

Full abstraction by translation

Guy McCusker

1 March 1996

Abstract

This paper shows how a fully abstract model for a rich metalanguage like *FPC* can be used to prove theorems about other languages. In particular, we use results obtained from a game semantics of *FPC* to show that the natural translation of the lazy λ -calculus into the metalanguage is fully abstract, thus obtaining a new full abstraction result from an old one. The proofs involved are very easy—all the hard work was done in giving the original games model. So far we have been unable to prove the completeness of our translation without recourse to the denotational model; we therefore have an indication of the worth of such fully abstract models.

1 Introduction

Plotkin, in his CSLI notes [18], showed how denotational semantics can be viewed as a two-stage process. First one defines a *metalanguage* which describes elements of the intended semantic model, usually some category of domains. Then to give semantics to a language \mathcal{L} it suffices to translate it into the metalanguage. While this is really no more than a formalization of the usual method of writing semantic equations, Plotkin demonstrated that it has an important advantage. Suppose that the translation of a term M of \mathcal{L} into the metalanguage is $\llbracket M \rrbracket$, and that the element of the denotational model which an expression e of the metalanguage describes is $\llbracket e \rrbracket$. The following *soundness* property is essential.

$$\llbracket \llbracket M \rrbracket \rrbracket = \llbracket \llbracket N \rrbracket \rrbracket \Rightarrow M \sim N$$

where \sim is the notion of program equivalence of interest for \mathcal{L} . By equipping the metalanguage with an operational semantics and hence a syntactic notion of equivalence \simeq , this requirement can be split into two.

$$\llbracket e \rrbracket = \llbracket f \rrbracket \Rightarrow e \simeq f. \tag{1}$$

$$\llbracket M \rrbracket \simeq \llbracket N \rrbracket \Rightarrow M \sim N. \tag{2}$$

In general, proving (1) is a non-trivial task; but if the metalanguage is sufficiently rich, it can be used in the semantics of a large variety of languages \mathcal{L} , and for each of these one only needs to prove (2) to establish soundness. What's more, the proof of (2) is essentially routine, so the burden of proof is greatly reduced by the generic soundness result of part (1). This philosophy has also been taken by Crole and Gordon [6,9].

Recent work on game semantics [14,15] has provided a model of Plotkin's metalanguage, now known as *FPC*, which is not only sound but also *fully abstract*, i.e. it satisfies the following *completeness* condition.

$$e \simeq f \Rightarrow \llbracket e \rrbracket = \llbracket f \rrbracket.$$

It is therefore natural to ask whether full abstraction results may also be factored into two: can one easily prove the following?

$$M \sim N \Rightarrow \langle M \rangle \simeq \langle N \rangle.$$

Of course, the hope is that when this is possible, it can be done easily and syntactically. Such a result would not only provide a fully abstract model of \mathcal{L} , but also the assurance that, even if a model of \mathcal{L} via *FPC* were not fully abstract, the loss of completeness did not occur at the syntactic translation stage.

This paper considers the simple example of the lazy λ -calculus [2,5,16]. We give a translation into *FPC* and perform the easy proof of its soundness. However, the completeness half is not so simple without the aid of some results about *FPC* obtained by the use of game semantics. Implicit in the proof of full abstraction in [14,15] is a strong characterization of program equivalence for *FPC*. We show how this can be used to establish completeness of the translation from lazy λ -calculus to *FPC*, thus arriving at a new full abstraction result bred from an old one. (A fully abstract games model of the lazy λ -calculus has already been described in [3]; but this used a different category of games from the one in which *FPC* is modelled, so we obtain here a genuinely new, if not altogether surprising, full abstraction theorem.) A useful characterization of *FPC* equivalence has been obtained by Gordon without the use of denotational semantics [10,11], but it is weaker than the one we present here, and in particular not good enough to establish our completeness theorem. In [19], Ritter and Pitts show that a certain translation between SML and a λ -calculus with reference types is fully abstract, again without recourse to a denotational model. This therefore raises two questions.

- Can the characterization of equivalence for *FPC* be obtained without the use of a fully abstract denotational model? This is important because there are many programming language features for which a fully abstract model is lacking.

- Can completeness of the translation perhaps be obtained in some other, more syntactic, way?

2 The metalanguage FPC

This section presents the syntax of *FPC* and equips it with a call-by-name operational semantics very similar to that of lazy functional languages such as Miranda¹. It should be noted that this is not the same as the semantics used by Plotkin, which is call-by-value. This language has also appeared in [7, 8, 12, 20]. We define the notion of program equivalence for *FPC*, and give a coinductive characterization of it, due to Gordon [11]. This is then strengthened using results obtained from the fully abstract model of *FPC* in [14, 15].

2.1 Syntax

There are two syntactic classes of variables: `TypeVar` for type variables, and `Var` for expression variables. The syntax of *FPC* is defined as follows.

$$\begin{aligned}
T &\in \text{TypeVar.} \\
\tau \in \text{Types} &::= T \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \mu T. \tau. \\
x &\in \text{Var.} \\
M \in \text{Exp} &::= x \\
&\mid \text{inl}_{\tau_1, \tau_2}(M) \mid \text{inr}_{\tau_1, \tau_2}(M) \\
&\mid \text{case } M \text{ of inl}(x_1).M_1 \text{ or inr}(x_2).M_2 \\
&\mid (M_1, M_2) \\
&\mid \text{fst}(M) \mid \text{snd}(M) \\
&\mid \lambda x : \tau. M \\
&\mid M_1(M_2) \\
&\mid \text{intro}_{\mu T. \tau}(M) \\
&\mid \text{elim}(M).
\end{aligned}$$

The type tags on $\text{inl}_{\tau_1, \tau_2}(M)$, $\text{inr}_{\tau_1, \tau_2}(M)$, and $\text{intro}_{\mu T. \tau}(M)$ are necessary to ensure that a given term-in-context can have only one type. However, we will omit them whenever we think we can get away with it.

In fact the syntax given above is abbreviated: in general it is useful to have sums of all arities in the language rather than just binary sums. In particular, we will make use of the unary case, which corresponds to *lifting*. Let us introduce some special informal syntax for this type constructor. The

¹When used as the name of a functional programming system, Miranda is a trade mark of Research Software Ltd.

lifted version of the type τ will be written as τ_{\perp} . The injection, corresponding to the unary version of $\text{inl}()$ and $\text{inr}()$, is $\text{up}()$, and the destructor term, corresponding to $\text{case } M \text{ of } \text{inl}(\mathbf{x}_1).M_1 \text{ or } \text{inr}(\mathbf{x}_2).M_2$, is $\text{conv } M \text{ in } \mathbf{x}.N$. The type formation rules, expression formation rules and operational semantics for lifting should be clear from those for the sums, which are detailed below.

A well-formed type consists of a list of distinct type variables Θ and a type τ , all of whose free variables appear in Θ . We will write $\Theta \vdash \tau$ to indicate that τ is a well-formed type in context Θ . The well-formed types are defined inductively in Figure 1. The variable T is bound in $\mu\mathsf{T}.\tau$ and we denote substitution of a type τ' for the free occurrences of T in τ by $\tau[\mathsf{T} \mapsto \tau']$. As usual, we identify types up to α -equivalence. For the most

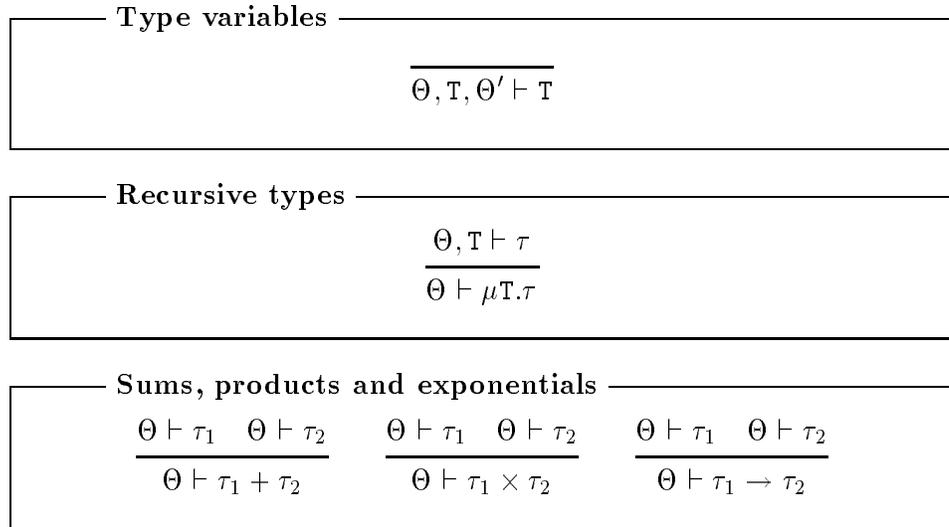


Figure 1: Well-formed types of *FPC*.

part we are going to be concerned with closed types, i.e. those types τ such that $\vdash \tau$ is derivable.

An expression context $\Theta, ?$ consists of a list of distinct type variables, Θ , and a list of (variable,type) pairs, $?$. The variables occurring in $?$ must all be distinct, and if τ is a type occurring in $?$ then $\Theta \vdash \tau$ must be a well-formed type. Each entry in $?$ is written as $\mathbf{x} : \tau$. Well-formed expressions are given by judgements $\Theta, ? \vdash M : \tau$ where $\Theta, ?$ is an expression context; the inductive definition is given in Figure 2. The expression $\text{case } M \text{ of } \text{inl}(\mathbf{x}_1).M_1 \text{ or } \text{inr}(\mathbf{x}_2).M_2$ binds \mathbf{x}_1 in M_1 and \mathbf{x}_2 in M_2 , while $\lambda\mathbf{x} : M$. binds \mathbf{x} in M . Expressions are identified up to α -equivalence, and we denote the substitution of N for free occurrences of \mathbf{x} in M by $M[N/\mathbf{x}]$. Notice that the type context Θ plays very little part in this definition. In

Variables

$$\frac{}{\Theta, ?_1, \mathbf{x} : \tau, ?_2 \vdash \mathbf{x} : \tau}$$

Sums

$$\frac{\Theta, ? \vdash M : \tau \quad \Theta \vdash \tau'}{\Theta, ? \vdash \text{inl}_{\tau, \tau'}(M) : \tau + \tau'} \quad \frac{\Theta, ? \vdash M : \tau \quad \Theta \vdash \tau'}{\Theta, ? \vdash \text{inr}_{\tau', \tau}(M) : \tau' + \tau}$$

$$\frac{\Theta, ? \vdash M : \tau_1 + \tau_2 \quad \Theta, ?, \mathbf{x}_1 : \tau_1 \vdash M_1 : \tau \quad \Theta, ?, \mathbf{x}_2 : \tau_2 \vdash M_2 : \tau}{\Theta, ? \vdash \text{case } M \text{ of } \text{inl}(\mathbf{x}_1).M_1 \text{ or } \text{inr}(\mathbf{x}_2).M_2 : \tau}$$

Products

$$\frac{\Theta, ? \vdash M_1 : \tau_1 \quad \Theta, ? \vdash M_2 : \tau_2}{\Theta, ? \vdash (M_1, M_2) : \tau_1 \times \tau_2}$$

$$\frac{\Theta, ? \vdash M : \tau_1 \times \tau_2}{\Theta, ? \vdash \text{fst}(M) : \tau_1} \quad \frac{\Theta, ? \vdash M : \tau_1 \times \tau_2}{\Theta, ? \vdash \text{snd}(M) : \tau_2}$$

Exponentials

$$\frac{\Theta, ?, \mathbf{x} : \tau_1 \vdash M : \tau_2}{\Theta, ? \vdash \lambda \mathbf{x} : \tau_1. M : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Theta, ? \vdash M_1 : \tau_1 \rightarrow \tau_2 \quad \Theta, ? \vdash M_2 : \tau_1}{\Theta, ? \vdash M_1(M_2) : \tau_2}$$

Recursive types

$$\frac{\Theta, ? \vdash M : \tau[\mathbf{T} \mapsto \mu \mathbf{T}. \tau]}{\Theta, ? \vdash \text{intro}_{\mu \mathbf{T}. \tau}(M) : \mu \mathbf{T}. \tau}$$

$$\frac{\Theta, ? \vdash M : \mu \mathbf{T}. \tau}{\Theta, ? \vdash \text{elim}(M) : \tau[\mathbf{T} \mapsto \mu \mathbf{T}. \tau]}$$

Figure 2: Well-formed expressions of *FPC*.

fact, we will mainly work with terms of closed type, so that Θ is empty. However, as it stands the language supports *parametricity*; this is not important for us, but Fiore treats it in his thesis [7].

Definition An *FPC program* is a closed term of closed type, i.e. an expression M such that $\vdash M : \tau$ is derivable. We write Prog for the set of programs, tagged with their types. The notion of *context* $C[-]$ with hole of a given type can also be defined; informally, a context is just a term with (possibly several occurrences of) a “hole” in it, and $C[M]$ denotes the result of filling in each hole with the expression M . Unlike expressions, contexts are *not* identified modulo α -equivalence. We will be interested in closed contexts of type τ , that is those contexts $C[-]$ such that if M is a closed expression of the same type as the hole, then $C[M]$ is a closed expression of type τ . If $C[-]$ is such a context, we write $C[-] : \tau$.

Before finishing our discussion of the syntax, it is worth remarking that though there is no explicit recursion combinator in the language, recursion is nonetheless supported. This is because the Curry fixed point combinator

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

of the untyped λ -calculus can be encoded in *FPC*, by the power of recursive types. This is left as an exercise for the interested reader; the bored reader can find it in Gunter’s book [12].

An important consequence of this is that there is a divergent term of each type, given by $Y(\lambda \mathbf{x} : \tau.\mathbf{x})$. We will denote this term by Ω , or Ω_τ when the type is important.

2.2 Operational semantics

We equip the language *FPC* with an operational semantics. In contrast to the work of Fiore and Plotkin, our semantics is call-by-name. As usual it is given in terms of a “big-step” evaluation relation \Downarrow . For readability, we assume all the terms which appear in the definition below are well-formed in some expression context, and omit this context. The definition of the evaluation relation is given in Figure 3.

2.3 Program equivalence in FPC

We are now in a position to define the notion of program equivalence that we are interested in. As usual, this is given in terms of an *observational preorder*, defined as follows. Given two programs M and N of type τ , define

$$M \sqsubseteq N \iff \forall C[-] : \tau_1 + \tau_2 [C[M]\Downarrow \Rightarrow C[N]\Downarrow]$$

Sums

$$\begin{array}{c}
\overline{\text{inl}_{\tau,\tau'}(M) \Downarrow \text{inl}_{\tau,\tau'}(M)} \quad \overline{\text{inr}_{\tau,\tau'}(M) \Downarrow \text{inr}_{\tau,\tau'}(M)} \\
\\
\frac{M \Downarrow \text{inl}(M') \quad M_1[M'/\mathbf{x}_1] \Downarrow M''}{\text{case } M \text{ of } \text{inl}(\mathbf{x}_1).M_1 \text{ or } \text{inr}(\mathbf{x}_2).M_2 \Downarrow M''} \\
\\
\frac{M \Downarrow \text{inr}(M') \quad M_2[M'/\mathbf{x}_2] \Downarrow M''}{\text{case } M \text{ of } \text{inl}(\mathbf{x}_1).M_1 \text{ or } \text{inr}(\mathbf{x}_2).M_2 \Downarrow M''}
\end{array}$$

Products

$$\begin{array}{c}
\overline{(M_1, M_2) \Downarrow (M_1, M_2)} \\
\\
\frac{M \Downarrow (M_1, M_2) \quad M_1 \Downarrow M'}{\text{fst}(M) \Downarrow M'} \quad \frac{M \Downarrow (M_1, M_2) \quad M_2 \Downarrow M'}{\text{snd}(M) \Downarrow M'}
\end{array}$$

Exponentials

$$\begin{array}{c}
\overline{\lambda \mathbf{x} : \tau. M \Downarrow \lambda \mathbf{x} : \tau. M} \\
\\
\frac{M_1 \Downarrow \lambda \mathbf{x} : \tau. M \quad M[M_2/\mathbf{x}] \Downarrow M'}{M_1(M_2) \Downarrow M'}
\end{array}$$

Recursive types

$$\begin{array}{c}
\overline{\text{intro}_{\mu\tau,\tau}(M) \Downarrow \text{intro}_{\mu\tau,\tau}(M)} \\
\\
\frac{M \Downarrow \text{intro}(M') \quad M' \Downarrow M''}{\text{elim}(M) \Downarrow M''}
\end{array}$$

Figure 3: Operational semantics of *FPC*.

where τ_1 and τ_2 are any closed types, and $M \Downarrow$ means that there exists some N such that $M \Downarrow N$. We then define program equivalence \simeq by

$$M \simeq N \iff M \sqsubseteq N \wedge N \sqsubseteq M.$$

We have chosen to observe only convergence of programs in contexts of (binary) sum type. Two programs are equivalent if, whatever context of sum type they are put in, either they both converge or they both diverge. A consequence of our decision to observe convergence at sum types alone is that the following equivalences hold.

- $\lambda \mathbf{x} : \tau. \Omega_{\tau'} \simeq \Omega_{\tau \rightarrow \tau'}$.
- $(\Omega_{\tau}, \Omega_{\tau'}) \simeq \Omega_{\tau \times \tau'}$.

The first would not hold if we allowed observation of convergence at function types; the second would fail if we allowed observation at product types. The decision to observe only convergence at sum types is by no means the only possible one. For example, Haskell [13] allows observation of convergence at product types too. Our notion of equivalence is more like that of Miranda.

It is easy to show that it is equivalent to observe convergence at sum types of any arity, and in particular at lifted types. We will make use of this fact later on.

Andrew Gordon has given a coinductive characterization of \simeq using the technique of *applicative bisimulation*. Applicative bisimulation is the analogue in functional programming of Park's bisimulation [17] from concurrency, and was first used by Abramsky in his work on the lazy λ -calculus. We now review and refine Gordon's work. To make clear the connection with bisimulation as used in concurrency theory, Gordon presents applicative bisimulation in terms of a labelled transition system on *FPC* terms, as follows. It is a family of relations $(\xrightarrow{\alpha} \subseteq \text{Prog} \times \text{Prog} \mid \alpha \in \text{Act})$ indexed by the set *Act* of **actions**.

$$\text{Act} = \{\text{inl}, \text{inr}, \text{fst}, \text{snd}, \text{elim}\} \cup \{\text{@}M \mid M \in \text{Prog}\}.$$

The labelled transition system is defined inductively as follows.

$$\begin{array}{c} \frac{M \Downarrow \text{inl}(M')}{M \xrightarrow{\text{inl}} M'} \quad \frac{M \Downarrow \text{inr}(M')}{M \xrightarrow{\text{inr}} M'} \\ \frac{M : A \times B}{M \xrightarrow{\text{fst}} \text{fst}(M)} \quad \frac{M : A \times B}{M \xrightarrow{\text{snd}} \text{snd}(M)} \\ \frac{M : A \rightarrow B \quad N : A}{M \xrightarrow{\text{@}N} MN} \\ \frac{M : \mu T. \tau}{M \xrightarrow{\text{elim}} \text{elim}(M)} \end{array}$$

Notice the difference in the rules for sum types: there the term is evaluated in order to calculate its transitions, whereas at other types, we merely apply a ‘destructor’ term. This reflects the fact that we can observe convergence at sum types directly.

Let \mathcal{R} be a binary relation on *FPC* programs such that if $M\mathcal{R}N$ then M and N have the same type. We say \mathcal{R} is a **bisimulation** if, for each $M\mathcal{R}N$,

- if $M \xrightarrow{\alpha} M'$ then there exists N' such that $N \xrightarrow{\alpha} N'$ and $M'\mathcal{R}N'$;
- if $N \xrightarrow{\alpha} N'$ then there exists M' such that $M \xrightarrow{\alpha} M'$ and $M'\mathcal{R}N'$.

Define **similarity** \approx to be the largest bisimulation. As usual this means that \approx is the greatest fixed point of a monotone operator on relations; the practical upshot of this is that to show that $M \approx N$, it suffices to find any bisimulation \mathcal{R} such that $M\mathcal{R}N$.

The following theorem appears in [10].

Theorem 1 (Gordon) Similarity and observational equivalence are the same relation. That is to say, for any *FPC* programs M and N of the same type,

$$M \simeq N \iff M \approx N.$$

In [15], a fully abstract game semantics is given for *FPC*. This model allows us to prove the above result very easily, although of course it requires a good deal of effort to construct the model in the first place. However, the proof in fact shows a stronger result, namely that in the clause for terms of function type, it suffices to consider transitions $\xrightarrow{@N}$ where N is a program whose denotation in the games model is finite. Fortunately, the finite elements can easily be characterized syntactically, by the following grammar. We will call the terms generated by this grammar the **compact** terms.

$$\begin{aligned} K & ::= \Omega \mid \text{inl}(K) \mid \text{inr}(K) \\ & \quad \mid \lambda \mathbf{x} : \tau. K \mid (K_1, K_1) \mid \text{intro}(K) \\ & \quad \mid \text{case } \alpha \text{ of } \text{inl}(\mathbf{x}_1).K_1 \text{ or } \text{inr}(\mathbf{x}_2).K_2 \\ \alpha & ::= \mathbf{x} \mid \text{fst}(\alpha) \mid \text{snd}(\alpha) \\ & \quad \mid \text{outl}(\alpha) \mid \text{outr}(\alpha) \mid \text{elim}(\alpha) \\ & \quad \mid \alpha K_1 \dots K_n \end{aligned}$$

In the clause for case expressions, the terms K_1 and K_2 are of sum type, K_1 does not contain \mathbf{x}_1 and K_2 does not contain \mathbf{x}_2 . The term outl is shorthand for

$$\lambda \mathbf{x} : \tau_1 + \tau_2. \text{case } \mathbf{x} \text{ of } \text{inl}(\mathbf{x}_1). \mathbf{x}_1 \text{ or } \text{inr}(\mathbf{x}_2). \Omega_{\tau_1}.$$

so that **outl** projects from $\tau_1 + \tau_2$ to τ_1 . The term **outr** is defined symmetrically. In the unary case we write **dn** for this term, dually to **up**.

This grammar is particularly restricted in its use of application. One can only apply a variable to other compact terms, and only then if the result will be used as the argument to a case expression; having tested for convergence, control then passes to the subterm K_1 or K_2 which makes no further use of the application. This means that the types of subterms of a compact term are much more predictable than for general terms, a fact which facilitates the proof of completeness to follow.

We can now define a new transition system which has the same rules as before except that the rule for function types is replaced by the more restricted rule below.

$$\frac{M : A \rightarrow B \quad N : A \quad N \text{ compact}}{M \xrightarrow{@N} MN}$$

We define **compact bisimulation** with respect to the new transition system in the same way as before, arriving at **compact similarity**. We then have the following characterizations of program equivalence for *FPC*.

Theorem 2

1. Compact similarity, similarity and observational equivalence coincide.
2. Given terms M and N of type τ , $M \simeq N$ if and only if for all compact terms $\mathbf{x} : \tau \vdash f : \tau_1 + \tau_2$ we have

$$f[M/\mathbf{x}]\Downarrow \iff f[N/\mathbf{x}]\Downarrow.$$

3 The lazy λ -calculus

We present the form of the lazy λ -calculus which contains the “sequential convergence testing” constant **C**; this language is known as $\lambda\mathcal{C}$. After describing the operational semantics and the notion of operational equivalence, we define a translation from $\lambda\mathcal{C}$ into *FPC* which can be seen as a description of its denotational semantics: the translation induces a model of $\lambda\mathcal{C}$ in any model of *FPC*, including the games model.

3.1 Syntax and operational semantics

The syntax of $\lambda\mathcal{C}$ is that of untyped λ -calculus with an additional constant **C**.

$$M ::= x \mid \lambda x.M \mid MM \mid \mathbf{C}.$$

We can give an inductive definition of the well formed terms $x_1, \dots, x_n \vdash M$ in a similar style to that of *FPC* terms in Figure 2, but we omit it here because it is so simple.

The operational semantics of $\lambda\mathcal{C}$ is defined as follows.

$$\frac{\overline{\lambda x.M \Downarrow \lambda x.M}}{\quad} \quad \frac{\overline{C \Downarrow C}}{\quad}$$

$$\frac{M \Downarrow \lambda x.P \quad P[Q/x] \Downarrow N}{MQ \Downarrow N} \quad \frac{M \Downarrow C \quad N \Downarrow}{MN \Downarrow \lambda x.x}$$

Again we define observational equivalence in terms of an observational preorder; in this case we can observe convergence of any term, because there are no types (or rather, there is only one type).

$$M \preceq N \iff \forall C[-]. C[M] \Downarrow \Rightarrow C[N] \Downarrow$$

where $C[-]$ denotes a closed context. We write the associated equivalence relation as \sim . Abramsky [5] shows that this relation is the largest applicative bisimulation, where a relation \mathcal{R} is an applicative bisimulation if whenever $M \mathcal{R} N$, then $M \Downarrow$ if and only if $N \Downarrow$, and in that case, $M P R N P$ for all programs P .

3.2 Denotational semantics via FPC

We now give a “denotational semantics” for $\lambda\mathcal{C}$ by translating it into *FPC*, so using *FPC* as a metalanguage for some semantic category, such as the category of games mentioned previously, or more traditionally a category of domains. Terms of $\lambda\mathcal{C}$ will be interpreted by terms of *FPC* of type D , where D is the solution to the “domain equation”

$$D = (D \rightarrow D)_\perp$$

i.e. D is the type $\mu\mathbf{T}.\mathbf{T} \rightarrow \mathbf{T}_\perp$. For more discussion of the use of this domain equation, see [5].

We now inductively define a translation $\llbracket - \rrbracket$ taking $\lambda\mathcal{C}$ terms to *FPC* terms. The translation is precisely the semantics that one would ordinarily give to $\lambda\mathcal{C}$ in a domain satisfying the above equation. The only difference is in the level of formality. First, we associate with each variable x of $\lambda\mathcal{C}$ a variable $\mathbf{x} : D$ of *FPC*. Given a list $?$ of $\lambda\mathcal{C}$ -variables, $\llbracket ? \rrbracket$ denotes the list of associated *FPC* variables. A term $? \vdash M$ of $\lambda\mathcal{C}$ is translated to $\llbracket ? \rrbracket \vdash m : D$ for some *FPC* term m , as follows.

$$\begin{aligned} \llbracket ? , x \vdash x \rrbracket &= \llbracket ? \rrbracket , \mathbf{x} : D \vdash \mathbf{x} : D \\ \llbracket ? \vdash C \rrbracket & \\ = \llbracket ? \rrbracket \vdash \mathbf{intro}(\mathbf{up}(\lambda\mathbf{x} : D.\mathbf{conv}(\mathbf{elim}(x)) \mathbf{in} \mathbf{y}.\mathbf{intro}(\mathbf{up}(\lambda\mathbf{z} : D.\mathbf{z})))) & \\ \llbracket ? , x \vdash M \rrbracket &= \llbracket ? \rrbracket , \mathbf{x} : D \vdash m : D \\ \llbracket ? \vdash \lambda x.M \rrbracket &= ? \vdash \mathbf{intro}(\mathbf{up}(\lambda\mathbf{x} : D.m)) : D \end{aligned}$$

$$\frac{\langle ? \vdash M \rangle = \langle ? \rangle \vdash m : D \quad \langle ? \vdash N \rangle = \langle ? \rangle \vdash n : D}{\langle ? \vdash MN \rangle = \langle ? \rangle \vdash \text{conv}(\text{elim}(m)) \text{ in } \mathbf{x.x}n}$$

4 Soundness and completeness

Now that we have defined the metalanguage FPC , our object language $\lambda_{\mathcal{C}}$ and the translation from $\lambda_{\mathcal{C}}$ into FPC , we can set about proving that the translation is fully abstract. The first part of the proof, soundness, follows from a computational adequacy result which is very easy to establish. The second part, completeness, makes essential use of the characterization of program equivalence in FPC given in section 2.

4.1 Soundness

We now give the straightforward proof of the soundness of our translation, which hinges on a computational adequacy result. This proof should be compared with the original soundness proof for the metalanguage itself [15], which is much longer and relies on the method of “formal approximation relations” used by Plotkin [18].

First we have a trivial substitution lemma for the translation.

Lemma 3 If $? \vdash M$ and $?, x \vdash N$ then $? \vdash N[M/x]$ and

$$\langle N[M/x] \rangle = \langle N \rangle[\langle M \rangle/\mathbf{x}].$$

Proof Follows from the compositionality of the translation. \square

The next lemma shows that the semantics respects evaluation as it should.

Lemma 4 If $M \Downarrow N$ then $\langle M \rangle \Downarrow \langle N \rangle$.

Proof By induction on the derivation of $M \Downarrow N$. The base cases, for $\lambda x.M$ and \mathbf{C} , are trivial. For the inductive step, we shall just consider the case of the following rule.

$$\frac{M \Downarrow \lambda x.M' \quad M'[N/x] \Downarrow P}{MN \Downarrow P}$$

(The other rule is very similar.) By the inductive hypothesis, we have

$$\langle M \rangle \Downarrow \text{intro}(\text{up}(\lambda \mathbf{x} : D. \langle M' \rangle))$$

and

$$\langle M' \rangle[\langle N \rangle/\mathbf{x}] \Downarrow \langle P \rangle,$$

using the substitution lemma. But by definition of the translation,

$$\langle MN \rangle = \text{conv} \text{elim}(\langle M \rangle) \text{ in } \mathbf{x.x} \langle N \rangle.$$

The operational semantics of FPC gives us $\text{elim}(\langle M \rangle) \Downarrow \text{up}(\lambda \mathbf{x} : D. \langle M' \rangle)$ and $(\lambda \mathbf{x} : D. \langle M' \rangle) \langle N \rangle \Downarrow \langle P \rangle$, so we can conclude that $\langle MN \rangle \Downarrow \langle P \rangle$ as required. \square

We now prove the computational adequacy result which tightens the correspondence between the operational properties of terms and their translations.

Lemma 5 If $\llbracket M \rrbracket \Downarrow$ then $M \Downarrow$.

Proof By induction on the length of derivation that $\llbracket M \rrbracket \Downarrow$. The base case arises if M is $\lambda x.M'$ or C , but in these cases $M \Downarrow$ trivially. This just leaves the case of MN . Suppose that $\llbracket MN \rrbracket \Downarrow$. We have

$$\llbracket MN \rrbracket = \text{conv elim}(\llbracket M \rrbracket) \text{ in } \mathbf{x.x}(\llbracket N \rrbracket).$$

so $\llbracket MN \rrbracket \Downarrow$ must have been derived from

$$\text{elim}(\llbracket M \rrbracket) \Downarrow \text{up}(e) \text{ and } e(\llbracket N \rrbracket) \Downarrow f$$

for some *FPC* terms e and f . The fact that $\text{elim}(\llbracket M \rrbracket) \Downarrow \text{up}(e)$ must itself have been derived from

$$\llbracket M \rrbracket \Downarrow \text{intro}(e') \text{ and } e' \Downarrow \text{up}(e)$$

so by the inductive hypothesis, we have $M \Downarrow P$ for some P . There are now two cases.

- P is $\lambda x.M'$. Then we have

$$\llbracket M \rrbracket \Downarrow \text{intro}(\text{up}(\lambda x : D.\llbracket M' \rrbracket))$$

so $e = \lambda x : D.\llbracket M' \rrbracket$, and we have $\lambda x : D.\llbracket M' \rrbracket(\llbracket N \rrbracket) \Downarrow f$. This was derived from $\llbracket M' \rrbracket[\llbracket N \rrbracket/\mathbf{x}] \Downarrow f$, which is to say $\llbracket M'[N/x] \rrbracket \Downarrow f$. So by the inductive hypothesis, $M'[N/x] \Downarrow$ and hence $MN \Downarrow$ as required.

- P is C . In this case, we have

$$\llbracket M \rrbracket \Downarrow \text{intro}(\text{up}(\lambda x : D.\text{conv}(\text{elim}(x)) \text{ in } \mathbf{y.intro}(\text{up}(\lambda z : D.z))))$$

so $e = \lambda x : D.\text{conv}(\text{elim}(x)) \text{ in } \mathbf{y.intro}(\text{up}(\lambda z : D.z))$ and we have

$$\lambda x : D.\text{conv}(\text{elim}(x)) \text{ in } \mathbf{y.intro}(\text{up}(\lambda z : D.z))(\llbracket N \rrbracket) \Downarrow.$$

This must be derived from

$$\text{conv}(\text{elim}(\llbracket N \rrbracket)) \text{ in } \mathbf{y.intro}(\text{up}(\lambda z : D.z)) \Downarrow$$

which itself comes from the fact that $\llbracket N \rrbracket \Downarrow$, so by the inductive hypothesis, $N \Downarrow$ and hence $MN \Downarrow$ as required. \square

Finally we can prove soundness.

Proposition 6 If $\llbracket M \rrbracket \simeq \llbracket N \rrbracket$ then $M \sim N$.

Proof Let M and N be terms of $\lambda_{\mathcal{C}}$ such that $\llbracket M \rrbracket \simeq \llbracket N \rrbracket$. Let $C[-]$ be any closed $\lambda_{\mathcal{C}}$ -context. By compositionality of the translation together with the fact that \simeq is a congruence on FPC terms, we have $\llbracket C[M] \rrbracket \simeq \llbracket C[N] \rrbracket$. If $C[M] \Downarrow$ then we have $\llbracket C[M] \rrbracket \Downarrow \mathbf{intro}(\mathbf{up}(e))$ for some e , so $\mathbf{elim}(\llbracket C[M] \rrbracket) \Downarrow$, and this is of sum type, so $\mathbf{elim}(\llbracket C[N] \rrbracket) \Downarrow$, and hence $\llbracket C[N] \rrbracket \Downarrow$, so by adequacy, $C[N] \Downarrow$. Similarly, if $C[N] \Downarrow$ then $C[M] \Downarrow$, so we conclude that $M \sim N$. \square

This proof is completely routine and very easy to carry out. It is this fact which makes the methodology of using a metalanguage for denotational semantics attractive. If instead we gave semantics to the lazy λ -calculus directly, much more effort would be required to establish soundness.

4.2 Completeness

In this section we make use of the strong characterization of program equivalence in FPC given in section 2.3 to show that the translation from $\lambda_{\mathcal{C}}$ into FPC is not just sound but also complete. As usual for such completeness results, the proof rests on *definability*: we show that sufficiently many FPC programs are in fact the denotation of some term of $\lambda_{\mathcal{C}}$. Here, “sufficiently many” means enough to distinguish any two distinct FPC programs, so that if we have $\llbracket M \rrbracket \not\simeq \llbracket N \rrbracket$ we can find some FPC context $C[-]$ such that $C[\llbracket M \rrbracket] \Downarrow$ but not $C[\llbracket N \rrbracket] \Downarrow$, and moreover this FPC context is the denotation of a $\lambda_{\mathcal{C}}$ context $C'[-]$, so we have $C'[M] \Downarrow$ but not $C'[N] \Downarrow$, which allows us to conclude that $M \not\sim N$.

Without the notion of compact terms introduced earlier, such a definability result would be hard to prove. The first temptation is to attempt to prove by structural induction that all FPC terms of type D are $\lambda_{\mathcal{C}}$ definable. This approach fails immediately because in a term MN , the subterms M and N may be of types very different from D . However, for compact terms, this case cannot arise, and as we have seen, the compact terms suffice to distinguish any two distinct FPC programs.

Proposition 7 For any compact term $\mathbf{x}_1 : D, \dots, \mathbf{x}_2 : D \vdash K : D$ there exists a $\lambda_{\mathcal{C}}$ -term $x_1, \dots, x_n \vdash M$ such that $\llbracket M \rrbracket \simeq K$, that is to say K is *definable*. Similarly, for any compact term K of type $(D \rightarrow D)_{\perp}$ the term $\mathbf{intro}(K)$ is definable, and for any compact term K of type $D \rightarrow D$, the term $\mathbf{intro}(\mathbf{up}(K))$ is definable.

Proof By induction on the structure of K . The case of Ω is trivial, since there is the $\lambda_{\mathcal{C}}$ -term $(\lambda x.xx)(\lambda x.xx)$ which diverges and hence has divergent denotation, by computational adequacy. The case of $\mathbf{intro}(K)$ comes directly from the inductive hypothesis, as does the case of $\mathbf{up}(K)$.

The cases of (K, K) , $\text{inl}(K)$ and $\text{inr}(K)$ are all vacuous. The only case of the form $\lambda \mathbf{x}.K$ which is of interest has type $D \rightarrow D$, so we need to show that $\text{intro}(\text{up}(\lambda \mathbf{x} : D.K))$ is definable. But the inductive hypothesis says that $?, \mathbf{x} \vdash K$ is definable, so this follows by definition of the translation of λ -abstractions. This just leaves the case statement

$$\text{case } \alpha \text{ of } \text{inl}(\mathbf{x}_1).K_1 \text{ or } \text{inr}(\mathbf{x}_2).K_2$$

(and similar statements for sums of different arities.) Note that since all the free variables have type D , the α in question can only be of type $(D \rightarrow D)_\perp$. So we in fact have

$$\text{conv } \alpha \text{ in } \mathbf{x}.K$$

where K has ‘sum’ type, which must therefore be $(D \rightarrow D)_\perp$, so we are to show that $\text{intro}(\text{conv } \alpha \text{ in } \mathbf{x}.K)$ is definable, and it can easily be shown that this is equivalent to $\text{conv } \alpha \text{ in } \mathbf{x}.\text{intro}(K)$. The inductive hypothesis immediately tells us that $\text{intro}(K)$ is definable by some term M of $\lambda\mathcal{C}$. Inspection of the grammar for α shows that we must have $\alpha = \text{elim}(\alpha')$ for some α' of type D . Suppose for now that α' is definable, so that $\alpha' = \llbracket N \rrbracket$ for some N . We need to show that $\text{conv } \text{elim}(\llbracket N \rrbracket) \text{ in } \mathbf{x}.\llbracket M \rrbracket$ is definable. But it’s easy to show that this is equivalent to $\llbracket (CN)M \rrbracket$ —one converges if and only if the other does, and then to the same value, namely that of $\llbracket M \rrbracket$. Therefore we just need to show that each α' of type D is definable. This we do by induction on the structure of α' .

- The base case: $\alpha' = \mathbf{x}$. This is trivially definable.
- The inductive step: we need only consider the case

$$\alpha' = \text{dn}(\text{elim}(\alpha''))(\llbracket P \rrbracket)$$

for some $\lambda\mathcal{C}$ -term P , since by the inductive hypothesis each argument we supply to α' is definable. Our inner inductive hypothesis tells us that $\alpha'' = \llbracket Q \rrbracket$ for some Q . We can calculate as follows.

$$\begin{aligned} \alpha' &= (\text{conv } \text{elim}(\llbracket Q \rrbracket) \text{ in } \mathbf{y}.\mathbf{y})(\llbracket P \rrbracket) \\ &\simeq \text{conv } \text{elim}(\llbracket Q \rrbracket) \text{ in } \mathbf{y}.\mathbf{y}(\llbracket P \rrbracket) \\ &= \llbracket QP \rrbracket. \end{aligned} \quad \square$$

Completeness, and hence full abstraction, is now easy to establish.

Theorem 8 The translation from $\lambda\mathcal{C}$ into FPC is fully abstract.

Proof We already have soundness, so we just need completeness. Suppose for some $\lambda\mathcal{C}$ terms M and N that $\llbracket M \rrbracket \not\approx \llbracket N \rrbracket$. Then using Theorem 2, there is a compact term $\mathbf{x} : D \vdash K : D$ such that (wlog) $f[\llbracket M \rrbracket / \mathbf{x}] \Downarrow$ but $f[\llbracket N \rrbracket / \mathbf{x}]$ does not converge. By the above, $f \simeq \llbracket P \rrbracket$ for some term $x \vdash P$ of $\lambda\mathcal{C}$, and

then by adequacy we have $P[M/x]\Downarrow$ but not $P[N/x]\Downarrow$. Hence $M \not\approx N$. We have shown that

$$(\Downarrow M) \not\approx (\Downarrow N) \Rightarrow M \not\approx N$$

and taking the contrapositive of this gives completeness. \square

5 Conclusions

We have shown how the metalanguage *FPC* can be used as a stepping stone in giving semantics to a different programming language, and how this facilitates an easy proof of soundness of this semantics. Furthermore, using some results obtained by studying a fully abstract games model of *FPC*, we were able to show that for a particular language, the lazy λ -calculus, the translation into *FPC* is not only sound but fully abstract, so any fully abstract model of *FPC* induces a fully abstract model of $\lambda_{\mathcal{C}}$ too. In particular, we now know that the games model in [15] contains a fully abstract model of $\lambda_{\mathcal{C}}$, subsuming the earlier result of [3]. (In fact, the games considered in [1, 3, 4] provide a fully abstract model of an impoverished version of *FPC* which only has lifting, not more general sum types.) The proof of this result was extremely simple, thanks to the power of the theorem imported from game semantics.

So far, the search for a self-contained proof of completeness of the translation presented here, that is to say a proof which does not rely on the existence of a fully abstract model, has been unsuccessful. The method of Ritter and Pitts [19] does not seem applicable. It would involve translating back from *FPC* to $\lambda_{\mathcal{C}}$ and proving terms are invariant under translation to the other language and back. The type system of *FPC* is so rich that it is not at all clear how such a translation might work. In particular, the usual coding of sum and product types in the λ -calculus is not suitable.

References

- [1] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF, 1995. To appear.
- [2] Samson Abramsky. The lazy λ -calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 4, pages 65–117. Addison Wesley, 1990.
- [3] Samson Abramsky and Guy McCusker. Games and full abstraction for the lazy λ -calculus. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 234–243. IEEE Computer Society Press, 1995.

- [4] Samson Abramsky and Guy McCusker. Games for recursive types. In Chris L. Hankin, Ian C. Mackie, and Rajagopal Nagarajan, editors, *Theory and Formal Methods of Computing 1994: Proceedings of the Second Imperial College Department of Computing Workshop on Theory and Formal Methods*. Imperial College Press, October 1995.
- [5] Samson Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105(2):159–267, August 1993.
- [6] Roy Crole and Andrew Gordon. Factoring an adequacy proof. In C. J. van Rijsbergen, editor, *1993 Glasgow Functional Programming Workshop*, Springer Verlag Workshops in Computing, pages 9–25, 1994.
- [7] Marcelo P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. PhD thesis, University of Edinburgh, 1994.
- [8] Marcelo P. Fiore and Gordon D. Plotkin. An axiomatization of computationally adequate domain theoretic models of FPC. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 92–102. IEEE Computer Society Press, 1994.
- [9] Andrew D. Gordon. *Functional programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.
- [10] Andrew D. Gordon. Bisimilarity as a theory of functional programming. In *Participants proceedings of the Eleventh Conference on the Mathematical Foundations of Computer Science, New Orleans, 1995*, volume 1 of *Electronic notes in Theoretical Computer Science*. Elsevier, 1995. To appear.
- [11] Andrew D. Gordon. Bisimilarity as a theory of functional programming: mini-course. Notes Series BRICS-NS-95-3, BRICS, Department of Computer Science, University of Aarhus, July 1995.
- [12] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.
- [13] P. Hudak and P. Wadler. Report on the functional programming language Haskell. Technical Report YALEU/DCS/RR666, Department of Computer Science, Yale University, November 1988.
- [14] Guy McCusker. Games and full abstraction for FPC. Submitted to LiCS 11, 1995.
- [15] Guy McCusker. *On functions and games*. PhD thesis, Department of Computing, Imperial College, University of London, 1996. To appear.

- [16] C. H. L. Ong. *The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming*. PhD thesis, Imperial College of Science and Technology, 1988.
- [17] D.M. Park. Concurrency on automata and infinite sequences. In P. Deussen, editor, *Conference on Theoretical Computer Science*, Berlin, 1981. Springer-Verlag. Lecture Notes in Computer Science Vol. 104.
- [18] Gordon Plotkin. Lectures on predomains and partial functions. Notes for a course given at the Center for the Study of Language and Information, Stanford, 1985.
- [19] E. Ritter and A. M. Pitts. A fully abstract translation between a λ -calculus with reference types and Standard ML. In *2nd Int. Conf. on Typed Lambda Calculus and Applications, Edinburgh, 1995*, volume 902 of *Lecture Notes in Computer Science*, pages 397–413. Springer-Verlag, Berlin, 1995.
- [20] G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing. The MIT Press, Cambridge, Massachusetts, 1993.