Fully Abstract Semantics of Additive Aspects by Translation

S. B. Sanjabi C.-H. L. Ong
Oxford University Computing Laboratory

Abstract

We study the denotational semantics of an aspect calculus by *compositional translation* to a functional language with higher-order store and ML-style references. The calculus is designed to construct only "additive" aspects i.e. those that do not elide the execution of the base computation. Such an aspect calculus is sufficiently expressive to encode `before()`, `after()` and `around()` advice which calls `proceed()`. We prove that our translation is *adequate* i.e. it reflects observational equivalence. Further if a standard object-oriented view of labels is adopted, the translation is *fully abstract* i.e. it preserves and reflects observational equivalence. A pleasing consequence is that full abstraction of the target-language semantics is thereby inherited by the source-language semantics. This yields the first fully abstract game model for a functional language of additive aspects.

1. Introduction

Aspect-oriented programming languages such as ASPECTJ [6] endow the programmer with the ability to associate a code fragment with a predicate matching a set of program points in an existing code base. Whenever program control reaches such a program point, the code fragment, called *advice*, is executed if the base program's execution state satisfies the predicate, known as a *pointcut*. In general, advice may influence the runtime behaviour of the base program, alter its final value, or even prevent it from executing altogether. Because of these features, aspect-oriented programs can be formidably hard to understand and reason about.

Denotational semantics by compositional translation

Researchers in programming languages have recently taken the first steps in understanding aspect-oriented computation: some have introduced prototypical calculi for studying aspectual features in purified form [4, 5]; others have proposed new features, or restrictions of existing features, that are designed to bring runtime behaviour under closer control [3, 8]. This paper is concerned with the denotational semantics of aspects that do not allow advice to suppress the execution of base code. Following [4], we shall call this class of aspects *additive* in that they only *add* code to the advised base computation; eg. harmless advice considered by Dantas and Walker in [3] is additive. We introduce a functional calculus of additive aspects – which is a slight variant of the Core Aspect Calculus of Walker et al. [4], and construct its denotational semantics by *compositional translation* to a functional language with higher-order store in the style of ML-references.

By *compositional*, we mean the property that the translate of a program is a composite of the translates of its component phrases; compositional translations are typically defined by recursion on program syntax. Because the translation is compositional, a denotational semantics of the target language can be transformed to a denotational semantics of the source language simply by precomposing the valuation function with the translation map. Further, the more faithful the translation, the greater the extent to which goodness-of-fit properties of the target-language semantics are inherited by the source-language semantics.

Translation and its faithfulness criteria

Fix a source language $\mathbb{S}$ and a target language $\mathbb{T}$. Let P, Q range over units of interest of $\mathbb{S}$ (be they terms-in-context, programs, configurations, etc.) and let M, N range over the corresponding units of $\mathbb{T}$. Assume that each language has a notion of evaluation, denoted by $\Downarrow$ annotated by the language name. We write $P \Downarrow_{\mathbb{S}} U$ to mean "P *evaluates* to value U". An important and compelling notion of program equivalence we shall study in the paper is observational equivalence. Intuitively two terms are *observationally equivalent*, written $P \simeq_{\mathbb{S}} Q$, just if one can be replaced by the other in *all* programs without causing any observable difference in the computational outcome. (For a precise formulation, see Definition 3.1.)

A basic property one asks of a translation $\ulcorner - \urcorner$ from units of $\mathbb{S}$ to those of $\mathbb{T}$ is the following:

> **Property E** (*Preservation and reflection of evaluation*). *For every P and U, $P \Downarrow_{\mathbb{S}} U$ iff for some V, $\ulcorner P \urcorner \Downarrow_{\mathbb{T}} V$ and $V \simeq_{\mathbb{T}} \ulcorner U \urcorner$.*

Writing $P{\Downarrow_{\mathbb{T}}}$ to mean "there is some U to which P evaluates", sometimes a weaker property is sufficient for our purpose:

> **Property T** (*Preservation and reflection of termination*). *For every P, $P{\Downarrow_{\mathbb{S}}}$ iff $\ulcorner P \urcorner {\Downarrow_{\mathbb{T}}}$.*

In case a translation satisfies Property E (respectively T), the target language can be regarded as an emulator of the evaluation relation (respectively termination property) of the source language.

A good translation should preserve and reflect stronger behavioural properties than termination. We call a translation *adequate* if it reflects observational equivalence, and *fully abstract* if, in addition, it preserves observational equivalence.

> **Property A** (*Adequacy*). *For every P and Q, if $\ulcorner P \urcorner \simeq_{\mathbb{T}} \ulcorner Q \urcorner$ then $P \simeq_{\mathbb{S}} Q$.*
>
> **Property F** (*Full Abstraction*). *For every P and Q, $P \simeq_{\mathbb{S}} Q$ iff $\ulcorner P \urcorner \simeq_{\mathbb{T}} \ulcorner Q \urcorner$.*

An important aim of this paper is to exhibit a fully abstract translation. A useful technical condition, often a pre-condition of full abstraction, is the following:

AOSD '07 March 12-16, Vancouver, British Columbia, Canada

Pproperty D (*Definability*). *For every $M \in \mathbb{T}$, there is some $P \in \mathbb{S}$ such that $\ulcorner P \urcorner \simeq_{\mathbb{S}} M$.*

This states that, modulo observational equivalence, the translation map is a surjection.

Recall that a denotational model for a language is *fully abstract* if the equational theory of the model coincides with observational equivalence. Fully abstract models are thus powerful and highly accurate models. An important reason for studying fully abstract translation is that it offers a way to build a fully abstract semantics for a language. Riecke [9] and McCusker [7] have shown that if the translation is adequate (respectively fully abstract), then the semantics inherited by the source language enjoys the property of adequacy (respectively full abstraction) provided the semantics of the target language does.

Contributions of the paper

This paper studies the semantics of a language of additive aspects, called CoreAML. Our approach is by a compositional translation $\ulcorner - \urcorner$ from CoreAML to a language with higher-order store in the style of ML-references [16], called GRef*. Both the source and the target languages share a common core of a call-by-value simply-typed λ-calculus. In the target language GRef*, for each type τ, the reference type

$$\mathtt{ref}[\tau] \;=\; (\mathbb{1} \to \tau) \times (\tau \to \mathbb{1})$$

is identified with the product of its read method (which has type $\mathbb{1} \to \tau$) and its write method (which has type $\tau \to \mathbb{1}$), in an object-oriented style. A consequence of the identification is the presence of so-called "bad references" (or bad variables). Even though every term of type $\mathtt{ref}[\tau]$ can be assigned to and dereferenced, not all terms of the type behave as *bona fide* references (for example, reads need not be causally related to writes). We show that the translation from CoreAML to GRef* preserves and reflects evaluation and termination; it is also adequate (i.e. it reflects observational equivalence) and has the definability property. But full abstraction fails; we trace this to a fundamental mismatch between the labels of type τ (say) and their translates, which are references of type $\ulcorner \tau \urcorner \to \ulcorner \tau \urcorner$ (and so, just pairs of read and write methods).

To fix the problem, we "object-orientate" the types of labels (and hence aspects) in CoreAML into their component accessor methods, much as the types of references in GRef* are products of the corresponding read and write methods. In other words we endow the source language CoreAML with the power to construct "bad labels" – the analogue of bad references, and call the augmented language CoreAML*. We show that the new translation from CoreAML*-to-GRef* does have the desired properties (T, D, A and F). Further, since the target language GRef* has a fully abstract semantics [16], we obtain a fully abstract semantics for the aspect language CoreAML* *by translation*. We tabulate the main technical results of the paper concerning the various translation in Theorem 6.4.

We summarise the main contributions of our work as follows.

- We have given *compositional translations* from functional languages of additive aspects, CoreAML and CoreAML*, to a language with higher-order store and ML-style reference GRef*.
- We have examined the *faithfulness* of the translations (i.e. the extent to which behavioural properties of the source language are preserved and reflected by the translation) and calibrated it in terms of Properties E, T, D, A and F.
- We have constructed the first *fully abstract* (game) model for a functional language of additive aspects (with an object-oriented view of labels) CoreAML* by the translation method.
- Our method gives an *adequate* (game) model for CoreAML (and hence Walker's MinAML), in which one can reason soundly (but not completely) about observational equivalence in the aspect language.

Previous models of aspect orientation, while covering larger segments of the paradigm, are not as fitting and accurate as the game semantics presented in our work. Wand et al. [12] define a monadic semantics for dynamic join points but they do not prove any of its formal properties. Closer to our work, Andrews [10] translates a language of static aspects into a CSP-like process algebra, but does not address the goodness-of-fit of the semantics. We believe that our results are new and the translation method can be extended to model non-additive aspects and dynamic pointcuts.

Outline of the paper

The remainder of this paper is structured as follows: Section 2 gives an overview of Walker et al.'s aspect calculus; Section 3 defines the target and source languages; Sections 4 and 5 define the translations and examine their properties; Section 6 presents a fully abstract translation and gives a complete picture of the "translation space"; and finally Section 7 concludes and discusses future directions.

2. The core aspect calculus

This Section gives an informal introduction to a language of additive aspect called CoreAML. In their ICFP03 paper [4], Walker et al. define a simple aspect-oriented programming language called MinAML; it extends a small functional language with AspectJ-like features such as before, after, and around advice, using function calls as the only join points. The semantics of MinAML is given by translation to a core aspect calculus called Core Mini AML. What we shall call CoreAML – our object of study – is this calculus, but less the `return` operator, thus restricting it to additive advice.

CoreAML has a distinguished type $\mathtt{lab}[\tau]$, whose values are taken from a countable set of *labels*. New labels are created (in block structured scopes) by a `new` command. CoreAML also has a type $\mathtt{asp}[\tau]$ of *aspects*, whose values take the form $\{\ell.x \to e\}$, where ℓ is a label, and e (the *advice*) is a term of the same type as the variable x. Terms are evaluated in an environment comprising a sequence of aspects A, which can be extended by the constructs $a \ll e$ and $a \gg e$ – the former prepends the aspect a to the head of the sequence (and then proceeds to evaluate e) and the latter appends the aspect to the tail of A.

Labelled program points $\ell\langle e \rangle$ are an essential construct of CoreAML, providing the mechanism for aspect substitution. In aspect-orientation terminology, they form the join points of the language. After e has been evaluated to a value v, the term $\ell\langle e \rangle$ causes any advice associated with the label ℓ to be triggered. If there are aspects of the form $\{\ell.x \to e'\}$ in A, then $\ell\langle e \rangle$ reduces to $e'[v/x]$.

Labels can be used to tag distinct control flow points; thus the term $(\ell\langle v_1 \rangle \;\mathtt{AND}\; \ell\langle v_2 \rangle)$ triggers two instances of the advice in $a \equiv \{\ell.x \to e\}$, passing the value v_1 in one instance and the value v_2 in the other. In general, several pieces of advice may be associated to the same label ℓ. In this case, evaluation of $\ell\langle v \rangle$ results in v being passed to the *composition* of all of the advice associated to ℓ (i.e. advice whose pointcut is ℓ) in the order specified in the list A. The importance of advice ordering is illustrated by the following evaluations (assuming for the moment that the language has natural number arithmetic):

$$\mathtt{newlab}\ \ell\ \mathtt{in}\ \{\ell.x \to x+1\} \ll \{\ell.y \to y^2\} \ll \ell\langle 3 \rangle \quad \Downarrow \quad 10$$

$$\mathtt{newlab}\ \ell\ \mathtt{in}\ \{\ell.x \to x+1\} \ll \{\ell.y \to y^2\} \gg \ell\langle 3 \rangle \quad \Downarrow \quad 16$$

In the first case, the incrementing advice is installed at the head of the advice sequence, followed by the installation of the advice

which squares its input *ahead of it*. Thus, when $\ell\langle 3\rangle$ is evaluated, it is first squared, then incremented. In the second case, the squaring advice is installed at the tail of the list, and thus is executed *after* the increment.

These simple features already endow the underlying simply-typed λ-calculus of CoreAML with the ability to write non-terminating programs. Let v be a value and consider the following term

$$\texttt{newlab}\ \ell\ \texttt{in}\ \{\ell.x \to \ell\langle x\rangle\} \ll \ell\langle v\rangle$$

which has an infinite reduction sequence, even though the underlying term (the base code) is already a value. In fact, this trick of invoking a given aspect from within its own body can be used to encode recursive functions. Take, for example, the following naïve definition of the exponential function b^a:

$$\begin{aligned}\texttt{exp}(b,a) \triangleq\ & \texttt{newlab}\ \ell\ \texttt{in}\\ & \{\ell.x \to \texttt{cond}\ (x=0)\ \ (1)\ \ (b * \ell\langle x-1\rangle)\} \gg\\ & \ell\langle a\rangle\end{aligned}$$

Note how closely the syntax of labelled join points and their associated aspects resembles that of named procedure calls. However the code behaves quite differently from a stack-based recursive procedure. The term installs advice that implements the body of the basic exponential function, and then invokes it immediately with the input argument. The recursive "call" is made by invoking the advice from within its own body with a decremented argument. This implementation is, of course, quite inefficient, as a new piece of advice is installed each time the `exp` function is called, but this is required in order to ensure that no external references to the label ℓ are possible. and hence better understood

The power of aspects does not stop at recursion. Walker et al. [4] give a term that uses aspects to implement a reference cell. We adopt the same technique to prove our definability result (Proposition 5.7). Indeed the object of our work is to make this connection precise, by showing that not only can these aspectual features emulate references, but that the reverse is also possible.

A remark on obliviousness

We take the Core Aspect Calculus of Walker et al. to be our prototypical aspect-oriented language, even although it does not enjoy *obliviousness* [13]. We would argue that this is not a deficiency of our approach because obliviousness is not a linguistic feature (such as assignment); rather it refers to the situation that the "base code" programmer is unaware that his code may be advised. For instance, the surface language MinaML of Walker et al. would be considered oblivious; however, a programmer who is aware that MinaML functions are join points could apply "dummy" identity functions to arbitrary program points in order to artificially invoke advice. This is exactly the feature provided by labels in the aspect calculus of Walker et al. While obliviousness may be a useful practical abstraction, it has arguably no real effect on the formal semantics of the language.

3. Language definitions

This Section defines the source and target languages of the compositional (and fully abstract) translation. We begin with a brief treatment of the core λ-calculus that underlies both of them.

A core functional calculus $\mathcal{L}$

The common fragment shared by both languages is a typed, call-by-value λ-calculus $\mathcal{L}$ whose types and expressions are given by the following grammars:

$$\begin{aligned}\tau,\sigma,\rho \quad &::= \quad \mathbb{1} \mid \texttt{bool} \mid \sigma\times\tau \mid \sigma\to\tau\\ e,f,g \quad &::= \quad x \mid \texttt{skip} \mid \texttt{tt} \mid \texttt{ff} \mid \texttt{cond}\, e\, e_1\, e_2 \mid\\ &\qquad\quad \langle e_1,e_2\rangle \mid \pi_i(e) \mid \lambda x^\tau.e \mid e_1\cdot e_2\end{aligned}$$

The typing rules and operational semantics are standard. We shall use $\vdash e$ to mean that the term e is well-typed i.e. that $\Gamma\vdash e:\tau$ is provable for some Γ and τ. The type $\mathbb{1}$ is the unit type (i.e. the type of commands) with the sole value `skip`. The operational semantics is given in terms of an evaluation relation $\Downarrow$ between expressions and *values* i.e. terms generated by the following grammar:

$$v,u \ ::= \ \texttt{skip} \mid \texttt{tt} \mid \texttt{ff} \mid \langle u,v\rangle \mid \lambda x^\tau.e$$

We equate terms up to α-renaming of bound names. Given a sequence $\overline{g}$ of terms of type $\tau\to\tau$, we write $g_1\circ\cdots\circ g_n$ to mean the sequential composition of the terms in the sequence i.e.

$$\lambda x^\tau.(g_1\cdot(g_2\cdot(\ \cdots\ \cdot(g_n\cdot x)\ \cdots)))$$

CoreAML: A language of additive aspects

Given a countable set of *label names* ranged over by $\mathtt{l}$, CoreAML adds to $\mathcal{L}$ a type of labels and a type of aspects, together with the respective constructs to generate (locally scoped) labels, label program points, and create and install aspects. The types, expressions, and values of this language are defined by adding the following rules to those defining $\mathcal{L}$

$$\begin{aligned}\tau,\sigma,\rho \quad &::= \quad \cdots \mid \texttt{lab}[\tau] \mid \texttt{asp}[\tau]\\ e,f,g \quad &::= \quad \cdots \mid \texttt{newlab}_\tau \mid \{e_1.x\to e_2\}\\ &\qquad\quad \mid e_1\langle e_2\rangle \mid e_1 \gg e_2\\ v,u \quad &::= \quad \cdots \mid \mathtt{l} \mid \{v.x\to e\}\end{aligned}$$

Though label names appear in the syntax, they only arise as the outcome of an evaluation (i.e. values), and may not appear in user programs. The typing rules and operational semantics are presented in Table 1. By a *configuration*, we mean a triple of the form $(\mathsf{L},\mathsf{A},e)$, comprising a map L from labels to types, a sequence A of aspects, and a term e to be evaluated; in case the expression e is a value, we shall call the configuration a *value configuration*. The operational semantics of CoreAML is given in terms of an evaluation relation between configurations and value configurations; note that label mappings and aspect sequences are omitted from the rules where possible so as to keep the presentation concise. The rules defining the operational semantics of $\mathcal{L}$ are naturally extended to the corresponding configurations (i.e. these rules make no changes to the L or A components). The rules use the notation $a :: \mathsf{A}$ and $\mathsf{A} :: a$ to denote the sequence obtained by respectively prepending and appending the element a to the sequence A. The empty sequence is denoted by ε, $\texttt{labs}(e)$ is defined as the set of label names appearing in the term e, $\texttt{dom}(F)$ as the domain of the map F, and $F\{x\mapsto n\}$ as the same map as F except that $x\in\texttt{dom}(F)$ is now mapped to n. Finally, for an aspect sequence A, we write $\mathsf{A}(\mathtt{l})$ as the subsequence $\overline{g}$ of $\mathtt{l}$-labelled advice occuring in A.

A term e is *closed* if it contains no free variables, and *open* otherwise. We refer to terms that contain no label names as *user terms*. A *program* is a closed user-term of ground type. A configuration $P\equiv(\mathsf{L},\mathsf{A},e)$ is *well-typed* (denoted $\vdash P$) if each $a\in\mathsf{A}$ is of type $\texttt{asp}[\tau]$ for some τ, $\texttt{labs}(e)\subseteq\texttt{dom}(\mathsf{L})$, and the mapping L correctly realises the typing of the labels of e. The semantics also uses an auxiliary family of functions $\mathbb{C}_{\mathsf{L},\mathsf{A}}$ which, given a label $\mathtt{l}$, return the term v obtained by functionally composing all $\mathtt{l}$-labelled advice in A in sequential order. We use upper-case letters P, Q to range

Type System

$$\frac{\Gamma \vdash e_1 : \mathtt{lab}[\tau] \qquad \Gamma, x:\tau \vdash e_2 : \tau}{\Gamma \vdash \{e_1.x \to e_2\} : \mathtt{asp}[\tau]}\ (\text{TP ASPECT}) \qquad \frac{}{\Gamma \vdash \mathtt{newlab}_\tau : \mathtt{lab}[\tau]}\ (\text{TP NEWLAB})$$

$$\frac{\Gamma \vdash e_1 : \mathtt{asp}[\sigma] \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \gg e_2 : \tau}\ (\text{TP INSTALL}) \qquad \frac{\Gamma \vdash e_1 : \mathtt{lab}[\tau] \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\langle e_2\rangle : \tau}\ (\text{TP INVOKE})$$

Operational Semantics

$$\frac{e_1 \Downarrow (\mathsf{L}, \mathsf{A}, a) \qquad (\mathsf{L}, \mathsf{A} :: a, e_2) \Downarrow v}{e_1 \gg e_2 \Downarrow v}\ (\Downarrow \text{INSTALL}) \qquad \frac{e_1 \Downarrow v}{\{e_1.x \to e\} \Downarrow \{v.x \to e\}}\ (\Downarrow \text{ASPECT})$$

$$\frac{}{(\mathsf{L}, \mathsf{A}, \mathtt{newlab}_\tau) \Downarrow (\mathsf{L}\{\mathtt{l} \mapsto \tau\}, \mathsf{A}, \mathtt{l})}\ \mathtt{l} \notin \mathtt{dom}(\mathsf{L})\ (\Downarrow \text{NEWLAB})$$

$$\frac{e_1 \Downarrow (\mathsf{L}, \mathsf{A}, \mathtt{l}) \qquad e_2 \Downarrow u \qquad e[u/x] \Downarrow v}{e_1\langle e_2\rangle \Downarrow v}\ [\mathbb{C}_{\mathsf{L},\mathsf{A}}(\mathtt{l}) = \lambda x.e]\ (\Downarrow \text{INVOKE})$$

$$\mathbb{C}_{\mathsf{L},\mathsf{A}} \triangleq \lambda \mathtt{l}. \begin{cases} \mathtt{id}[\mathsf{L}(\mathtt{l})] & \text{if } \mathtt{l} \in \mathtt{dom}(\mathsf{L}) \text{ and } \mathsf{A}(\mathtt{l}) = \varepsilon \\ g_n \circ \cdots \circ g_1 & \text{if } \mathtt{l} \in \mathtt{dom}(\mathsf{L}) \text{ and } \mathsf{A}(\mathtt{l}) = \overline{g} \\ \bot & \text{if } \mathtt{l} \notin \mathtt{dom}(\mathsf{L}) \end{cases}$$

Table 1. Typing Rules and Operational Semantics of CoreAML

over well-typed configurations of CoreAML, and an upper-case U to range over value configurations.

The evaluation relation $\Downarrow$ coincides with the transition semantics (denoted here by $\rightarrowtail$) defined for CoreAML in [4][1] i.e.

Proposition 3.1. *For any well-typed configuration P, we have $P \Downarrow U$ if and only if $P \rightarrowtail^* U$, writing $\rightarrowtail^*$ for the reflexive, transitive closure of $\rightarrowtail$.*

For a well-typed configuration P, we write $P\Downarrow$ (read "P converges") if there exists a configuration U such that $P \Downarrow U$ and $P\Uparrow$ otherwise. We also use this notation for well-typed user terms, writing $e\Downarrow$ if $(\bot, \varepsilon, e)$ converges and $e\Uparrow$ for divergence.

Observational equivalence is a compelling notion of program equivalence. Intuitively we say that two terms are observationally equivalent just in case one term can be replaced by the other in every program without any observable difference in the computational outcome.

Definition 3.1. Precisely we say that two well-typed user terms e_1 and e_2 of CoreAML are *observationally equivalent* if for any context $\mathcal{C}[-] : \mathbb{1}$ such that $\mathcal{C}[e_i]$ is a program, we have

$$\mathcal{C}[e_1] \Downarrow \quad \Longleftrightarrow \quad \mathcal{C}[e_2] \Downarrow$$

and denote this situation by $e_1 \simeq e_2$.

We can define syntactic sugar for sequential composition, the `let` construct, and block structured labels (`new ... in ...`) by using λ abstraction as follows:

$$\begin{aligned} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 &\triangleq (\lambda x.e_2) \cdot e_1 \\ \mathtt{newlab}\ x\ \mathtt{in}\ e &\triangleq (\lambda x.e) \cdot \mathtt{newlab} \\ e_1; e_2 &\triangleq (\lambda d.e_2) \cdot e_1 \quad (d \notin \mathtt{fv}(e_2)) \end{aligned}$$

Finally we use $\mathtt{id}[\tau]$ as a shorthand for the identity function $\lambda x^\tau.x$ of type τ.

[1] For expository purposes, we omit Walker's other installation primitive to the head of the aspect sequence. While we could easily include it, it is not used in the encoding of MinAML, nor is it needed to encode general references.

GRef*: A language of general references

The language GRef* is defined by extending $\mathcal{L}$ with higher-order store in the style of ML-references (the "*" superscript in the language indicates the presence of "bad references", for which more anon). Given a countable set of *references* whose elements are ranged over by $\mathtt{r}$, the expressions of the language are defined by extending the expression grammar of $\mathcal{L}$ with the following rules:

$$e ::= \cdots \mid \mathtt{newref}_\tau(e) \mid e_1 := e_2 \mid\ !e$$

The typing rules and operational semantics of these constructs are defined in Table 2. Rather than introducing a distinguished type for references, we take an approach now standard in (game) semantics of viewing the reference type $\mathtt{ref}[\tau]$ as a product of a "read method" and a "write method", *a la* Reynolds [14]. The write method assigns a value to the location and has type $\tau \to \mathbb{1}$; the read method retrieves the value currently stored there and has type $\mathbb{1} \to \tau$. Thus we use the following shorthand:

$$\mathtt{ref}[\tau] \quad \triangleq \quad (\tau \to \mathbb{1}) \times (\mathbb{1} \to \tau)$$

A consequence of the identification is the presence of so-called "bad references" (or bad variables). Even though every term of type $\mathtt{ref}[\tau]$ can be assigned to and dereferenced, not all terms of the type behave as *bona fide* references (for example, reads need not be causally related to writes).

Remark 3.1. To our knowledge, all known *fully abstract* game models of Algol-like languages interpret reference types as products of read and write methods. Consequently bad references live in all these game models. Interested readers may wish to consult [16, §2.4] for a discussion on whether the presence of bad references should be regarded as a defect.

The operational semantics of GRef* is defined as an evaluation relation between triples $(\mathsf{R}, \mathsf{S}, e)$ comprising a map R from references to types, a store S mapping references to values, and a term e to be evaluated. The rules omit the R and S components of the triple when they play no role in a particular evaluation. The grammar of values is extended with reference names:

$$v ::= \cdots \mid \mathtt{r}$$

By definition a reference is a pair, so we must be able to project it in order to retrieve its "components"; the rules ($\Downarrow$REFPROJ$_i$) give us the capability. The rules ($\Downarrow$DEREF-BAD) and ($\Downarrow$ASSIGN-BAD) respectively define the behaviour of executing the read and write operations on a bad reference. The "*" superscript in the language indicates the presence of bad references. The language GREF (without the superscript) is defined by regarding the reference type $\mathtt{ref}[\tau]$ as primitive (as opposed to a composite of other types), and removing the rules for bad-reference assignment and dereferencing, and those for projections over references.

We use upper-case letters M and N to range over well-typed configurations of GREF*, and V to range over value configurations. As for COREAML, we write $M\Downarrow$ if M is well-typed and there exists a V such that $M \Downarrow V$, and write $e\Downarrow$ just if $(\bot, \bot, e)\Downarrow$, where $\bot$ is the map that leaves all elements in its domain undefined. Observational equivalence of GREF* terms is defined as for COREAML.

Shorthands for the `let` construct, sequentical composition, and the identity function are defined as in COREAML, and the `new` ... `in` ... construction for references is defined as

$$\mathtt{newref}\ x = e_1\ \mathtt{in}\ e_2 \ \triangleq\ (\lambda x.e_2)\cdot\mathtt{newref}(e_1)$$

The language GREF* is identical to the language of Abramsky et al. [16] except for the absence of distinguished reference types (and hence also the absence of an explicit "bad reference" constructor `mkref`), and the fact that we initialize newly created references. These differences have no semantic consequences. In fact, the game model of the language of Abramsky et al. is fully abstract for GREF*.

Remark 3.2. The preceding statement deserves an explanation. In the language of Abramsky et al., for each type τ, there is a *primitive* reference type $\mathtt{ref}[\tau]$; in addition there is a "bad reference" constructor that takes a write method $e_1 : \tau \to \mathbb{1}$ and a read method $e_2 : \mathbb{1} \to \tau$ and cast them as a reference $\mathtt{mkref}\,\langle e_1, e_2\rangle : \mathtt{ref}[\tau]$. Crucially the reference type $\mathtt{ref}[\tau]$ is isomorphic to the product type $(\tau \to \mathbb{1}) \times (\mathbb{1} \to \tau)$

$$\mathtt{ref}[\tau] \underset{\mathtt{mkref}}{\overset{\mathscr{P}_{\mathcal{R}}(-)}{\rightleftarrows}} (\tau \to \mathbb{1}) \times (\mathbb{1} \to \tau)$$

The isomorphism is witnessed in one direction by the bad-reference constructor `mkref`; in the other direction, a term e of reference type $\mathtt{ref}[\tau]$ can be tranformed to a pair of type $(\tau \to \mathbb{1}) \times (\mathbb{1} \to \tau)$ as follows:

$$\mathscr{P}_{\mathcal{R}}(e) \quad \triangleq \quad \langle\, \lambda x^{\tau}.(e := x)\ ,\ \lambda d^{\mathbb{1}}.(!e)\,\rangle$$

The two transformations are inverse of each other, modulo observational equivalence; that is to say, we have

$$\mathtt{mkref}\,\mathscr{P}_{\mathcal{R}}(e) \simeq e \quad : \quad \mathtt{ref}[\tau]$$
$$\mathscr{P}_{\mathcal{R}}(\mathtt{mkref}\,\langle f, g\rangle) \simeq \langle f, g\rangle \quad : \quad (\tau \to \mathbb{1}) \times (\mathbb{1} \to \tau).$$

It follows that the types $\mathtt{ref}[\tau]$ and $(\tau \to \mathbb{1}) \times (\mathbb{1} \to \tau)$ – which are distinct syntactic objects – must have isomorphic denotations in every fully abstract model. In the game model in [16]; the types have the same denotations.

As noted above, the language GREF* is identical to the language of Abramsky et al. except that its reference types are not primitive; indeed $\mathtt{ref}[\tau]$ is just a shorthand for $(\tau \to \mathbb{1}) \times (\mathbb{1} \to \tau)$. For this reason, GREF* has no bad-reference constructor such as `mkref` (there is no need for it), but it does need projections i.e. ($\Downarrow$REFPROJ$_{\{1,2\}}$) to extract the corresponding read and write methods of a reference. The game model of [16] is therefore also a fully abstract model of GREF*.

Types

$$\begin{aligned}
\ulcorner\mathtt{lab}[\tau]\urcorner &\triangleq \mathtt{ref}[\ulcorner\tau \to \tau\urcorner] \\
\ulcorner\mathtt{asp}[\tau]\urcorner &\triangleq \ulcorner\mathtt{lab}[\tau]\urcorner \times \ulcorner\tau \to \tau\urcorner
\end{aligned}$$

Terms

$$\begin{aligned}
\ulcorner\mathtt{l}\urcorner &\triangleq \mathscr{L}(\lfloor\mathtt{l}\rfloor) \\
\ulcorner\mathtt{newlab}_{\tau}\urcorner &\triangleq \mathtt{newref}\ x^{\ulcorner\tau\to\tau\urcorner} = \mathtt{id}[\ulcorner\tau\urcorner]\ \mathtt{in}\ \mathscr{L}(x) \\
\ulcorner e_1 \gg e_2\urcorner &\triangleq \Big[\mathtt{let}\ a = \ulcorner e_1\urcorner\ \mathtt{in}\ \pi_1(a) := \pi_2(a)\Big]; \ulcorner e_2\urcorner \\
\ulcorner\{e_1.x \to e_2\}\urcorner &\triangleq \langle\ulcorner e_1\urcorner, \lambda x.\ulcorner e_2\urcorner\rangle \\
\ulcorner e_1\langle e_2\rangle\urcorner &\triangleq\ !\ulcorner e_1\urcorner\cdot\ulcorner e_2\urcorner
\end{aligned}$$

Label Mapping

$$\begin{aligned}
\ulcorner\bot\urcorner &\triangleq \bot \\
\ulcorner\mathsf{L}\{\mathtt{l} \mapsto \tau\}\urcorner &\triangleq \ulcorner\mathsf{L}\urcorner\{\lfloor\mathtt{l}\rfloor \mapsto \ulcorner\tau \to \tau\urcorner\}
\end{aligned}$$

Aspect Sequences

$$\begin{aligned}
\ulcorner\varepsilon\urcorner_{\mathsf{L}} &\triangleq \mathbb{ID}_{\ulcorner\mathsf{L}\urcorner} \\
\ulcorner\langle\mathtt{l}, v\rangle :: \mathsf{A}\urcorner_{\mathsf{L}} &\triangleq \ulcorner\mathsf{A}\urcorner_{\mathsf{L}}\{\lfloor\mathtt{l}\rfloor \mapsto \ulcorner v\urcorner\}
\end{aligned}$$

Configurations

$$\ulcorner(\mathsf{L}, \mathsf{A}, e)\urcorner \triangleq (\ulcorner\mathsf{L}\urcorner, \ulcorner\mathsf{A}\urcorner_{\mathsf{L}}, \ulcorner e\urcorner)$$

Table 3. Translation from COREAML to GREF*

4. Translating aspects to references

This Section introduces a compositional translation $\ulcorner-\urcorner$ from COREAML into GREF*; see Table 3 for the key clauses of the definition. In the following, we comment on the salient features of the translation.

The main intuition is to translate a COREAML label of type τ into a GREF* reference of type $\ulcorner\tau\urcorner \to \ulcorner\tau\urcorner$, and to translate an aspect into the pairing of the respective translates of its pointcut (the label) and advice (a function). Installing an aspect then corresponds to composing the advice with the current content of the location, and invoking an aspect simply dereferences the location and applies the result to the value of the underlying term. We give the translation of the types of labels and aspects at the top of Table 3. The translation acts trivially (and compositionally) on the remaining types of COREAML (i.e. the types of the underlying calculus $\mathcal{L}$): it is the identity map on ground types, and the translate of a product type is the product of the respective translates of its components, similarly for function type.

For the translation of the terms of $\mathcal{L}$, we assume a bijection between label names and references names, writing $\lfloor\mathtt{l}\rfloor$ as the image of `l` under the bijection. Intuitively we will be storing the composite of all the advice associated to a label `l` in the reference location $\lfloor\mathtt{l}\rfloor$. Since the only way to access advice in COREAML is by accessing the entire composite, we are free to compose it at "storage time" without losing any expressive power. The translation uses a shorthand

$$\mathscr{L}(e) \quad \triangleq \quad \langle\, \lambda x^{\tau\to\tau}.(e := x \circ !e)\ ,\ \lambda d^{\mathbb{1}}.!e\,\rangle$$

for any GREF* term e such that $\Gamma \vdash e : \mathtt{ref}[\tau \to \tau]$ is derivable for some τ. This term is the heart of the translation; it defines a bad reference of type $\tau \to \tau$ whose assignment function composes its input with the current contents (rather than simply overwriting them). This allows aspect installation to be modelled by an assignment to the corresponding reference, and aspect invocation then

Type System

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{newref}_\tau(e) : \mathtt{ref}[\tau]}\ (\text{TP NEWREF}) \qquad \frac{\Gamma \vdash e : \mathtt{ref}[\tau]}{\Gamma \vdash !e : \tau}\ (\text{TP DEREF}) \qquad \frac{\Gamma \vdash e_1 : \mathtt{ref}[\tau] \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \mathbb{1}}\ (\text{TP ASSIGN})$$

Operational Semantics

$$\frac{e \Downarrow (\mathsf{R}, \mathsf{S}, v)}{\mathtt{newref}_\tau(e) \Downarrow (\mathsf{R}\{\mathtt{r} \mapsto \tau\}, \mathsf{S}\{\mathtt{r} \mapsto v\}, \mathtt{r})}\ (\Downarrow \text{NEWREF}) \quad \mathtt{r} \notin \mathtt{dom}(\mathsf{R})$$

$$\frac{e_1 \Downarrow \mathtt{r} \qquad e_2 \Downarrow (\mathsf{R}, \mathsf{S}, v)}{e_1 := e_2 \Downarrow (\mathsf{R}, \mathsf{S}\{\mathtt{r} \mapsto v\}, \mathtt{skip})}\ (\Downarrow \text{ASSIGN}) \qquad \frac{e_1 \Downarrow \langle v_1, v_2 \rangle \qquad e_2 \Downarrow v \qquad v_1 \cdot v \Downarrow \mathtt{skip}}{e_2 := e_2 \Downarrow \mathtt{skip}}\ (\Downarrow \text{ASSIGN-BAD})$$

$$\frac{e \Downarrow (\mathsf{R}, \mathsf{S}, \mathtt{r})}{!e \Downarrow (\mathsf{R}, \mathsf{S}, v)}\ [\mathsf{S}(\mathtt{r}) = v]\ (\Downarrow \text{DEREF}) \qquad \frac{e \Downarrow \langle v_1, v_2 \rangle \qquad v_2 \cdot \mathtt{skip} \Downarrow v}{!e \Downarrow v}\ (\Downarrow \text{DEREF-BAD})$$

$$\frac{e \Downarrow \mathtt{r}}{\pi_1(e) \Downarrow \lambda x^\tau.(\mathtt{r} := x)}\ (\Downarrow \text{REFPROJ}_1) \qquad \frac{e \Downarrow \mathtt{r}}{\pi_2(e) \Downarrow \lambda d^{\mathbb{1}}.(!\mathtt{r})}\ (\Downarrow \text{REFPROJ}_2)$$

Table 2. Typing Rules and Operational Semantics of GREF*

simply amounts to dereferencing the current location. Label creation, $\mathtt{newlab}_\tau$, translates to the creation of a new reference (of the above kind) initialised to the appropriately typed identity function. As with the pairing transform $\mathscr{P}_\mathcal{R}(-)$, we can use a `let` statement to ensure that the $\mathscr{L}(e)$ converges if and only if e does. Since there is exactly one `newref` statement in $\ulcorner e \urcorner$ for each `newlab` statement in e, we assume that if the new label chosen by evaluating e is `l`, the corresponding `newref` in $\ulcorner e \urcorner$ generates $\rfloor \mathtt{l} \lfloor$.

We extend the translation to act on configurations of COREAML, producing configurations of GREF*. First a label mapping L that maps `l` to τ is translated to the reference mapping that takes each $\rfloor \mathtt{l} \lfloor$ to the translate of the function type $\tau \to \tau$. An aspect sequence A must be translated with respect to the label mapping because of the case in which the sequence is empty. In this case any labels not appearing in A but appearing in L must be mapped to the appropriate identity function in the GREF* store. This is achieved by the $\mathbb{ID}_\mathsf{R}$ function (for R a reference/type mapping from an GREF* configuration) defined as follows:

$$\mathbb{ID}_\mathsf{R} \triangleq \lambda \mathtt{r}. \begin{cases} \mathtt{id}[\tau] & \text{if } \mathsf{R}(\mathtt{r}) = (\tau \to \tau) \\ \bot & \text{otherwise} \end{cases}$$

Informally this is just the store that maps each reference `r` to the identity function if `r` is defined in R and is of function type with identical input and output types. Every type in R will be of this form if it has been translated from COREAML, so our definition is well formed. Further, for an GREF* store S, location `r` and value v of type $\tau \to \tau$, we write $\mathsf{S}\{\mathtt{r} \twoheadrightarrow v\}$ for the store obtained by remapping `r` to its current contents composed with v:

$$\mathsf{S}\{\mathtt{r} \twoheadrightarrow v\} \triangleq \mathsf{S}\{\mathtt{r} \mapsto (\mathsf{S}(\mathtt{r}) \circ v)\}$$

We will often drop the subscript from $\ulcorner \mathsf{A} \urcorner_\mathsf{L}$ when it is clear which L was used. Finally the translate of complete configurations is defined as the pointwise translation of each of its components.

We first establish some basic properties of the translation:

Proposition 4.1. *(1) If $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$ is valid in* COREAML *then $x_1 : \ulcorner \tau_1 \urcorner, \ldots, x_n : \ulcorner \tau_n \urcorner \vdash \ulcorner e \urcorner : \ulcorner \tau \urcorner$ is valid in* GREF*.

(2) If v is a COREAML*-value, then $\ulcorner v \urcorner$ is a* GREF*-value.*

(3) For COREAML*-terms $g_1, \cdots, g_n$ of type $\tau \to \tau$, we have*

$$\ulcorner g_1 \circ \cdots \circ g_n \urcorner \ = \ \ulcorner g_1 \urcorner \circ \cdots \circ \ulcorner g_n \urcorner.$$

(4) If $\vdash (\mathsf{L}, \mathsf{A}, e)$ in COREAML *then $\vdash \ulcorner (\mathsf{L}, \mathsf{A}, e) \urcorner$ in* GREF*.*

Proof. (1) is proved by a straightforward induction on the structure of e. (2) follows immediately from the definition of the translation over the core calculus $\mathcal{L}$ and the fact that $\mathscr{L}(v)$ is a value for any value v. (3) is trivially proved by following the definition of the translation over λ-terms. (4) follows from the fact that every label `l` in the environment is translated into $\rfloor \mathtt{l} \lfloor$, and each label in e is translated into $\mathscr{L}(\rfloor \mathtt{l} \lfloor)$, which itself only contains the reference $\rfloor \mathtt{l} \lfloor$. □

5. Properties of the translation: adequacy and definability

How faithful is the compositional translation from COREAML into GREF*? In this Section, we examine the extent to which behavioural properties of user terms of COREAML (as considered in Section 1) are preserved (and reflected) by the translation. We prove that the translation is adequate, and satisfies the *definability property* i.e. every term of GREF* is the translate of some COREAML-term. However the translation fails to be fully abstract; we use a simple counterexample to explain why. In the next Section, we shall show how a fully abstract translation can be achieved by a slight modification to COREAML.

Basic properties of the translation

We present some basic properties of the translation that will help us reason about typing judgements involving substitution. In the first two lemmas, we let e and v be well-typed user terms of COREAML where v is a value.

Lemma 5.1. *If $\vdash e[v/x]$ holds in* COREAML*, then $\vdash \ulcorner e \urcorner[\ulcorner v \urcorner / x]$ holds in* GREF*.*

Proof. By assumption there are Γ, τ, σ such that

$$\Gamma, x : \sigma \vdash e : \tau \quad \text{and} \quad \Gamma \vdash v : \sigma$$

are valid. By Proposition 4.1, we have

$$\ulcorner\Gamma\urcorner, x : \ulcorner\sigma\urcorner \vdash \ulcorner e\urcorner : \ulcorner\tau\urcorner \quad \text{and} \quad \ulcorner\Gamma\urcorner \vdash \ulcorner v\urcorner : \ulcorner\sigma\urcorner$$

are valid in GREF*. Then, since $\ulcorner v\urcorner$ is a value, we know that $\ulcorner e\urcorner[\ulcorner v\urcorner/x]$ is a well-typed term of GREF*. □

The converse of the lemma does not hold i.e. we could well have terms e and v such that $\ulcorner e\urcorner[\ulcorner v\urcorner/x]$ is well-typed in GREF*, but $e[v/x]$ is not well-typed in COREAML. For instance, let e be a variable x of type $\mathtt{lab}[\tau]$ and

$$v \equiv \langle \lambda y^{\tau\to\tau}.\mathtt{skip}, \lambda y^{\mathbb{1}}.\mathtt{id}[\tau]\rangle \ : \ \mathtt{ref}[\tau \to \tau].$$

It is precisely the lack of this property that we shall later exploit to show the failure of full abstraction. Therefore the remedy proposed in the following section is designed to ensure that the translation on types is a bijection.

Next we prove a useful substitution lemma, which states that the translate of a substitution is a substitution of the respective translates.

Lemma 5.2. *If* $\vdash e[v/x]$ *holds in* COREAML *then* $\ulcorner e[v/x]\urcorner = \ulcorner e\urcorner[\ulcorner v\urcorner/x]$ *holds in* GREF*.

Proof. By the composition of the translation, and by the result of the previous lemma, we know that both sides of the desired equality are well-typed terms of GREF*. The equality is proved by an easy induction on the derivation of

$$\Gamma, x : \sigma \vdash e : \tau$$

whose existence is implied by $\vdash e[v/x]$. The proof requires a standard weakening lemma in the case of application. □

The last basic lemma establishes the correctness of the translation of COREAML environments. It states that for a given environment (L, A), the translate of the composite $\mathbb{C}_{\mathsf{L},\mathsf{A}}$ is the GREF* store $\ulcorner\mathsf{A}\urcorner_{\mathsf{L}}$.

Lemma 5.3. *For any label* $\mathtt{l}$*, we have* $\ulcorner\mathbb{C}_{\mathsf{L},\mathsf{A}}(\mathtt{l})\urcorner = \ulcorner\mathsf{A}\urcorner_{\mathsf{L}}(\lfloor\mathtt{l}\rfloor)$.

Proof. If $\mathtt{l} \notin \mathtt{dom}(\mathsf{L})$, then $\lfloor\mathtt{l}\rfloor \notin \mathtt{dom}(\ulcorner\mathsf{L}\urcorner_{\mathsf{L}})$ and so both sides of the equation are equal to $\bot$. For the remaining cases, denote $\mathsf{A} \restriction \mathtt{l}$ as the restriction of A to $\mathtt{l}$ (i.e. the subsequence of $\mathtt{l}$-labelled advice of A), and note that

$$\mathbb{C}_{\mathsf{L},\mathsf{A}}(\mathtt{l}) = \mathbb{C}_{\mathsf{L},(\mathsf{A}\restriction\mathtt{l})}(\mathtt{l}) \quad \text{and} \quad \ulcorner\mathsf{A}\urcorner_{\mathsf{L}}(\lfloor\mathtt{l}\rfloor) = \ulcorner\mathsf{A}\restriction\mathtt{l}\urcorner_{\mathsf{L}}(\lfloor\mathtt{l}\rfloor)$$

We therefore only need to consider aspect sequences A that are comprised entirely of $\mathtt{l}$-labelled advice. In this case, if $\mathsf{A} = \varepsilon$, then $\mathbb{C}_{\mathsf{L},\mathsf{A}} = \mathtt{id}[\mathsf{L}(\mathtt{l})]$. By the definition of the translation we have

$$\ulcorner\mathtt{id}[\mathsf{L}(\mathtt{l})]\urcorner = \mathtt{id}[\ulcorner\mathsf{L}(\mathtt{l})\urcorner]$$

which is exactly $\ulcorner\mathsf{A}\urcorner_{\mathsf{L}}(\lfloor\mathtt{l}\rfloor)$ because $\ulcorner\varepsilon\urcorner_{\mathsf{L}} = \mathbb{ID}_{\ulcorner\mathsf{L}\urcorner}$. If A is not empty, then $\mathbb{C}_{\mathsf{L},\mathsf{A}} = (g_n \circ \cdots \circ g_1)$ for some sequence $\overline{g}$ of advice. The translate of this term is therefore $(\ulcorner g_n\urcorner \circ \cdots \circ \ulcorner g_1\urcorner)$ by Proposition 4.1(2). But then a routine induction easily shows that this is exactly the value of $\ulcorner\mathsf{A}\urcorner_{\mathsf{L}}(\lfloor\mathtt{l}\rfloor)$. □

Preservation and reflection of termination and evaluation

We are now ready to prove the main properties of the translation. The first result shows that the evaluation of *user terms* is preserved. However a stronger induction hypothesis is needed to push through the argument; we thus prove the result for all possible configurations, and with respect to syntactic equality (up to α-equivalence) rather than observational equivalence.

Proposition 5.4. *For every well-typed* COREAML *configuration* P*, if* $P \Downarrow U$ *then* $\ulcorner P\urcorner \Downarrow V$ *and* $V = \ulcorner U\urcorner$.

Proof. The proof is by induction on the derivation of $P \Downarrow U$. The base case where P is a value configuration follows immediately from the fact that the translation preserves values, i.e. Proposition 4.1(2). Cases for the rules stemming from $\mathcal{L}$ follow easily from the induction hypothesis, we present the case for application:

Case ($\Downarrow$APP)**.** The induction hypothesis applied to the premises of this rule yield derivations of

$$\mathcal{D}_1 \equiv \frac{\vdots}{\ulcorner e_1\urcorner \Downarrow \lambda x.\ulcorner e\urcorner} \quad \mathcal{D}_2 \equiv \frac{\vdots}{\ulcorner e_2\urcorner \Downarrow \ulcorner u\urcorner} \quad \mathcal{D}_3 \equiv \frac{\vdots}{\ulcorner e[u/x]\urcorner \Downarrow \ulcorner v\urcorner}$$

Applying Lemma 5.2 to $\mathcal{D}_3$ yields

$$\ulcorner e\urcorner[\ulcorner u\urcorner/x] \Downarrow \ulcorner v\urcorner$$

and combined with $\mathcal{D}_1$ and $\mathcal{D}_2$, this yields a derivation of

$$\ulcorner e_1\urcorner \cdot \ulcorner e_2\urcorner \Downarrow \ulcorner v\urcorner$$

in GREF*, which by definition is exactly the required derivation of

$$\ulcorner e_1 \cdot e_2\urcorner \Downarrow \ulcorner v\urcorner$$

The remaining cases are label creation, aspect installation, aspects, and join points. The case for rule ($\Downarrow$ASPECT) follows immediately from the inductive hypothesis as above, the remaining cases are presented below:

Case ($\Downarrow$NEWLAB)**.** For this case, we simply carry out the evalutation of the translated term using the GREF* evaluation rules. The translation of $(\mathsf{L}, \mathsf{A}, \mathtt{newlab}_\tau)$ is

$$(\ulcorner\mathsf{L}\urcorner, \ulcorner\mathsf{A}\urcorner_{\mathsf{L}}, \mathtt{newref}\ x^{\ulcorner\tau\to\tau\urcorner} = \mathtt{id}[\ulcorner\tau\urcorner]\ \mathtt{in}\ \mathscr{L}(x))$$

which, using rule ($\Downarrow$ NEWREF), immediately evaluates to

$$(\ulcorner\mathsf{L}\urcorner\{\lfloor\mathtt{l}\rfloor \mapsto \ulcorner\tau \to \tau\urcorner\}, \ulcorner\mathsf{A}\urcorner_{\mathsf{L}}\{\lfloor\mathtt{l}\rfloor \mapsto \mathtt{id}[\ulcorner\tau\urcorner]\}, \mathscr{L}(\lfloor\mathtt{l}\rfloor))$$

but one can routinely check that this is exactly the translation of $(\mathsf{L}\{\mathtt{l} \mapsto \tau\}, \mathsf{A}, \mathtt{l})$ as required.

Case ($\Downarrow$INSTALL)**.** The induction hypothesis yields GREF* derivations of

$$\mathcal{D}_1 \equiv \frac{\vdots}{\ulcorner e_1\urcorner \Downarrow (\ulcorner\mathsf{L}\urcorner, \ulcorner\mathsf{A}\urcorner_{\mathsf{L}}, \ulcorner a\urcorner)}$$

$$\mathcal{D}_2 \equiv \frac{\vdots}{(\ulcorner\mathsf{L}\urcorner, \ulcorner\mathsf{A} :: a\urcorner_{\mathsf{L}}, \ulcorner e_2\urcorner) \Downarrow \ulcorner v\urcorner}$$

Supposing, without loss of generality, that $a \equiv \{\mathtt{l}.x \to e\}$, we calculate the following:

$$\begin{aligned} \ulcorner a\urcorner &= \langle \mathscr{L}(\lfloor\mathtt{l}\rfloor), \lambda x.\ulcorner e\urcorner\rangle \\ \ulcorner\mathsf{A} :: a\urcorner_{\mathsf{L}} &= \ulcorner\mathsf{A}\urcorner_{\mathsf{L}}\{\lfloor\mathtt{l}\rfloor \twoheadrightarrow (\lambda x.\ulcorner e\urcorner)\} \end{aligned}$$

Note that the result of assigning $\lambda x.\ulcorner e\urcorner$ to $\mathscr{L}(\lfloor\mathtt{l}\rfloor)$ in environment $\ulcorner\mathsf{A}\urcorner_{\mathsf{L}}$ is $\mathtt{skip}$ in the updated environment $\ulcorner\mathsf{A} :: a\urcorner_{\mathsf{L}}$. We therefore know that we can derive

$$\pi_1(\ulcorner a\urcorner) := \pi_2(\ulcorner a\urcorner); \ulcorner e_2\urcorner \Downarrow \ulcorner v\urcorner$$

and since $\ulcorner a\urcorner$ is already a value, we can certainly $\mathtt{let}$ abstract it in the assignment without affecting the evaluation, yielding

$$\Big[\mathtt{let}\ x = \ulcorner a\urcorner\ \mathtt{in}\ \pi_1(x) := \pi_2(x)\Big]; \ulcorner e_2\urcorner \Downarrow \ulcorner v\urcorner$$

and finally, by derivation $\mathcal{D}_1$, we know that $\ulcorner e_1 \urcorner$ evaluates to $\ulcorner a \urcorner$, therefore we can conclude that

$$\Big[\texttt{let}\ x = \ulcorner e_1 \urcorner\ \texttt{in}\ \pi_1(x) := \pi_2(x)\Big]; \ulcorner e_2 \urcorner \Downarrow \ulcorner v \urcorner$$

which is precisely the desired derivation of

$$\ulcorner e_1 \gg e_2 \urcorner \Downarrow \ulcorner v \urcorner$$

Case ($\Downarrow$INVOKE)**.** Applying the induction hypothesis to the first, third, and fourth premises of this rule yields GREF* derivations of the following evaluations:

$$\mathcal{D}_1 \equiv \frac{\vdots}{\ulcorner e_1 \urcorner \Downarrow (\ulcorner \mathsf{L} \urcorner, \ulcorner \mathsf{A} \urcorner_{\mathsf{L}}, \mathscr{L}(\lfloor \mathtt{l} \rfloor))}$$

$$\mathcal{D}_2 \equiv \frac{\vdots}{\ulcorner e_2 \urcorner \Downarrow \ulcorner u \urcorner} \quad \mathcal{D}_3 \equiv \frac{\vdots}{\ulcorner e[u/x] \urcorner \Downarrow \ulcorner v \urcorner}$$

Note firstly that dereferencing $\mathscr{L}(\lfloor \mathtt{l} \rfloor)$ is equivalent to dereferencing $\lfloor \mathtt{l} \rfloor$, and secondly that applying the store correctness lemma we have $\ulcorner \mathsf{A} \urcorner_{\mathsf{L}}(\lfloor \mathtt{l} \rfloor) = \lambda x.\ulcorner e \urcorner$. Combining these two facts with $\mathcal{D}_1$ yields a derivation of

$$\frac{\vdots}{!\ulcorner e_1 \urcorner \Downarrow \lambda x.\ulcorner e \urcorner}$$

Using this together with $\mathcal{D}_2$ and $\mathcal{D}_3$ (and the substitution lemma), we can apply rule ($\Downarrow$APP) to construct a derivation of

$$!\ulcorner e_1 \urcorner \cdot \ulcorner e_2 \urcorner \Downarrow \ulcorner v \urcorner$$

which is exactly the desired derivation of

$$\ulcorner e_1 \langle e_2 \rangle \urcorner \Downarrow \ulcorner v \urcorner$$

□

The next proposition proves the converse of the preceding result, thus showing that that the translation from COREAML to GREF* *preserves and reflects evaluation* in the sense of Property E. It follows that the translation also *preserves and reflects termination* of configurations.

Proposition 5.5. *For every well-typed* COREAML *configuration* P*, if* $\ulcorner P \urcorner \Downarrow V$ *then* $P \Downarrow U$ *and* $V = \ulcorner U \urcorner$*.*

Proof. By induction on the derivation of $\ulcorner P \urcorner \Downarrow V$. The base case occurs when P is a value, and is trivially true by Proposition 4.1(2). As in the previous proof, the rules of the underlying language $\mathcal{L}$ follow relatively straightforwardly from the induction hypothesis, the most complex case being application (which requires the substitution lemma):

Case ($\Downarrow$APP)**.** The induction hypothesis on the first two premises allows us to deduce that $\ulcorner e_1 \urcorner$ must evaluate to $\lambda x.\ulcorner e \urcorner$ for some COREAML term e, that $\ulcorner e_2 \urcorner$ evaluates to $\ulcorner u \urcorner$ for some COREAML value u, and that the following derivations exist:

$$\mathcal{D}_1 \equiv \frac{\vdots}{e_1 \Downarrow \lambda x.e} \quad \mathcal{D}_2 \equiv \frac{\vdots}{e_2 \Downarrow u}$$

For the third premise, note that since the original term was well typed in COREAML, then $e[u/x]$ must also be well-typed. This allows us to apply Lemma 5.2 to yield that $\ulcorner e \urcorner[\ulcorner u \urcorner/x] = \ulcorner e[u/x] \urcorner$, and therefore allowing us to apply the induction hypothesis to the third premise to conclude that the substitution must evaluate to $\ulcorner v \urcorner$ for some COREAML term v, and that

$$\mathcal{D}_3 \equiv \frac{\vdots}{e[u/x] \Downarrow v}$$

is derivable in COREAML. Finally we use $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$ as premises of rule ($\Downarrow$APP) in COREAML to construct the desired derivation of $e_1 \cdot e_2 \Downarrow v$.

The remaining cases are those enumerated by the rules added to $\mathcal{L}$ to define COREAML. The rule ($\Downarrow$ASPECT) follows easily from the induction hypothesis, leaving the cases for the creation of new labels, and installing and invoking advice.

Case ($\Downarrow$NEWLAB)**.** Translating $P \equiv (\mathsf{L}, \mathsf{A}, \mathtt{newlab}_\tau)$ and evaluating it using the evaluation rules of GREF* yields a derivation that the following configuration:

$$(\ulcorner \mathsf{L} \urcorner, \ulcorner \mathsf{A} \urcorner_{\mathsf{L}}, \mathtt{newref}\ x^{\ulcorner \tau \to \tau \urcorner} = \mathtt{id}[\ulcorner \tau \urcorner]\ \mathtt{in}\ \mathscr{L}(x))$$

evaluates to the following value:

$$(\ulcorner \mathsf{L} \urcorner\{\lfloor \mathtt{l} \rfloor \mapsto \ulcorner \tau \to \tau \urcorner\}, \ulcorner \mathsf{A} \urcorner_{\mathsf{L}}\{\lfloor \mathtt{l} \rfloor \mapsto \mathtt{id}[\ulcorner \tau \urcorner]\}, \mathscr{L}(\lfloor \mathtt{l} \rfloor))$$

This is exactly a translation of the COREAML configuration

$$(\mathsf{L}\{\mathtt{l} \mapsto \tau\}, \mathsf{A}, \mathtt{l})$$

which, as required, is exactly the value to which P evaluates.

Case ($\Downarrow$INSTALL)**.** Assuming the antecedent, evaluating the translate of $e_1 \gg e_2$ yields the following derivation:

$$\mathcal{E} \equiv \frac{\vdots}{\Big[\texttt{let}\ x = \ulcorner e_1 \urcorner\ \texttt{in}\ \pi_1(x) := \pi_2(x)\Big]; \ulcorner e_2 \urcorner \Downarrow V}$$

for some value configuration V. The first command in the sequence (i.e. $\texttt{let}\ x = \cdots$) was derived from an evaluation of $\ulcorner e_1 \urcorner$. We apply the induction hypothesis to this derivation, yielding that it evaluates to a value configuration $(\ulcorner \mathsf{L} \urcorner, \ulcorner \mathsf{A} \urcorner_{\mathsf{L}}, \ulcorner a \urcorner)$ which is the translate of a COREAML aspect a, and that

$$\mathcal{D}_1 \equiv \frac{\vdots}{e_1 \Downarrow (\mathsf{L}, \mathsf{A}, a)}$$

Supposing without losing generality that $a = \{\mathtt{l}.x \to e\}$, then the body of the let command amounts to an evaluation of

$$(\ulcorner \mathsf{L} \urcorner, \ulcorner \mathsf{A} \urcorner_{\mathsf{L}}, \mathscr{L}(\lfloor \mathtt{l} \rfloor) := \lambda x.\ulcorner e \urcorner)$$

which evaluates to

$$(\ulcorner \mathsf{L} \urcorner, \ulcorner \mathsf{A} \urcorner_{\mathsf{L}}\{\lfloor \mathtt{l} \rfloor \twoheadrightarrow \lambda x.\ulcorner e \urcorner\}, \mathtt{skip})$$

by the definition of $\mathscr{L}(-)$. This then, is the environemnt under which $\ulcorner e_2 \urcorner$ must have been evaluated in order to yield derivation $\mathcal{E}$. By the definition of the translation, the GREF* configuration

$$(\ulcorner \mathsf{L} \urcorner, \ulcorner \mathsf{A} \urcorner_{\mathsf{L}}\{\lfloor \mathtt{l} \rfloor \twoheadrightarrow \lambda x.\ulcorner e \urcorner\}, \ulcorner e_2 \urcorner)$$

is precisely the translation of $(\mathsf{L}, \mathsf{A} :: a, e_2)$, and so we may apply the induction hypothesis to its evaluation to yield that it evaluates to some value configuration $V = \ulcorner U \urcorner$, and that

$$\mathcal{D}_2 \equiv \frac{\vdots}{(\mathsf{L}, \mathsf{A} :: a, e_2) \Downarrow U}$$

Finally, using $\mathcal{D}_1$ and $\mathcal{D}_2$, we can construct the desired derivation

$$\frac{\vdots}{e_1 \gg e_2 \Downarrow U}$$

using rule ($\Downarrow$INSTALL).

Case ($\Downarrow$INVOKE)**.** Assuming the antecedent, translating $e_1\langle e_2\rangle$ and evaluating it yields the following GREF* derivation:

$$\mathcal{E} \equiv \frac{\vdots}{!\ulcorner e_1 \urcorner \cdot \ulcorner e_2 \urcorner \Downarrow V}$$

for some value configuration V. This must have been derived from a derivation that $!\ulcorner e_1 \urcorner$ evaluates to some λ term g, and this must have in turn been derived from an evaluation of $\ulcorner e_1 \urcorner$. We apply the induction hypothesis to the latter to deduce that $\ulcorner e_1 \urcorner \Downarrow (\ulcorner \mathsf{L} \urcorner, \ulcorner \mathsf{A} \urcorner_{\mathsf{L}}, \mathscr{L}(\lfloor \mathtt{l} \rfloor))$ and that

$$\mathcal{D}_1 \equiv \frac{\vdots}{e_1 \Downarrow (\mathsf{L}, \mathsf{A}, \mathtt{l})}$$

for some label $\mathtt{l}$ (WLOG). The second premise of the evaluation of $!\ulcorner e_1 \urcorner$ must therefore have been the application of $\mathtt{skip}$ to the second component of $\mathscr{L}(\lfloor \mathtt{l} \rfloor)$ (because this is a bad reference). By definition, this is exactly the same as dereferencing $\lfloor \mathtt{l} \rfloor$ directly, and therefore we must have $\ulcorner \mathsf{A} \urcorner_{\mathsf{L}}(\lfloor \mathtt{l} \rfloor) = g$. Now by Lemma 5.3 we know that g must be the translate of $\mathbb{C}_{\mathsf{L},\mathsf{A}}(\mathtt{l})$, and hence the translate of some CoreAML λ term $\lambda x.e$, i.e. $g = \lambda x.\ulcorner e \urcorner$ and hence

$$\mathcal{D}_2 \equiv [\mathbb{C}_{\mathsf{L},\mathsf{A}}(\mathtt{l}) = \lambda x.e]$$

Thirdly, we can apply the induction hypothesis to the subevaluation of $\ulcorner e_2 \urcorner$ to yield that $\ulcorner e_2 \urcorner \Downarrow \ulcorner u \urcorner$ for some CoreAML value u, and that

$$\mathcal{D}_3 \equiv \frac{\vdots}{e_2 \Downarrow u}$$

and doing the same to the last premise of the application $\mathcal{E}$, and using the substitution lemma, we get $\ulcorner e[u/x] \urcorner \Downarrow \ulcorner U \urcorner$ (where $\ulcorner U \urcorner = V$), and that

$$\mathcal{D}_4 \equiv \frac{\vdots}{e[u/x] \Downarrow U}$$

and finally, we note that $\mathcal{D}_1, \cdots, \mathcal{D}_4$ form exactly the premises of the required derivation that

$$e_1\langle e_2\rangle \Downarrow U$$

□

Adequacy

We can now prove that the translation satisfies *adequacy* i.e. observational equivalence of CoreAML-terms is *reflected* by the translation. Termination preservation and reflection, as well as the substitution lemma, are the only results required to prove adequacy.

Theorem 5.6 (Adequacy)**.** *For every well-typed* CoreAML *open user terms e_1 and e_2, if $\ulcorner e_1 \urcorner \simeq \ulcorner e_2 \urcorner$ then $e_1 \simeq e_2$.*

Proof. Let e_1 and e_2 be such that $\ulcorner e_1 \urcorner \simeq \ulcorner e_2 \urcorner$, and let $\mathcal{C}[-]$ be an arbitrary CoreAML context such that both $\mathcal{C}[e_1]$ and $\mathcal{C}[e_2]$ are programs (i.e. closed terms of type $\mathbb{1}$). By Lemma 5.1, $\ulcorner \mathcal{C} \urcorner[-]$ is similarly a closing context for $\ulcorner e_1 \urcorner$ and $\ulcorner e_2 \urcorner$. Since observational equivalence is a congruence, we have $\ulcorner \mathcal{C} \urcorner[\ulcorner e_1 \urcorner] \simeq \ulcorner \mathcal{C} \urcorner[\ulcorner e_2 \urcorner]$; but then by Lemma 5.2 we have $\ulcorner \mathcal{C} \urcorner[\ulcorner e_i \urcorner] = \ulcorner \mathcal{C}[e_i] \urcorner$, and hence $\ulcorner \mathcal{C}[e_1] \urcorner \simeq \ulcorner \mathcal{C}[e_2] \urcorner$. Now suppose $\mathcal{C}[e_1]\Downarrow$; by Proposition 5.4 we know that $\ulcorner \mathcal{C}[e_1] \urcorner \Downarrow$. Since $\ulcorner \mathcal{C}[e_2] \urcorner \simeq \ulcorner \mathcal{C}[e_2] \urcorner$, we have $\ulcorner \mathcal{C}[e_2] \urcorner \Downarrow$. Hence by Proposition 5.5 we can deduce that $\mathcal{C}[e_2]\Downarrow$. Symmetrically, we can prove that if $\mathcal{C}[e_2]$ converges then $\mathcal{C}[e_1]$ converges. Therefore we have $e_1 \simeq e_2$ as desired. □

The Theorem tells us that we can soundly reason about the observational equivalence of CoreAML-terms by reasoning about the observational equivalence of their respective translates in GREF*. Since the translation is *compositional*, every denotational model of the target language GREF* becomes a denotational model of the source language CoreAML by translation. Further, thanks to the adequacy of the translation, since the game model of is adequate (see GREF* [16]), the inherited semantics for CoreAML is also adequate.

Definability

The Adequacy Theorem tells us that, modulo observational equivalence, there are "at least as many" terms in GREF* as there are in CoreAML i.e. the target language is at least as rich as the source. The next result tells us that, modulo observational equivalence, the target language is "no richer" than the (translate of the) source. More precisely, for every GREF* term, there is some CoreAML term whose translate is observationally equivalent to it.

Proposition 5.7 (Definability)**.** *For every open term e of* GREF**, there exists a user term of* CoreAML $\wr e \int$ *such that $\ulcorner \wr e \int \urcorner \simeq e$.*

Proof. The proof is by induction on the structure of e. Most cases follow immediately from the induction hypothesis, the notable exception is the $\mathtt{newref}_\tau(e)$ construct. We don't need to handle open reference names because we only care about the definability of user terms. Following [4], we essentially modify the reference cell term defined by Walker et al. and tweak it to serve our purposes. Assume (by the induction hypothesis) that there is an $\wr e \int$ such that $e \simeq \ulcorner \wr e \int \urcorner$. A reference of type τ is modelled by creating a label of type τ and using the ability to preserve local state to mimic a reference cell. Omitting type annotations, consider the CoreAML term

$$\begin{aligned}
\mathtt{cell}(e) \quad \triangleq \quad & \mathtt{newlab}\ \ell\ \mathtt{in} \\
& \mathtt{let}\ \mathsf{init} = \wr e \int\ \mathtt{in} \\
& \langle \\
& \quad \lambda \mathsf{v}.\Big[\{\ell.d \to \mathsf{v}\} \gg \mathtt{skip}\Big], \\
& \quad \lambda d.\ell\langle \mathsf{init} \rangle \\
& \rangle
\end{aligned}$$

the intuition is that ℓ will be the label modelling the newly created reference cell, init will be the initial value of the reference cell, and v will be bound to the value we will wish to assign to the cell. Now consider the term's translation into GREF* term $\ulcorner \mathtt{cell}(e) \urcorner$:

$$\begin{aligned}
& \mathtt{newref}\ x = (\lambda y.y)\ \mathtt{in} \\
& \mathtt{let}\ \mathsf{init} = \ulcorner \wr e \int \urcorner\ \mathtt{in} \\
& \quad \langle \\
& \qquad \lambda \mathsf{v}.\Big[\Big(\mathtt{let}\ a = \langle \mathscr{L}(x), \lambda d.\mathsf{v} \rangle\ \mathtt{in}\ \pi_1(a) := \pi_2(a)\Big); \mathtt{skip}\Big], \\
& \qquad \lambda d.\Big[!\mathscr{L}(x) \cdot \mathsf{init}\Big] \\
& \quad \rangle
\end{aligned}$$

The $\mathscr{L}(-)$s appear because of the shorthand we used to encode the $\mathtt{new} \ldots \mathtt{in}$ constructs, we've α-renamed ℓ to x to indicate that it is no longer binding a label.

We claim that this term is equivalent to $\mathtt{newref}(e)$. First note that if ℓ was a label of type τ in the original term, then $\ulcorner \mathtt{cell}(e) \urcorner$ is a term of type $\mathtt{ref}[\ulcorner \tau \urcorner]$. We first observe that $\langle \mathscr{L}(x), \lambda d.\mathsf{v} \rangle$ is already a value, and that sequential composition of a term of type $\mathbb{1}$ with $\mathtt{skip}$ is equivalent to just the term. Using these facts, and the induction hypothesis that $e \simeq \ulcorner \wr e \int \urcorner$, we can simplify $\ulcorner \mathtt{cell}(e) \urcorner$

to

$$
\begin{array}{l}
\texttt{newref}\ x = (\lambda y.y)\ \texttt{in} \\
\texttt{let}\ \mathsf{init} = e\ \texttt{in} \\
\quad \langle \\
\qquad \lambda \mathsf{v}.\Big[\mathscr{L}(x) := (\lambda d.\mathsf{v})\Big], \\
\qquad \lambda d.\Big[!\mathscr{L}(x) \cdot \mathsf{init}\Big] \\
\quad \rangle
\end{array}
$$

Note that the initial value stored in the new reference is the identity function, and because it is guarded by a `new`, we know that this location will appear nowhere else in any enclosing program. This means that the only way to store any other value into the location created here is by applying a value to the first component of the reference pair. Therefore the only thing that can ever subsequently be assigned to the location is a constant function composed with its current contents (because of the definition of $\mathscr{L}(-)$), and thus the new location will only ever contain a constant function. We can now complete the argument by noting the following observations:

- Because of the guarding `let`, this term has the same termination behaviour as $\texttt{newref}(e)$, i.e. it converges if and only if e converges.
- Suppose the new reference location created by evaluating this term is `r`. There are three possible imperative actions one can take on this location:
  1. If this is immediately dereferenced, then the operational semantics dictates that the result of the evaluation will be the result of $(!\texttt{r}) \cdot \mathsf{init}$, which (since `r` contains the identity function in this case) evaluates to init: the result of evaluating e. So an immediately accessed reference returns its inital value as expected.
  2. Any assignment of a term $\hat{e}$ will result in divergence if and only if $\hat{e}$ diverges, and otherwise results in the *constant function* returning the value $\hat{v}$ (to which $\hat{e}$ converges) to be composed with the contents of `r` (this is due to the definition of assignment to $\mathscr{L}(-)$).
  3. Thus, any subsequent dereferencing will converge to the value $\hat{v}$ last assigned to the term, because this operation results in the application of the initial value init to the contents of `r`, but since each function in the composition $!\texttt{r}$ ignore any input, this application immediately returns the constant value of the last function in the sequence (i.e. the last function assigned).

If we project the reference cell, we simply obtain the corresponding function which we've shown above correspond exactly to assignment and dereferencing to the cell. Therefore these points exactly describe the observational behaviour of $\texttt{newref}(e)$, thereby completing the proof of definability.

□

Though adequate, the translation from CoreAML to GRef* is *not* fully abstract i.e. it does not preserve observational equivalence. This is somewhat surprising, especially in view of the definability result (Proposition 5.7); it suggests that the translation is not fully abstract for a subtle reason. Take the following CoreAML user terms and context:

$$
\begin{array}{rcl}
e_1 & = & \lambda \ell^{\texttt{lab}[\tau]}.\{\ell.x \to x\} \gg \texttt{skip} \\
e_2 & = & \lambda \ell^{\texttt{lab}[\tau]}.\texttt{skip} \\
\mathcal{C}[-] & = & [-] \cdot \langle \lambda x.\Omega, \lambda x.\Omega \rangle
\end{array}
$$

where Ω denotes a divergent term. Note that $\mathcal{C}[-]$ (viewed as a term with a single free variable) paired with either of the e_i's, forms a witness to the failure of the converse of Lemma 5.1. Now consider the latter terms, the first takes a label as input and then installs the trivial identity advice to it before evaluating to `skip`, while the second simply evaluates to `skip` right away. We have $e_1 \simeq e_2$ in CoreAML. Now consider their respective translates in GRef*: $\ulcorner e_1 \urcorner$ assigns to the reference that is bound to ℓ, whereras $\ulcorner e_2 \urcorner$ does no such thing. Therefore, if ℓ is bound to a bad reference whose components immediately diverge (say by plugging the e_i into $\mathcal{C}[-]$), we obtain a term $\mathcal{C}[\ulcorner e_1 \urcorner]$ which diverges and another $\mathcal{C}[\ulcorner e_2 \urcorner]$ which does not. Therefore the translates of the two terms are not observationally equivalent in GRef*, violating full abstraction. To summarise, we have:

Proposition 5.8 (Non full-abstraction)**.** *There are* CoreAML *user terms e_1 and e_2 such that $e_1 \simeq e_2$ but $\ulcorner e_1 \urcorner \not\simeq \ulcorner e_2 \urcorner$.*

In the next Section we will show how the situation can be remedied by "breaking down" the types of CoreAML so that the translation both preserves and reflects subsitution.

6. Fully abstract translation and the complete picture

The failure of full abstraction can be traced to a fundamental mismatch between labels of type τ (say) and their translates, which are references of type $\ulcorner \tau \urcorner \to \ulcorner \tau \urcorner$ (which are just pairs of read and write methods). To fix the problem, we "object-orientate" the types of labels (and hence aspects) in CoreAML into their component accessor methods, much as the types of references in GRef* are products of the corresponding read and write methods. In other words we need to endow the source language CoreAML with the power to construct "bad labels" – the analogue of bad references.

Definition 6.1. Formally, we define the language CoreAML* to be the same as CoreAML except that the types of labels and aspects are viewed as shorthands for the following:

$$
\begin{array}{rcl}
\texttt{lab}[\tau] & \triangleq & ((\tau \to \tau) \to \mathbb{1}) \times (\mathbb{1} \to (\tau \to \tau)) \\
\texttt{asp}[\tau] & \triangleq & \texttt{lab}[\tau] \times (\tau \to \tau)
\end{array}
$$

Under these definitions, the translation function $\ulcorner - \urcorner$ defined in Section 4 is just the identity on types, and so, we no longer need to distinguish between a CoreAML* type τ and its translate $\ulcorner \tau \urcorner$. The intuition for the type of aspects should be obvious: we simply separate an aspect explicitly into its pointcut (i.e. label) and advice, and allow users to project on it to retrieve each component. The type of labels pairs an *installation* function – which takes an advice and installs an associated aspect into the environment – with an *invocation* function – which returns the composite of all the advice associated with the label. Since the translation on types is now bijective, the converse of Lemma 5.1 is immediate, and we state it here without proof:

Lemma 6.1. *Let e and v be user terms of* CoreAML* *where v is a value. If $\vdash \ulcorner e \urcorner[\ulcorner v \urcorner/x]$ holds in* GRef* *then $\vdash e[v/x]$ holds in* CoreAML*.

The rules defining typing judgements and operational semantics in Table 1 are carried over to CoreAML* directly, except that the types of labels $\texttt{lab}[\tau]$ and aspects $\texttt{asp}[\tau]$ are now viewed as the above shorthands; similarly $\{e_1.x \to e_2\}$ is viewed as a shorthand for $\langle e_1, \lambda x.e_2 \rangle$. The rule ($\Downarrow$ASPECT) is therefore subsumed by the rule for tuples, and no longer necessary. As was the case for references, *any* terms of the appropriate types may now be used in installation and labelling statements, therefore the behaviour of these constructs on pairs must be defined. Furthermore, since label

names are now of type $\mathtt{lab}[\tau]$ (which is a pair), we must define the behaviour of projecting over them. Thus we must add the following rules to the operational semantics:

$$(\Downarrow \text{INSTALL-BAD})\quad \frac{e_1 \Downarrow \langle\langle v_1, v_2\rangle, g\rangle \qquad v_1 \cdot g \Downarrow \mathtt{skip} \qquad e_2 \Downarrow v}{e_1 \gg e_2 \Downarrow v}$$

$$(\Downarrow \text{INVOKE-BAD})\quad \frac{e_1 \Downarrow \langle v_1, v_2\rangle \qquad v_2 \cdot \mathtt{skip} \Downarrow \lambda x.e \qquad e_2 \Downarrow u \qquad e[u/x] \Downarrow v}{e_1\langle e_2\rangle \Downarrow v}$$

$$(\Downarrow \text{LABPROJ}_1)\quad \frac{e \Downarrow \mathtt{l}}{\pi_1(e) \Downarrow \lambda x^{\tau\to\tau}.\Big[\langle \mathtt{l}, x\rangle \gg \mathtt{skip}\Big]} \qquad (\Downarrow \text{LABPROJ}_2)\quad \frac{e \Downarrow \mathtt{l}}{\pi_2(e) \Downarrow \lambda d^{\mathbb{1}}.\Big[\lambda x.\mathtt{l}\langle x\rangle\Big]}$$

Preservation and reflection of termination

Lemmas 5.1, 5.2, and 5.3 still hold under the new translation with basically the same proofs as before. However the translation no longer preserves evaluation (w.r.t. syntactic equality) because of the new cases for projection over labels, as can be evidenced by considering the configuration $P \equiv (\mathsf{L}, \mathsf{A}, \pi_2(\mathtt{l}))$ which evaluates to

$$V \equiv (\mathsf{L}, \mathsf{A}, \lambda d.\lambda x.\mathtt{l}\langle x\rangle).$$

Now, omitting the environments, we have

$$\ulcorner P \urcorner \;=\; \pi_2(\mathscr{L}(\lfloor \mathtt{l} \rfloor)) \Downarrow \lambda d.!\lfloor \mathtt{l} \rfloor \;\neq\; \ulcorner V \urcorner.$$

We could consider an appropriate variant notion of equivalence between configurations, but we have not been able to find any such notion that would support a suitably strong inductive hypothesis. For example, an inductive hypothesis that $\ulcorner P \urcorner \Downarrow$ holds does not imply that P evaluates to the translate of a valid CoreAML* term, and so, we cannot apply the hypothesis to a term involving substitution. Fortunately, using a continuation-based technique introduced by Pitts and Stark [1], we are able to prove that the translation preserves and relfects the evaluation of configurations (and hence evaluation of user terms).

Proposition 6.2. *For every well-typed* CoreAML* *configuration P, we have $P\Downarrow$ if and only if $\ulcorner P \urcorner \Downarrow$.*

Proof. (*Sketch*) The idea is to introduce a variant notion of termination (for both languages) whose rules permit inductive reasoning. Let $\mathbf{C}$ be the set of configurations, and $\mathbf{K}$ be the set of *continuations* for an appropriate model of computation for the language. We define a termination relation $\downarrow \subseteq \mathbf{C} \times \mathbf{K}$, where $(P, K)\downarrow$ means "P terminates for some continuation K". The point is that the rules defining the new termination relation $\downarrow$ enjoys a kind of *subformula property* (each premise is a subformula of the conclusion); and so it can quite easily be proved that

$$(P, K)\downarrow \text{ if and only if } \ulcorner (P, K) \urcorner \downarrow$$

for all continuations K and configurations P (assuming a translation between continuations). The result follows from the following correspondence between the two notions of termination

$$P\Downarrow \text{ if and only if } (P, \bullet)\downarrow$$

where $\bullet$ is the empty continuation. □

Fully abstract translation

To prove full abstraction, note that the proof of our earlier adequacy result (Theorem 5.6) only requires that the translation preserves and reflects termination of user terms. Therefore an identical proof can be used to prove the adequacy of the translation from CoreAML* to GRef*. The definability result also carries over because we have not reduced the expressive power of CoreAML at all by adding bad labels to it. Therefore it remains to prove the other direction of full abtraction, namely, that the translation preserves observational equivalence. We prove the contrapositive. Suppose for some CoreAML* user terms e_1 and e_2 that $\ulcorner e_1 \urcorner \not\simeq \ulcorner e_2 \urcorner$. By definition, there exists a closing GRef* context $\mathcal{C}[-]$ such that (without losing generality) $\mathcal{C}[\ulcorner e_1 \urcorner]\Downarrow$ but $\mathcal{C}[\ulcorner e_2 \urcorner]\Uparrow$. By definability, $\mathcal{C}[-] \simeq \ulcorner \mathcal{D}[-] \urcorner$ for some CoreAML context $\mathcal{D}[-]$. Since $\simeq$ is a congruence, we have $\ulcorner \mathcal{D} \urcorner[\ulcorner e_1 \urcorner]\Downarrow$ and $\ulcorner \mathcal{D} \urcorner[\ulcorner e_2 \urcorner]\Uparrow$. Now we apply Lemma 6.1 to deduce that $\mathcal{D}[e_1]$ and $\mathcal{D}[e_2]$ are well-typed terms of CoreAML*, and therefore by Lemma 5.2 we have

$$\ulcorner \mathcal{D}[e_1] \urcorner \Downarrow \quad \text{and} \quad \ulcorner \mathcal{D}[e_2] \urcorner \Uparrow .$$

But we are now done, because Proposition 6.2 tells us that $\mathcal{D}[e_1] \Downarrow$ and $\mathcal{D}[e_2] \Uparrow$. Therefore $e_1 \not\simeq e_2$ as required. To summarise we have:

Theorem 6.3 (Full abstraction)**.** *For well-typed* CoreAML* *user terms e_1 and e_2, we have $e_1 \simeq e_2$ if and only if $\ulcorner e_1 \urcorner \simeq \ulcorner e_2 \urcorner$.*

A pleasing consequence of the Theorem is that the fully abstract semantics of the target language GRef* is at once inherited by the source language CoreAML*. Since observational equivalence of CoreAML* coincides with the equational theory of the fully abstract model, we may therefore reason about the former in the model.

Space of translation: The complete picture

We have considered compositional translations from CoreAML to GRef* (Section 5), and from CoreAML* to GRef* (Section 6) – recall that the superscript "*" indicates the presence of bad labels or bad references. In each case, we have examined the faithfulness of the translation and calibrated it in terms of the satisfaction (or not) of Properties E, T, D, A and F. It is natural to ask if there is a compositional translation from CoreAML (with only good labels) to GRef (with only good references)? The answer is yes, and the translation can be defined by moving the composition behaviour of assignment to the aspect installation case. Precisely we substitute the following for the corresponding rules in Table 3:

$$\begin{aligned} \ulcorner \mathtt{newlab}_\tau \urcorner &\triangleq \mathtt{newref}\ x^{\ulcorner \tau\to\tau \urcorner} = \mathtt{id}[\ulcorner \tau \urcorner]\ \mathtt{in}\ x \\ \ulcorner e_1 \gg e_2 \urcorner &\triangleq \Big[\mathtt{let}\ \langle \mathtt{l}, g\rangle = \ulcorner e_1 \urcorner\ \mathtt{in}\ \mathtt{l} := (g \circ !\mathtt{l})\Big]; \ulcorner e_2 \urcorner \\ \ulcorner \mathtt{l} \urcorner &\triangleq \lfloor \mathtt{l} \rfloor \end{aligned}$$

We present the properties satisfied by the CoreAML-to-GRef translation, along with those of other translations, as follows.

Theorem 6.4. *Properties satisfied by the respective translations from* CoreAML/CoreAML* *to* GRef/GRef* *are as follows:*

	Properties of the Translation				
labs. / refs.	**E**	**T**	**D**	**A**	**F**
good / good	*Yes*	*Yes*	*No*	*Yes*	*No*
good / bad	*Yes*	*Yes*	*Yes*	*Yes*	*No*
bad / good	*No*	*No*	*No*	*No*	*No*
bad / bad	*No*	*Yes*	*Yes*	*Yes*	*Yes*

(Reading of the Table: E.g. the second row summarises properties satisfied by the CoreAML*-to-*GRef* *translation (see Section 5), which satisfies Properties E, T, D and A, but not F.)*

Proof. (*Sketch*) The fourth row summarises the findings of Section 6. We consider the first row, namely, the translation from

CoreAML to GRef. The translation preserves and reflects evaluation and hence termination because we do not need to consider projections over labels, and also because any assignment appearing in the translated term is a direct consequence of an installation of advice in the source term. Therefore it is irrelevant whether the aspect composition behaviour is encapsulated in the translate of the label or of the installation[2]. Definability (Property D) is not satisfied by the translation because no term translates into a term of type $\texttt{ref}[\tau]$ (where τ is not of the form $\sigma \to \sigma$).

Lastly we include the third row for the sake of completeness, although the "obvious" translation from bad-labels to good-references would not satisfy even the most basic properties. Consider $\ulcorner b\langle x\rangle \urcorner$ for some bad label b and variable x; it is not even a well-typed GRef term because it would result in the dereferencing of the pair $\ulcorner b \urcorner$. □

7. Further directions

In the concluding Section, we discuss possible applications of the fully abstract semantics and further directions.

Possible applications of the fully abstract semantics

In a fully abstract translation, the target language can be thought of as a highly accurate emulator of the source language. In the case of the CoreAML*-to-GRef* translation, since the target language has a fully abstract (game) model, the model *inherited* by CoreAML* is also fully abstract [7]. I.e. the equational theory of the inherited model coincides with observational equivalence (in CoreAML*). Thus reasoning in the model is equivalent to reasoning about the corresponding terms.

Fully abstract game models have recently found novel applications in software model checking [15]. For instance, the strategy-denotations of terms in the second-order fragment of Idealized Algol can be represented as regular expressions [2], but these results only apply to ground-type references. We would like to construct the game semantics of an Algol-like language with a store for second-order terms. Such a target language would correspond (via the kind of translations we have considered) to an aspectual language in which only program points of order at most one can be advised.

Further directions

Our translation suggests an implementation strategy for additive aspects in a functional setting. Specifically our work shows that (this fragment of) additive aspects can be encoded as a library in a language such as ML. It is unclear whether such an implementation would be efficient, but the fact that the advice is only composed once, at installation time, suggests that there are potential gains.

We also intend to extend our approach to *non-additive* aspects, thus giving a similar characterisation of constructs such as AspectJ's `around()` advice [6]. We believe that GRef* augmented by locally declared exceptions in the style of [11] is a suitable target language for a fully abstract translation. However such a translation would not give a fully abstract model for the source language because the game model described in [11] is for a language with only ground-type references. Further extensions would include dynamic pointcuts such as AspectJ's `cflow()`.

Finally current work on the semantics of aspect orientation assumes that aspects and base programs are in the same global scope. This simplified perspective does not reflect the important reality that program components are often developed / compiled separately and then composed together. The impact of applying aspects from an imported module to an existing module (and vice versa) is hard to understand. We think that one cannot claim to understand aspectual programs until and unless this issue – which goes to the heart of "obliviousness" – is properly addressed. The semantics of oblivious advice is in essence the semantics of *open* program fragments; whence we believe game semantics is a promising setting in which to study the problem.

[2] In fact this version of the translation could have been used for the "good/bad" translation as well, but not the "bad/bad" one

References

[1] A.Pitts and I.Stark. Operational reasoning for functions with local state. In Andrew Gordon and Andrew Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Publications of the Newton Institute, Cambridge University Press, 1998.

[2] D.Ghica and G.McCusker. The regular-language semantics of second-order idealised algol. *Theoretical Computer Science*, 2003.

[3] D.S.Dantas and D.Walker. Harmless advice. In *Workshop on Foundations of Object-Oriented Languages*, January 2005.

[4] D.Walker, S.Zdancewic, and J.Ligatti. A theory of aspects. In *Proceedings of the 8^{th} ACM SIGPLAN International Conference on Functional Programming*, pages 127–139, 2003.

[5] G.Bruns, R.Jagadeesan, A.Jeffrey, and J.Riely. μABC: A minimal aspect calculus. In *Proceedings of the 15^{th} International Conference on Concurrency Theory (CONCUR 2004)*, September 2004.

[6] G.Kiczales, E.Hilsdale, J.Hugunin, M.Kersten, J.Palm, and W.G.Grisold. An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP 2001)*, volume 2072 of *Lecture Notes In Computer Science*, pages 327–353, June 2001.

[7] G.McCusker. Full abstraction by translation. In *Proceedings of the Third Workshop of the Theory and Formal Methods Section*, 1996.

[8] J.Aldrich. Open modules: A proposal for modular reasoning in aspect-oriented programming. In *Proceedings of the FOAL Workshop*, March 2004.

[9] J.G.Riecke. Fully abstract translations between functional languages. In *Eighteenth Symposium on Principles of Programming Languages*, pages 245–254, 1991.

[10] J.H.Andrews. Process algebraic foundations of aspect oriented programming. In *3^{rd} International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes In Computer Science*. Springer-Verlag, 2001.

[11] J.Laird. A fully abstract games semantics of local exceptions. In *Proceedings, 16^{th} Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2001.

[12] M.Wand, G.Kiczales, and C.Dutchyn. A semantics for advice and dynamic join points in aspect oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, September 2004.

[13] R.E.Filman and D.P.Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation Concerns (OOPSLA 2001)*, 2000.

[14] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North Holland, 1978.

[15] S.Abramsky, D.Ghica, L.Ong, and A.Murawski. Applying game semantics to compositional software modelling and verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science*, pages 421–435. Springer-Verlag, 2004.

[16] S.Abramsky, K.Honda, and G.McCusker. A fully abstract game semantics for general references. In *Proceedings, 13^{th} Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1998.