Correctness of Program Translations for Observational Semantics [1]

MANFRED SCHMIDT-SCHAUSS and DAVID SABEL
Goethe-University Frankfurt am Main, Germany
and
JOACHIM NIEHREN
INRIA, Lille, France, Mostrare Project
and
JAN SCHWINGHAMMER
Saarland University, Programming Systems Lab, Saarbrücken, Germany

We investigate methods and tools to analyze translations between programming languages with respect to observational semantics. We consider a framework where the behavior of programs is observed by (possibly multiple) convergence predicates in arbitrary contexts. While *observational correctness* is taken to be the fundamental notion of correctness of a translation, we emphasize the role of other correctness conditions like *adequacy* and *full abstraction* of translations. As a tool for proving these correctness properties, we propose a notion of *convergence equivalence* as a means for proving adequacy for compositional translations which avoids explicit reasoning about contexts.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Program; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics*; D.3.1 [**Programming Language**]: Formal Definitions and Theory—*Semantics*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Contextual equivalence, correctness, semantics, translations

[1]A previous version of this article appeared at IFIP TCS 2008 ([Schmidt-Schauß et al. 2008]). In the present article, the framework for OSP-calculi and for translations has been generalized to also cover call-by-value semantics, more examples are given, and some proofs are extended.

Author addresses: M. Schmidt-Schauß and David Sabel, Institut für Informatik, Fachbereich Informatik und Mathematik, Goethe-Universität Frankfurt, 60054 Frankfurt am Main, Germany.
J. Niehren, NRIA, Lille, France, Mostrare Project.
J. Schwinghammer, Saarland University, Programming Systems Lab, Saarbrücken, Germany.

1. INTRODUCTION

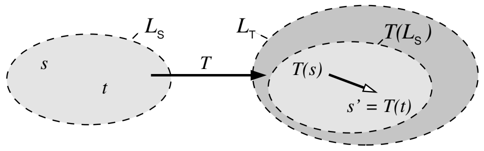
Translating programs is an important operation in computer science. There are three main tasks where translations play an important role: (1) Translation is the standard task of a compiler, where this is usually a conversion from a high-level language into an intermediate or low-level one, like an assembly language; (2) translations are also required in programming languages for explaining the meaning of surface language constructs by decomposing them into a number of more primitive operations in the core part of the programming language; (3) translations are used to obtain expressivity results between different languages or programming models. Correctness of these translations is an indispensable prerequisite for their use.

There are different approaches to the foundations of program and translation correctness, based on logical methods, operational semantics, and denotational semantics. For languages of restricted expressiveness, these methods are well-developed, and, more essentially, there is usually little disagreement about the notion of correctness that one tries to capture. Proving the correctness of program translations (for instance, on the basis of operational semantics as in the recent [Matthews and Findler 2007; Sanjabi and Ong 2007]) is an ongoing research topic that is, however, still poorly understood when it comes to concurrency and mutable state. Nevertheless these features have to be accommodated when reasoning about the implementations of language extensions in terms of a core language (which 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 [Alice 2007; Niehren et al. 2006; Schwinghammer et al. 2009], or using MVars in Concurrent Haskell [Peyton Jones et al. 1996].

To study translations, in this paper we adopt an *observational semantics* based on convergence and a set of contexts. (More precisely, as a more comprehensive term, we will often use "observer" instead of "context" in the following.) Two programs p, p' are considered equivalent if they exhibit the same convergence behavior in all contexts O, denoted as $p \sim p'$. For non-deterministic and concurrent programming languages a single (may-) convergence predicate $p{\downarrow}$ is insufficient (in the sense that the induced program equivalence does not distinguish programs that exhibit intuitively distinct behaviours). Instead, for non-deterministic and concurrent languages, a suitable equivalence arises from a combination of may- and must-convergence (see e.g. [De Nicola and Hennessy 1984; de'Liguoro and Piperno 1992; Carayol et al. 2005; Sabel and Schmidt-Schauß 2008; Niehren et al. 2007; Schmidt-Schauß and Sabel 2010]). Accordingly, we will consider an observational semantics which may be based on *multiple* convergence predicates.

In this setting, we consider translations $T : L_{\mathsf{S}} \to L_{\mathsf{T}}$ between source and target languages L_{S} and L_{T} that are both equipped with an observational semantics. We view *convergence equivalence* as a basic requirement for correct translations, stating that convergence is not changed by the translation: $p{\downarrow} \iff T(p){\downarrow}$. However, this requirement is not sufficient since it does not tell us to which extent interactions with program contexts are preserved. The fundamental semantical requirement for a correct translation is therefore *observational correctness.* This condition means that T is *convergence equivalent* and *compositional up to observations*, where the latter says that $O(p){\downarrow} \iff T(O)(T(p)){\downarrow}$ holds, i.e., observations are preserved by the translation.

Of course, often the language L_T is larger and provides more contexts than L_S; consequently, for a convergence equivalent translation T there may be programs with $p_1 \sim_{L_\mathsf{S}} p_2$ but $T(p_1) \not\sim_{L_\mathsf{T}} T(p_2)$. There are other important correctness properties of translations: A translation $T : L_\mathsf{S} \to L_\mathsf{T}$ is *adequate* if $T(s) \sim_{L_\mathsf{T}} T(t)$ implies $s \sim_{L_\mathsf{S}} t$ for all programs s and t of L_S. Adequacy is implied by observational correctness, and it ensures that program transformations of the target language L_T can be soundly applied with respect to observations made in the source language L_S. More precisely, suppose a translated program $T(s)$ is optimized (by a program transformation) to an equivalent program $s' \sim_{L_\mathsf{T}} T(s)$ and that s' is the translation of some L_S-program t, i.e. $T(t) = s'$, then $s \sim_{L_\mathsf{S}} t$ is guaranteed by adequacy of T.



Full abstraction extends adequacy by the inverse property, i.e., that program equivalence is reflected and preserved by the translation. If T is fully abstract, then L_S and $T(L_\mathsf{S}) \subseteq L_\mathsf{T}$ have the same equivalences, and L_T conservatively extends $T(L_\mathsf{S})$. Thus the contexts in L_T cannot make more distinctions in $T(L_\mathsf{S})$ than the ones that are provided by the contexts of $T(L_\mathsf{S})$ already. However, L_T may be strictly greater than $T(L_\mathsf{S})$. The property of full abstraction is often too strong as a requirement; for instance, compilation of high-level languages to abstract machine languages are typically far from being fully abstract, but nevertheless intuitively correct. In the case of language extensions full abstraction exactly means conservativeness. If the translation is a bijection on the equivalence classes, then we can even say that the translation is an *isomorphism*.

Besides convergence predicates and contexts, a further ingredient in the framework that we propose is a notion of *closedness* of programs. This concept is necessary in order to have a common framework for higher-order languages with call-by-value and call-by-need reduction strategies. For instance, the exact condition for the observational equivalence in call-by-value languages is often the following: $p \sim_{L_\mathsf{S}} p'$ holds if for all contexts O *where* $O[p]$ *and* $O[p']$ *are closed* (in the sense of having no free variables), $O[p]{\downarrow}$ iff $O[p']{\downarrow}$. In general, this relation differs from the relation defined without the closedness condition. We also consider types for programs and contexts in the framework, and allow only type correct application of contexts to programs. Correspondingly, observational equivalence (and a corresponding preorder) are only considered on programs with equal types.

In order to illustrate our framework and demonstrate its applicability, we point to rather diverse examples. First, we look at the pure and untyped call-by-name and call-by-value lambda-calculus, and then use PCF as a well-known example of a typed lambda-calculus both in its call-by-value and call-by-name variants (as well as some other variations). We also demonstrate how the non-structural and non-compositional translation of lambda-terms into deBruijn-indexed terms matches our framework, and show that this is a fully abstract translation. We also refer to several published papers on contextual equivalences and translations and explain how these make use of the framework. As a more fully worked out example we consider the standard Church encoding of pairs in a call-by-value lambda calculus; this example shows that our basic requirement for correctness, convergence equivalence, *fails* without an appropriate notion of typing.

While all these examples are rather small, we believe that they illustrate well the key properties of our framework. For instance, the typing issues raised by the encoding of pairs is an instance of the general situation where an abstract data type is implemented in terms of some operations on a representation type. A real world example – the correctness of encoding buffers in the concurrent core of Alice ML – has been worked out by the authors in [Schwinghammer et al. 2009].

Related work. Various proof methods have been developed for establishing contextual equivalences. These include context lemmas (e.g., [Milner 1977]), bisimulation methods (for instance, [Gordon 1999]), diagram-based methods (e.g., [Kutzner and Schmidt-Schauß 1998; Niehren et al. 2007]), and characterizations of contextual equivalence in terms of logical relations (e.g. [Pitts 2000]).

In most cases, translations and 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 [Ritter and Pitts 1995], 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 [Riecke 1991], via fully abstract models (necessitating the addition of parallel constructs to the languages) and domain-theoretic techniques. Milner [1990] shows that the call-by-name as well as the call-by-value lambda calculus can be encoded into the π-calculus and shows adequacy of these encodings. A similar result is obtained for the call-by-name lambda calculus with McCarthy's amb-operator in [Carayol et al. 2005]. The fact that adequate (and fully abstract) translations compose is exploited in [McCusker 1996], where a syntactic translation is used to lift semantic models for FPC to ones for the lazy lambda calculus. In a similar vein, Sanjabi and Ong [2007] develop a translation from an aspect-oriented language to an ML-like language, to obtain a model of the former. Their adequacy proof follows a similar pattern to ours, but does not abstract away from the particularities of the concrete languages. In [Johann and Voigtländer 2006; Voigtländer and Johann 2007], the effect of adding a strictness operator to lambda calculus with parametric polymorphism is exhibited.

Shapiro [1991] 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 [1991] and Mitchell [1993]. 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 observational correctness and adequacy.

For parallel and concurrent languages, approaches to prove compiler correctness can be found in [Wand 1995; Gladstein and Wand 1996]. While these results make use of a denotational semantics (and its domain is a common "intermediate language" for both the source and the target language), the recent [Hu and Hutton 2009] does not use a denotation, but shows correctness more directly. Nevertheless, the approach taken in [Hu and Hutton 2009] requires that the values of the source and the target language are comparable by a bisimulation equivalence. The work on certified compilers of C [Leroy 2009] uses also behavioral criteria for correctness, however, the correctness notion differs from ours, since mainly convergence equivalence of the translations is shown where erroneous source programs are not considered. In [Sabel 2008] observational correctness and also full-abstraction of a translation from a non-deterministic call-by-need lambda calculus with McCarthy's amb [McCarthy 1963] into a concurrent abstract machine was shown w.r.t. fair may- and must-convergence.

Using the techniques from this paper, Schmidt-Schauß et al. [2010] prove some isomorphisms between different deterministic extended lambda calculi with cyclic let and Abramsky's lazy lambda calculus [Abramsky 1990], based on contextual equivalence.

Outline. In Section 2 we introduce our framework for program calculi with observational semantics. In Section 3 we define the notion of a translation, introduce the fundamental properties of translations, discuss which is the right correctness notion and finally show some relations between these properties. In Section 4 we consider the specific case of language extensions and analyze the conditions under which full abstraction can be deduced. In Section 5 we provide two larger examples for translations: Church's encoding of pairs and the de Bruijn encoding of the lambda calculus. For both examples we show observational correctness of the corresponding translations.

2. CALCULI WITH OBSERVATIONAL SEMANTICS

In this section we present a general framework within which notions of correctness of language translations can be analyzed and discussed. The ingredients of the framework are types, programs (or expressions, processes,...), a closedness test, contexts (as well as a subset of closing contexts), and convergence observations, which then allow us to explain program equivalence and a notion of correctness for translations. We will motivate and illustrate these ingredients by classic, standard examples of programming language abstractions: these consist usually of calculi equipped with a contextually defined program equivalence that is based on convergence behavior. After introducing our definitions, we illustrate that this framework captures a wide range of situations: call-by-value, call-by-name and call-by-need, small-step and big-step operational semantics, deterministic and non-deterministic reduction, contextual and test-function based equivalence, and simple forms of typing.

2.1 Examples for Abstract Calculi with Observational Semantics

We illustrate the ideas behind the framework by varying standard examples of idealized programming languages, like lambda calculi and PCF. As a first example we consider the lazy lambda calculus:

Example 2.1 *The call-by-name lambda calculus.* The call-by-name (or lazy) lambda calculus [Plotkin 1975; Abramsky 1990] is an untyped pure λ-calculus. The expressions of this calculus, given by the grammar $t ::= x \mid (t_1\ t_2) \mid \lambda x.t$, consist of variables x from some infinite set, applications $(t_1\ t_2)$, and functional abstractions $\lambda x.t$. Contexts C of the call-by-name lambda calculus are like expressions where a single subexpression is replaced by the context *hole* $[\cdot]$. $C[s]$ is the expression obtained by filling the hole of C by the expression s. We define convergence of a term t to an abstraction v (denoted with $t \downarrow v$) by a big-step semantics, i.e. it is defined inductively as follows:

- $\lambda x.s \downarrow \lambda x.s$
- $\dfrac{s \downarrow \lambda x.s' \qquad s'[t/x] \downarrow v}{(s\ t){\downarrow}v}$

The observation $t{\downarrow}$ ("*t converges*") holds iff there exists an abstraction v such that $t \downarrow v$ holds. An equivalent definition of the operational semantics and convergence can be given by a small-step semantics, where the reduction relation $\to$ is defined by beta reduction steps applicable in reductions contexts that satisfy the grammar $R ::= [\cdot] \mid (R\ t)$, i.e., $R[((\lambda x.t_1)\ t_2)] \to R[t_1[t_2/x]]$. Convergence $t{\downarrow}$ holds iff $t \to^* v$ for some abstraction v by a sequence of call-by-name beta-reductions. Contextual equivalence $t \sim t'$ holds iff for all contexts C: $C[t]{\downarrow} \Leftrightarrow C[t']{\downarrow}$. It is easy to verify that the relation $\sim$ is a congruence.

We also summarize easy observations for the pure call-by-value calculus:

Example 2.2 *The call-by-value lambda calculus.* The call-by-value (cbv) lambda calculus [Plotkin 1975] is an untyped pure λ-calculus. The expressions and contexts are the same as for the lazy lambda calculus. The operational semantics and convergence can be given by a small-step semantics (and also alternatively by a big-step semantics), where the reduction relation $\to_{cbv}$ is defined by value-beta reduction steps applicable in reductions contexts that satisfy the grammar $R ::= [\cdot] \mid (R\ t) \mid (v\ R)$, where v is a variable or an abstraction, i.e., $R[((\lambda x.t)\ v)] \to R[t[v/x]]$ where v is an abstraction or a variable. Convergence w.r.t call-by-value $t{\downarrow_{cbv}}$ holds iff $t \to^*_{cbv} v$ for some variable or abstraction v by a sequence $\to^*_{cbv}$ of call-by-value beta-reductions.
Contextual equivalence $t \sim_{cbv} t'$ holds iff for all contexts C: $C[t]{\downarrow_{cbv}} \Leftrightarrow C[t']{\downarrow_{cbv}}$.

Another often used variant of contextual equivalence is to restrict the contexts in the definition to closing contexts: $t \sim t'$ iff for all contexts C such that $C[t]$ and $C[t']$ are closed, $C[t]{\downarrow} \Leftrightarrow C[t']{\downarrow}$. It is folklore that the observational equivalence of the call-by-name λ-calculus is not changed by this restriction. The proof technique is to "close" contexts C by transferring them into $(\lambda x_1, \dots x_n.C)\ \Omega\ \dots \Omega$, where $\Omega := (\lambda x.(x\ x))\ (\lambda x.(x\ x))$. The same result holds for the (pure) call-by-value λ-calculus, where closing C is done by transferring it into $(\lambda x_1, \dots x_n.C)\ (\lambda x.\Omega)\ \dots\ (\lambda x.\Omega)$.
However, if Boolean constants and conditionals like `if-then-else` are added to a call-by-value calculus, then observational equivalence including all contexts may

differ from the observational equivalence restricted to closing contexts. E.g., in the typed call-by-value lambda calculus with Boolean values and `if-then-else` the equivalence $C[\texttt{if}\ x\ \texttt{then true else true}]{\downarrow} \iff C[\texttt{true}]{\downarrow}$ holds for all (type-correct) closing contexts C, but does not hold for arbitrary contexts: for the empty context, ($\texttt{if}\ x\ \texttt{then true else true}$) cannot converge, while $\texttt{true}{\downarrow}$. The usual approach for call-by-value languages is to consider closing contexts only (e.g. [Plotkin 1975; Mason et al. 1996; Pitts and Stark 1998]).

The example calculi considered so far indicate that the equivalence using closing contexts is more general than the equivalence defined for all contexts. Experience shows that this holds for most calculi, hence the framework will support this form of definition.

The prototypical example of an idealized (functional) programming language is PCF with contextual equivalence, where we will use variations of PCF, like call-by-value and call-by-name reduction strategy, and also other variations of reductions of constants. We briefly recall call-by-value PCF.

Example 2.3 *Call-by-value PCF.* The language PCF (in its call-by-value variant) [Plotkin 1977; Pierce 2002] is the simply-typed lambda calculus over two base types B and N (Booleans and numbers), with constants for Boolean values and non-negative numbers, arithmetic operations **pred**, **succ**, and **zero?**, and fixed point combinators at all function types. *Contexts* are defined like PCF terms C, but with a "hole" which can be filled with appropriately typed terms t (possibly capturing free variables of t) to obtain another term $C[t]$. The language is usually presented with a big-step operational semantics. We will use here the equivalent small-step operational semantics $t \to t'$, which is generated by call-by-value beta reduction $(\lambda x.t)\,v \to t[v/x]$, and reduction axioms for the constants, such as $\mathbf{fix}\ \lambda x.t \to t[\lambda y.(\mathbf{fix}\ \lambda x.t)\,y/x]$. The reduction must take place in reduction contexts R defined as $R ::= [.] \mid (R\ s) \mid (v\ R) \mid \texttt{if}\ R\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \mid \mathbf{pred}\ R \mid \mathbf{succ}\ R \mid \mathbf{zero?}\ R$. Closed values are defined as boolean values, integer values, function constants, or closed abstractions. Variables are seen as non-closed values. Due to the fixpoint reduction axiom, programs may diverge, and we write $t{\downarrow}$ if there is some value v such that $t \to^* v$. *Contextual equivalence* on equally-typed terms is defined by observing termination in all *closing* contexts: $t \sim t'$ if for all appropriately typed contexts C and if $C[t], C[t']$ are closed, $C[t]{\downarrow} \Leftrightarrow C[t']{\downarrow}$. This captures the intuition that t and t' may be freely exchanged in any larger program without affecting its observable behavior. Technically, $\sim$ forms a congruence on terms (see Proposition 2.10), which follows from the fact that contexts can be "composed" by plugging one context C into the hole of a second context C' to yield a context $C'[C]$. As already remarked, the contextual equivalence defined w.r.t. all contexts is not the same as $\sim$: the two expressions $\texttt{if}\ x\ \texttt{then true else true}$ and `true` are witnesses for this difference.

Example 2.4 *Call-by-name PCF.* The call-by-name variant of PCF has the following differences compared with call-by-value PCF: beta-reduction is applicable for any argument s: $(\lambda x.t)\,s \to t[s/x]$, the reduction contexts are different, since they do not force evaluation of arguments in expressions like $(\lambda x.t)\,s$, and there is a modified fixpoint reduction: $\mathbf{fix}\ \lambda x.t \to t[(\mathbf{fix}\ \lambda x.t)/x]$. The reduction contexts are $R ::= [.] \mid (R\ s) \mid \texttt{if}\ R\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \mid \mathbf{pred}\ R \mid \mathbf{succ}\ R \mid \mathbf{zero?}\ R$. Analyzing the observational equivalence shows that contextual equivalence w.r.t. all contexts is the same as for closed contexts.

Remark 2.5 *(eta)-conversion.* The conversion (eta) is usually not correct under observational equivalence, since $\Omega \not\sim \lambda x.\Omega$. Depending on the calculus, some instances of (eta) may hold: The equivalence $v \sim_{cbv} \lambda y.v\ y$ usually holds, where v is a value. It holds in the call-by-name calculus of Example 2.1 for abstractions v, and in the call-by-value calculus of Example 2.2 if v is an abstraction or a variable.

It is useful to allow for more than a single observation, in particular for calculi that model non-determinism or concurrency, as the following two examples show.

Example 2.6 *Non-deterministic call-by-value PCF.* Suppose that call-by-value PCF (with a small-step reduction semantics, see Example 2.3) is extended by a choice construct $t \oplus t'$, with reductions $v_1 \oplus v_2 \to v_i$. For this non-deterministic language it makes sense to observe not only whether a program *may* terminate, but also if its termination is inevitable: for instance, a form of *must-convergence*, written $t\Downarrow$, is defined if the possibility of convergence is retained by reduction. Formally, $t\Downarrow$ if $t \to^* t'$ implies $t'\downarrow$. Then, contextual equivalence is defined with respect to both may- and must-convergence behavior, i.e., $t \leq_{may} t'$ if for all appropriately typed contexts C such that $C[s], C[t]$ are closed: $C[t]\downarrow \implies C[t']\downarrow$ and, similarly, $t \leq_{must} t'$ if $C[t]\Downarrow \implies C[t']\Downarrow$ for these contexts. Then $t \sim_{may} t'$ iff $t \leq_{may} t'$ and $t' \leq_{may} t$, $t \sim_{must} t'$ iff $t \leq_{must} t'$ and $t' \leq_{must} t$, and $t \sim t'$ iff $t \sim_{may} t'$ and $t \sim_{must} t'$. As in Example 2.3, contextual equivalence is a congruence relation.

There is at least one other form of must-convergence used in the literature for non-deterministic calculi: $t\Downarrow'$ if there is no infinite reduction sequence starting from t. Here, we just want to make the point that neither $\downarrow$ nor $\Downarrow$ alone suffice to capture the intuitive notion of equivalence. For example, $\lambda x.x \oplus \lambda x.\Omega \sim_{may} \lambda x.x$ holds, whereas, if must-convergence is considered, we can distinguish these expressions: $\lambda x.x \oplus \lambda x.\Omega \not\sim_{must} \lambda x.x$, for each of the must-convergence notions.

Example 2.7 *Terminating PCF.* Let the language $\text{PCF}_{-\mu,0}$ be call-by-value PCF (see Example 2.3) modified as follows: the fixed point operators are dropped and (**pred** 0) is defined as resulting in 0 (instead of being an error). Then all closed programs terminate with a value, and therefore the observation of termination does no longer make sense. Instead, a sensible notion of program equivalence may be obtained by observing if a program results in a particular (integer) constant: writing $t\downarrow_n$ if $t \to^* v$ implies $v = \underline{n}$, contextual equivalence $t \sim_n t'$ holds if for all appropriately typed contexts C such that $C[t], C[t']$ are closed: $C[t]\downarrow_n \Leftrightarrow C[t']\downarrow_n$, and $t \sim t'$ if for all $n : t \sim_n t'$.

Observe that in Example 2.7 $t \sim_n t'$ is equivalent to $t \sim_1 t'$. Hence, in contrast to the Example 2.6, here a single convergence predicate would be sufficient to define $\sim$. In other words, while it can be *natural* to consider sets of observations and the equivalences they induce, this generality is not always *essential*.

2.2 Program Calculi with Observational Semantics

Abstracting from these examples, a calculus in our framework consists of a collection of *types*, typed *programs*, typed *contexts*, a notion of *observation* and a notion of *closedness*. In the following we use a slightly more neutral terminology and, instead of contexts, speak of *tests* or *observers*. This makes it easier to fit formalisms without an obvious notion of context into the framework, like abstract machines.

Definition 2.8 *(OSP calculus).* A *program calculus with observational semantics (OSP-calculus)* is a tuple $(\mathcal{T}, \mathcal{P}, \mathcal{O}, \mathcal{OBS}, \mathcal{G}, \mathcal{CO})$ where

—$\mathcal{T}$ is a set of *types*, ranged over by τ.
—$\mathcal{P}$ is a family of sets $\mathcal{P}_\tau$ for every type $\tau \in \mathcal{T}$, where $\mathcal{P}_\tau$ is the set of *programs of type* τ, ranged over by p. In abuse of notation we write $\mathcal{P}$ for $\bigcup_{\tau \in \mathcal{T}} \mathcal{P}_\tau$.
—$\mathcal{O}$ is a family of sets of functions $\mathcal{O}_{\tau_1,\tau_2}$ for every pair $\tau_1, \tau_2 \in \mathcal{T}$ of types, called *observers*, with $O : \mathcal{P}_{\tau_1} \to \mathcal{P}_{\tau_2}$ for $O \in \mathcal{O}_{\tau_1,\tau_2}$, such that
  —the identity function Id_τ is included in $\mathcal{O}_{\tau,\tau}$ for every type τ.
  —$\mathcal{O}$ is closed wrt. function composition, i.e. for all $\tau_1, \tau_2, \tau_3 \in \mathcal{T}$ and all $O_1 \in \mathcal{O}_{\tau_1,\tau_2}, O_2 \in \mathcal{O}_{\tau_2,\tau_3}$: $O_2 \circ O_1 \in \mathcal{O}_{\tau_1,\tau_3}$.
  For notational convenience we write $\mathcal{O}$ for $\bigcup_{\tau_1,\tau_2 \in \mathcal{T}} \mathcal{O}_{\tau_1,\tau_2}$.
—$\mathcal{OBS} = \{\Downarrow_1, \Downarrow_2, \ldots\}$ is a set of *observations* (or convergence predicates) where each $\Downarrow_i$ is a predicate[2] on $\mathcal{P}$ written in postfix notation. For an observation $\Downarrow_i$ we write $\Uparrow_i$ for its negation, i.e. $p \Uparrow_i$ iff $p{\Downarrow_i}$ does not hold.
—$\mathcal{G}$ is a predicate on $\mathcal{P}$, where we write $\mathcal{G}(p)$ if $\mathcal{G}$ holds for p. Intuitively, we think of $\mathcal{G}(p)$ as meaning that p is a closed program.
—$\mathcal{CO}$ (closing observers) is a family of sets of functions $\mathcal{CO}_\tau$ for every type $\tau \in \mathcal{T}$, such that $\emptyset \neq \mathcal{CO}_\tau \subseteq \mathcal{O}_{\tau,\tau}$ and such that $\mathcal{CO}$ is closed w.r.t. function composition, i.e. for all $\tau_1, \tau_2, \tau_3 \in \mathcal{T}$ and all $D_1 \in \mathcal{CO}_{\tau_1,\tau_2}, D_2 \in \mathcal{CO}_{\tau_2,\tau_3}$: $D_2 \circ D_1 \in \mathcal{CO}_{\tau_1,\tau_3}$.

The following conditions relating $\mathcal{CO}$ and $\mathcal{G}$ must hold:

(OSP-C1) For all types τ and all $p \in \mathcal{P}_\tau$, there is some $D \in \mathcal{CO}_\tau$, such that $\mathcal{G}(D(p))$ holds.
(OSP-C2) For all programs $p \in \mathcal{P}_\tau$, all $D \in \mathcal{CO}_\tau$: $\mathcal{G}(p) \implies \mathcal{G}(D(p))$.
(OSP-C3) For all programs $p \in \mathcal{P}_\tau$, all $D \in \mathcal{CO}_\tau$ and for all i:
  $\mathcal{G}(p) \implies (D(p){\Downarrow_i} \iff p{\Downarrow_i})$.

The above examples match this definition by taking $\mathcal{P}_\tau$ to be the set of (all or only the closed) expressions of type τ, $\mathcal{O}_{\tau_1,\tau_2}$ the set of τ_2-valued contexts with hole of type τ_1, and where $C(t)$ is $C[t]$. In Examples 2.1, 2.2, 2.3 and 2.4, the observations are $\{\Downarrow_1, \Downarrow_2, \ldots\} = \{\downarrow\}$, in Example 2.6 we have $\{\Downarrow_1, \Downarrow_2, \ldots\} = \{\downarrow, \Downarrow\}$, and in Example 2.7 the observations are $\{\Downarrow_1, \Downarrow_2, \ldots\} = \{\downarrow_n \mid n \in \mathbb{Z}\}$.

The motivation for including $\mathcal{G}$ and $\mathcal{CO}$ (test for closedness and a set of closing observers, respectively) is to provide a common framework for call-by-name and call-by-value calculi and their contextual equivalence relation (see Example 2.3). The closing observers from $\mathcal{CO}$ may be used to close expressions before applying the tests. However, $\mathcal{CO}$ should be chosen as a particular small subset of the observers that behave similar to substitutions that replace free variables by closed expressions. For example, in lambda calculi, an appropriate choice of $\mathcal{CO}$ would be the closed (substitutive) contexts $(\lambda x_1, \ldots x_n.[\cdot])\ s_1 \ldots s_n$ and their compositional closure. It is also possible to permit the substitutions $\{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$ where a_i are

[2]A predicate is understood like a function from programs to Boolean values

closed values (or other closed programs), provided the substitutions are observers. On the other hand, if $\mathcal{CO}$ would be chosen as all closed contexts, then the last property of Definition 2.8 is in general false, since for example in call-by-value lambda calculi: $\texttt{true}\downarrow$, and for $D := (\Omega\ [\cdot])$, we have $D[\texttt{true}]\uparrow$. The abstract properties of OSP-calculi are sufficient to ensure that observational equivalence is a congruence (proved in Proposition 2.10).

In our example calculi (Example 2.1,2.2, 2.3, 2.4, 2.6, and 2.7) so far, the conditions on $\mathcal{G}$ and $\mathcal{CO}$ hold using the substitutive contexts and their compositions, or the substitutions: Every expression can be closed with some observer from $\mathcal{CO}$ and for closed expressions e, the expression $D[e]$ for $D \in \mathcal{CO}$ has the same convergence behavior as e.

In Example 2.1, the lazy lambda calculus, the most appropriate modeling is to use the definition of contextual equivalence via closing contexts. However, it is also possible to choose $\mathcal{G}$ to be true for all programs and use the singleton containing the empty context (i.e. the identity observer) as the set $\mathcal{CO}$, which results in the same contextual equivalence.

Definition 2.8 allows for untyped calculi by considering a single, 'universal' type. Note that in this case the conditions simply state that $\mathcal{O}$ is a monoid with unit Id.

The definition of contextual equivalence generalizes in the evident way from the above examples. In fact, we will also consider a preorder that allows more flexibility and is an analogue of the domain-theoretic information preorder.

Definition 2.9 *Observational preorder and observational equivalence.* For a fixed OSP calculus, and for each $\tau \in \mathcal{T}$, we define the following relations on $\mathcal{P}_\tau$:

—$p_1 \leq_{\Downarrow_i,\tau} p_2$ iff for all $\tau' \in \mathcal{T}$ and all $O \in \mathcal{O}_{\tau,\tau'}$, if $\mathcal{G}(O(p_1))$ and $\mathcal{G}(O(p_2))$, then $O(p_1)\Downarrow_i$ implies $O(p_2)\Downarrow_i$.

—$p_1 \leq_\tau p_2$ iff $\forall i : p_1 \leq_{\Downarrow_i,\tau} p_2$.

—$p_1 \sim_{\Downarrow_i,\tau} p_2$ iff $p_1 \leq_{\Downarrow_i,\tau} p_2$ and $p_2 \leq_{\Downarrow_i,\tau} p_1$.

—$p_1 \sim_\tau p_2$ 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*, and the relations $\sim_{\Downarrow_i,\tau}$ and $\sim_\tau$ are *congruences*, in the following sense:

PROPOSITION 2.10 (PRE-) CONGRUENCE.

(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 $O(p_1) \leq_{\Downarrow_i,\tau'} O(p_2)$ for all $O \in \mathcal{O}_{\tau,\tau'}$ (similarly for $\leq_\tau$).*

(2) *The relations $\sim_{\Downarrow_i,\tau}$ and $\sim_\tau$ are* congruences*, i.e. they are precongruences and equivalence relations.*

PROOF. It is easy to see that $\leq_{\Downarrow_i,\tau}$ is reflexive. In order to check that each $\leq_{\Downarrow_i,\tau}$ is transitive, let $p_1 \leq_{\Downarrow_i,\tau} p_2 \leq_{\Downarrow_i,\tau} p_3$ and O be an observer in $\mathcal{O}_{\tau,\tau'}$, such that $\mathcal{G}(O(p_1))$ and $\mathcal{G}(O(p_3))$, and such that $O(p_1)\Downarrow_i$. We have to show that $O(p_3)\Downarrow_i$. If $\mathcal{G}(O(p_2))$, then $O(p_1)\Downarrow_i$ implies $O(p_2)\Downarrow_i$, which in turn implies $O(p_3)\Downarrow_i$. In the other case, there is some observer $D \in \mathcal{CO}_{\tau'}$, such that $\mathcal{G}(D(O(p_2)))$. By the conditions on $\mathcal{G}$, $D(O(p_1))\Downarrow_i \iff O(p_1)\Downarrow_i$ and $D(O(p_3))\Downarrow_i \iff O(p_3)\Downarrow_i$, hence $D(O(p_1))\Downarrow_i$. Since $D \circ O$ is also an observer, we obtain $D(O(p_2))\Downarrow_i$, and also

$D(O(p_3))\Downarrow_i$, since $\mathcal{G}(D(O(p_3)))$, and thus $O(p_3)\Downarrow_i$. It remains to show that $\leq_{\Downarrow_i,\tau}$ is compatible with observers: Let $p_1 \leq_{\Downarrow_i,\tau} p_2$ and $O \in \mathcal{O}_{\tau,\tau'}$. For any observer $O' \in \mathcal{O}_{\tau',\tau''}$ with $\mathcal{G}(O'(O(p_1)))$ and $\mathcal{G}(O'(O(p_2)))$ the inequation $p_1 \leq_{\Downarrow_i,\tau} p_2$ obviously implies $O'(O(p_1))\Downarrow_i \implies O'(O(p_2))\Downarrow_i$, since $O' \circ O$ is also an observer.

Part (2) of the proposition follows immediately from the first part and the definition of $\sim_\tau$ in terms of $\leq_\tau$. $\square$

Lemma 2.11. *Let p_1, p_2 be programs of the same type τ with $\mathcal{G}(p_1), \mathcal{G}(p_2)$. If $p_1\Downarrow_i$ and one of the following holds: $p_1 \sim_\tau p_2$, $p_1 \sim_{\Downarrow_i,\tau} p_2$, $p_1 \leq_\tau p_2$, or $p_1 \leq_{\Downarrow_i,\tau} p_2$, then also $p_2\Downarrow_i$.*

In the following, types are sometimes omitted in the notation, and we implicitly assume that type information follows from the context.

2.3 Further Examples

The framework may be instantiated by lambda calculi with a small-step operational semantics, like the lazy lambda calculus, call-by-name and call-by-value PCF as in our previous examples, etc. Also abstract machines fit into the framework where machine environments, stacks, heaps etc. may be modelled as observers. We also do not rely only on small-step semantics, also calculi with big-step operational semantics fit into our framework. We sketch some further examples to illustrate the range of situations that fit the definition of an abstract OSP calculus. In particular, Definition 2.8 captures not only variants of the lambda calculus, but can also be applied to process calculi:

Example 2.12 *CCS.* CCS [Milner 1989] may be viewed as an (untyped) OSP calculus: for a fixed action set Σ, both programs and observers are given by the set of CCS processes $P, Q, \ldots$, and $P \circ Q$ as well as $P(Q)$ are given by the parallel composition $P \mid Q$. More precisely, observers are given by the functions f_P with $f_P(Q) = P \mid Q$. By considering observation predicates $\downarrow_\sigma$ for every $\sigma \in \Sigma^\omega$ such that $P\downarrow_\sigma$ holds if σ is a trace of P, we obtain a trace-based testing equivalence $\sim$ on processes. Variations are possible, for instance by restricting the observations to finite traces $\sigma \in \Sigma^*$ (see [Nain and Vardi 2007]).

The term "calculus" in Definition 2.8 is to be understood in a loose sense. For instance, also semantic models fit in:

Example 2.13 *Cpos.* A semantic counterpart to PCF, as described in Example 2.3, is given by ω-complete pointed partial orders (cppos) and continuous maps. More precisely, if $\mathcal{D}_B$ and $\mathcal{D}_N$ are the flat cppos with underlying sets $\{0, 1\}$ and $\mathbb{Z}$ respectively, we let $\mathcal{D}_{\tau_1\to\tau_2} = \mathcal{D}_{\tau_1} \to (\mathcal{D}_{\tau_2})_\perp$ be the set of strict continuous functions from $\mathcal{D}_{\tau_1}$ to $\mathcal{D}_{\tau_2}$ extended with a new least element, and order $\mathcal{D}_{\tau_1\to\tau_2}$ pointwise. We can then take $\mathcal{P}_\tau$ to be the underlying set of $\mathcal{D}_\tau$. The observers are continuous maps, i.e., $\mathcal{O}_{\tau_1\to\tau_2} = \mathcal{D}_{\tau_1} \to \mathcal{D}_{\tau_2}$, and for $a \in \mathcal{P}_\tau$ the observation $a\downarrow$ holds if $a \neq \perp$. In this example, $a \sim_\tau a'$ if and only if $a = a'$.

3. TRANSLATIONS

In this section we discuss mappings between OSP calculi. Such mappings often arise very concretely when relating two calculi; examples are compilations of one programming language into another, which may induce a mapping between possibly rather different calculi, or the removal of syntax sugar, which may be expressed as a mapping from an extended calculus into a core calculus, or the embedding of a calculus in its extended version. Also expressivity results between different calculi are usually obtained by mapping one calculus into another one.

A simple concrete example of the removal of syntax sugar is an extension of PCF with n-ary functions (see also the extended Example 4.7), which can be viewed as a syntactic convenience and expressed as curried function abstraction and application:

$$\begin{array}{rcl}
(\tau_1,\ldots,\tau_n) \to \tau & \leadsto & \tau_1 \to (\ldots \to (\tau_n \to \tau))\\
\lambda(x_1,\ldots,x_n).t & \leadsto & \lambda x_1.\ \ldots \lambda x_n.t\\
t(t_1,\ldots,t_n) & \leadsto & ((t\,t_1)\ldots)\,t_n
\end{array}$$

By extending this scheme homomorphically to all other language constructs, we obtain a mapping between n-ary PCF and the basic PCF language.

Call-by-value PCF extended with data constructors like pairs and lists permits well-behaved translations into PCF under certain typing restrictions, as shown in Section 5.1. The de Bruijn-encoding of lambda-calculi can also be seen as a translation (see Section 5.2).

A question that arises is in which sense such translations between OSP calculi are correct, i.e., how does the semantics in source and target calculus relate with respect to the translation, and what are the minimal correctness requirements?

3.1 Definition of Translations

In the following we mainly consider translations between OSP calculi that have the same number of observation predicates $\{\Downarrow_1, \Downarrow_2, \ldots\}$, in a fixed ordering. However, we generalize this by permitting different kinds and numbers of such predicates in order to facilitate comparisons. We define some characterizing notions of translations. In the remainder of this section we exhibit their dependencies and prove some consequences.

Definition 3.1 *Translation.* A *translation* $T : \mathcal{C} \to \mathcal{C}'$ between two OSP-calculi $\mathcal{C} = (\mathcal{T}, \mathcal{P}, \mathcal{O}, \mathcal{OBS}, \mathcal{G}, \mathcal{CO})$ and $\mathcal{C}' = (\mathcal{T}', \mathcal{P}', \mathcal{O}', \mathcal{OBS}', \mathcal{G}', \mathcal{CO}')$ is a mapping from types to types $T : \mathcal{T} \to \mathcal{T}'$, programs to programs $T : \mathcal{P}_\tau \to \mathcal{P}'_{T(\tau)}$, observers to observers $T : \mathcal{O}_{\tau,\tau'} \to \mathcal{O}'_{T(\tau),T(\tau')}$, observation predicates to observation predicates $T : \mathcal{OBS} \to \mathcal{OBS}'$, and closing observers to closing observers $T : \mathcal{CO}_\tau \to \mathcal{CO}'_{T(\tau)}$, such that $T(Id_\tau) = Id_{T(\tau)}$.

3.2 Correctness of Translations

The following definition captures the fundamental requirements for correct translations:

Definition 3.2 *Observational correctness of translations.* A translation $T : \mathcal{C} \to \mathcal{C}'$ is *observationally correct* (oc, for short) if it is convergence equivalent and compositional up to observations, where a translation T is:

convergence equivalent (ce) if for all p: $\mathcal{G}(p) \iff \mathcal{G}'(T(p))$ (also called *closedness equivalent* cle), and if for all i and all p: $\mathcal{G}(p)$ implies $\big(p\Downarrow_i \iff T(p)\Downarrow'_{T(i)}\big)$, and

compositional up to observations (cuo) if the following two conditions hold: (i) for all types τ, τ', all observers $O \in \mathcal{O}_{\tau,\tau'}$, and all programs $p \in \mathcal{P}_\tau$: $\mathcal{G}(O(p)) \Leftrightarrow \mathcal{G}'(T(O)(T(p)))$; and (ii) for all i, all types τ, τ', all observers $O \in \mathcal{O}_{\tau,\tau'}$, and all programs $p \in \mathcal{P}_\tau$: $\mathcal{G}(O(p))$ implies $T(O(p))\Downarrow'_{T(i)} \iff T(O)(T(p))\Downarrow'_{T(i)}$.

A trivial example of a translation that satisfies all the properties above is the identity translation $\mathcal{C} \to \mathcal{C}$.
Since there is an identity observer for every type, we have:

Lemma 3.3. *If a translation T is* (cuo)*, then for all p: $\mathcal{G}(p) \iff \mathcal{G}'(T(p))$ holds.*

Observational correctness has a more explicit description by a homomorphism-like condition: The translation retains the results of applying observers and then applying a convergence test.

Lemma 3.4 Alternative characterization of (oc). *Suppose $T : \mathcal{C} \to \mathcal{C}'$ is a translation. Then the following are equivalent:*

(1) T is observationally correct.

(2) (acooc): For all $\tau \in \mathcal{T}$, all $p \in \mathcal{P}_\tau$, $O \in \mathcal{O}_{\tau,\tau'}$:
$\mathcal{G}(O(p)) \iff \mathcal{G}'(T(O)(T(p)))$,
and if $\mathcal{G}(O(p))$, then for all i: $O(p)\Downarrow_i \iff T(O)(T(p))\Downarrow'_{T(i)}$.

Proof. (1) $\implies$ (2): Let T be (oc), i.e., convergence equivalent and compositional up to observations; let O be an observer and p be a program such that $\mathcal{G}(O(p))$. Then

$$O(p)\Downarrow_i \overset{\mathsf{ce}}{\iff} T(O(p))\Downarrow'_{T(i)} \overset{\mathsf{cuo}}{\iff} T(O)(T(p))\Downarrow'_{T(i)}$$

and thus the claim holds.

(2) $\implies$ (1): Instantiating O with the unit Id_τ we first obtain that $\mathcal{G}(p) \iff \mathcal{G}'(T(p))$ holds. Furthermore, the condition $\mathcal{G}(p)$ implies

$$p\Downarrow_i \iff Id_\tau(p)\Downarrow_i \overset{\mathsf{acooc}}{\iff} T(Id_\tau)(T(p))\Downarrow'_{T(i)} \iff T(p)\Downarrow'_{T(i)} \qquad (1)$$

since $T(Id_\tau) = Id_{T(\tau)}$. Hence T is (ce).

It remains to show that T is (cuo): Let $\mathcal{G}(O(p))$ hold, then:

$$T(O(p))\Downarrow'_{T(i)} \overset{\mathsf{ce}}{\iff} O(p)\Downarrow_i \overset{\mathsf{acooc}}{\iff} T(O)(T(p))\Downarrow'_{T(i)}. \quad \square$$

Compositionality up to observations is a generalization of compositionality and its variants, with easy proofs:

Lemma 3.5. *Let $T : \mathcal{C} \to \mathcal{C}'$ be a translation that is* cle*. Then each of the following conditions implies that T is* (cuo)*:*

(1) T is compositional*, meaning that for all O and p: $\mathcal{G}(O(p))$ implies $T(O(p)) = T(O)(T(p))$.*

(2) T is compositional modulo $\sim$*, meaning that for all (appropriately typed) O and p: $\mathcal{G}'(T(O(p))) \iff \mathcal{G}'(T(O)(T(p)))$ and $\mathcal{G}(O(p)) \implies T(O(p)) \sim' T(O)(T(p))$.*

PROOF. Let T be compositional. Then $\mathcal{G}(O(p)) \Leftrightarrow \mathcal{G}'(T(O(p)))$ holds, since T is closedness equivalent. Compositionality shows $\mathcal{G}'(T(O(p))) \Leftrightarrow \mathcal{G}'(T(O)(T(p)))$ and thus $\mathcal{G}(O(p)) \Leftrightarrow \mathcal{G}'(T(O)(T(p)))$. The remaining part of (cuo) holds obviously.

Now let T be compositional modulo $\sim$. Then $\mathcal{G}(O(p)) \Leftrightarrow \mathcal{G}'(T(O(p)))$ by closedness equivalence. The compositionality modulo $\sim$-conditions imply $\mathcal{G}(O(p)) \Leftrightarrow \mathcal{G}'(T(O)(T(p)))$. For the other part of (cuo) let $\mathcal{G}(O(p))$ hold. From closedness equivalence we have $\mathcal{G}'(T(O(p)))$ and also $\mathcal{G}'(T(O)(T(p)))$ by compositionality modulo $\sim$. Now Lemma 2.11 is applicable and shows $T(O(p))\Downarrow'_{T(i)} \iff T(O)(T(p))\Downarrow'_{T(i)}$. □

This lemma cannot be improved (see Proposition 3.16.(3)).
We define properties of translations w.r.t. the observational preorder.

Definition 3.6 *Adequacy and full abstraction of translations.* The following properties w.r.t. observational preorder $\leq_\tau$ are defined for a translation $T : \mathcal{C} \to \mathcal{C}'$: We say that T is

adequate if for all τ, and all $p_1, p_2 \in \mathcal{P}_\tau$: $T(p_1) \leq'_{T(\tau)} T(p_2) \implies p_1 \leq_\tau p_2$, i.e., if T is $\leq$-reflecting;

fully abstract if for all τ, and all $p_1, p_2 \in \mathcal{P}_\tau$: $p_1 \leq_\tau p_2 \iff T(p_1) \leq'_{T(\tau)} T(p_2)$, i.e., if T is $\leq$-preserving and $\leq$-reflecting;

an isomorphism if T is a bijection on the types, T is a bijection between $\mathcal{P}/\sim$ and $\mathcal{P}'/\sim'$, and if T is fully abstract.

A trivial example that satisfies all the properties above is again given by the identity translation $\mathcal{C} \to \mathcal{C}$.

As motivated in the introduction, we consider observational correctness as the fundamental requirement for correct translations. As explained below in Subsection 3.4, observational correctness captures the intuition that compiled tests applied to compiled programs have the same result as in the source language. However, the target language may have more testing capabilities, and usually has access to details of the compilation which are inaccessible from the source language. Fully abstract translations and isomorphisms are of course superior and important notions when expressivity of calculi is considered, but in general compilation from high-level typed languages into low-level (untyped, or less typed) languages is often not fully abstract. An interesting discussion about the difficulties and challenges of security in connection with compilation and the implications of adequacy and full abstraction is in [Ahmed and Blume 2008], which may open a third perspective on the properties of translations.

Theorems 3.13 and 3.14 exhibit invariant properties for observational correctness. Observational correctness is a sufficient criterion for adequacy (see Proposition 3.8). Full abstraction is not necessary for the adequacy of translations. But when full abstraction holds in addition to observational correctness then, for surjective translations that moreover preserve the type structure, it means that both program calculi are identical with respect to the observational preorder, i.e. they are isomorphic.

By straightforward arguments it can be shown that translations compose:

PROPOSITION 3.7 CLOSURE UNDER COMPOSITION. *Let $\mathcal{C}, \mathcal{C}', \mathcal{C}''$ be program calculi, and $T : \mathcal{C} \to \mathcal{C}'$, $T' : \mathcal{C}' \to \mathcal{C}''$ be translations. Then $T' \circ T : \mathcal{C} \to \mathcal{C}''$ is also a translation, and for every property P from Definitions 3.2 and 3.6, if T and T' have property P, then so has their composition $T' \circ T$.*

PROPOSITION 3.8. *If a translation $T : \mathcal{C} \to \mathcal{C}'$ is observationally correct, then T is also adequate.*

PROOF. 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 $\mathcal{G}(O(p_1))$, $\mathcal{G}(O(p_2))$ and $O(p_1)\Downarrow_i$. By the characterization of observational correctness in Lemma 3.4, this implies $T(O)(T(p_1))\Downarrow'_{T(i)}$, where $\mathcal{G}'(T(O)(T(p_1)))$ and $\mathcal{G}'(T(O)(T(p_2)))$ hold by the conditions of (oc). From $T(p_1) \leq_{T(\tau)} T(p_2)$, we obtain $T(O)(T(p_2))\Downarrow'_{T(i)}$, and observational correctness, using the other direction of the equivalence, implies $O(p_2)\Downarrow_i$. This proves $p_1 \leq_\tau p_2$. □

To conclude this section, let us emphasize that Definitions 3.1, 3.2 and 3.6 are stated only in terms of abstract OSP calculi, and hence they can be used for all calculi with such a description. In the case of two calculi with observations defined in terms of a small-step semantics, the definition of translation also allows for reduction sequences in the target that may lead outside of the image of the translation, i.e., the reduction sequences as such may not be retranslatable.

3.3 Examples

The following examples show some easy uses of translations:

Example 3.9. Let $\text{PCF}_{-\mu}$ be call-by-value PCF as in Example 2.3 with the following modification: fixed point operators are dropped, and let PCF_{cbv} be call-by-value PCF. The difference between $\text{PCF}_{-\mu}$ and $\text{PCF}_{-\mu,0}$ from Example 2.7 is that (**pred** 0) results in an error as in PCF. The embedding $\iota : \text{PCF}_{-\mu} \to \text{PCF}_{\text{cbv}}$ is defined as the identity on types and expressions, and hence ι is compositional and (ce), hence adequate by Proposition 3.8. Below in Example 4.10 we show that it is also fully abstract.

In call-by-name or call-by-need calculi changing the closedness predicate from trivial closedness to proper closedness is in general an isomorphism. We illustrate this for call-by-name PCF:

Example 3.10. In order to illustrate the possibility of varying the closedness predicate, let $\text{PCF}_{\text{cbn},1}$ be call-by-name PCF where $\mathcal{G}$ is always true, and where the closing observers are only the empty contexts of different types, and let $\text{PCF}_{\text{cbn},2}$ be call-by-name PCF where $\mathcal{G}$ is the test for closed expressions and closing contexts are composed of contexts of the form $\lambda x_1 \ldots x_n.[\cdot]\ v_1 \ldots v_n$ where v_i are closed values. Let $T : \text{PCF}_{\text{cbn},1} \to \text{PCF}_{\text{cbn},2}$ be the identity on all types, programs, and observers. Unfortunately, while T is compositional and $\mathcal{G}(p) \implies \big(p\Downarrow_i \iff T(p)\Downarrow_i\big)$ holds, T violates the condition cle, i.e., $\mathcal{G}(p) \iff \mathcal{G}'(T(p))$ might be wrong; this means that our tools cannot be used. Still, it can be shown (using methods specific to PCF) that the translation is fully abstract, and since T is a bijection on programs and types it is also an isomorphism.

Example 3.11. We give an example of an adequate, but not fully abstract, embedding: Let $\text{PCF}_{-\mu,0}$ be PCF as in Example 2.7 (see also Example 2.3) where fixed point operators are dropped and $(\textbf{pred}\ 0) = 0$ and let PCF_0 be call-by-value PCF with the modification that $(\textbf{pred}\ 0) = 0$. The embedding $\iota : \text{PCF}_{-\mu,0} \to \text{PCF}_0$ is the identity on types and expressions, and hence the embedding is compositional and (ce), hence adequate. However, the embedding is not fully abstract: The expressions p_1, p_2 with $p_1 = \lambda x.0$ and $p_2 = \lambda x.\texttt{if}\ (\textbf{zero?}\ (x\ 0))\ \texttt{then}\ 0\ \texttt{else}\ 0$ are observationally equivalent w.r.t $\text{PCF}_{-\mu,0}$, but are not observationally equivalent w.r.t. PCF_0: They can be distinguished by applying both to $\lambda y.\bot$, where $\bot$ is a non-converging PCF_0-expression. Another explanation for the failure of full abstraction is that η-equivalence (with respect to $\sim$) holds in $\text{PCF}_{-\mu,0}$, but not in PCF_0.

Finally, we compare call-by-name PCF and call-by-value PCF.

Example 3.12. Let PCF_{cbn} be call-by-name PCF and PCF_{cbv} be call-by-value PCF, where the closing observers are generated by contexts of the form $\lambda x_1 \dots x_n.[\cdot]\ v_1 \dots v_n$ where v_i are closed values. Let $T : \text{PCF}_{\text{cbn}} \to \text{PCF}_{\text{cbv}}$ be given by the identity on types, programs, and observers. Then T is compositional, but it is not (ce) since, for example, the expression $(\lambda x.0)\ \bot$ has different convergence behaviors in call-by-name and call-by-value PCF. Since $(\lambda x.0)\ \bot \sim 0$ in PCF_{cbn} but not in PCF_{cbv}, the translation is not fully abstract. In the converse direction, the expressions $\lambda x.0$ and $\lambda x.\big(\texttt{if}\ x\ \texttt{then}\ 0\ \texttt{else}\ 0\big)$ are equivalent in PCF_{cbv}, but not in PCF_{cbn}, hence the translation is also not adequate.

3.4 Minimal Correctness Requirement

The minimal sensible correctness requirement for a translation (seen as a compilation) is that convergence of programs and testing results are unchanged by a translation. This is exactly the observational correctness property. Proposition 3.8 shows that such a translation is adequate. A compilation of a high-level program into a rather low level programming language is usually not fully abstract, since it is possible to test the implemented program or function for implementation details, which is impossible in the high-level language. However, the observers that are images of high-level observers should make the same observations. Thus, only taking into account the observers that are image observers under the translation may help to establish a restricted form of full abstractness. In the next theorem we show that observing the translated programs using translated observers under type restrictions makes the same distinctions between the original and the translated programs if observational correctness holds.

THEOREM 3.13. *Let $\mathcal{C}, \mathcal{C}'$ be OSP-calculi and $T : \mathcal{C} \to \mathcal{C}'$ be an observationally correct translation. Let $\mathcal{C}'' := T(\mathcal{C})$ be the subcalculus of $\mathcal{C}'$ consisting of the object-, observer- and observation-images under T, and let $\leq_T$ be the preorder defined on the T-image $\mathcal{C}''$ where the type restrictions of $\mathcal{C}$ remain in effect:*
Let $p'_1 \leq_T p'_2$ hold if for all types τ, all p_1, p_2 of type τ and all observers $O \in \mathcal{O}_{\tau,\tau'}$: If $p'_1 = T(p_1)$, $p'_2 = T(p_2)$, $\mathcal{G}'(T(O)(p'_1))$, $\mathcal{G}'(T(O)(p'_2))$, and $T(O)(p'_1){\Downarrow}'_{T(i)}$, then also $T(O)(p'_2){\Downarrow}'_{T(i)}$. Then for all types τ and programs p_1, p_2: $p_1 \leq_\tau p_2 \iff T(p_1) \leq_T T(p_2)$.

PROOF. In the proof we omit reasoning about $\mathcal{G}$-conditions, since these are obvious. Let $p'_1 \leq_T p'_2$ hold, let τ be a type, p_1, p_2 be programs of type τ with $T(p_1) = p'_1, T(p_2) = p'_2$, and let $O \in \mathcal{O}_{\tau,\tau'}$ be an observer such that $\mathcal{G}'(T(O)(p'_1))$, $\mathcal{G}'(T(O)(p'_2))$, and $O(p_1)\Downarrow_i$. Observational correctness implies $T(O)(p'_1)\Downarrow'_{T(i)}$. The definition of $\leq_T$ shows that $T(O)(p'_2)\Downarrow'_{T(i)}$ also holds. Again by observational correctness, we obtain $O(p_2)\Downarrow_i$. Thus $p_1 \leq_\tau p_2$.

Let $p_1 \leq_\tau p_2$ and let O' be a $\mathcal{C}''$ observer such that $O' = T(O)$ for some $O \in \mathcal{O}_{\tau,\tau'}$: and let $\mathcal{G}'(O'(T(p_1)))$, $\mathcal{G}'(O'(T(p_2)))$, and $O'(T(p_1))\Downarrow'_{T(i)}$. We show that $O'(T(p_2))\Downarrow'_{T(i)}$. Observational correctness implies $\mathcal{G}(O(p_1))$ and $\mathcal{G}(O(p_2))$. Observational correctness and the characterization in Lemma 3.4 now shows $O'(T(p_1))\Downarrow'_{T(i)} \Leftrightarrow T(O)(T(p_1))\Downarrow'_{T(i)} \implies O(p_1)\Downarrow_i \implies O(p_2)\Downarrow_i$. This in turn implies $T(O)(T(p_2))\Downarrow'_{T(i)}$ which is equivalent to $O'(T(p_2))\Downarrow'_{T(i)}$. Since this holds for all observers O', we have shown $T(p_1) \leq''_{T(\tau)} T(p_2)$. □

As an improvement of the previous theorem we show that observational correctness of a translation provides an isomorphism as translation $T : \mathcal{C} \to T(\mathcal{C})$, if the translation on the types is injective. If T is surjective on programs and observers then T is an isomorphism, provided T is an isomorphism on the type structure.

THEOREM 3.14. *Let $\mathcal{C}, \mathcal{C}'$ be OSP-calculi and $T : \mathcal{C} \to \mathcal{C}'$ be an observationally correct translation such that T is injective on the types. Let $\mathcal{C}'' := T(\mathcal{C})$ be the subcalculus of $\mathcal{C}'$ consisting of the program-, observer- and observation-images under T, and let $\leq''$ be the preorder defined on the T-image $\mathcal{C}''$.*
Then for all types τ and programs p_1, p_2: $p_1 \leq_\tau p_2 \iff T(p_1) \leq''_{T(\tau)} T(p_2)$, i.e. the translation is fully abstract as translation $T : \mathcal{C} \to \mathcal{C}''$. Moreover, the translation T is also an isomorphism.

PROOF. We denote the observational preorder of $\mathcal{C}''$ with $\leq''$. Adequacy follows from Proposition 3.8 by applying it to the restricted translation $T : \mathcal{C} \to \mathcal{C}''$.

Let $p_1 \leq_\tau p_2$ and let O' be a $\mathcal{C}''$-observer such that $\mathcal{G}'(O'(T(p_1)))$, $\mathcal{G}'(O'(T(p_2)))$, and $O'(T(p_1))\Downarrow'_{T(i)}$. We show that $O'(T(p_2))\Downarrow'_{T(i)}$. Since T is surjective on $\mathcal{O}_{\tau,\tau'}$, there is an observer O with input type τ, such that $T(O) = O'$. Observational correctness implies $\mathcal{G}(O(p_1))$ and $\mathcal{G}(O(p_2))$. Observational correctness and the characterization in Lemma 3.4 now shows $O'(T(p_1))\Downarrow'_{T(i)} \Leftrightarrow T(O)(T(p_1))\Downarrow'_{T(i)} \implies O(p_1)\Downarrow_i \implies O(p_2)\Downarrow_i$. This in turn implies $T(O)(T(p_2))\Downarrow'_{T(i)}$ which is equivalent to $O'(T(p_2))\Downarrow'_{T(i)}$. Since this holds for all observers O', we have shown $T(p_1) \leq''_{T(\tau)} T(p_2)$. □

For many translations, being injective on types is too strict a requirement, and thus the previous theorem is not applicable. Nevertheless, if the translation is observationally correct then Theorem 3.13 is applicable. An example is:

Example 3.15. The classical Church-encoding of the untyped lambda-calculus with pairs and selectors into the pure untyped lambda calculus is neither convergence equivalent nor adequate. An observationally correct (and thus also adequate) encoding can be established if the lambda calculus with pairs is simply typed and the lambda calculus without pairs is untyped. In Section 5.1 we explain why the usual encoding fails without typing, and prove observational correctness and thus adequacy of the same encoding under typing restrictions.

3.5 Independence of Properties of Translations

As the following proposition shows, (ce) is in general not sufficient for adequacy, and full abstraction is not implied by observational correctness. Similarly, (ce) is not even implied by full abstraction (and thus neither by adequacy).

Proposition 3.16. *The following holds:*

(1) *Convergence equivalence does not imply adequacy.*

(2) *Observational correctness does not imply full abstraction.*

(3) *Convergence equivalence is not implied by the conjunction of compositionality up to observations and preservation and reflection of $\leq$.*

(4) *Convergence equivalence and preservation and reflection of $\leq$ do not imply observational correctness.*

Proof. (1) Let the OSP-calculus $\mathcal{C}$ have three programs: a, b, c with $a \Uparrow$, $b \Downarrow$ and $c \Downarrow$. Assume there are two observers O_1, O_2 with $O_1 = Id$ and $O_2(a) = a, O_2(b) = a, O_2(c) = c$. Then $b \not\sim_{\mathcal{C}} c$. The language $\mathcal{C}'$ has three programs A, B, C with $A \Uparrow$, $B \Downarrow$ and $C \Downarrow$. There is only the identity observer Id' in $\mathcal{C}'$. Then $B \sim_{\mathcal{C}'} C$. Let the translation be defined as $T : \mathcal{C} \to \mathcal{C}'$ with $T(a) = A, T(b) = B, T(c) = C$, and $T(O_1) = T(O_2) = Id'$. Then convergence equivalence holds, but neither adequacy nor observational correctness. The translation T is not (cuo), since $T(O_2(b)) = A$ and thus $\neg T(O_2(b)) \Downarrow$ while $T(O_2)(T(b)) = Id(B) = B$ and thus $T(O_2)(T(b)) \Downarrow$.

(2) Example 3.11 is a witness for this. Another simple example taken from [Mitchell 1993] is the identity encoding into (call-by-name) PCF with product types from PCF but without the projections **fst** and **snd**. Then, in the restricted calculus, all pairs are indistinguishable but the presence of the observers **fst** $[\cdot]$ and **snd** $[\cdot]$ in PCF with products permits more distinctions to be made.

(3) A trivial example is given by two calculi $\mathcal{C}$ with $p\Downarrow$ for all p, and $\mathcal{C}'$ with the same programs and $\neg p \Downarrow'$ for all p. For the identity 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, and also that the translation is compositional up to observations, but clearly T does not preserve convergence.

(4) Let $\mathcal{C}$ have two programs a, b, the identity observer, and one observer O with $O(a) = b$ and $a\Downarrow$, $b \Uparrow$. Let $\mathcal{C}'$ consists of A, B with $A\Downarrow$, and $B \Uparrow$, and the identity observer. Let $T : \mathcal{C} \to \mathcal{C}'$ be the translation defined by $T(a) = A, T(b) = B$, and $T(O) = Id'$. Then the translation is $\leq$-preserving and reflecting, since there are no relevant equalities. It is also convergence equivalent. But it is not observationally correct, since $T(O(a)) = T(b) = B$, i.e. $T(O(a)) \Uparrow$, and $T(O)(T(a)) = A$, i.e. $T(O)(T(A))\Downarrow$. $\square$

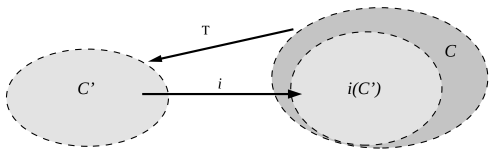
4. LANGUAGE EXTENSIONS

We now consider extensions of languages or, taking a slightly more general point of view, embeddings of one language into another. A typical case is that new language primitives are added to a calculus, together with their (operational) semantics, which are then encoded by the translation. There are two issues: One is whether the extension is conservative, i.e. whether the embedding of the non-extended language into the extended one is fully abstract. The second issue is whether the extension is ‘syntactic sugar’, i.e. whether the extended language can be translated into the base language in a fully abstract way.

Definition 4.1 *Extension and embedding.* An OSP-calculus $\mathcal{C}$ is an *extension* of the OSP-calculus $\mathcal{C}'$ iff there is a translation $\iota : \mathcal{C}' \to \mathcal{C}$ which is observationally correct, i.e. (ce) and (cuo), and which is injective on the types. In this case the translation ι is called an *embedding.*

Note that an embedding ι is adequate by Proposition 3.8, i.e. injective modulo $\sim$, but not necessarily fully abstract.

Informally, the embedding situation can be described (after identifying $\mathcal{C}'$-programs with their image under ι and modulo $\sim$) as follows: every $\mathcal{C}'$-type is also a $\mathcal{C}$-type, $\mathcal{P}'_\tau \subseteq \mathcal{P}_\tau$, and $\mathcal{O}'_{\tau,\tau'}$ can be seen as a subset of $\mathcal{O}_{\tau,\tau'}$, more precisely: $\mathcal{O}'_{\tau,\tau'}$ is the same as $\{O\lceil_{\iota(\mathcal{P}')} \mid O \in \mathcal{O}_{\tau,\tau'}\}$, (where $f\lceil_A$ is the function f restricted to the set A) and the observation predicates coincide on $\mathcal{C}'$-programs.



If $\mathcal{C}$ is an extension of $\mathcal{C}'$, then an observationally correct translation $T : \mathcal{C} \to \mathcal{C}'$ (plus some conditions) has the nice consequence of T and ι being fully abstract (see Proposition 4.3 and Corollary 4.4).

An example for an embedding is the trivial embedding $inc : \Lambda \to \Lambda_p$ of the untyped lambda calculus Λ into the lambda calculus Λ_p extended with pairing and projections. The embedding allows us to reason about contextual equivalence in Λ_p and transfer this result to Λ, i.e. a proof of $t_1 \sim_{\Lambda_p} t_2$ where t_1, t_2 are expressions that do not contain pairing or projections directly shows that $t_1 \sim_\Lambda t_2$. Disproving an equivalence in the lambda calculus with pairing and projections, however, does *not* imply that this equivalence is false in Λ (see subsection 5.1).

Definition 4.2. For an OSP-calculus $(\mathcal{T}, \mathcal{P}, \mathcal{O}, \mathcal{OBS}, \mathcal{G}, \mathcal{CO})$ and two observers $O_A, O_B \in \mathcal{O}_{\tau_1,\tau_2}$ and a set $M \subseteq \mathcal{P}_{\tau_1}$ of programs of type τ_1, we write $O_A \approx_M O_B$ iff for all $p \in M$ and all i: $\mathcal{G}(O_A(p)) \Leftrightarrow \mathcal{G}(O_B(p))$ holds and $\mathcal{G}(O_A(p))$ implies that $O_A(p)\Downarrow_i \Leftrightarrow O_B(p)\Downarrow_i$ holds.

PROPOSITION 4.3 FULL ABSTRACTION FOR EXTENSIONS. *Let $\mathcal{C}$ be an extension of $\mathcal{C}'$, i.e. an embedding (an (oc)-translation) $\iota : \mathcal{C}' \to \mathcal{C}$ and let $T : \mathcal{C} \to \mathcal{C}'$ be an observationally correct translation such that $T \circ \iota$ is the identity on $\mathcal{OBS}$ and on $\mathcal{C}'$-types, and $(T \circ \iota)(p) \sim p$ for all $\mathcal{C}'$-programs p. Then ι is fully abstract and T is adequate.*
If the following “surjectivity” condition holds:
(†) *For all $\mathcal{C}$-types τ_1, τ_2 and $O' \in \mathcal{O}'_{T(\tau_1),T(\tau_2)}$, and every set $M \subseteq T(\mathcal{P}_{\tau_1})$ of programs with $|M| \leq 2$, there is an observer $O \in \mathcal{O}_{\tau_1,\tau_2}$ with $T(O) \approx_M O'$*
then T is also fully abstract.

PROOF. Note that the conditions imply that ι is injective on the types and on observations, and that T is surjective on types and on observations.

Adequacy of ι and T follows from Proposition 3.8.
First we show full abstraction of ι. Let p_1, p_2 be $\mathcal{C}'$ programs of type τ, let $p_1 \leq_\tau p_2$ and let O be a $\mathcal{C}$-observer of the right type such that $\mathcal{G}(O(\iota(p_1)))$, $\mathcal{G}(O(\iota(p_2)))$ and $O(\iota(p_1))\Downarrow_i$. We must show that $O(\iota(p_2))\Downarrow_i$. We can apply T and obtain, by observational correctness, that $\mathcal{G}'(T(O)(T(\iota(p_1))))$ and $T(O)(T(\iota(p_1)))\Downarrow'_{T(i)}$. We want to use $T(\iota(p_1)) \sim p_1$, but $\mathcal{G}'(T(O)(p_1))$ may be wrong. So let $D_1 \in \mathcal{CO}'$ be an (existing) observer, such that $\mathcal{G}'(D_1(T(O)(p_1)))$. For the same D_1 we obtain also that $\mathcal{G}'(D_1(T(O)(T(\iota(p_1)))))$ and $(D_1(T(O)(T(\iota(p_1)))))\Downarrow'_{T(i)}$, which implies $D_1(T(O)(p_1))\Downarrow'_{T(i)}$ by Lemma 2.11. From $p_1 \leq p_2$ we again obtain a $D_2 \in \mathcal{CO}'$ such that $\mathcal{G}'((D_2 \circ D_1 \circ T(O))(p_2))$ and from $(D_2 \circ D_1 \circ T(O))(p_1)\Downarrow'_{T(i)}$ we derive $(D_2 \circ D_1 \circ T(O))(p_2)\Downarrow'_{T(i)}$. The assumptions on translations and $\mathcal{G}(O(\iota(p_2)))$ imply that $\mathcal{G}'(T(O)(T(\iota(p_2))))$. The equation $T(\iota(p_2)) \sim p_2$ implies $(D_2 \circ D_1 \circ T(O))(T(\iota(p_2)))\Downarrow'_{T(i)}$ and by the assumptions on D_i this also implies $T(O)(T(\iota(p_2)))\Downarrow'_{T(i)}$. By observational correctness, we obtain $O(\iota(p_2))\Downarrow_i$.

It remains to show that T is fully abstract under the condition (†) above. Let p_1, p_2 be $\mathcal{C}$-programs of type τ, and assume $p_1 \leq_{\Downarrow_i,\tau} p_2$. We have to prove that $T(p_1) \leq'_{\Downarrow'_{T(i)},T(\tau)} T(p_2)$. Let O' be a $\mathcal{C}'$-observer such that $\mathcal{G}'(O'(T(p_1)))$, $\mathcal{G}'(O'(T(p_2)))$ and $O'(T(p_1))\Downarrow'_{T(i)}$. Let O be the existing $\mathcal{C}$-observer for the set $M := \{T(p_1), T(p_2)\}$ due to the condition (†) with $T(O) \approx_M O'$ and such that $O(p_1)$ is defined. The condition $O' \approx_M T(O)$ implies $\mathcal{G}'(T(O)(T(p_1)))$. Observational correctness of T using (†) and $T(O)(T(p_1))\Downarrow'_{T(i)}$ implies $O(p_1)\Downarrow_i$. Moreover $\mathcal{G}(O(p_1))$ holds since T is an observationally correct translation. Using $\mathcal{G}'(O'(T(p_2)))$ and $O' \approx_M T(O)$ we see that $\mathcal{G}'(T(O)(T(p_2)))$. Since T is (cuo), this implies $\mathcal{G}(O(p_2))$. From $p_1 \leq_{\Downarrow_i,\tau} p_2$ we now derive $O(p_2)\Downarrow_i$. Again, observational correctness of T can be applied and shows that $T(O)(T(p_2))\Downarrow'_{T(i)}$. This is equivalent to $O'(T(p_2))\Downarrow'_{T(i)}$, again using $O' \approx_M T(O)$. Since the observer $O' \in \mathcal{O}'_{\tau,\tau'}$ was chosen arbitrarily, we have $T(p_1) \leq'_{\Downarrow_i,T(\tau)} T(p_2)$. $\square$

Injectivity of T on types is a special case of the condition in Proposition 4.3:

Corollary 4.4 Full Abstraction for Extensions; injectivity. *If in Proposition 4.3 the condition* (†) *is replaced by: T is injective (i.e. bijective) on types, then all claims of Proposition 4.3 hold.*

Proof. Assume injectivity of T on types. Given τ_1, τ_2 and $O' \in \mathcal{O}_{T(\tau_1),T(\tau_2)}$, we define $O := \iota(O')$ and have to show that $O' \approx_{T(\mathcal{P}_{\tau_1})} T(\iota(O'))$. The precondition $(T \circ \iota)(p_1) \sim p_1$ for all type-correct $\mathcal{C}'$-programs p_1 and observational correctness of T and ι show that for all $p \in T(\mathcal{P}_{\tau_1})$: $\mathcal{G}(O'(p)) \Leftrightarrow \mathcal{G}(T(\iota(O'))(p))$ and that for all $p \in T(\mathcal{P}_{\tau_1})$: $\mathcal{G}(O'(p))$ implies $O'(p)\Downarrow_i \Leftrightarrow T(\iota(O'))(p)\Downarrow_i$.
The condition $O' \sim T(\iota(O'))$ holds by the precondition that $(T \circ \iota)(p) \sim p$ for all $\mathcal{C}'$-programs p. If $\mathcal{G}(p)$ holds, then by convergence equivalence of $T \circ \iota$, this is equivalent to $\mathcal{G}((T \circ \iota)(p))$, hence the condition holds. $\square$

Example 4.5. In general, Proposition 4.3 and Corollary 4.4 will not hold without assumption (†) or the assumption that T is injective on types, respectively. To see this, let $\mathcal{C}'$ be the OSP-calculus with one type A, four programs a_1, a_2, a_3, a_4 of type A, the identity as well as an observer $f \in \mathcal{O}_{A,A}$ with $f(a_1) = f(a_3) = a_3$, $f(a_2) = f(a_4) = a_4$, and $a_1\Downarrow$, $a_2\Downarrow$, $a_3\Downarrow$, but $\neg a_4\Downarrow$. Thus, $a_1 \not\sim a_2$.

Let $\mathcal{C}$ be an extension with additional type B and programs b_1, b_2 of type B, with only the identity observer, and such that $b_1\Downarrow$, $b_2\Downarrow$. Hence $b_1 \sim b_2$. Let $T : \mathcal{C} \to \mathcal{C}'$ be defined by:

$$\begin{array}{lcl} T(A) & = & T(B) \ = \ A \\ T(f) & = & f \\ T(a_i) & = & a_i \\ T(b_1) & = & a_1 \\ T(b_2) & = & a_2 \end{array}$$

Note that T is not injective on the types, since $T(A) = T(B) = A$.

Then T is compositional and convergence equivalent, hence also observationally correct. Moreover the embedding $\iota : \mathcal{C}' \to \mathcal{C}$ satisfies that $T \circ \iota$ is the identity on $\mathcal{C}'$. But T is not fully abstract, since $b_1 \sim b_2$, but $T(b_1) = a_1$ and $T(b_2) = a_2$, and $a_1 \not\sim a_2$. Thus, we cannot omit the injectivity assumption in Corollary 4.4.
Our example also does not satisfy the condition (†) of Proposition 4.3, since for the type B, elements a_1, a_2 and observer $f \in \mathcal{O}_{A,A}$, there is no observer $O \in \mathcal{O}_{B,B}$, such that $T(O) \approx_{a_2} f$, since $T(O)$ can only be Id, and $a_2\Downarrow$, but $\neg f(a_2)\Downarrow$. Hence, we cannot omit the condition (†).

Example 4.6. Let $\mathcal{C}'$ be call-by-value PCF (see Example 2.3) and let $\mathcal{C}$ be call-by-value PCF extended by strict `let`-expressions of the form (`let` $x = e_1$ `in` e_2). The reduction of $\mathcal{C}$ extends $\mathcal{C}'$-reduction by reducing first inside the binding of `let`-expressions and then applying the rule (`let` $x = v$ `in` e_2) $\to e_2[v/x]$ if v is a value. A translation $T : \mathcal{C} \to \mathcal{C}'$ which removes the `let`-expressions can be defined as follows: $T($`let` $x = e_1$ `in` $e_2) := (\lambda x.T(e_2))\ T(e_1)$ and for all other cases T translates the expressions homomorphically with respect to the term structure. T is the identity on types (and hence also injective on types) and can be extended to contexts in the obvious way. The embedding $\iota : \mathcal{C}' \to \mathcal{C}$ is the identity on types, expressions and contexts. Obviously, $T \circ \iota$ is the identity on $\mathcal{C}'$-expressions. Both translations T, ι are compositional and also convergence equivalent, since `let`-reductions exactly correspond to call-by-value beta-reductions and reductions inside `let`-bindings exactly correspond to reductions inside arguments of applications. Since the first conditions of Proposition 4.3 hold and since T is injective on types, we can apply Corollary 4.4 and conclude that T and ι are fully abstract.

Example 4.7. We give an example of an application of Proposition 4.3 using a slightly unusual contextual semantics for PCF. Let $\mathcal{C}'$ be call-by-name PCF where the observations are only convergence of Boolean expressions, i.e., $s \leq t$ iff for all contexts C of Boolean type: if $C[s], C[t]$ are closed, then $C[s]\Downarrow$ implies $C[t]\Downarrow$. Let $\mathcal{C}$ be an extension of $\mathcal{C}'$ with n-ary functions, i.e., there are also n-ary function types $(\tau_1, \ldots, \tau_n) \to \tau$, n-ary lambda-expressions, written as $\lambda(x_1, \ldots, x_n).t$, and n-ary applications $t\ (t_1, \ldots, t_n)$. Lambda-reduction in $\mathcal{C}$ is permitted as $(\lambda(x_1, \ldots, x_n).t)\ (t_1, \ldots, t_n) \to t[t_1/x_1, \ldots t_n/x_n]$. We assume that there are no explicit tuples and no variables of a tuple type. Similar as above, we only observe convergence of Boolean expressions.

It is not hard to see that the η-axiom holds for all expressions of function type. That is, for $t : \tau_1 \to \tau_2$ and x not free in t, we have $t \sim \lambda x.(t\ x)$. Correspondingly, in the case $t : (\tau_1, \ldots, \tau_n) \to \tau$, the equivalence $t \sim \lambda(x_1, \ldots, x_n).(t\ (x_1, \ldots, x_n))$

holds for fresh $x_1, \dots, x_n$.
The embedding $\iota : \mathcal{C}' \to \mathcal{C}$ is defined as the identity on types, expressions and observers, and the translation T translates types $(\tau_1, \dots, \tau_n) \to \tau$ to $\tau_1 \to \dots \tau_n \to \tau$, $T(\lambda(x_1, \dots, x_n).t) = \lambda x_1. \dots \lambda x_n.T(t)$, $T(t(t_1, \dots, t_n)) = (((T(t)\, T(t_1)) \dots)\, T(t_n))$, and all other constructs homomorphically with respect to the term structure. The following properties hold: $T \circ \iota$ is the identity, the embedding ι is compositional and also (ce). The translation T is also compositional, since there are no special syntactic conditions. The translation is also (ce), since reductions $s_1 \xrightarrow{*} s_2$ for closed s_1 can be translated as $T(s_1) \xrightarrow{*} T(s_2)$. We argue that the condition (†) of Proposition 4.3 holds. The main argument is that η holds, so that for given $\mathcal{C}$-types τ_1, τ_2, finite set M of programs, and an observer $O' \in \mathcal{O}'_{T(\tau_1),T(\tau_2)}$, an observer $O \in \mathcal{P}_{\tau_1,\tau_2}$ with $T(O) \approx_M O'$ can be found: for this (inductive) construction of O, eta-long normal forms are used. Since the cardinality of the set M that has to be covered is at most two, it is always possible to find fresh variable names when the eta-rule has to be applied to a context where the hole is in the scope of the fresh variable.

Remark 4.8. In contrast to the previous example, Proposition 4.3 cannot be applied to call-by-value PCF since the condition (†) does not hold. It is sufficient to show that T is not surjective on the programs of a fixed type: Let $\tau := ((N \to N \to N), N, N) \to N$ and consider the "partial application" $p_1 := \lambda x_1.\lambda x_2.(x_1\ x_2)$ of type $T(\tau) = ((N \to N \to N) \to N \to N \to N$. Then there is no p of type $((N \to N \to N), N, N) \to N$ such that $T(p) \sim p_1$. For assume otherwise, then obviously $T(p)$ cannot be $\bot$. Hence p converges and we can assume that p is a lambda-expression in the extension: $\lambda(y_1, y_2, y_3).s$. Then $T(\lambda(y_1, y_2, y_3).s)\ t_1\ t_2 = (\lambda y_1.\lambda y_2.\lambda y_3.T(s))\ t_1\ t_2$ always converges. However, $p_1\ (\lambda x.\bot)\ 0$ diverges, hence p_1 is not an image of an expression of type τ under T. Note that the key to this counterexample is the failure of (η) in call-by-value PCF with respect to observational equivalence.

Now we provide a criterion for full abstractness of the embedding in the case where the extended calculus cannot be translated into the base calculus. The main idea is to use a *family* of translations as an approximation of a translation that cannot be represented. For example, if the translation has to deal with recursive programs (or types), then it may be possible to consider (all) the finitary approximations instead.

Proposition 4.9 Families of Translations. *Let $\mathcal{C}$ be an extension of $\mathcal{C}'$, i.e. with an observationally correct translation $\iota : \mathcal{C}' \to \mathcal{C}$. Let J be an partially ordered index set, such that for j_1, j_2, there is some $j_3 \in J$ with $j_1 \leq j_3$ and $j_2 \leq j_3$. Let $\{T_j\}_j$ be a J-indexed family of translations $T_j : \mathcal{C} \to \mathcal{C}', j \in J$ such that the following conditions hold:*

—*For all $j \in J$: $T_j \circ \iota$ is the identity on $\mathcal{OBS}$ and on $\mathcal{C}'$-types, and*

—*for all $j \in J$: $(T_j \circ \iota)(p) \sim p$ for all $\mathcal{C}'$-programs p;*

—*for every observer $O \in \mathcal{O}_{\tau,\tau'}$ and every program $p \in \mathcal{P}_\tau$ with $\mathcal{G}(O(p))$, there is some $k \in J$, such that for all $j \in J$ with $j \geq k$: $\mathcal{G}'(T_j(O)(T_j(p)))$ and in addition for all i: $O(p){\Downarrow}_i \Leftrightarrow T_j(O)(T_j((p))){\Downarrow}'_{T_j(i)}$.*

Then ι is fully abstract.

Proof. It is obvious that ι is adequate.
In order to show full abstraction of ι, let p'_1, p'_2 be programs of type τ such that $p'_1 \leq'_{\Downarrow'_i,\tau} p'_2$ and let O be an arbitrary $\mathcal{C}$-observer such that $\mathcal{G}(O(\iota(p'_1)))$, $\mathcal{G}(O(\iota(p'_2)))$, and $O(\iota(p'_1))\Downarrow_{\iota(i)}$, but $O(\iota(p'_2)) \Uparrow_{\iota(i)}$. Then there is some k_1 such that for all $j_1 \geq k_1$: $\mathcal{G}'(T_{j_1}(O)(T_{j_1}(\iota(p'_1))))$ and $T_{j_1}(O)(T_{j_1}(\iota(p'_1)))\Downarrow'_i$. There is also some k_2 such that for all $j_2 \geq k_2$: $\mathcal{G}'(T_{j_2}(O)(T_{j_2}(\iota(p'_2))))$ and but $T_{j_2}(O)(T_{j_2}(\iota(p'_2))) \Uparrow'_i$. Due to the condition on the order, there is some common $j_3 \in J$ with $j_1 \leq j_3$, $j_2 \leq j_3$ such that for all $j \geq j_3$: $\mathcal{G}'(T_j(O)(T_j(\iota(p'_1))))$, $\mathcal{G}'(T_j(O)(T_j(\iota(p'_2))))$, and $T_j(O)(T_j(\iota(p'_1)))\Downarrow'_i$, but $T_j(O)(T_j(\iota(p'_2))) \Uparrow'_i$. We use the conditions above and Lemma 2.11. In a similar way as in the proof of Proposition 4.3 we can show that $\mathcal{G}'(D \circ T_j(O)(p'_1))$ and $\mathcal{G}'(D \circ T_j(O)(p'_2))$ for some appropriate D, and that $D \circ T_j(O)(p'_1)\Downarrow'_i$ and $D \circ T_j(O)(p'_2) \Uparrow'_i$, which contradicts the assumption. □

Example 4.10. Using call-by-value PCF we illustrate Proposition 4.9 to show full abstractness of embeddings of restricted PCF into call-by-value PCF: As in Example 3.9 let $\iota : \text{PCF}_{-\mu} \to \text{PCF}$ be the identity on types and expressions in the two PCF-variants. The embedding ι is compositional and (ce), and hence adequate. While there appears to be no (recursion-eliminating) translation $T : \text{PCF} \to \text{PCF}_{-\mu}$ as required for Proposition 4.3, it is possible to find a family of translations T_j as in Proposition 4.9: Let T_j be the translation that unfolds every occurrence of a fixed point operator j times and then replaces all further occurrences by $\bot$. Using induction on the length of reductions it can be shown that the condition of Proposition 4.9 holds, hence ι is fully abstract.

The following example shows that Proposition 4.3 may be violated if T does not satisfy the precondition for full abstraction. (The translation T in the example is not injective on types.) This counter example is somewhat unfortunate: it highlights the fact that the corollary cannot be applied to show full abstraction when the translation is given by an encoding of an abstract data type (such as products or lists in the lambda calculus) in terms of an implementation type in a subcalculus.

Example 4.11. Assume, we have extended call-by-value PCF with the data types `List` and `Set` (over numbers N), called $\text{PCF}_{\texttt{List},\texttt{Set}}$, and the constructors `Cons` and `Nil` for lists. We use the notation $[a_1, \dots, a_m]$ for a list with the n elements $a_i, i = 1, \dots, n$, and the selectors `head` and `tail`. In order to generate sets we assume a function `listToSet`, as well as functions `elem`, `union`, `intersection`, and `setEqual` operating on sets. For example `setEqual`(`listToSet`$[1,2]$), (`listToSet`$[2,1]$)) should result in `True`. Now we assume that there is an implementation of sets as lists, written as a translation $T : \text{PCF}_{\texttt{List},\texttt{Set}} \to \text{PCF}_{\texttt{List},\texttt{Set}}$. We can assume that T is observationally correct, and hence adequate.
Now we focus on the question whether T is fully abstract. The data type `Set` and the implementation T should make sense, i.e. we expect that `listToSet`$[1,2] \sim_{\texttt{Set}}$ `listToSet`$[2,1]$ holds. This enforces that T(`listToSet`$[1,2]$) and T(`listToSet`$[2,1]$) result in the same list, for otherwise there is a context that can distinguish the expressions `listToSet`$[1,2]$ and `listToSet`$[2,1]$. In order to obtain full abstractness, we could try to apply Corollary 4.4 or Proposition 4.3.

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

Fig. 1. Syntax of λ_{pair}

Corollary 4.4 can not be applied, since T is not injective on the types. The precondition of Proposition 4.3 may be valid, since only programs in the T-image are required as set of comparing list-observers and set-observers. We conjecture that full abstractness of T holds, but this proof may be not obvious.

Further examples for the application, successes and problems can be found in [Schwinghammer et al. 2009], where two synchronization primitives, buffers and so-called handled futures, are compared in a concurrent higher-order language: by extending a simply typed concurrent call-by-value calculus L to languages L_b, L_h, and L_{bh} (where L_{bh} is the language containing both primitives), the translations are analyzed by describing the embeddings and the encodings of one primitive by the other one. Our technique helps to show almost all the translation to be fully abstract. The translation that encodes buffers by handles could be shown adequate, and is conjectured to be fully abstract. However, as in the above Example 4.11, the buffer type was encoded into another type so that the translation was not injective on types, and this prevented us from applying the extension theorem.

5. DISCUSSION OF EXTENDED EXAMPLES

In this section we discuss two extended examples. First we analyze the Church encoding of pairs and selectors in the call-by-value lambda calculus. We show that the encoding becomes observationally correct if the calculus with pairs is restricted to simple typing while the target calculus is untyped. In the second example we inspect the de Bruijn encoding of lambda expressions and show that also for this encoding observational correctness holds.

5.1 Church's Encoding of Pairs in the Call-by-Value Lambda Calculus

As a larger example in this section, we recall the call-by-value lambda calculus with a fixed point operator and present its observational semantics on the basis of convergence. We illustrate that Church's encoding of pairs is observationally correct under typing restrictions and show why Church's encoding of pairs fails to be observationally correct in the untyped case.

5.1.1 *Languages.* The calculus λ_{pair} is the usual call-by-value lambda calculus extended by a call-by-value fixed point operator **fix** 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_{pair} and values Val_{pair} is shown in Fig. 1. Note that only values are syntactically permitted as components of a pair. The subcalculus λ_{cbv} is the calculus without pairs and selectors and will be used as target language. We use Exp_{cbv} (Val_{cbv}, resp.) for the set of λ_{cbv}-expressions (λ_{cbv}-values, resp.).

For both calculi we require call-by-value evaluation contexts $\mathbb{E}$ which are intro-

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

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

(β-CBV)	$\mathbb{E}[(\lambda x.t)\ w] \to \mathbb{E}[t[w/x]]$
(FIX)	$\mathbb{E}[\mathbf{fix}\ \lambda x.t] \to \mathbb{E}[t[(\lambda y.(\mathbf{fix}\ \lambda x.t)y)/x]]$
(SEL-F)	$\mathbb{E}[\mathbf{fst}\,(w_1,w_2)] \to \mathbb{E}[w_1]$
(SEL-S)	$\mathbb{E}[\mathbf{snd}\,(w_1,w_2)] \to \mathbb{E}[w_2]$

Fig. 3. Small-Step Reduction

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

Fig. 4. Translation of λ_{pair} into λ_{cbv}

duced in Fig. 2. With $s_1[s_2/x]$ we denote the capture-free replacement 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 $\to_{pair}$ of λ_{pair} is the union of all four rules, and small step reduction $\to_{cbv}$ of λ_{cbv} is the union of the first two rules. We assume that reduction preserves the distinct variable convention by implicitly performing α-renaming if necessary.

Convergence $\downarrow_{pair}$ in λ_{pair} is defined as $e{\downarrow_{pair}}$ iff $\exists v \in Val_{pair} : e \xrightarrow{*}_{pair} v$, and for λ_{cbv} convergence $\downarrow_{cbv}$ is defined accordingly as $e{\downarrow_{cbv}}$ iff $\exists v \in Val_{cbv} : e \xrightarrow{*}_{cbv} v$. Observers $\mathcal{O}$ are all contexts of the respective calculus. For the predicate $\mathcal{G}$ we use the closedness of expressions, and the closing observers $\mathcal{CO}$ are the contexts of the form $(\lambda x_1, \ldots x_n.[\cdot])\ v_1 \ldots v_n$, where v_i are closed values and their compositions. Obviously, λ_{pair} and λ_{cbv} are OSP-calculi. For the contextual preorders and equivalences for both calculi we index the relations with *pair* or *cbv*.

As a notation we use $\uparrow_{pair}$ and $\uparrow_{cbv}$ for the divergence predicates, i.e. for $e \in \Lambda_i$: $e{\uparrow_i} \iff \neg(e{\downarrow_i})$ for $i \in \{pair, cbv\}$.

5.1.2 *Removing Pairs.* We will mainly investigate the translation *enc* of λ_{pair} into λ_{cbv} as defined in Fig. 4 under different restrictions. The translation performs the classical removal of pairs as given by Church.

Note that conversely, it is trivial to encode λ_{cbv} into λ_{pair} via the identity translation $inc(s) = s$.

Since abstractions are translated into abstractions and pairs and selectors are translated into abstractions, obviously the following holds:

LEMMA 5.1. *For all $s \in \lambda_{pair}$: s is a λ_{pair}-value iff $enc(s)$ is a λ_{cbv}-value.*

We are able to show that convergence is preserved by the translation, i.e.

LEMMA 5.2. *Let $t \in \lambda_{pair}$ with $t{\downarrow_{pair}}$, then $enc(t){\downarrow_{cbv}}$.*

PROOF. Let $t_0 \in \lambda_{pair}$ with $t{\downarrow_{pair}}$, so $t_0 \to_{pair} t_1 \to_{pair} \cdots \to_{pair} t_n$ where t_n is a value. We show by induction on n that $enc(t_0){\downarrow_{cbv}}$. If $n = 0$ then t_0 is a value and $enc(t_0)$ must be a value, too, by Lemma 5.1. For the induction step we assume the

(.,.)	::	$\forall\alpha,\beta.\alpha \to \beta \to (\alpha,\beta)$			
fst	::	$\forall\alpha,\beta.(\alpha,\beta) \to \alpha$	**unit**	::	unit
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 schemes for constants in λ^T_{pair}

induction hypothesis $enc(t_1)\downarrow_{cbv}$. Hence, it suffices to show $enc(t_0) \xrightarrow{*}_{cbv} enc(t_1)$. If $t_0 \to_{pair} t_1$ is a (β-CBV) or (FIX) reduction, then the same reduction can be used in λ_{cbv}, and $enc(t_0) \to_{cbv} enc(t_1)$. If $t_0 \to_{pair} t_1$ by (SEL-F) or (SEL-S), then three (β-CBV) steps are necessary in λ_{cbv}, i.e., $enc(t_0) \xrightarrow{3}_{cbv} enc(t_1)$. $\square$

Nevertheless, we cannot prove reflection of convergence, since the following counter example shows that the implementation of pairs is not correct in the untyped setting.

Example 5.3. Let $t := \mathbf{fst}(\lambda z.z)$. Then $t\uparrow_{pair}$, since t is irreducible and not a value. However, the translation $enc(t)$ results in the expression $t' := (\lambda p.p\ (\lambda x.\lambda y.x))\ (\lambda z.z)$, which reduces by some (β-CBV)-reductions to $\lambda x.\lambda y.x$, hence $enc(t)\downarrow_{cbv}$. This is clearly not a correct translation, since it removes an error. Therefore, the observations are not preserved by this translation. This example also shows that enc is not adequate, since it invalidates the implication $T(p_1) \leq_{cbv} T(p_2) \implies p_1 \leq_{pair} p_2$, since $enc(t') = t'$, and hence $enc(t') = t' \leq_{cbv} t' = enc(t)$, but $t' \not\leq_{pair} t$ by the arguments above.

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.

One solution to prevent the counter example 5.3 is to consider a simply typed variant λ^T_{pair} of λ_{pair} 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_{pair}, where we assume a hole $[\cdot]_\tau$ for every type τ. For typing, we treat pairs, projections, the unit value, and the operator **fix** as a family of constants with the type schemes given in Fig. 5. Type safety can be stated by a preservation theorem for all expressions and a progress theorem for closed expressions. The condition of OSP-calculi in Definition 2.8 can easily be satisfied by defining $\mathcal{CO}$ to be the composition closure of the contexts of the form $(\lambda x_1,\dots x_n,[\cdot])\ v_1 \dots v_n$, where v are closed values. Note that for every type there is a closed value. Now it is easy to prove adequacy via observational correctness of the translations.

PROPOSITION 5.4. *For λ^T_{pair}, the (correspondingly restricted) translation enc : $\lambda^T_{pair} \to \lambda_{cbv}$ is compositional and convergence equivalent, and hence observationally correct and adequate.*

PROOF. Compositionality follows from the definition of enc (see Fig. 4). Lemma 5.1 also holds if enc is restricted to λ^T_{pair}. We show convergence equivalence:

(1) $t\downarrow_{pair} \implies enc(t)\downarrow_{cbv}$: Follows from Lemma 5.2.

(2) $enc(t){\downarrow_{cbv}} \implies t{\downarrow_{pair}}$: An inspection of the reductions shows that if the λ^T_{pair}-expression t_1 is reducible, then for every reduction *Red* of $enc(t_1)$ to a value, there is some t_2 with $t_1 \to_{pair} t_2$ and $enc(t_1) \xrightarrow{+}_{cbv} enc(t_2)$ is a prefix of *Red*. We use induction 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 5.1. 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 these 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. □

Note that Proposition 4.3 cannot be applied since λ^T_{pair} is not an extension of untyped λ_{cbv}. 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 $s \sim_{pair,(\mathsf{unit},\mathsf{unit})\to(\mathsf{unit},\mathsf{unit})} t$ holds in λ^T_{pair} by standard reasoning, but after translation to λ_{cbv}, we have $enc(s) \not\sim_{cbv} enc(t)$. The latter can be seen with the context $C = ([\cdot]\ \mathbf{unit})$, since $C[enc(s)]$ is divergent while $C[enc(t)]$ converges.

The extension situation could perhaps be regained by a System F-like type system, which we leave for future research. Here we just observe that the use of a simple type system for λ_{cbv} is insufficient since the encoding of pairs with components of different types cannot be simply typed. (The same holds for Hindley-Milner polymorphic typing.)

In [Schmidt-Schauß et al. 2008] we have shown that an adequacy result also holds if nondeterminism is added to both calculi, and if arbitrary expressions (instead of just values) are allowed as components of pairs.

5.2 The Encoding of de Bruijn

We demonstrate that our framework also encompasses encodings like the de Bruijn index encoding of variables (see [de Bruijn 1972]) in a (call-by-name, lazy) lambda calculus. This encoding translates variables as numbers, where the number indicates the distance of a variable to its binding position, measured in the number of crossings of lambda-expressions. For example $(\lambda x.x\ (\lambda y.y\ x))$ is translated as $(\lambda.1\ (\lambda.1\ 2))$. This encoding is sometimes used during the compilation of functional and functional-logic programming languages.

In the following we argue that our methods are powerful enough to show that for closed lambda-expressions the encoding is fully abstract and also an isomorphism. For open expressions and contexts the encoding has to be defined with some care in order to obtain full abstraction and the isomorphism property.

Let L be the closed expressions of the untyped pure lambda calculus (see Example 2.1) and let B be the closed expressions of the untyped pure lambda calculus using de Bruijn indices, i.e. the syntax of B is $E ::= n \mid (E\ E) \mid \lambda.E$, where n represents positive integers 1,2,....

Let S range over (finite) sequences of variables, index(x, S) be the position of the leftmost x in S, [] be the empty sequence and $x : S$ be the sequence S extended by the head element x. The translation T maps closed lambda expressions to its de

Bruijn encoding. T is defined as $T(t) = T'(t, [])$ where T' is defined as:

$$\begin{array}{lcl} T'(\lambda x.s, S) &=& T'(s, x : S) \\ T'(s\ t, S) &=& (T'(s, S)\ T'(t, S)) \\ T'(x, S) &=& \mathrm{index}(x, S) \end{array}$$

For example, $T(\lambda x.x) = \lambda.1$ and $T(\lambda x.(\lambda y.x\ (y\ x)) = \lambda.(\lambda.2\ (1\ 2))$. We also assume that the de Bruijn beta-reduction is a one step-reduction such that for closed lambda-expressions s_1, s_2 the following diagram holds:

$$\begin{tikzcd} s_1 \arrow[r, "T"] \arrow[d, "\beta"] & T(s_1) \arrow[d, "dB\beta"] \\ s_2 \arrow[r, "T"] & T(s_2) \end{tikzcd}$$

For example $\lambda.(\lambda.\lambda.\lambda.3)\ (\lambda.2) \xrightarrow{dB\beta} \lambda.(\lambda.\lambda.\lambda.4)$. Let the observers of L and B be the corresponding closed contexts, where T treats contexts like expressions. The convergence is defined w.r.t. an outermost call-by-name reduction to an abstraction, i.e. the lazy lambda calculus [Abramsky 1990]. Using the standard definitions, we see that the translation T is (ce): i.e. $t \Downarrow$ iff $T(t) \Downarrow$ for closed t. Since only closed contexts are applied to closed lambda expressions, the translation T satisfies compositionality: $T(C[s]) = T(C)[T(s)]$, and hence the translation is (cuo). Hence we can conclude that the translation T is observationally correct. T is not injective, since for example $T(\lambda x.x) = T(\lambda y.y) = \lambda.1$. However, if the closed expressions s, t are α-equivalent, then $T(s) = T(t)$. The following holds: Closed contexts and lambda expressions in a de Bruijn-encoding can be retranslated into lambda expressions (up to α-renaming). Since T is a bijection (modulo α-renaming) on contexts and expressions, the translation T is also fully abstract, and even an isomorphism, since the calculi are untyped and the translation is a bijection between the quotients $L/\sim$ and $B/\sim$.

Also the translation of open expressions into a de Bruijn-encoding turns out to be an isomorphism if some precautions are taken into account. The extra precautions are: A fixed enumeration $x_1, x_2, \ldots$ of all variables, and a modified translation of contexts C, where the scope of variables in the hole has to be adapted to the enumeration of variables by translating CD_k instead of C where $D_k = ((\lambda x_k.\lambda x_{k-1}.\ldots.\lambda x_1.[\cdot])\ x_k\ \ldots\ x_1)$.

6. CONCLUSIONS AND OUTLOOK

Motivated by translation problems between concurrent programming languages, this paper clarified the notions and the methods, and provided some tools for proving the correctness of translations. The presented framework can be applied directly to the operational semantics and the derived observational equivalences of programming calculi and thus does not rely on denotational models which are usually hard to find.

In future research the framework can be used to prove the correctness of various impementations, especially in concurrent settings where correctness of synchronization abstractions is often far from obvious.

REFERENCES

Abramsky, S. 1990. The lazy lambda calculus. In *Research Topics in Functional Programming*, D. Turner, Ed. -Addison-Wesley, 65–116.

Ahmed, A. and Blume, M. 2008. Typed closure conversion preserves observational equivalence. In *Proc. 13th ICFP*. 157–168.

Alice 2007. *The Alice Project.* Saarland University, `http://www.ps.uni-sb.de/alice`.

Carayol, A., Hirschkoff, D., and Sangiorgi, D. 2005. On the representation of McCarthy's amb in the pi-calculus. *Theoret. Comput. Sci. 330,* 3, 439–473.

de Bruijn, N. G. 1972. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae 34*, 381–392.

De Nicola, R. and Hennessy, M. 1984. Testing equivalences for processes. *Theoret. Comput. Sci. 34*, 83–133.

de'Liguoro, U. and Piperno, A. 1992. Must preorder in non-deterministic untyped lambda-calculus. In *17th CAAP*. Springer, 203–220.

Felleisen, M. 1991. On the expressive power of programming languages. *Sci. Comput. Programming 17,* 1–3, 35–75.

Gladstein, D. S. and Wand, M. 1996. Compiler correctness for concurrent languages. In *COORDINATION '96: Proceedings of the First International Conference on Coordination Languages and Models*. Springer-Verlag, London, UK, 231–248.

Gordon, A. D. 1999. Bisimilarity as a theory of functional programming. *Theoret. Comput. Sci. 228,* 1–2, 5–47.

Hu, L. and Hutton, G. 2009. Compiling Concurrency Correctly: Cutting out the Middle Man. In *Proceedings of the Symposium on Trends in Functional Programming*. Komarno, Slovakia. to appear.

Johann, P. and Voigtländer, J. 2006. The impact of seq on free theorems-based program transformations. *Fundamenta Informaticae 69,* 1–2, 63–102.

Kutzner, A. and Schmidt-Schauss, M. 1998. A nondeterministic call-by-need lambda calculus. In *Proc. ICFP*. ACM, 324–335.

Leroy, X. 2009. Formal verification of a realistic compiler. *Commun. ACM 52,* 7, 107–115.

Mason, I., Smith, S. F., and Talcott, C. L. 1996. From operational semantics to domain theory. *Inform. and Comput. 128*, 26–47.

Matthews, J. and Findler, R. B. 2007. Operational semantics for multi-language programs. In *34th ACM POPL*. ACM, 3–10.

McCarthy, J. 1963. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, Eds. North-Holland, Amsterdam, 33–70.

McCusker, G. 1996. Full abstraction by translation. In *Advances in Theory and Formal Methods of Computing*. IC Press.

Milner, R. 1977. Fully abstract models of typed lambda calculi. *Theoret. Comput. Sci. 4,* 1, 1–22.

Milner, R. 1989. *Communication and Concurrency.* Prentice Hall.

Milner, R. 1990. Functions as processes. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*. Springer-Verlag New York, Inc., New York, NY, USA, 167–180.

Mitchell, J. C. 1993. On abstraction and the expressive power of programming languages. *Sci. Comput. Programming 21,* 2, 141–163.

Nain, S. and Vardi, M. Y. 2007. Branching vs. linear time: Semantical perspective. In *ATVA*. 19–34.

Niehren, J., Sabel, D., Schmidt-Schauss, M., and Schwinghammer, J. 2007. Observational semantics for a concurrent lambda calculus with reference cells and futures. *Electron. Notes Theor. Comput. Sci. 173*, 313–337.

Niehren, J., Schwinghammer, J., and Smolka, G. 2006. A concurrent lambda calculus with futures. *Theoret. Comput. Sci. 364,* 3, 338–356.

Peyton Jones, S., Gordon, A., and Finne, S. 1996. Concurrent Haskell. In *23rd ACM POPL*. ACM, 295–308.

Pierce, B. C. 2002. *Types and Programming Languages*. The MIT Press.

Pitts, A. and Stark, I. 1998. Operational reasoning for functions with local state. In *Higher order operational techniques in semantics*, A. Pitts and A. Gordon, Eds. Publications of the Newton Institute, vol. 12. Cambridge university press, 227–273.

Pitts, A. D. 2000. Parametric polymorphism and operational equivalence. *Math. Structures Comput. Sci. 10*, 321–359.

Plotkin, G. D. 1975. Call-by-name, call-by-value, and the lambda-calculus. *Theoret. Comput. Sci. 1*, 125–159.

Plotkin, G. D. 1977. LCF considered as a programming language. *Theoret. Comput. Sci. 5,* 3, 225–255.

Riecke, J. G. 1991. Fully abstract translations between functional languages. In *18th ACM POPL*. ACM, 245–254.

Ritter, E. and Pitts, A. M. 1995. A fully abstract translation between a lambda-calculus with reference types and Standard ML. In *Proc. 2nd TLCA*. Springer, 397–413.

Sabel, D. 2008. Semantics of a call-by-need lambda calculus with McCarthy's amb for program equivalence. Ph.D. thesis, J. W. Goethe-Universität Frankfurt, Institut für Informatik. Fachbereich Informatik und Mathematik.

Sabel, D. and Schmidt-Schauss, M. 2008. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci. 18,* 3, 501–553.

Sanjabi, S. B. and Ong, C.-H. L. 2007. Fully abstract semantics of additive aspects by translation. In *Proc. 6th OASD*. ACM, 135–148.

Schmidt-Schauss, M., Niehren, J., Schwinghammer, J., and Sabel, D. 2008. Adequacy of compositional translations for observational semantics. In *5th IFIP TCS 2008*. IFIP, vol. 273. Springer, 521–535.

Schmidt-Schauss, M. and Sabel, D. 2010. Closures of may-, should- and must-convergences for contextual equivalence. *Information Processing Letters 110,* 6, 232 – 235.

Schmidt-Schauss, M., Sabel, D., and Machkasova, E. 2010. Simulation in the call-by-need lambda-calculus with letrec. In *RTA'10 (to appear)*.

Schwinghammer, J., Sabel, D., Schmidt-Schauss, M., and Niehren, J. 2009. Correctly translating concurrency primitives. In *ML '09: Proceedings of the 2009 ACM SIGPLAN workshop on ML*. ACM, New York, NY, USA, 27–38.

Shapiro, E. 1991. Separating concurrent languages with categories of language embeddings. In *23rd ACM STOC*. ACM, 198–208.

Voigtländer, J. and Johann, P. 2007. Selective strictness and parametricity in structural operational semantics, inequationally. *Theor. Comput. Sci 388,* 1–3, 290–318.

Wand, M. 1995. Compiler correctness for parallel languages. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*. ACM, New York, NY, USA, 120–134.

Received Month Year; revised Month Year; accepted Month Year