Adequacy of Compositional Translations for Observational Semantics

Manfred Schmidt-Schauß[1], Joachim Niehren[2], Jan Schwinghammer[3], and David Sabel[1]

[1] J. W. Goethe-Universität, Frankfurt, Germany
[2] INRIA, Lille, France, Mostrare Project
[3] Saarland University, Saarbrücken, Germany

Summary. We investigate methods and tools for analysing translations between programming languages with respect to observational semantics. The behaviour of programs is observed in terms of may- and must-convergence in arbitrary contexts, and *adequacy* of translations, i.e., the reflection of program equivalence, is taken to be the fundamental correctness condition. For compositional translations we propose a notion of *convergence equivalence* as a means for proving adequacy. This technique avoids explicit reasoning about contexts, and is able to deal with the subtle role of typing in implementations of language extensions.

1 Introduction

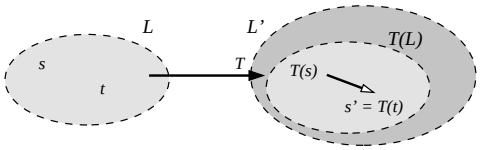
Proving correctness of program translations on the basis of operational semantics is an ongoing research topic (see e.g. the recent [7, 18]) that is still poorly understood when it comes to concurrency and mutable state. We are motivated by implementations of language extensions that are often packaged into the language's library. Typical examples are implementations of channels, buffers, or semaphores using mutable reference cells and futures in Alice ML [1, 12], or using MVars in Concurrent Haskell [13]. Ensuring the correctness of such implementations of higher-level constructs is obviously important.

In this paper we adopt an *observational semantics* based on may- and must-convergence. Two programs are considered equivalent if they exhibit the same may- and must-convergence behaviour in all contexts. This definition is flexible and has been applied to a wide variety of programming languages and calculi in the past. The observation of may- and must-convergence is particularly well-suited for dealing with nondeterminism as it arises in concurrent programming [2, 17, 11].

We study implementations of language extensions in the compilation paradigm, i.e., by viewing them as translations $T : L \to L'$ from a language L into another language L'. Such translations are usually *compositional* in that $T(C[t]) = T(C)[T(t)]$ for all contexts C and programs t of L. In a naive approach, one might even want to assume that L is a conservative extension of L' so that (non-)equivalences of L' continue to hold in L. However, this fails in many cases (see below) due to subtle typing problems.

A translation $T : L \to L'$ is *adequate* if $T(s) \sim_{L'} T(t)$ implies $s \sim_L t$ for all programs s and t of L, where $\sim_L$ and $\sim_{L'}$ are the program equivalences of the respective languages.

Adequacy is the basic correctness requirement to ensure that program transformations of the target language L' can be soundly applied with respect to observations made in the source language L.



Suppose a translation $T(s)$ is optimized to an equivalent program $s' \sim_{L'} T(s)$ and that s' is the translation of some t, i.e. $T(t) = s'$. Any useful notion of correctness must enforce that s and t must be indistinguishable, i.e. $s \sim_L t$. This is precisely what adequacy of T guarantees. With respect to implementations, adequacy opens the possibility of transferring contextual equivalences from the target language L' to the source language L. For non-deterministic and concurrent languages, such equivalences have been established for instance by inductive reasoning using diagram-based methods directly on an underlying small-step operational semantics [6, 11].

Full abstraction extends adequacy by the inverse property, i.e., that program equivalence is also preserved by the translation. In the general situation, however, the language L' may be more expressive than L and allows us to make more distinctions, also on the image $T(L)$. I.e., we can have $T(s) \not\sim_{L'} T(t)$ for some expressions s,t with $s \sim_L t$.

In denotational semantics, adequacy and full abstraction are well-studied concepts. In contrast, in this paper we provide a general criterion for proving adequacy of translations that is not tied to specific models. More precisely, we show that convergence equivalence implies adequacy of compositional translations, meaning it is enough to establish that all convergence tests yield the same results before and after the translation. We also provide a criterion for the full abstractness of compositional translations for which the target language is a conservative extension of the source language.

In order to demonstrate these tools, we consider the standard Church encoding of pairs in a call-by-value lambda calculus with a fixed point operator and nondeterministic choice. In order to reason that the encoding of pairs is adequate, one needs to check, for all lambda terms t with pairs and projections, that reduction from t may-converges (must-converges, respectively) if and only if reduction from its encoding $T(t)$ may-converges (must-converges, respectively). However, even in this seemingly well-understood example, this condition *fails* if the lambda calculus is untyped, since the implementation may remove errors, i.e., $T(t)$ terminates more often than t. If the source-language is typed so that stuck expressions are excluded, then our tools apply in a smooth way and show the adequacy of the standard translation, even for differently typed versions of the lambda calculus that is used as target language. Since neither simple typing nor Hindley-Milner polymorphic typing are sufficient to make the source language an extension of the target language, we cannot expect to have an extension situation under type systems that are commonly used in programming languages.

Related work. Various proof methods have been developed for establishing contextual equivalences. These include context lemmas (e.g., [9]), bisimulation methods (for instance, [5]), diagram-based methods (e.g., [6, 11]), and characterizations of contextual equivalence in terms of logical relations (e.g. [14]). In most cases, language extensions and their effect on equivalences are not discussed. There are some notable exceptions: a translation from the core of Standard ML into a typed lambda calculus is given in [16], and full abstraction is shown by exhibiting an inverse mapping, up to contextual equivalence. Adequate translations (with certain additional constraints) between call-by-name and call-by-value versions of PCF are considered in [15], via fully abstract models (necessitating the addition of parallel constructs to the languages) and domain-theoretic techniques. The fact that adequate (and fully abstract) translations compose is exploited in [8], where a syntactic translation is used to lift semantic

models for FPC to ones for the lazy lambda calculus. In a similar vein, the recent [18] develops a translation from an aspect-oriented language to an ML-like language, to obtain a model for the former. The adequacy proof follows a similar pattern to ours, but does not abstract away from the particularities of the concrete languages.

Shapiro [19] categorizes implementations and embeddings in concurrent scenarios, but does not provide concrete proof methods based on contextual equivalence. For deterministic languages (where may- and must-convergence agree), frameworks similar to our proposal were considered by Felleisen [4] and Mitchell [10]. Their focus is on comparing languages with respect to their expressive power; the non-deterministic case is only briefly mentioned by Mitchell. Mitchell's work is concerned with (the impossibility of) translations that additionally preserve representation independence of ADTs, and consequently assumes, for the most part, source languages with expressive type systems. Felleisen's work is set in the context of a Scheme-like untyped language. Although the paper discusses the possibility of adding types to get stronger expressiveness statements, the theory of expressiveness is developed by abandoning principles similar to adequacy.

Outline. Section 2 recalls the encoding of pairs in the non-deterministic lambda calculus, introduces rigorous notions of observables, and illustrates the need for types. In Section 3 a general framework for proving observational correctness as well as adequacy of translations is introduced. Section 4 shows the adequacy of the pair encoding using a simple type system and discusses two extensions.

2 Non-deterministic Call-by-Value Lambda Calculi

In this section, we recall the call-by-value lambda calculus with a fixed point operator and nondeterministic choice, and present its observational semantics on the basis of may- and must-convergence. We illustrate why Church's encoding of pairs in this calculus fails to be observationally correct in the untyped case.

2.1 Languages

The calculus λ_{cp} is the usual call-by-value lambda calculus extended by a (demonic, see [20]) choice operator, a call-by-value fixed point operator for recursion, pairs (w_1,w_2) and selectors **fst** and **snd** as data structure, and a constant **unit**. Fixing a set of variables *Var*, the syntax of expressions Exp_{cp} and values Val_{cp} is shown in Fig. 1. The subcalculus λ_c is the calculus without pairs and selectors and will be used as target language. We use Exp_c (Val_c, resp.) for the set of λ_c-expressions (λ_c-values, resp.).

A *context* C is an expression with a hole denoted with $[]$, $C[s]$ is the result of placing the expression s in the hole of C. For both calculi we require call-by-value evaluation contexts $\mathbb{E}$ which are introduced in Fig. 2. With $s_1[s_2/x]$ we denote the capture-free substitution of variable x with s_2 for all free occurrences of x in s_1. To ease reasoning we assume that the distinct variable convention holds for all expressions, i.e. that the bound variables of an expression are all distinct and free variables are distinct from bound variables.

The reduction rules for both calculi are defined in Fig. 3. Small step reduction $\to_{cp}$ of λ_{cp} is the union of all six rules, and small step reduction $\to_c$ of λ_c is the union of the first four rules. We assume that reduction preserves the distinct variable convention by implicitly performing α-renaming if necessary.

$$
\begin{array}{rcl}
x, y & \in & \mathit{Var} \\
r, s, t & \in & \mathit{Exp}_{cp} ::= w \mid t_1\ t_2 \mid t_1 \oplus t_2 \\
v, w & \in & \mathit{Val}_{cp} ::= x \mid \lambda x.t \mid \mathbf{unit} \mid \mathbf{fix} \\
 & & \quad \mid (w_1, w_2) \mid \mathbf{fst} \mid \mathbf{snd}
\end{array}
$$

Fig. 1. Syntax of λ_{cp}

$$\mathbb{E} ::= [\,] \mid \mathbb{E}\,t \mid w\,\mathbb{E} \mid \ \mid \mathbb{E} \oplus t \mid w \oplus \mathbb{E}$$

Fig. 2. Evaluation Contexts $\mathbb{E}$

$$
\begin{array}{ll}
(\beta\text{-CBV}) & \mathbb{E}[(\lambda x.t)\ w] \to \mathbb{E}[t[w/x]] \\
(\text{FIX}) & \mathbb{E}[\mathbf{fix}\ \lambda x.t] \to \mathbb{E}[t[(\lambda y.(\mathbf{fix}\ \lambda x.t)y)/x]] \\
(\oplus\text{L}) & \mathbb{E}[w_1 \oplus w_2] \to \mathbb{E}[w_1] \\
(\oplus\text{R}) & \mathbb{E}[w_1 \oplus w_2] \to \mathbb{E}[w_2] \\
(\text{SEL-F}) & \mathbb{E}[\mathbf{fst}\,(w_1, w_2)] \to \mathbb{E}[w_1] \\
(\text{SEL-S}) & \mathbb{E}[\mathbf{snd}\,(w_1, w_2)] \to \mathbb{E}[w_2]
\end{array}
$$

Fig. 3. Small-Step Reduction

2.2 Contextual Equivalence

Let *Exp* be a language, let $\mathit{Val} \subseteq \mathit{Exp}$ be a set of values and $\to$ be a reduction relation. Then *may-convergence* for expressions $s \in \mathit{Exp}$ is defined as $s\!\downarrow$ iff $\exists v \in \mathit{Val} : s \xrightarrow{*} v$, and *must-convergence* is defined as $s\!\Downarrow$ iff $\forall s' : s \xrightarrow{*} s' \implies s'\!\downarrow$. For a discussion and motivations for the latter notion see [2, 17, 11]. Note that there is also another notion of must-convergence found in the literature (e.g. [3]), which holds if an expression has no infinite evaluation (i.e. if $s \not\to^{\omega}$).

For an expression s we also write $s\!\Uparrow$ if $s\!\downarrow$ does not hold, and say that s is *must-divergent*. We write $s\!\uparrow$ if s is not must-convergent and then say s is *may-divergent*. Note that may-divergence can equivalently be defined as $s\!\uparrow$ iff $\exists s' \in \mathit{Exp} : s \xrightarrow{*} s'$ and $s'\!\Uparrow$. This view allows us to use inductive proofs for showing may-divergences. For $\mathit{Exp}_c, \mathit{Val}_c$, and $\to_c$ we use $\downarrow_c$ for may-convergence and $\Downarrow_c$ for must-convergence. Accordingly for $\mathit{Exp}_{cp}, \mathit{Val}_{cp}$, and $\to_{cp}$ we use $\downarrow_{cp}$ and $\Downarrow_{cp}$ for the predicates.

Contextual equivalence for a (non-deterministic) calculus $(\mathit{Exp}, \mathit{Val}, \to)$ is defined by observing may- and must-convergence in all contexts. We first define two preorders for both predicates:

$$s_1 \leq_{\downarrow} s_2 \text{ iff } \forall C : C[s_1]\!\downarrow \implies C[s_2]\!\downarrow \qquad\qquad s_1 \leq_{\Downarrow} s_2 \text{ iff } \forall C : C[s_1]\!\Downarrow \implies C[s_2]\!\Downarrow$$

These are combined to obtain the contextual preorder $\leq$ as their intersection $\leq_{\downarrow} \cap \leq_{\Downarrow}$, and the contextual equivalence $\sim$ as $\leq \cap \geq$. To distinguish between the relations for λ_c and λ_{cp}, we index the symbols for the preorders and equivalence with c or cp, respectively, e.g. contextual equivalence in λ_c is $\sim_c$, and contextual preorder in λ_{cp} is $\leq_{cp}$.

2.3 Implementation of Pairs

We will mainly investigate the translation *enc* of λ_{cp} into λ_c as defined in Fig. 4 under different restrictions. Conversely, it is trivial to encode λ_c into λ_{cp} via the identity $\mathit{inc}(s) = s$ (which is more an *embedding* than a translation).

The following counter example shows that the implementation of pairs is not correct in the untyped setting.

Example 2.1. Let $t := \mathbf{fst}(\lambda z.z)$. Then $t\!\Uparrow_{cp}$, since t is irreducible and not a value. However, the translation $\mathit{enc}(t)$ results in the expression $t' := (\lambda p.p\ (\lambda x.\lambda y.x))\ (\lambda z.z)$, which deterministically reduces by some $(\beta\text{-CBV})$-reductions to $\lambda x.\lambda y.x$, hence $\mathit{enc}(t)\!\Downarrow_c$. This is clearly not a correct translation, since it removes an error. Therefore, the observations are not preserved by this translation. This example also invalidates the implication $T(p_1) \leq_c T(p_2) \implies p_1 \leq_{cp} p_2$,

$$
\begin{array}{lll}
enc(x) & = x & enc(\mathbf{fix}) = \mathbf{fix} \\
enc(\mathbf{unit}) & = \mathbf{unit} & enc((w_1,w_2)) = \lambda s.\ (s\ enc(w_1)\ enc(w_2)) \\
enc(\lambda x.t) & = \lambda x.enc(t) & enc(\mathbf{fst}) = \lambda p.\ (p\ \lambda x.\lambda y.x) \\
enc(t_1\ t_2) & = enc(t_1)\ enc(t_2) & enc(\mathbf{snd}) = \lambda p.\ (p\ \lambda x.\lambda y.y) \\
enc(t_1 \oplus t_2) & = enc(t_1) \oplus enc(t_2) &
\end{array}
$$

Fig. 4. Translation of λ_{cp} into λ_c

since $enc(t') = t'$, and hence $enc(t') = t' \leq_c t' = enc(t)$, but $t' \not\leq_{cp} t$ by the arguments above. In the terms of Definition 3.2 below, the translation *enc* is not adequate.

This counter example is also valid for deterministic calculi, where may- and must-convergence coincide. There, it is possible to circumvent the problem by weakening the definition of correctness to only one direction of the logical equivalence, $s\downarrow \Longrightarrow T(s)\downarrow$, but this results in weaker properties and is not the appropriate notion for compilations. In particular, this notion of correctness of a translation (which is called *weak expressibility* in [4]) implies the correctness of a trivial translation that maps all expressions to a (may-) convergent expression.

One potential remedy to the failure of the untyped approach to correctness of translations is to distinguish divergence from typing errors. From a different point of view, this simply means that only correctly typed programs should be considered by a translation: in Section 4.1 we will obtain adequacy after adding a type system to λ_{cp}.

3 Adequacy of Translations

We present a general framework for reasoning about different notions of language translations which are related to correctness.

We assume that languages come equipped with a small-step operational semantics and a notion of observables, expressed through convergence tests, with respect to which contextual equivalence can be defined. Since we are interested in concurrent calculi, a typical case will be the observations of may- and must-termination behavior, as introduced in the previous section. In the following we generalize slightly and, instead of contexts, speak of observers: this makes it easier to fit formalisms without an obvious notion of context into the framework, like abstract machines.

Definition 3.1. *A* program calculus with observational semantics (OSP-calculus) *consists of the following components:*

- *A set* $\mathscr{T}$ *of* types*, ranged over by* τ*.*
- *For every type* τ*, a set* $\mathscr{P}_\tau$ *of* programs*, ranged over by p.*
- *For every pair* τ_1, τ_2 *of types, a set of functions* $\mathscr{O}_{\tau_1,\tau_2}$ *with* $O : \mathscr{P}_{\tau_1} \to \mathscr{P}_{\tau_2}$ *for* $O \in \mathscr{O}_{\tau_1,\tau_2}$*, called* observers*, such that also the identity function* Id_τ *is included in* $\mathscr{O}_{\tau,\tau}$ *for every type* τ*, and such that* $\bigcup_{\tau_1,\tau_2\in\mathscr{T}} \mathscr{O}_{\tau_1,\tau_2}$ *is closed under function composition whenever the types are appropriate.*
- *A set* $\{\Downarrow_1,\ldots,\Downarrow_n\}$ *of* convergence tests *with* $\Downarrow_i$: $\bigcup_{\tau\in\mathscr{T}} \mathscr{P}_\tau \to \{true,false\}$ *for all* $i = 1,\ldots,n$*.*

This definition is also applicable to the special case of deterministic calculi, where usually only a single termination predicate is considered. Moreover, it allows for untyped calculi like λ_{cp} by considering a single, 'universal' type. The calculus λ_{cp} then fits this definition of OSP-calculus, after identifying a context C with the map $t \mapsto C[t]$, and taking $\{\Downarrow_1, \Downarrow_2\} = \{\downarrow_{cp}, \Downarrow_{cp}\}$.

Since this framework has arbitrary observers (not only contexts) and there are types, the observational preorders at type τ are defined as follows, where $p_1, p_2 \in \mathscr{P}_\tau$:

- $p_1 \leq_{\Downarrow_i,\tau} p_2$ iff for all $\tau' \in \mathscr{T}$ and all $O : \tau \to \tau'$, $O(p_1) \Downarrow_i$ implies $O(p_2) \Downarrow_i$.
- $p_1 \leq_\tau p_2$ iff $\forall i : p_1 \leq_{\Downarrow_i,\tau} p_2$.
- $p_1 \sim_\tau p_2$ iff $p_1 \leq_\tau p_2$ and $p_2 \leq_\tau p_1$.

The relations $\leq_{\Downarrow_i,\tau}$ and $\leq_\tau$ are *precongruences*, i.e. they are preorders, and $p_1 \leq_{\Downarrow_i,\tau} p_2$ implies $\forall O : \tau \to \tau' : O(p_1) \leq_{\Downarrow_i,\tau'} O(p_2)$. For proving the latter implication let O' be an observer with $O'(O(p_1)) \Downarrow_i$. Then $O' \circ O$ is also an observer, hence $O' \circ O(p_2) \Downarrow_i$. Obviously, the same holds for $\leq_\tau$. The relation $\sim_\tau$ is a *congruence*, i.e. it is a precongruence and an equivalence relation.

In the following we only consider translations between OSP-calculi that have the same number n of convergence tests $\{\Downarrow_1, \ldots, \Downarrow_n\}$, in a fixed ordering. We define some characterizing notions of translations. In the following we exhibit their dependencies and prove some consequences.

Definition 3.2. *A translation $T : \mathscr{C} \to \mathscr{C}'$ between two calculi $\mathscr{C} = (\mathscr{T}, \mathscr{P}, \mathscr{O}, \leq)$ and $\mathscr{C}' = (\mathscr{T}', \mathscr{P}', \mathscr{O}', \leq')$ maps types to types $T : \mathscr{T} \to \mathscr{T}'$, programs to programs $T : \mathscr{P}_\tau \to \mathscr{P}'_{T(\tau)}$, and observers to observers $T : \mathscr{O}_{\tau,\tau'} \to \mathscr{O}'_{T(\tau),T(\tau')}$ such that their types correspond for all $\tau, \tau' \in \mathscr{T}$ and such that $T(Id_\tau) = Id_{T(\tau)}$ for all τ.*

Adequacy. *A translation T is* adequate *iff for all τ, and $p_1, p_2 \in \mathscr{P}_\tau$, $T(p_1) \leq'_{T(\tau)} T(p_2) \implies p_1 \leq_\tau p_2$.*

Full abstraction. *A translation T is* fully abstract *iff for all τ, and $p_1, p_2 \in \mathscr{P}_\tau$, $p_1 \leq_\tau p_2 \iff T(p_1) \leq'_{T(\tau)} T(p_2)$.*

Observational correctness. *A translation T is* observationally correct *iff for all τ, $p \in \mathscr{P}_\tau$,$O \in \mathscr{O}_{\tau,\tau'}$ and all i: $O(p) \Downarrow_i$ if and only if $T(O)(T(p)) \Downarrow'_i$.*

Convergence equivalence. *A translation T is* convergence equivalent *(i.e.* preserves and reflects convergence*) iff for all p and convergence tests $\Downarrow_i$: $p \Downarrow_i$ if and only if $T(p) \Downarrow'_i$.*

Compositionality. *A translation T is* compositional *iff for all types $\tau, \tau' \in \mathscr{T}$, for all observers $O \in \mathscr{O}_{\tau,\tau'}$ and all programs $p \in \mathscr{P}_\tau$ we have $T(O(p)) = T(O)(T(p))$.*

If in the following types are omitted, we implicitly assume that type information follows from the context.

As motivated in the Introduction, we consider adequacy as the right notion of correctness. Observational correctness is a sufficient criterion for adequacy (see Proposition 3.3). Convergence equivalence is implied by observational correctness, since T preserves identity observers. For compositional translations, the converse is true, i.e., it is sufficient to prove convergence equivalence in order to prove observational correctness. Full abstraction is not necessary for the adequacy of translations. If it holds in addition, for surjective translations it means that both program calculi are identical w.r.t. $\leq_c$.

Note that Definition 3.2 is stated only in terms of convergence tests and sets of observers, and hence only relying on the syntax and the operational semantics. Thus it can be used in all calculi with such a description. In the case of two calculi with convergence tests defined in terms of a small-step semantics, the definition also allows for reduction sequences in the translation that may lead outside of the image of the translation, i.e., that may not be retranslatable.

Proposition 3.3. *For a translation T the following hold:*

1. *If T is compositional, then T is convergence equivalent if and only if T is observationally correct.*
2. *If T is observationally correct, then T is adequate.*

Proof. 1. The only if direction holds, since T preserves identity observers: $Id_\tau(p)\Downarrow_i \iff T(Id_\tau)T(p)\Downarrow'_i \iff Id_{T(\tau)}T(p)\Downarrow'_i \iff T(p)\Downarrow'_i$.
For the if-direction let us assume that T is compositional and convergence equivalent. If $O(p)\Downarrow_i$, then preservation of convergence yields $T(O(p))\Downarrow'_i$. Compositionality implies $T(O(p)) = T(O)(T(p))$, hence $T(O)(T(p))\Downarrow'_i$. If $T(O)(T(p))\Downarrow'_i$ then compositionality implies $T(O(p))\Downarrow'_i$ so that reflection of convergence yields $O(p)\Downarrow_i$.

2. To show adequacy, let us assume that $T(p_1) \leq_{T(\tau)} T(p_2)$. We must prove that $p_1 \leq_\tau p_2$. Thus let O be such that $O(p_1)\Downarrow_i$. By observational correctness this implies $T(O)(T(p_1))\Downarrow'_i$. From $T(p_1) \leq_{T(\tau)} T(p_2)$, we obtain $T(O)(T(p_2))\Downarrow'_i$, since $T(O)$ is an admissible observer. Observational correctness in the other direction implies $O(p_2)\Downarrow_i$. This proves $p_1 \leq_\tau p_2$. □

As the following counter examples show, convergence equivalence is in general not sufficient for adequacy, and full abstraction is not implied by observational correctness. Similarly, convergence equivalence is not even implied by full abstraction (and thus neither by adequacy):

Example 3.4 (Convergence equivalence does not imply adequacy). Let the OSP-calculus L have three programs: a,b,c with $a\uparrow$, $b\downarrow$ and $c\downarrow$. Assume there are two observers O_1, O_2 with $O_1(x) = x$ and $O_2(a) = a, O_2(b) = a, O_2(c) = c$. Then $b \not\sim_L c$. The language L' has three programs A,B,C with $A\uparrow$, $B\downarrow$ and $C\downarrow$. There is only the identity observer O in L'. Then $B \sim_{L'} C$. Let the translation be defined as $T : L \to L'$ with $T(a) = A, T(b) = B, T(c) = C$, and $T(O_1) = T(O_2) = O$. Then convergence equivalence holds, but neither equational adequacy nor observational correctness. Note that T is not compositional, since $T(O_2(b)) = A$ while $T(O_2)(T(b)) = O(B) = B$.

Example 3.5 (Observational correctness does not imply full abstraction). A simple example taken from [10] is the identity encoding from the OSP-calculus λ_{cp} without the projections **fst** and **snd** into full λ_{cp}. Then, in the restricted OSP-calculus, all pairs are indistinguishable but the presence of the observers (here simply taken as contexts) **fst**$[\cdot]$ and **snd**$[\cdot]$ in λ_{cp} permits more distinctions to be made.

Example 3.6 (Convergence equivalence is not implied by full abstraction). A trivial example is given by two calculi $\mathscr{C}$ with $p\Downarrow$ for all p, and $\mathscr{C}'$ with the same programs and $\neg p\Downarrow'$ for all p. For the translation $T(p) = p$ for all p it is clear that $\forall p_1, p_2 : p_1 \leq p_2 \iff T(p_1) \leq' T(p_2)$ holds, but T does not preserve convergence.

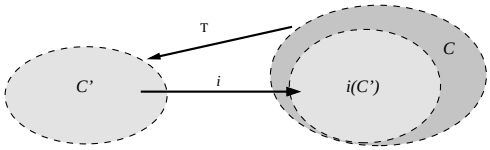
By standard arguments it can be shown that translations compose:

Proposition 3.7. *Let $\mathscr{C}, \mathscr{C}', \mathscr{C}''$ be program calculi, and $T : \mathscr{C} \to \mathscr{C}'$, $T' : \mathscr{C}' \to \mathscr{C}''$ be translations. Then $T' \circ T : \mathscr{C} \to \mathscr{C}''$ is also a translation, and for every property P from Definition 3.2, if T, T' have property P, then also the composition $T' \circ T$.*

We now consider the case that only new language primitives are added to a language, which are then encoded by the translation. This is usually known as removing 'syntactic sugar'.

Definition 3.8. *An OSP-calculus $\mathscr{C}$ is an* extension *of the OSP-calculus $\mathscr{C}'$ iff there is a compositional translation $\iota : \mathscr{C}' \to \mathscr{C}$, called an* embedding*, which is injective on the expressions, types and observers, and is convergence equivalent.*

Informally, this can be described (after identifying $\mathscr{C}'$-programs with their image under ι) as follows: every $\mathscr{C}'$-type is also a $\mathscr{C}$-type, $\mathscr{P}'_\tau \subseteq \mathscr{P}_\tau$, and $\mathscr{O}'_{\tau,\tau'}$ is a subset of $\mathscr{O}_{\tau,\tau'}$, and the test-predicates coincide on $\mathscr{C}'$-programs. The embedding of $\mathscr{O}'_{\tau,\tau'}$ into $\mathscr{O}_{\tau,\tau'}$ is slightly more involved, since the $\mathscr{C}'$-observers are restrictions (as functions) of $\mathscr{C}$-observers. Note that for the case of contexts as observers, the embedding of $\mathscr{O}'_{\tau,\tau'}$ into $\mathscr{O}_{\tau,\tau'}$ is unique. The conditions imply that an embedding ι is adequate, but not necessarily fully abstract.



If $\mathscr{C}$ is an extension of $\mathscr{C}'$, then an observationally correct translation $T : \mathscr{C} \to \mathscr{C}'$ (plus some obvious conditions) has the nice consequence of T and ι being fully abstract.

An example for an embedding is the trivial embedding $inc : \lambda_c \to \lambda_{cp}$, which is adequate by Proposition 3.3, since the embedding *inc* is compositional and convergence equivalent. This allows us to reason about contextual equivalence in λ_{cp} and transfer this result to λ_c, i.e. a proof of $t_1 \sim_{cp} t_2$ where t_1, t_2 are also expressions of λ_c directly shows $t_1 \sim_c t_2$. Disproving an equivalence in λ_{cp}, however, does *not* imply that this equivalence is false in λ_c.

Proposition 3.9 (Full Abstraction for Extensions). *Let $\mathscr{C}$ be an extension of $\mathscr{C}'$, and let $T : \mathscr{C} \to \mathscr{C}'$ be an observationally correct translation, such that $T \circ \iota$ is the identity on $\mathscr{C}'$-programs, on $\mathscr{C}'$-observers, and on $\mathscr{C}'$-types. Then the translation T as well as the embedding ι are fully abstract.*

Proof. First we show that T is fully abstract: adequacy follows from Proposition 3.3. It remains to show the inverse condition of full abstraction. Let p_1, p_2 be $\mathscr{C}$-programs of type τ, and assume $p_1 \leq_{\Downarrow_i,\tau} p_2$. We have to show that $T(p_1) \leq'_{\Downarrow_i,T(\tau)} T(p_2)$. Let O' be a $\mathscr{C}'$-observer with $O'(T(p_1)) \Downarrow'_i$. Then by definition of ι there exists an observer O of $\mathscr{C}$ with $O := \iota(O')$. Since $T \circ \iota$ is the identity, we have $T(O) = O'$ and thus we obtain $T(O)(T(p_1)) \Downarrow'_i$. Observational correctness implies that $O(p_1) \Downarrow_i$. From $p_1 \leq_{\Downarrow_i,\tau} p_2$ we now derive $O(p_2) \Downarrow_i$. Again observational correctness can be applied and shows that $T(O)(T(p_2)) \Downarrow'_i$. This is equivalent to $O'(T(p_2)) \Downarrow'_i$. Since the observer O' was chosen arbitrarily, we have $T(p_1) \leq'_{\Downarrow_i,T(\tau)} T(p_2)$.

The embedding ι is already shown to be adequate. The missing direction, i.e. that $\iota(p_1) \leq'_{\Downarrow_i,T(\tau)} \iota(p_2)$ implies $p_1 \leq'_{\Downarrow_i,\tau} p_2$ follows from full abstraction of T and the assumption that $T \circ \iota$ is the identity. □

4 Adequacy of Pair Encoding

We analyze the translation *enc* on the untyped language λ_c. Inspecting the definition of *enc* the following lemma is easy to verify:

Lemma 4.1. *For all $s \in \lambda_{cp}$: s is a λ_{cp}-value iff $enc(s)$ is a λ_c-value.*

Lemma 4.2. *Let $t \in \lambda_{cp}$ with $t \downarrow_{cp}$, then $enc(t) \downarrow_c$.*

$(.,.) :: \forall\alpha,\beta.\alpha \to \beta \to (\alpha,\beta)$ **unit** :: unit
fst $:: \forall\alpha,\beta.(\alpha,\beta) \to \alpha$ $\oplus$ $:: \forall\alpha.\alpha \to \alpha \to \alpha$
snd $:: \forall\alpha,\beta.(\alpha,\beta) \to \beta$ **fix** $:: \forall\alpha,\beta.((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)$

Fig. 5. Types of constants

Proof. Let $t_0 \in \lambda_{cp}$ with $t \downarrow_{cp}$, so $t_0 \to_{cp} t_1 \to_{cp} \cdots \to_{cp} t_n$ where t_n is a value. We show by induction on n that $enc(t_0)\downarrow_c$. If $n = 0$ then t_0 is a value and $enc(t_0)$ must be a value, too, by Lemma 4.1. For the induction step we assume the induction hypothesis $enc(t_1)\downarrow_c$. Hence, it suffices to show $enc(t_0) \xrightarrow{*}_c enc(t_1)$. If $t_0 \to_{cp} t_1$ is a (β-CBV), (FIX), ($\oplus$L), or ($\oplus$R) reduction, then the same reduction can be used in λ_c, and $enc(t_0) \to_c enc(t_1)$. If $t_0 \to_{cp} t_1$ by (SEL-F) or (SEL-S), then three (β-CBV) steps are necessary in λ_c, i.e., $enc(t_0) \xrightarrow{3}_c enc(t_1)$. □

For the other direction, i.e., for proving the claim $enc(t)\downarrow_c \implies t\downarrow_{cp}$ the counter example 2.1 shows that the translation *enc* is not adequate and not observationally correct. Moreover, this example shows that an untyped language does in general not permit an adequate – and hence also not an observationally correct – translation into a subset of itself.

4.1 Typing λ_{cp}

One solution to prevent the counter example 2.1 is to consider a simply typed variant λ^T_{cp} of λ_{cp} as follows. The types are given by $\tau ::= \mathsf{unit} \mid \tau \to \tau \mid (\tau,\tau)$, and only typed expressions and typed contexts are in the language λ^T_{cp}, where we assume a hole $[\cdot]_\tau$ for every type τ. For typing, we treat pairs, projections, the unit value, and the operators $\oplus$ and **fix** as a family of constants with the types given in Figure 5. Type safety can be stated by a preservation theorem for all expressions and a progress theorem for closed expressions. The framework now permits to prove adequacy via observational correctness of the translations.

Proposition 4.3. *For λ^T_{cp}, the (correspondingly restricted) translation $enc : \lambda^T_{cp} \to \lambda_c$ is compositional and convergence equivalent, and hence adequate.*

Proof. Compositionality follows from the definition of *enc*. Lemma 4.1 also holds if *enc* is restricted to λ^T_{cp}. We split the proof into four parts:

1. $t\downarrow_{cp} \implies enc(t)\downarrow_c$: Follows from Lemma 4.2
2. $enc(t)\downarrow_c \implies t\downarrow_{cp}$: An inspection of the reductions shows that if t_1 is reducible, then for every reduction *Red* of $enc(t_1)$ to a value, there is some t_2 with $t_1 \to_{cp} t_2$ and $enc(t_1) \xrightarrow{+}_c enc(t_2)$ is a prefix of *Red*. We use iduction on the length of a reduction *Red* of $enc(t)$ to a value to show that a corresponding reduction can be constructed. The base case is proved in Lemma 4.1. If t is an irreducible non-value, then due to typing it is an open expression of one of the forms $\mathbb{E}[(x\ r)], \mathbb{E}[\mathbf{fix}\ x], \mathbb{E}[\mathbf{fst}\ x], \mathbb{E}[\mathbf{snd}\ x]$, where x is a free variable. But the cases are not possible, since $enc(t)$ is either an irreducible non-value, or $enc(t)$ reduces in one step to an irreducible non-value.
3. $enc(t)\Downarrow_c \implies t\Downarrow_{cp}$: We prove that $t\uparrow_{cp} \implies enc(t)\uparrow_c$ by induction on the length of a reduction $t \xrightarrow{*}_{cp} t'$, where $t'\Uparrow_{cp}$. For the base case $t\Uparrow_{cp}$ and (2) show that $enc(t)\Uparrow_c$. The induction consists in computing a reduction sequence $enc(t) \xrightarrow{*}_c r$ where $r\Uparrow_{cp}$ and the correspondence is as in the proof of Lemma 4.2, such that $t \xrightarrow{*}_{cp} t'$ and $r = enc(t')$. By type preservation, t' is well-typed and now the base-case reasoning applies.

$w \in Val_{cpig}$ iff $w \in Val_{cp}$
$t \in Exp_{cpig} ::= w \mid t_1\ t_2 \mid t_1 \oplus t_2 \mid (t_1,t_2)$

Fig. 6. Syntax of λ_{cpig}

$\mathbb{E}_{cpig} ::= [\,] \mid \mathbb{E}_{cpig}\, t \mid w\, \mathbb{E}_{cpig} \mid \ \mid \mathbb{E}_{cpig} \oplus t \mid t \oplus \mathbb{E}_{cpig} \mid (\mathbb{E}_{cpig},t) \mid (t,\mathbb{E}_{cpig})$

Fig. 7. Evaluation Contexts $\mathbb{E}_{cpig}$ for λ_{cpig}

4. $t\Downarrow_{cp} \Longrightarrow enc(t)\Downarrow_c$: Proving $enc(t)\uparrow_c \Longrightarrow t\uparrow_{cp}$ can be done using the same technique as in the previous parts. □

Note that Proposition 3.9 cannot be applied since λ^T_{cp} is not an extension of untyped λ_c. As expected, full abstraction does not hold. For instance, let $s = \lambda p.((\lambda y.\lambda z.(y,z))\ (\mathbf{fst}\ p)\ (\mathbf{snd}\ p))$, and $t = \lambda p.p$. Then the equation $s \sim_{cp,(\mathsf{unit},\mathsf{unit})\to(\mathsf{unit},\mathsf{unit})} t$ holds in λ^T_{cp} by standard reasoning, but after translation to λ_c, we have $enc(s) \not\sim_c enc(t)$. The latter can be seen with the context $C = ([\cdot]\ \mathbf{unit})$, since $C[enc(s)]$ is must-divergent while $C[enc(t)]$ must-converges.

The extension situation could perhaps be regained by a System F-like type system, which we leave for future research. However, using a simple type system for λ_c is insufficient since the encoding of pairs with components of different types cannot be simply typed. The same holds for Hindley-Milner polymorphic typing: Let $s,r \in \lambda_{cp}$ where s is defined as before and $r = s\ (\mathbf{unit},\lambda x.x)$. The most general type of $enc(s)$ in a Hindley-Milner system is $((\alpha \to \alpha \to \alpha) \to \beta) \to (\beta \to \beta \to \gamma) \to \gamma$, which essentially means that the encoding requires the components of a pair to have equal type. The reason for the insufficient type is the monomorphic use of the argument variable p of $enc(s)$. Hence, $enc(r)$ is not typeable using a Hindley-Milner system.

One can establish a fully-abstract translation between λ^T_{cp} and a variant of λ_c by using a 'virtual typing' in λ_c which, intuitively, restricts λ_c to the image of the translation (see Appendix B).

4.2 Modifying Reduction Strategies

As a final example we extend λ^T_{cp} to λ_{cpig} by allowing pairs with arbitrary expressions as components (see Fig. 6). Secondly, we relax the reduction strategy, by allowing interleaving evaluation of pair components and of the arguments of the choice-operator. This is established by using the evaluation contexts $\mathbb{E}_{cpig}$ (see Fig. 7) for the calculus λ_{cpig}. The translation $enc_{ig} : \lambda_{cpig} \to \lambda^T_{cp}$ is the identity except for the case $(t_1,t_2) \to (\lambda x\, y.(x,y))\ enc_{ig}(t_1)\ enc_{ig}(t_2)$ where t_1,t_2 are not variables.

It is possible to split the translation enc_{ig} into two translations $enc_i : \lambda_{cpig} \to \lambda_{cpg}$ and $enc_g : \lambda_{cpg} \to \lambda^T_{cp}$, where the intermediate calculus λ_{cpg} has general pairs, but not the general reduction strategy of λ_{cpig}. One can then prove that enc_i and enc_g are fully abstract by applying Proposition 3.9 on extensions. Full abstraction of the composition $enc_{ig} = enc_i \circ enc_g$ follows from Proposition 3.7. The details can be found in Appendix A.

Conclusions and Outlook

Motivated by translation problems between concurrent programming languages, this paper succeeded in clarifying the methods, and providing tools, to assess the correctness of translations. The framework is general enough to apply directly to an operational semantics and the derived contextual equivalences, without relying on the availability of models.

In future research we want to exploit these results, to prove the correctness of various implementations of synchronization constructs in concurrent languages.

References

1. *The Alice Project*. Saarland University, `http://www.ps.uni-sb.de/alice`, 2007.
2. A. Carayol, D. Hirschkoff, and D. Sangiorgi. On the representation of McCarthy's amb in the pi-calculus. *Theoret. Comput. Sci.*, 330(3):439–473, 2005.
3. R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoret. Comput. Sci.*, 34:83–133, 1984.
4. M. Felleisen. On the expressive power of programming languages. *Sci. Comput. Programming*, 17(1–3):35–75, 1991.
5. A. D. Gordon. Bisimilarity as a theory of functional programming. *Theoret. Comput. Sci.*, 228(1–2):5–47, 1999.
6. A. Kutzner and M. Schmidt-Schauß. A nondeterministic call-by-need lambda calculus. In *Proc. ICFP*, pages 324–335. ACM, 1998.
7. J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *34th ACM POPL*, pages 3–10. ACM, 2007.
8. G. McCusker. Full abstraction by translation. In *Advances in Theory and Formal Methods of Computing*. IC Press, 1996.
9. Robin Milner. Fully abstract models of typed lambda calculi. *Theoret. Comput. Sci.*, 4(1):1–22, 1977.
10. J. C. Mitchell. On abstraction and the expressive power of programming languages. *Sci. Comput. Programming*, 21(2):141–163, 1993.
11. J. Niehren, D. Sabel, M. Schmidt-Schauß, and J. Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. *Electron. Notes Theor. Comput. Sci.*, 173:313–337, 2007.
12. J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoret. Comput. Sci.*, 364(3):338–356, 2006.
13. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *23rd ACM POPL*, pages 295–308. ACM, 1996.
14. A. D. Pitts. Parametric polymorphism and operational equivalence. *Math. Structures Comput. Sci.*, 10:321–359, 2000.
15. J. G. Riecke. Fully abstract translations between functional languages. In *18th ACM POPL*, pages 245–254. ACM, 1991.
16. E. Ritter and A. M. Pitts. A fully abstract translation between a lambda-calculus with reference types and Standard ML. In *Proc. 2nd TLCA*, pages 397–413. Springer, 1995.
17. D. Sabel and M. Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 2008. accepted for publication.
18. S. B. Sanjabi and C.-H. L. Ong. Fully abstract semantics of additive aspects by translation. In *Proc. 6th OASD*, pages 135–148. ACM, 2007.
19. E. Shapiro. Separating concurrent languages with categories of language embeddings. In *23rd ACM STOC*, pages 198–208. ACM, 1991.
20. H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *Comput. J.*, 35(5):514–523, 1992.

$$\mathbb{E}_{cpg} ::= [\,] \mid \mathbb{E}_{cpg}\, t \mid w\, \mathbb{E}_{cpg} \mid \mathbb{E}_{cpg} \oplus t \mid w \oplus \mathbb{E}_{cpg} \mid (\mathbb{E}_{cpg}, t) \mid (w, \mathbb{E}_{cpg})$$

Fig. 8. Evaluation Contexts for λ_{cpg}

Appendix

A Correctness of Modifications of the Reduction Strategies

In this section we prove that the translation enc_{ig} introduced in Subsection 4.2 is fully abstract. We split the translation into two translations enc_i and enc_g, such that $enc_{ig} = enc_g \circ enc_i$. The intermediate calculus λ_{cpg} which is the codomain of enc_i extends λ^T_{cp} by allowing arbitrary pairs (i.e. pairs with arbitrary expressions as components), but does not allow interleaved evaluation. The calculus λ_{cpig} then extends λ_{cpg} by allowing concurrent reduction of the components of pairs and choice.

Permitting General Pairs

We consider the extension λ_{cpg} of the language λ^T_{cp} where λ_{cpg} is simply typed, and where pairs are not restricted to values. The syntax is identical to the syntax of λ_{cpig} which is shown in Fig. 6. The reductions and evaluation contexts in λ_{cpg} are extended w.r.t. λ^T_{cp}, but restricted w.r.t. λ_{cpig}. They are introduced in Fig. 8. The translation $enc_g : \lambda_{cpg} \to \lambda^T_{cp}$ is the same translation as enc_{ig}. We show that enc_g is a fully abstract translation from λ_{cpg} to λ^T_{cp}, and hence nothing is lost by restricting pairs to values. Type preservation and progress also hold for λ_{cpg}. Moreover, enc_g is compositional and is easily seen to map well-typed terms of λ_{cpg} to well-typed terms of λ^T_{cp}.

Lemma A.1. *For the translation enc_g the following holds: For all s, if s is a λ_{cpg}-value, then $enc_g(s)$ is must-convergent and has a deterministic reduction to a value. Moreover, for all s, if $enc_g(s)$ is a value, then s is a λ_{cpg}-value.*

Proof. By induction on the size of expressions and inspection of all cases. This holds also for the case $(w_1,w_2) \mapsto (\lambda x\, y.(x,y))\ enc_g(w_1)\ enc_g(w_2)$, since $enc_g(w_1)$, $enc_g(w_2)$ are must-convergent and independently reduce to values, and then two deterministic beta-reductions reduce the resulting expression to a value. □

Proposition A.2. *The translation enc_g is fully abstract.*

Proof. By Proposition 3.9, and since the identity: $\lambda^T_{cp} \to \lambda_{cpg}$ is an embedding (see Definition 3.8), it suffices to prove observational correctness of the translation. Note that $enc_g(t) = t$, for all λ^T_{cp}-terms t, which makes Proposition 3.9 applicable. We have to show four implications.

1. $t \downarrow_{cpg} \implies enc_g(t) \downarrow_{cp}$: This follows by a straightforward translation from the $t \downarrow_{cpg}$-reduction into a reduction of $enc_g(t)$. In the case of non-value pairs, (β-CBV)-reductions have to be added to produce pairs in λ^T_{cp}.
2. $enc_g(t) \downarrow_{cp} \implies t \downarrow_{cpg}$: A reduction $enc_g(t) \downarrow_{cp}$ can be retranslated into one of t, by observing that (t_1,t_2) on the λ_{cpg}-side may correspond to three different possibilities on the λ_{cp}-side: it may be (t'_1,t'_2), $(\lambda xy.(x,y))\ t'_1\ t'_2$ or $(\lambda y.(t'_1,y))\ t'_2$.

3. $t \Downarrow_{cpg} \Longrightarrow enc_g(t) \Downarrow_{cp}$: We show $enc_g(t) \uparrow_{cp} \Longrightarrow t \uparrow_{cpg}$. Again the reductions correspond, up to the $(\beta\text{-CBV})$-reductions for the pair-encoding. The base case is that $enc_g(t) \Uparrow_{cp} \Longrightarrow t \Uparrow_{cpg}$, which follows from (1).
4. $enc_g(t) \Downarrow_{cp} \Longrightarrow t \Downarrow_{cpg}$: We show $t \uparrow_{cpg} \Longrightarrow enc_g(t) \uparrow_{cp}$. As above, the reductions correspond up to the $(\beta\text{-CBV})$-reductions for the pair-encoding. The base case is $t \Uparrow_{cpg} \Longrightarrow enc_g(t) \Uparrow_{cp}$, and follows from (2). □

Remark A.3. The combined translation from λ_{cpg} to λ_c is thus $enc_{gc} := enc \circ enc_g$. It operates on pairs as follows: $enc_{gc}((s,t)) = enc(\lambda xy.(x,y))\ enc_{gc}(s)\ enc_{gc}(t) = (\lambda xy.(\lambda p.p\ x\ y))\ enc_{gc}(s)\ enc_{gc}(t)$. The naive translation $T'((s,t)) = (\lambda p.p\ T'(s)\ T'(t)))$ is not convergence equivalent, since for example $T'((\Omega,\Omega)) = \lambda p.p\ \Omega\ \Omega$. However, (Ω,Ω) must-diverges, whereas $\lambda p.p\ \Omega\ \Omega$ is a value and thus converges.

Permitting Independent Reductions

In this subsection we will show that it is also correct to modify the reduction strategy in the OSP-calculus λ_{cpg}, where we allow that the arguments of choice and of pairs may be evaluated independently (i.e. interleaved, in any order). The OSP-calculus λ_{cpig}, i.e. its syntax and the evaluation contexts $\mathbb{E}_{cpig}$ used for reduction have been introduced in Subsection 4.2 (Fig. 6 and Fig. 7).

The translation $enc_i : \lambda_{cpig} \to \lambda_{cpg}$ is just the identity. However, it is not immediately obvious that the convergence predicates of λ_{cpig} and λ_{cpg} are the same, due to the independent reduction possibilities in λ_{cpig}. We denote the reduction in λ_{cpig} with $\rightarrow_{cpig}$ and the reduction in λ_{cpg} with $\rightarrow_{cpg}$.

Proposition A.4. *The identity translation enc_i from λ_{cpig} into λ_{cpg} is fully abstract.*

Proof. Obviously enc_i (and its inverse) are compositional. Thus, to prove observational correctness it suffices to establish convergence equivalence. We have to show four implications:

1. $enc_i(t) \downarrow_{cpg} \Longrightarrow t \downarrow_{cpig}$: This follows by using the same reduction sequence.
2. $t \downarrow_{cpig} \Longrightarrow enc_i(t) \downarrow_{cpg}$: A reduction corresponding to $t \downarrow_{cpig}$ can be rearranged until it is a reduction w.r.t. λ_{cpg}, since the reductions are at independent positions, and the final result is a value without any reductions.
3. $enc_i(t) \Downarrow_{cpg} \Longrightarrow t \Downarrow_{cpig}$: We show the equivalent $t \uparrow_{cpig} \Longrightarrow enc_i(t) \uparrow_{cpg}$. Let *Red* be a λ_{cpig}-reduction of $enc_i(t)$ to a must-divergent expression. We use induction on the measure (l,n), where l is the number of reductions and n is the number of non-value surface positions of $enc_i(t)$, i.e. positions not within abstractions. Now consider the λ_{cpg}-redex in $enc_i(t)$. If the reduction of the redex is contained in *Red*, then we can shift it to the start, and we obtain a shorter reduction, i.e. l is decreased. Otherwise, if the reduction of the redex is not contained in *Red*, there are two possibilities. If the redex is must-divergent, then we are finished, since then $enc_i(t)$ is also must-divergent. Otherwise, if the redex is not must-divergent, then we simply select a converging reduction of the redex to a value. This reduction can be integrated into *Red*. In this case the number of reductions does not change, but the number n of the measure will be reduced. In any case, we can use induction. The base case follows from (1).
4. $t \Downarrow_{cpig} \Longrightarrow enc_i(t) \Downarrow_{cpg}$: We show $enc_i(t) \uparrow_{cpg} \Longrightarrow t \uparrow_{cpig}$. We can leave the reduction unchanged. The base case is $enc_i(t) \Uparrow_{cpg} \Longrightarrow t \Uparrow_{cpig}$, which follows from (2).

Finally, full abstraction follows from Proposition 3.9, since the proof also shows that the inverse of enc_i is convergence equivalent. □

Remark A.5. Note that in languages with shared variable concurrency (for instance, extensions of λ_{cp} with reference cells) the modification of the reduction strategy given in this subsection is no longer correct: permitting interleaving reductions of the arguments can be observed through their read and write effects on shared variables.

Using Proposition 3.7 we have:

Theorem A.6. *The translation enc_{ig} is fully abstract. For $enc : \lambda_{cp}^{T} \to \lambda_c$ the combined translation $enc \circ enc_{ig} : \lambda_{cpig} \to \lambda_c$ is adequate.*

B Using Virtual Typing

We consider the issue of full abstraction of the pair encoding for simply typed λ_{cp}^{T} and assume a variation of simple typing for λ_c in order to describe the structure of the image of λ_{cp}^{T} under the translation $enc(.)$.

We define the language λ_c^{VT} as a typed variant of λ_c that is sufficiently large to serve as a target language for *enc*. The syntax of expressions in λ_c^{VT} is extended as follows. We assume that every expression s and subexpression is decorated with a pair $\langle \tau, \beta \rangle$ of labels: a type label τ, and a *selector-label* β, written as $s :: \langle \tau, \beta \rangle$, where we write $s :: \tau$, if only the type label is of interest. Here, τ is either a λ_{cp}^{T}-type (i.e., including pair types), or the special symbol † (indicating no type), and the selector-label can be either **fst**, **snd** or #. Intuitively, # can be interpreted as the absence of a selector-label. The objects of the language λ_c^{VT} are triples $(s, \langle \tau, \beta \rangle)$. Thus there may be different objects corresponding to the same λ_c-expression. Below, we give more conditions that will only accept certain triples as valid λ_c^{VT}-expressions. We assume that variables are partitioned by assigning a fixed type (or †) to each, which is also its type-label. Constants are labeled with a type that is an instance of the type as given in Fig. 5.

Instead of type derivation rules, we assume that the following consistency rules must be satisfied by λ_c^{VT}-expressions and their type- and selector-labeling. That is, types are not inferred for expressions and subexpressions, but verified against the term structure and the type- and selector-labeling.

Definition B.1. *The type consistency rules for λ_c^{VT} are:*

- *An application $(s\ t)$ is type-labeled as: $((s :: \tau_1 \to \tau_2)\ \ (t :: \tau_1)) :: \tau_2$, if it is not one of the exceptions in* **??** *below.*
- *There are different possibilities for abstractions. The expression $\lambda x.t$ is consistently typed if the term and the type labeling satisfies one of the following patterns:*
  1. $\lambda (x :: \tau_1).(t :: \tau_2) :: \tau_1 \to \tau_2$.
  2. $(\lambda (sel :: \dagger).sel\ (w_1 :: \tau_1)\ (w_2 :: \tau_2)) :: (\tau_1, \tau_2)$*, the variable sel does not occur free in s,t, w_1, w_2 are λ_c-values after stripping off the labels, and w_1, w_2 have to be type-consistent. Type consistency is not necessary for the applications $(sel\ (w_1 :: \tau_1))$ and $sel\ (w_1 :: \tau_1)\ (w_2 :: \tau_2)$). This kind of expression is the only possibility for a variable to be labeled with † as a type.*
  3. $(\lambda p.p\ (\lambda x.\lambda y.x)) :: \langle (\tau_1, \tau_2) \to \tau_1, \mathbf{fst} \rangle$ *where* $p :: ((\tau_1 \to \tau_2 \to \tau_1) \to \tau_1)$*;* $(\lambda x.\lambda y.x) : (\tau_1 \to \tau_2 \to \tau_1)$.
  4. $(\lambda p.p\ (\lambda x.\lambda y.y)) :: \langle (\tau_1, \tau_2) \to \tau_2, \mathbf{snd} \rangle$ *where* $p :: ((\tau_1 \to \tau_2 \to \tau_2) \to \tau_2)$*;* $(\lambda x.\lambda y.y) : (\tau_1 \to \tau_2 \to \tau_2)$.

Whenever an expression has a selector-label **fst** *or* **snd***, then it must be one of the cases* (3),(4) *above.*

Typing of the constants is as for λ_{cp}^{T}. An expression t of λ_c^{VT} is *well-typed* of type τ if t is type-labeled τ, and the type consistency rules hold for the subexpression of t according to Definition B.1. A λ_c^{VT}-value is defined to be a (labeled) abstraction or a constant.

The action of the reduction rules in λ_c^{VT} on the expressions and hence the label components is the obvious one, with the exception of the cases where the redex is an application of a selector-labeled expression to a pair, which is defined explicitly:

Definition B.2. *We define the type behavior of the reduction rules in λ_c^{VT} for the critical cases of an application of an implemented selector to a pair.*

- *Let the redex be an application $(s\,t)$, where s has selector-label* **fst***,*
  $s = \lambda p.p\,(\lambda x.\lambda y.x) :: (\tau_1,\tau_2) \to \tau_1$, $t :: (\tau_1,\tau_2)$, $(s\,t) :: \tau_1$, *the term t must be an abstraction* $(\lambda(sel :: \dagger).sel\ (w_1 :: \tau_1)\ (w_2 :: \tau_2)) :: (\tau_1,\tau_2)$.
  Then the beta-reduction will produce the expression

$$(\lambda(sel' :: \tau_1 \to \tau_2 \to \tau_1).sel'\ (w_1 :: \tau_1)\ (w_2 :: \tau_2))\quad (\lambda x.\lambda y.x)$$

  with the type label τ_1. The selector-label **fst** *is removed.*
- *Similarly for selector-label* **snd***.*
- *Beta-reduction must give priority to the the selector-labels* **fst**, **snd** *over the label #. The latter may be overwritten.*

Note that the reductions on the underlying λ_c-expression are exactly the same as the untyped reductions in λ_c.
A case-analysis results in the following:

Proposition B.3. *The following holds for λ_c^{VT}:*

- *The type of a closed expression is not changed by reduction.*
- *A closed well-typed expression is either reducible or a value.*

As observers in λ_c^{VT} we use the contexts of λ_c^{VT} with the following restrictions: The hole is also typed, an expression with a selector-label cannot have a hole in it; and a context or a term cannot contain a free variable with type label †.

The translations $enc(.)$ and $inc(.)$ are adapted to the labeling (see Fig. **??** and 10): $enc(.)$ keeps the type labeling and adds the select-labels. The translation $inc(.)$ maps abstractions to pairs controlled by the type labeling, and uses the selector-labels to map abstractions to the appropriate selectors.

The type labeling of contexts shows that the translations between λ_{cp}^{T} and λ_c^{VT} are compositional, and that the type mapping is the identity.

It is easy to verify that if s is a well-typed expression in λ_{cp}^{T}, then $enc(s)$ is well-typed as a λ_c^{VT}-expression, and conversely if s in λ_c^{VT} is well-typed then $inc(s)$ is well-typed as a λ_{cp}^{T}-expression. Thus *enc* and *inc* are translations also w.r.t. the typed languages.

Lemma B.4. *For the mappings enc and inc, the following holds:*

- *For all s: s is a λ_{cp}^{T}-value iff $enc(s)$ is a value.*
- *For all s: s is a λ_c^{VT}-value iff $inc(s)$ is a value.*

Proof. Follows by inspecting all the cases. □

$$
\begin{array}{ll}
enc((w_1,w_2) :: (\tau_1,\tau_2)) & = \lambda s.\ ((s :: \dagger)\ enc(w_1) :: \tau_1\ enc(w_2) :: \tau_2) :: (\tau_1,\tau_2) \\
enc(\mathbf{fst} :: (\tau_1,\tau_2) \to \tau_1) & = \lambda p.\ (p\ \lambda x.\lambda y.x) :: \langle (\tau_1,\tau_2) \to \tau_1, \mathbf{fst} \rangle \\
enc(\mathbf{snd} :: (\tau_1,\tau_2) \to \tau_2) & = \lambda p.\ (p\ \lambda x.\lambda y.y) :: \langle (\tau_1,\tau_2) \to \tau_2, \mathbf{snd} \rangle
\end{array}
$$

Fig. 9. Adaptations of translation *enc*

$$
\begin{array}{ll}
inc(\lambda s.\ ((s :: \dagger)\ w_1 :: \tau_1\ w_2 :: \tau_2) :: \langle (\tau_1,\tau_2), \# \rangle) & = (inc(w_1), inc(w_2)) :: (\tau_1,\tau_2) \\
inc(\lambda p.\ (p\ \lambda x.\lambda y.x) :: \langle (\tau_1,\tau_2) \to \tau_1, \mathbf{fst} \rangle) & = \mathbf{fst} :: (\tau_1,\tau_2) \to \tau_1 \\
inc(\lambda p.\ (p\ \lambda x.\lambda y.y) :: \langle (\tau_1,\tau_2) \to \tau_2, \mathbf{snd} \rangle) & = \mathbf{snd} :: (\tau_1,\tau_2) \to \tau_2
\end{array}
$$

Only the top-types are indicated and the †-label of s.

Fig. 10. Adaptations of translation *inc*

Theorem B.5. *The translations inc and enc are fully abstract translations between* λ_{cp}^{T} *and* $\lambda_{c}^{\mathrm{VT}}$.

Proof. Compositionality follows from the definition of the translations. Values are preserved and observational correctness of *inc* holds. The proof of observational correctness of *enc* is analogous to the proof of Proposition 4.3.

Full abstraction of both translations holds, since types are preserved, and the translations are partial inverses of each other:
$enc(inc(s)) = s$ for each $\lambda_{c}^{\mathrm{VT}}$-expression s (modulo α-renaming), and
$inc(enc(s)) = s$ for each λ_{cp}^{T}-expression s (modulo α-renaming).
Then it is easy to show that the observers are also equivalent, and hence that full abstraction of both translations holds. □

This result can be interpreted as an isomorphism between λ_{cp}^{T} and $\lambda_{c}^{\mathrm{VT}}$, which may sloppily be formulated as "the pair-constructor can be encoded in the types". However, note that λ_{cp}^{T} is not an extension of $\lambda_{c}^{\mathrm{VT}}$, since the $\lambda_{c}^{\mathrm{VT}}$-expressions are not contained in λ_{cp}^{T}. In particular, an abstraction in λ_{cp}^{T} may correspond to several objects in $\lambda_{c}^{\mathrm{VT}}$ due to the type labeling.