}_i\}\}) \sqsubseteq \cup_{(\mathtt{n}, \bar{c}) \in d}(\bar{c} \cup \{(\mathtt{l}, []) \rightsquigarrow \{\mathtt{n}\}\})$ ♣

Proof of Lemma 11 (Well-Formedness of $\mathcal{A}(t)$)

Proof. By induction on the structure of types.

– Case int
By the definition of $\mathcal{A}$
$\mathcal{A}(int) = (\emptyset, [])$

By Definition 1
$\mathcal{A}(int)$ is well-formed

– Case $(t' * t_0) \xrightarrow{d} t_1$ where $t_0 = t \xrightarrow{d'} t_1$
By the definition of $\mathcal{A}$
$\mathcal{A}((t' * t_0) \xrightarrow{d} t_1) = (\hat{v}', \hat{E}')$
where
(1) $\hat{v}' = \{(\lambda_{\mathtt{n}_i}(\mathtt{x}\ \mathtt{k}_i)\ \mathtt{e}_i, [\mathtt{x}_i \mapsto \mathtt{l}_i]) \mid i = 1 \ldots q\}$
(2) $\hat{E}' = \hat{E}[(\mathtt{x}_i, \mathtt{l}_i) \mapsto \hat{v} \mid i = 1 \ldots q]$
(3) $(\hat{v}, \hat{E}) = \mathcal{A}(t)$

From (3), by induction
(4) $\mathcal{A}(t)$ is well-formed, i.e. $(\hat{v},\hat{E})$ is well-formed

From (2)(4), since $\mathtt{x}_i$ is fresh, by Definition 1
(5) $(\hat{v}, \hat{E}')$ is well-formed

From (5), by Definition 1, $\forall i$
(6) $([\mathtt{x}_i \mapsto \mathtt{l}_i], \hat{E}')$ is well-formed

From (6)(1)(2), by Definition 1
$\mathcal{A}((t' * t_0) \xrightarrow{d} t_1)$ is well-formed ♣

Proof of Lemma 14 (Simulation)

Proof. By induction on $\bar{c}_i$.

We note $\forall j \geq 1$
$\hat{\beta}'_j = \hat{\beta}_j[\mathtt{x}_i \mapsto \mathtt{l}_i][\mathtt{k}_i \mapsto \mathtt{l}_k]$
$\hat{E}'_j = \hat{E}_j[(\mathtt{x}_i, \mathtt{l}_i) \mapsto \hat{v}][(\mathtt{k}_i, \mathtt{l}_k) \mapsto \{Id\}]$

– Case $\bar{c}_i = []$
By the definition of $\mathcal{S}$
$\mathcal{S}([], \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) = (\mathtt{k}_i\ \mathtt{x}_i)_{\mathtt{l}}$

By the abstract semantics, since $\mathtt{k}_i$ is bound to Id
$\hat{\beta}'_1, \hat{E}'_1 \vdash (\mathtt{k}_i\ \mathtt{x}_i)_{\mathtt{l}} \rightarrow \hat{v}, \{(\mathtt{l}, \hat{\beta}'_1) \rightsquigarrow \{Id\}\}$

By the definition of $\mathcal{D}$
$\mathcal{D}(\{(\mathtt{l}, \hat{\beta}'_1) \rightsquigarrow \{Id\}\}) = \{(\mathtt{l}, []) \rightsquigarrow \{Id\}\}$

– Case $\bar{c}_i = \bar{c}'_i \cup \{(\mathtt{l}', []) \rightsquigarrow \{\mathtt{n}_1 \ldots \mathtt{n}_r\}\}$
By the definition of $\mathcal{S}$
(1) $\mathcal{S}(\bar{c}_i, \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) = \mathcal{S}'(\{\mathtt{n}_1 \ldots \mathtt{n}_r\}, \bar{c}'_i, \mathtt{l}', \mathtt{k}_i, \mathtt{x}_i, \mathtt{l})$

By the definition of $\mathcal{S}'$, $\forall j = 1 \ldots r$
(2) $\mathcal{S}'(\{\mathtt{n}_j \ldots \mathtt{n}_r\}, \bar{c}'_i, \mathtt{l}', \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) = (\ (\lambda_{\mathtt{n}_j}(\mathtt{k}_j)\ \mathcal{S}'(\{\mathtt{n}_{j+1} \ldots \mathtt{n}_r\}, \bar{c}'_i, \mathtt{l}', \mathtt{k}_k, \mathtt{x}_i, \mathtt{l}))\ \mathtt{k}_i\ )_{\mathtt{l}'}$
Note that $\{\mathtt{n}_{r+1} \ldots \mathtt{n}_r\} = \emptyset$

By induction on j, we prove that if
$\hat{\beta}'_j, \hat{E}'_j \vdash \mathcal{S}'(\{\mathtt{n}_j \ldots \mathtt{n}_r\}, \bar{c}'_i, \mathtt{l}', \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) \to \hat{v}, \hat{c}'$ then
$\mathcal{D}(\hat{c}') = \bar{c}'_i \cup \{(\mathtt{l}, []) \rightsquigarrow \{Id\}\} \cup \{(\mathtt{l}', []) \rightsquigarrow \{\mathtt{n}_j \ldots \mathtt{n}_r\}\}$
where
$\hat{c}' = \hat{c}'_i \cup \{(\mathtt{l}', \hat{\beta}'_j) \rightsquigarrow \{\mathtt{n}_j\}\} \ldots \{(\mathtt{l}', \hat{\beta}'_r) \rightsquigarrow \{\mathtt{n}_r\}\}$
$\hat{\beta}_{j+1} = \hat{\beta}_j[\mathtt{k}_j \mapsto \mathtt{l}']$
$\hat{E}_{j+1} = \hat{E}_j[(\mathtt{k}_j, \mathtt{l}') \mapsto \mathtt{k}_i]$

- Case $j = r + 1$
  By the definition of $\mathcal{S}'$, since $\{\mathtt{n}_{r+1} \ldots \mathtt{n}_r\} = \emptyset$
  (3) $\mathcal{S}'(\emptyset, \bar{c}'_i, \mathtt{l}', \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) = \mathcal{S}(\bar{c}'_i, \mathtt{k}_i, \mathtt{x}_i, \mathtt{l})$

  From (3), by induction on $\bar{c}_i$
  $\hat{\beta}'_{r+1}, \hat{E}'_{r+1} \vdash \mathcal{S}'(\emptyset, \bar{c}'_i, \mathtt{l}', \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) \to \hat{v}, \hat{c}'_i$ implies
  $\mathcal{D}(\hat{c}'_i) = \bar{c}'_i \cup \{(\mathtt{l}, []) \rightsquigarrow \{Id\}\}$

- Case $j = 1 \ldots r$
  By induction on j
  (4) $\hat{\beta}'_{j+1}, \hat{E}'_{j+1} \vdash \mathcal{S}'(\{\mathtt{n}_{j+1} \ldots \mathtt{n}_r\}, \bar{c}'_i, \mathtt{l}', \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) \to \hat{v}, \hat{c}'$ implies
  (5) $\mathcal{D}(\hat{c}') = \bar{c}'_i \cup \{(\mathtt{l}, []) \rightsquigarrow \{Id\}\} \cup \{(\mathtt{l}', []) \rightsquigarrow \{\mathtt{n}_{j+1} \ldots \mathtt{n}_r\}\}$
  where
  $\hat{c}' = \hat{c}'_i \cup \{(\mathtt{l}', \hat{\beta}'_{j+1}) \rightsquigarrow \{\mathtt{n}_{j+1}\}\} \ldots \{(\mathtt{l}', \hat{\beta}'_r) \rightsquigarrow \{\mathtt{n}_r\}\}$

  From (2)(4), by the abstract semantics
  $\hat{\beta}'_j, \hat{E}'_j \vdash \mathcal{S}'(\{\mathtt{n}_j \ldots \mathtt{n}_r\}, \bar{c}'_i, \mathtt{l}', \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) \to \hat{v}, \hat{c}' \cup \{(\mathtt{l}', \hat{\beta}'_j) \rightsquigarrow \{\mathtt{n}_j\}\}$

  From (5), by the definition of $\mathcal{D}$
  $\mathcal{D}(\hat{c}' \cup \{(\mathtt{l}', \hat{\beta}'_j) \rightsquigarrow \{\mathtt{n}_j\}\}) = \bar{c}'_i \cup \{(\mathtt{l}, []) \rightsquigarrow \{Id\}\} \cup \{(\mathtt{l}', []) \rightsquigarrow \{\mathtt{n}_j \ldots \mathtt{n}_r\}\}$

From (1), using the initial abstract value environments $\hat{\beta}'_1$ and $\hat{E}'_1$, we get :
$\hat{\beta}'_1, \hat{E}'_1 \vdash \mathcal{S}(\bar{c}_i, \mathtt{k}_i, \mathtt{x}_i, \mathtt{l}) \to \hat{v}, \hat{c}_i$ implies
$\mathcal{D}(\hat{c}_i) = \bar{c}_i \cup \{(\mathtt{l}, []) \rightsquigarrow \{Id\}\}$
where
$\hat{c}_i = \hat{c}'_i \cup \{(\mathtt{l}', \hat{\beta}'_1) \rightsquigarrow \{\mathtt{n}_1\}\} \ldots \{(\mathtt{l}', \hat{\beta}'_r) \rightsquigarrow \{\mathtt{n}_r\}\}$ ♣

Proof of Lemma 15 (Well-Formedness of $(\hat{\beta}_0, \hat{E}_0)$)

Proof. – By the definition of $(\hat{\beta}_0, \hat{E}_0)$
(1) $\forall \mathtt{x} \in Dom(\hat{\beta}_0),\ (\mathtt{x}, \hat{\beta}_0(\mathtt{x})) \in Dom(\hat{E}_0)$

(2) $\hat{E}_0(\mathtt{x}, \hat{\beta}_0(\mathtt{x})) = \hat{v}$
(3) $Dom(\hat{E}) \subseteq Dom(\hat{E}_0)$
(4) $(\hat{v}, \hat{E}) = \mathcal{A}(\mathcal{E}(\mathtt{x}))$

From (4), by Lemma 11
(5) $(\hat{v}, \hat{E})$ is well-formed

From (5)(3), by Definition 1
(6) $(\hat{v}, \hat{E}_0)$ is well-formed

From (1)(2)(6), by Definition 1
$(\hat{\beta}_0, \hat{E}_0)$ is well-formed ♣

Proof of Lemma 16 (Consistency of $(\hat{\beta}_0, \hat{E}_0)$)

Proof. – By the definition of $(\hat{\beta}_0, \hat{E}_0)$
(1) $\forall \mathtt{x} \in Dom(\mathcal{E}),\ \mathtt{x} \in Dom(\hat{\beta}_0)$
(2) $\hat{E}_0(\mathtt{x}, \hat{\beta}_0(\mathtt{x})) = \hat{v}$
(3) $Dom(\hat{E}) \subseteq Dom(\hat{E}_0)$
(4) $(\hat{v}, \hat{E}) = \mathcal{A}(\mathcal{E}(\mathtt{x}))$

Since $(\hat{\beta}_0', \hat{E}_0') : \mathcal{E}$, by Definition 8
(5) $\forall \mathtt{x} \in Dom(\hat{\beta}_0'),\ \mathtt{x} \in Dom(\mathcal{E})$
(6) $(\hat{E}_0'(\mathtt{x}, \hat{\beta}_0'(\mathtt{x})), \hat{E}_0') : \mathcal{E}(\mathtt{x})$

From (6), by Lemma 13
(7) $(\hat{E}_0'(\mathtt{x}, \hat{\beta}_0'(\mathtt{x})), \hat{E}_0') \leq \mathcal{A}(\mathcal{E}(\mathtt{x}))$

From (7)(4)
(8) $(\hat{E}_0'(\mathtt{x}, \hat{\beta}_0'(\mathtt{x})), \hat{E}_0') \leq (\hat{v}, \hat{E})$

From (8)(3), by Definition 4
(9) $(\hat{E}_0'(\mathtt{x}, \hat{\beta}_0'(\mathtt{x})), \hat{E}_0') \leq (\hat{v}, \hat{E}_0)$

From (5)(1)
(10) $\forall \mathtt{x} \in Dom(\hat{\beta}_0'),\ \mathtt{x} \in Dom(\hat{\beta}_0)$

From (10)(9)(2), by Definition 4
$(\hat{\beta}_0', \hat{E}_0') \leq (\hat{\beta}_0, \hat{E}_0)$ ♣