J. Functional Programming **10** (5): 453–499, September 2000. Printed in the United Kingdom 453

Uniform confluence in concurrent computation

JOACHIM NIEHREN

Programming Systems Laboratory, Universität des Saarlandes, 66041 Saarbrücken, Germany
(*e-mail:* niehren@ps.uni-sb.de)

Capsule Review

A novel technique for comparing complexities of rewriting systems is presented. The essential machinery is the notion of uniform confluence, which is very simple and yet pleasantly applicable to various useful systems. In particular, simulations work nicely when we have uniform confluence, and this observation provides a good basis for comparing complexities of systems.

The translations from functional programs to the concurrent setting (in particular, the π-calculus) have been studied by many people. This work adds a new interesting insight; using uniform confluence and simulations it is shown that there are complexity-respecting translations from the call-by-value lambda calculi into a fragment of the π-calculus.

This result enables us to compare the complexity of the call-by-value standard reduction with that of the call-by-need standard reduction. Though these reduction systems may not directly respect the complexities of the actual implementation, this result suggests an abstract and formal way to understand the efficiency of call-by-need over call-by-value, which has been a folklore and never been treated formally.

Abstract

Indeterminism is typical for concurrent computation. If several concurrent actors compete for the same resource then at most one of them may succeed, whereby the choice of the successful actor is indeterministic. As a consequence, the execution of a concurrent program may be nonconfluent. Even worse, most observables (termination, computational result, and time complexity) typically depend on the scheduling of actors created during program execution. This property contrast concurrent programs from purely functional programs. A functional program is uniformly confluent in the sense that all its possible executions coincide modulo reordering of execution steps. In this paper, we investigate concurrent programs that are uniformly confluent and their relation to eager and lazy functional programs. We study uniform confluence in concurrent computation within the applicative core of the π-calculus which is widely used in different models of concurrent programming (with interleaving semantics). In particular, the applicative core of the π-calculus serves as a kernel in foundations of concurrent constraint programming with first-class procedures (as provided by the programming language Oz). We model eager functional programming in the λ-calculus with weak call-by-value reduction and lazy functional programming in the call-by-need λ-calculus with standard reduction. As a measure of time complexity, we count application steps. We encode the λ-calculus with both above reduction strategies into the applicative core of the π-calculus and show that time complexity is preserved. Our correctness proofs employs a new technique based on uniform confluence and simulations. The strength of our technique is illustrated by proving a folk theorem, namely that the call-by-need complexity of a functional program is smaller than its call-by-value complexity.

1 Introduction

During the last 15 years, concurrency has been investigated for high-level programming. This kind of research lead to the development of a variety of new programming languages. Two major lines of research can be distinguished, concurrent constraint programming (Maher, 1987; Saraswat *et al*., 1991; Smolka, 1995) which originates from logic programming and concurrent functional programming (Reppy, 1992; Thomsen *et al*., 1993; Armstrong *et al*., 1996; Pierce and Turner, 1997; Fournet and Maranget, 1997). The work presented here was motivated by concurrent constraint programming but contributes mainly to the area functional programming.

Computation models of concurrent programming

It is standard that high-level programming languages are designed on the basis of a *computation model*. The level of abstraction on which these models are formulated often permits to relate quite distinct programming paradigms.

The most popular model of concurrent constraint programming is Saraswat's (1991) cc-model. It describes concurrent constraints that communicate over common logic variables residing in a global constraint store. In contrast to a memory store in the classical machine-oriented sense, a constraint store contains information on logic variables and increases monotonically as computation proceeds. Monotonicity is a central property needed for reliable synchronization. The ρ-calculus (Niehren and Smolka, 1994; Niehren and Müller, 1995) extends the cc-model with first-class procedures. It is a variation of Smolka's (1994) γ-calculus which models important aspects of the concurrent constraints language Oz (Smolka *et al*., 1995).

In the research line starting from functional computation, Milner (1992) proposed the π-calculus as a model of concurrent computation. The π-calculus describes concurrent actors that communicate over shared channels. For several years, the π-calculus served mainly for semantical reasoning. Later on, it was also used as the basis of a concurrent programming language named Pict (Pierce and Turner, 1997). Another concurrent computation model of interest is the join calculus (Fournet and Gonthier, 1996), which was introduced as a variation of the π-calculus. The join calculus underlies the join calculus language (Fournet and Maranget, 1997) which features distributed programming.

All computation models mentioned above are closely related. The relationship between the ρ-calculus and the π-calculus was first noticed by Smolka (1994) and was formally elaborated by Niehren and Müller (1995), Victor and Parrow (1996) and Niehren (1996). Most of the material of this article stems from Niehren (1996), but has undergone a major revision. The relationship between the join calculus and the π-calculus is investigated in Fournet and Gonthier (1996).

Confluence

Indeterminism is typical for concurrent computation. If several concurrent actors compete for the same resource then at most one of them may succeed eventually, whereby the choice of the successful actor is indeterministic. As a consequence,

the execution of a concurrent program may be non-confluent. Even worse, most observables (termination, computational result, and time complexity) depend on the scheduling of the actors created during program execution. This property contrasts concurrent programs to purely functional programs whose execution is expected to be confluent.

A functional program might be expected to be *uniformly confluent* in the sense that all its possible executions coincide modulo reordering of executable function calls. Of course, this property depends on the notion of an *executable* function call. In this article, function calls nested inside of function definitions are not considered executable. This view is consistent with all implementations of functional languages used in practice; it is also naturally reflected in the λ-calculus by means of some *weak* reduction strategy to which we will restrict ourselves.

1.1 Uniform confluence in concurrent computation

In this article, we investigate the class of concurrent programs that are uniformly confluent in the sense that all execution of a program coincide modulo reordering of execution steps. We then consider eager and lazy functional programs as uniformly confluent concurrent programs.

Our study of uniformly confluence in concurrent computation is based on the applicative core of the π-calculus which is widely used in different models of concurrent programming. Eager functional programming is modeled in the call-by-value λ-calculus with weak reduction and lazy functional programming in the call-by-need λ-calculus with standard reduction (Ariola *et al*., 1995). We measure the time complexity of the execution of a concurrent or functional program by counting application steps.

Uniform confluence

We consider a computation model as a *calculus* which is an abstract rewrite system (Dershowitz and Jouannaud, 1990; Klop, 1987) consisting of a set of (program) *expressions* denoted with E, a binary relation $\rightarrow$ between expressions that we call (one-step) *reduction*, and an equivalence relation $\equiv$ on expressions called *congruence*. In this paper, we mainly consider three different calculi: the λ-calculus with the weak call-by-value strategy, the call-by-need λ-calculus with standard reduction, and the applicative core of the π-calculus. Note that our abstract notion of a calculus applies to the π-calculus but deviates from Plotkin's (1975) usage, who considers the λ-calculus as an equational theory (not including a reduction strategy).

Following Niehren and Smolka (1994), we call an expression E *uniformly confluent* if $E_1 \leftarrow E \rightarrow E_2$ implies that either $E_1 \equiv E_2$ or there exists E' such that $E_1 \rightarrow E' \leftarrow E_2$. All executions of a uniformly confluent expression E coincide up to a reordering of reduction steps. Uniform confluence is an important notion for reasoning about (time) complexity where the complexity of the execution of a program expression is measured in the number of its reduction steps. The complexity of a uniformly confluent expression is independent of the scheduling of the concurrent actors created

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

during its execution. We call a calculus uniformly confluent if all its program expressions are uniform confluent. Note that a uniformly confluent calculus is confluent in the usual sense of (abstract) rewrite systems (Dershowitz and Jouannaud, 1990; Klop, 1987). Note also that the notions of confluence investigated by Nestmann (1996) and Philippou and Walker (1997) are quite unrelated to those for rewriting systems.

Contribution

We study uniformly confluent concurrent computation in the applicative core of the π-calculus. We present embeddings of both, the λ-calculus with weak call-by-value reduction and the call-by-need λ-calculus with standard reduction, into the applicative core of the π-calculus. We prove our encodings correct in the sense that they preserve complexity up to a constant factor. Our correctness proofs exploit a new technique we develop based on the notions of uniform confluence and simulations. The strength of this technique is illustrated by proving a folk theorem, namely that the call-by-need complexity of a functional program is smaller than its call-by-value complexity (measured in terms of β-reduction steps).

Complexity measures

One might hesitate to accept the number of β-reduction steps as a good measure for the complexity of functional programs. A first counter argument is that one might wish to choose some reduction strategy not considered in this article. However, the alternatives are few. The restriction to weak reduction is not problematic in this respect, as long as compile time optimizations (program transformations or partial evaluation) are ignored. The choice of a particular weak reduction strategy does not matter for call-by-value since all of them coincide in the number of β-reduction steps. For call-by-need, we do also not know about any alternative which significantly differs from the call-by-need λ-calculus with standard reduction, at least with respect to counting β-reduction steps.

A more severe doubt might be that the substitution operation of the λ-calculus is not appropriate for modeling time complexity in implementations. Of course, substitutions $[M/x]$ of arbitrary λ-terms M for some variable x are not realistic; In call-by-name reduction, these substitutions raise duplications of β-reduction steps; in weak call-by-name reduction, they even lead to non-confluence. In this article, we restrict ourselves to substitutions $[V/x]$ of some value V for a variable x. Still, one might object that substitutions $[V/x]$ are non realistic for implementations since many occurrences of x are replaced by V in one single step (whereas in an implementation one might access V in a closure several times). Therefore, one might opt for a calculus with explicit substitution (see, for instance, Abadi *et al.*, 1995) or for the call-by-let λ-calculus with some form of weak reduction (Maraist *et al.*, 1995). A more recent discussion on models of call-by-need complexity that is appropriate for actual implementations is given by Moran and Sands (1999).

For the formal approach of this article however, both of these alternatives seem

to be problematic since they require administrative steps for the treatment of substitutions or let-environments, which do not easily combine with uniform confluence. For instance, it seems difficult to define a weak reduction strategy for the call-by-let λ-calculus (Maraist *et al.*, 1995) which is at the same time maximally liberal in its reduction strategy and uniformly confluent (compare Example 7.2 below). Nevertheless, additional administrative steps have to be implemented and thus, their execution costs time. We conjecture that the costs of administrative steps can be safely ignored for a complexity analysis of *weak* reduction, in contrast to deep reduction (Asperti, 1997).

Structure of the article

In the remainder of this introduction, we survey our formal contributions, which concerns calculi and embeddings, complexity results and a proof technique based on uniform confluence. In sections 2, 3, and 4, we develop a theory of complexity based on uniform confluence and simulations. In section 5 we introduce concurrent computation in the applicative core of the π-calculus and investigate uniform confluence on basis of a linear type system. In sections 6 and 7 we translate eager and lazy functional computation respectively into concurrent computation and prove that complexity is preserved. Finally, section 8 contains a formal comparison between call-by-value and call-by-need complexity.

For lack of space, most of the simpler proofs are only scetched or completely omitted. They can however be found in an unabridged technical report preceeding this article (Niehren, 1999). This report also supplies an additional result of its own interest: an encoding of the δ-calculus (see section 1.3) into the applicative core of the π-calculus.

1.2 The applicative core of the π-calculus

Our study is based on the applicative core of the polyadic asynchronous π-calculus (Milner, 1991; Honda and Tokoro, 1991; Boudol, 1992) that we call π_0 here but δ_0 in Niehren (1999). Also, π_0 is a subcalculus of the ρ-calculus Niehren and Smolka (1994) and Niehren and Müller, 1995). Therefore, all results presented here also apply to concurrent constraint programming.

The fragment π_0 of the π-calculus

As with any other calculus, we define π_0 in terms of a set of expressions, a structural congruence, and a reduction relation. Expressions of π_0 are built from variables ranged over by x, y, z. An expression E of π_0 is either a (named) abstraction, an application, a concurrent composition, or a declaration, as given by the following abstract syntax:

$$E \ ::= \ x{:}x_1 \dots x_n/E \ \mid \ xx_1 \dots x_n \ \mid \ E|E' \ \mid \ (\nu x)E \qquad (n \geqslant 0)$$

The syntax of π_0 is borrowed from the ρ-calculus rather than from the π-calculus. A (named) abstraction $x{:}x_1 \dots x_n/E$ requires the (syntactic) value of variable x to

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

be the (anonymous) abstraction $x_1 \dots x_n/E$. An application $x x_1 \dots x_n$ applies the abstraction referred to by x with arguments referred to by $x_1, \dots, x_n$.

The calculus π_0 is identical to the ρ-calculus without cells and constraints. It also coincides with the polyadic asynchronous π-calculus without once-only input agents $x?(x_1 \dots x_n).E$. In the terminology of the π-calculus, variables are usually called channels. Following the syntax of Kobayashi *et al.* (1996), a (named) abstraction $x{:}x_1 \dots x_n/E$ would be called a replicated input agent $x?^*(x_1 \dots x_n).E$ and an application $x x_1 \dots x_n$ an output agent $x!(x_1 \dots x_n)$.

Reduction in π_0 is defined by the following application relation $\rightarrow_A$ where $\bar{y}$ and $\bar{z}$ stand for sequences of variables $y_1 \dots y_n$ and $z_1 \dots z_n$:

$$x{:}\bar{y}/E \mid x\bar{z} \;\rightarrow_A\; x{:}\bar{y}/E \mid E[\bar{z}/\bar{y}]$$

We do allow for reduction in every context except below abstraction. It was proved by Niehren and Smolka (1994) and Niehren (1994) that π_0 is uniformly confluent when restricted to expressions that remain consistent under reduction, i.e. that cannot be reduced to an expression containing two abstractions with the same name $x{:}\bar{y}/E$ and $x{:}\bar{z}/E'$ that are not congruent. For the expressions stemming from functional programming, we can ensure this invariant by an appropriate linear type system.

1.3 Call-by-value and call-by-need translation

The applicative core π_0 of the π-calculus is surprisingly expressive. An encoding of the λ-calculus with call-by-value reduction was already given by Niehren and Müller (1995). A analogous result for the full π-calculus was proved earlier by Milner (1992). As we show in this article, it is also possible to express lazy functional computation in π_0. We present an encoding of the call-by-need λ-calculus with standard reduction (Ariola *et al.*, 1995). An encoding of the call-by-need λ-calculus with standard reduction into the full π-calculus was proved before by Brock and Ostheimer (1995). Independently, Smolka (1994) formulated an analogous encoding but without correctness proof.

The δ-calculus

A major difficulty in finding an encoding of some λ-calculus into π_0 is to devise a mechanism for transporting values along reference chains. The only values considered in this article are anonymous abstractions. In π_0, abstractions can be transported when using continuation passing style. In a model of concurrent constraint programming, equational constraints $x{=}y$ can be used for this purpose.

To abstract from 'how to transport abstractions', we introduce an extension of π_0 that we call the δ-calculus. The δ-calculus extends π_0 extended with three new forms of expressions, which come with two additional reduction relations, forwarding $\rightarrow_F$ and triggering $\rightarrow_T$. The first new expressions are *forwarders* of the form $x = y$ which are directed from the right to the left (and not symmetric). Their operational semantics is to forward an abstraction from y to x (as soon as the abstraction referred to by y is available). The forwarding relation $\rightarrow_F$ is described by the

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

following rule where $\overline{z}$ stands for a sequence of variables $z_1 \ldots z_n$:

$$x{=}y \mid y{:}\overline{z}/E \quad \rightarrow_F \quad x{:}\overline{z}/E \mid y{:}\overline{z}/E$$

Note that an analogous operation for 'copying abstractions' exists implicitly in the call-by-value λ-calculus and explicitly in the call-by-need λ-calculus. Note also, that forwarders provide for single assignment as known from a directed usage of logic variables (Pingali, 1987) in the data-flow language Id (Arvind *et al.*, 1989).

The remaining additional constructions of the δ-calculus can be used to encode call-by-need control. There are *delay expressions* of the form $x.E$ and *trigger expressions* $\mathbf{tr}(x)$. The execution of an expression E nested into a delay expression $x.E$ is delayed until a trigger expression $\mathbf{tr}(x)$ becomes active. The trigger relation $\rightarrow_T$ is defined by the following rule:

$$x.E \mid \mathbf{tr}(x) \rightarrow_T E \mid \mathbf{tr}(x)$$

The δ-calculus can be encoded into π_0 such that complexity is preserved up to a constant factor. Proving this is of its own interest but not in the scope of the presented article. It can be found in Niehren (1999). Note also, that the encoding of the δ-calculus into π_0 given in Niehren (1996) is not complexity preserving.

Call-by-value translation

Encoding the λ-calculus with weak call-by-value reduction into the δ-calculus is quite simple. The idea is to name all entities of interest by variables, in particular functional abstractions and return values. Functional nesting can then be replaced by concurrent composition and declaration. For example, let I be the λ-identity $\lambda y.y.$ and z be a variables for naming it. The call-by-value translation $[\![I]\!]_z^{\mathrm{val}}$ of I with name z is a named abstraction:

$$[\![I]\!]_z^{\mathrm{val}} \;\equiv\; z{:}yy'/y'{=}y$$

An additional output argument y' is introduced whose value is related to the input argument y by means of a forwarder. Whenever z is applied, this forwarder passes the input value to the output argument.

Naming leads to a call-by-value translation such that every weak call-by-value β-reduction step corresponds to exactly one application plus at most two forwarding steps in the δ-calculus. Translation is adequate in that it preserves complexity up-to a constant factor. There exists a simulation which relates every weak call-by-value execution to a unique execution in the δ-calculus.

However, a bisimulation does not exist. Consider for instance the λ-term $I(II)$ which has a unique weak call-by-value execution where the inner redex is reduced first. Modulo an irrelevant simplification (which introduces sharing for all occurrences of the definition of I), the call-by-value translation $[\![I(II)]\!]_z^{\mathrm{val}}$ is:

$$[\![I(II)]\!]_z^{\mathrm{val}} \quad \approx \quad (\nu x)(\nu y)([\![I]\!]_x^{\mathrm{val}} \mid xxy \mid xyz)$$

It contains a named abstraction $[\![I]\!]_x^{\mathrm{val}}$ and two applications corresponding to the inner and out redex respectively. Thus, $[\![I(II)]\!]_z^{\mathrm{val}}$ can be reduced in two ways

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

depending on the scheduling of its applications. Those two executions corresponds to either first reducing the inner or outer redex of $I(II)$. In contrast, the outer redex can not be reduced by call-by-value reduction. Hence, call-by-value translation introduces new flexibility to the scheduling of applications. This property contrasts the presented call-by-value translation with those of Milner (1992), who considers deterministic reduction strategies only such that bisimulations exist.

According to a proposal of Philip Wadler during a personal communication the observation of the previous paragraph can be rephrased as the following conjecture: The mapping of M to $[\![M]\!]_z^{\mathrm{val}}$ rather encodes call-by-let λ-calculus (Maraist *et al.*, 1995) with some form of weak reduction than the λ-calculus with weak call-by-value reduction. In order to establish this statement formally, an appropriate weak reduction strategy for the call-by-let λ-calculus has to be defined such that in our example $I(II)$ above, both the inner and the outer redex in $I(II)$ could be reduced. As mentioned before, it is unclear how to define such a weak reduction strategy.

Call-by-need translation

We now consider the call-by-need λ-calculus with standard reduction. This calculus can also be encoded into the δ-calculus based on the idea of naming. In addition, we have to express call-by-need control which can be encoded by the delay and trigger mechanism of the δ-calculus. For instance, the call-by-need translation $[\![I]\!]_z^{\mathrm{need}}$ of I with name z is the a named abstraction:

$$[\![I]\!]_z^{\mathrm{need}} \;\equiv\; z\!:\!yy'/(y'\!=\!y \mid \mathbf{tr}(y))$$

The only difference to $[\![I]\!]_z^{\mathrm{val}}$ is that the computation of the value of the input argument y has to be triggered before and not only forwarded to the output argument y'. The call-by-need translation $[\![I(II)]\!]_z^{\mathrm{need}}$ looks as follows (modulo sharing of I):

$$[\![I(II)]\!]_z^{\mathrm{need}} \;\approx\; (vx)(vy)([\![I]\!]_x^{\mathrm{need}} \mid y.[\![xxy]\!]_y^{\mathrm{need}} \mid xyz)$$

Also, the definition of $[\![I(II)]\!]_z^{\mathrm{need}}$ is similar to that of $[\![I(II)]\!]_z^{\mathrm{val}}$ except that the application corresponding to the inner redex is delayed.

Our call-by-need translation is correct in that every execution in the call-by-need λ-calculus with standard reduction can be simulated by a unique execution in the δ-calculus and vice versa, i.e. there exists a bisimulation and not only a simulation as for call-by-value. Every application step in call-by-need λ-calculus with weak reduction corresponds to exactly one application plus some triggering and forwarding steps. Thus, our call-by-need translation preserves complexity measured by counting application steps.

A proof technique based on uniform confluence

An adequacy proof for the call-by-value translation has to deal with the fact that this embedding introduces new flexibility to the executions scheduling. This problem can be solved due to a proof technique that combines uniform confluence and

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

simulations (Niehren, 1994). It is sufficient to prove that every execution of M can be simulated by an execution of $[\![M]\!]_z^{\mathrm{val}}$ of the same length up to a constant factor. The uniform confluence of the λ-calculus with weak call-by-value reduction implies that all executions of M have the same length and the uniform confluence of the consistent fragment of the δ-calculus implies that all executions of $[\![M]\!]_z^{\mathrm{val}}$ have the same length. Hence all executions of M and $[\![M]\!]_z^{\mathrm{val}}$ have the same complexity up to a constant factor.

The proof technique based on simulations is less restrictive than Milner's (1992) bisimulation based proof technique, but only applicable once uniform confluence is available. The bisimulations technique is more general in that it allows to deal with observable indeterminism. Also, note that the definitions of the concrete simulations given in the technical part of this article are strongly inspired by the concrete bisimulations presented by Milner (1992).

Call-by-need versus call-by-value complexity

It is a folk theorem that – up to overhead – the time complexity of call-by-need computation is smaller than the time complexity of call-by-value computation. We formalize the folk theorem for a first time for the call-by-need and call-by-value λ-calculus and prove its correctness in this setting. Every closed λ-expression M satisfies:

$$\mathscr{C}^{\mathrm{need}}(M) \leqslant \mathscr{C}^{\mathrm{val}}(M)$$

where $\mathscr{C}^{\mathrm{need}}(M)$ denotes the complexity of M in the call-by-need λ-calculus with standard reduction and $\mathscr{C}^{\mathrm{val}}(M)$ the complexity M in the call-by-value λ-calculus with weak reduction (both measured in terms on β-reduction steps).

Our proof exploits the similarity of call-by-value and call-by-need translations into the δ-calculus, i.e. that $[\![M]\!]_z^{\mathrm{val}}$ and $[\![M]\!]_z^{\mathrm{need}}$ coincide for all M and z up to triggering and delay expressions expressing the call-by-need control. Therefore, every application step in an execution of $[\![M]\!]_z^{\mathrm{need}}$ corresponds to an application step for $[\![M]\!]_z^{\mathrm{val}}$. Hence every β-reduction step of M under standard call-by-need reduction corresponds to a β-reduction step of M under weak call-by-value reduction. This might seem surprising, because in the call-by-need λ-calculus an abstraction may be applied before its argument is evaluated, whereas in the λ-calculus with call-by-value reduction, an argument has to be evaluated first. This problem is solved due to the additional flexibility introduced by our call-by-value translation. In other words, we rather compare call-by-need with call-by-let than call-by-value. This does not matter because call-by-let and call-by-value reduction are indistinguishable with respect to complexity of weak reductions. They only differ in the flexibility of possible schedulings for applications.

Alternative approaches

Alternative formalizations of the folk theorem and respective correctness proofs can be derived from previous results on weak optimal reduction. Yoshida (1993) proves

for her λf-calculus that weak call-by-need reduction is optimal in that the complexity of weak call-by-need reduction is smaller than the complexity of any other weak reduction strategy. An explicit relationship between the λf-calculus and the call-by-value λ-calculus or call-by-need λ-calculus is not given. An earlier approach to weak optimal reduction based on calculi with explicit substitutions was explored by Maranget (1990). He also proved optimality of weak call-by-need reduction with respect to weak reduction.

We compare the complexity of call-by-value and call-by-need by means of an indirection through the δ-calculus. One might ask whether such an indirection is necessary. The problem that one has to deal with is that the scheduling of function calls in call-by-need and call-by-value are very different. We therefore need a common calculus in which to express both schedulings in a uniform way. We propose the δ-calculus for this purpose. Alternatively, it should also be possible to choose the call-by-let λ-calculus with some weak reduction strategy to be defined. Also it might be possible to deal with Yoshida's λf-calculus (1993) or a calculus with explicit substitutions as used by Maranget (1990).

Call-by-need computation models

A call-by-need model describes the complexity behavior of lazy functional programming. In other words, a call-by-need model is a call-by-name model with sharing of evaluations. Designing good call-by-need models turned out to be a difficult task which was solved only recently. Early approaches are based on calculi with explicit substitutions (Purushothaman and Seaman, 1992; Maranget, 1992) or graph reduction (Jeffrey, 1994). Launchbury's (1993) call-by-need model uses environments. It is base on a big-step semantics where complexity is reflected in the size of proof trees. Launchbury relates his call-by-need calculus to the λ-calculus with call-by-name reduction and proves correctness with respect to denotational semantics (but not with respect to complexity). A first small-step semantics for call-by-need computation based on λ-term notation was introduced by several authors (Ariola *et al.*, 1995; Ariola and Felleisen, 1997; Maraist *et al.*, 1998) under the name *call-by-need λ-calculus*. In the same papers, the relationship between the call-by-need λ-calculus and the λ-calculus with standard call-by-name reduction is established and proved correct.

2 Complexity and uniform confluence

We introduce a framework in which to reason about complexity in computation models. It provides several criteria for proving that an encoding preserves complexity. The basic notion of our framework is the notion of a calculus which can be considered as an abstract computation model. The notion of a calculus slightly generalizes Klop's (1987) abstract rewrite systems; it was first introduced by the author (1994).

We need the following notation throughout the paper. The symbol $\circ$ stands for relational composition; if $\rightarrow_1$ and $\rightarrow_2$ are two binary relations on some set $\mathscr{E}$ and

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

$E, E'' \in \mathscr{E}$, then $E \rightarrow_1 \circ \rightarrow_2 E''$ if and only if there exists $E' \in \mathscr{E}$ such that $E \rightarrow_1 E'$ and $E' \rightarrow_2 E''$.

Definition 2.1

A *calculus* is a triple $(\mathscr{E}, \equiv, \rightarrow)$, where $\mathscr{E}$ is a set, $\equiv$ an equivalence relation and $\rightarrow$ a binary relation on $\mathscr{E}$. Elements of $\mathscr{E}$ are called *expressions* of the calculus, $\equiv$ its *congruence*, and $\rightarrow$ its *reduction*. We require for every calculus that its reduction is *modulo its congruence*, i.e. that $(\equiv \circ \rightarrow \circ \equiv) \subseteq \rightarrow$ holds.

Typical examples for a calculus are the π-calculus, the ρ-calculus, the λ-calculus with some reduction strategy, abstract rewrite systems, Turing machines, etc.

2.1 Complexity

Intuitively, the complexity of an expression in some calculus is the maximal length of one of its (complete) executions. We now define formally what this means.

Let a calculus $(\mathscr{E}, \equiv, \rightarrow)$ be given. We call an expression $E \in \mathscr{E}$ *irreducible* if there exists no $E' \in \mathscr{E}$ such that $E \rightarrow E'$. A *partial execution* in the given calculus is a finite or infinite sequence $(E_i)_{i=1}^{n}$ or $(E_i)_{i=0}^{\infty}$ of expressions such that $E_i \rightarrow E_{i+1}$ holds for all consecutive elements. A *partial execution of an expression* E is a partial execution whose first element is congruent to E. An *execution of* E is a *maximal* partial execution of E, i.e. an infinite partial execution, or a finite one whose last element is irreducible. The least transitive relation containing $\rightarrow$ and $\equiv$ is denoted with $\rightarrow^*$. We also define for all $n \geqslant 0$ the relation $\rightarrow^n$ via $\rightarrow^{n+1} \stackrel{\text{def}}{=} \rightarrow \circ \rightarrow^n$ and $\rightarrow^0 \stackrel{\text{def}}{=} \equiv$. Furthermore, we set $\rightarrow^{\leqslant n} \stackrel{\text{def}}{=} \cup_{i=0}^{n} \rightarrow^i$. The *length* of a finite partial execution $(E_i)_{i=0}^{n}$ is n and the length of an infinite execution $(E_i)_{i=0}^{\infty}$ is ∞.

Definition 2.2 (*Complexity*)

The *complexity* $\mathscr{C}(E)$ of an expression E is the least upper bound of the lengths of the executions of E.

$$\mathscr{C}(E) = \sup\{m \mid m \text{ is the length of an execution of } E\}$$

Note that $0 \leqslant \mathscr{C}(E) \leqslant \infty$. Instead of considering complete executions, we may also consider finite partial executions; for all E:

$$\mathscr{C}(E) = \sup\{m \mid m \text{ is the length of a finite partial execution of } E\}$$

In general, distinct executions of the same expression E may have distinct lengths. For instance, consider the calculus with two expressions a and b, the trivial congruence (equal to the set of pairs $\{(a,a),(b,b)\}$), and the reduction given by $a \rightarrow a$ and $a \rightarrow b$. In this calculus, the expression a has executions of arbitrary length greater than 1. For example, the sequence $a \rightarrow a \rightarrow a \rightarrow b$ defines an execution of a of length 3. Note that the considered calculus is confluent because every partial execution of a can be extended such that it terminates with b.

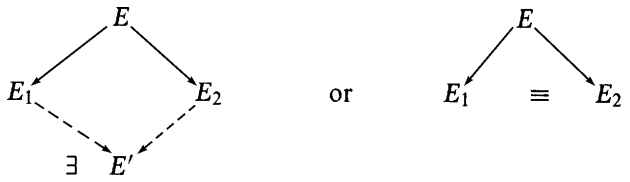
https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

2.2 Uniform confluence

We introduce the notion of uniform confluence and show that every execution of an expression in a uniformly confluent calculus has the same complexity.

Definition 2.3 (*Uniform confluence*)

A calculus is *uniformly confluent* if its reduction $\rightarrow$ and congruence $\equiv$ satisfy the inclusion $(\leftarrow \circ \rightarrow) \subseteq ((\rightarrow \circ \leftarrow) \cup \equiv)$ visualized below.



Proposition 2.4

A uniformly confluent calculus is confluent. For every expression E of a uniformly confluent calculus every execution of E has the same length.

Proof

By a standard inductive argument as for the notion of strong confluence (Huet, 1980) which is implied by uniform confluence. More precisely, we can prove the following property for every expression E, E_1, E_2 and natural numbers m_1 and m_2 by simultaneous induction over m_1 and m_2: If $E_1 \ {}^{m_1}\!\leftarrow E \rightarrow^{m_2} E_2$ then there exists an expression E' and a natural number $m \leqslant \min\{m_1, m_2\}$ such that $E_1 \rightarrow^{m_1-m} E' \rightarrow^{m_2-m} E_2$. □

3 Complexity in unions of calculi

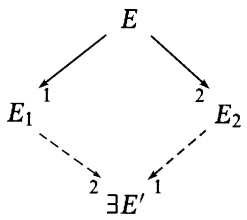
Throughout the paper, we will consider several calculi which are defined as unions of others. This additional structure renders our theory surprisingly rich.

Definition 3.1 (*Union of calculi*)

We define the *union* of two calculi $(\mathscr{E}, \equiv, \rightarrow_1)$ and $(\mathscr{E}, \equiv, \rightarrow_2)$ to be the calculus $(\mathscr{E}, \equiv, \rightarrow_1 \cup \rightarrow_2)$.

3.1 Uniform confluence for unions

Under the assumption of *commutativity*, one can conclude the uniform confluence of a union from the uniform confluence of its components. We say that the relations $\rightarrow_1$ and $\rightarrow_2$ *commute* iff $({}_1\!\leftarrow \circ \rightarrow_2) \subseteq (\rightarrow_2 \circ {}_1\!\leftarrow)$, i.e. if the following diagram can be completed for all E, E_1, E_2.



https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

Lemma 3.2 (*Reformulation of the Hindley–Rosen Lemma*)
The union of uniformly confluent calculi with commuting reductions is uniformly confluent.

Lemma 3.2 implies the classical Hindley–Rosen Lemma (see, for instance, Barendregt, 1981), which states that the reflexive transitive closure of a confluent relation is confluent. This follows from that a relation is confluent if and only if its reflexive transitive closure is uniformly confluent.

We will use sequence notation where we freely omit index bounds if they are not relevant. This means that we may write $(x_j)_j$ for some sequence $(x_j)_{j=k}^{l}$ where k is a natural number and $l \geqslant k$ is a natural number or $l = \infty$.

Definition 3.3 (*i-steps and i-complexity*)
Let $(\mathscr{E}, \equiv, \rightarrow_1 \cup \ldots \cup \rightarrow_n)$ be a union of calculi, $1 \leqslant i \leqslant n$, and $(E_j)_j$ a partial execution. An *i-step in* $(E_j)_j$ is an index k of the sequence $(E_j)_j$ such that $E_k \rightarrow_i E_{k+1}$ (and $k+1$ is also an index of $(E_j)_j$). We define the *i-complexity* $\mathscr{C}_i(E)$ of an expression $E \in \mathscr{E}$ to be the least upper bound of the number of i-steps in a partial execution of E:

$$\mathscr{C}_i(E) = \sup\{m \mid m \text{ is number of } i\text{-steps in a finite partial execution of E}\}$$

Lemma 3.4
Let $(\mathscr{E}, \equiv, \rightarrow_1 \cup \ldots \cup \rightarrow_n)$ be a union of calculi. For all $E \in \mathscr{E}$ and all $1 \leqslant i \leqslant n$: $\mathscr{C}_i(E) \leqslant \mathscr{C}(E)$.

3.2 Additivity and orthogonality

Given the union of two calculi, say $(\mathscr{E}, \equiv, \rightarrow_1 \cup \rightarrow_2)$, one might wish complexity to be additive in that $\mathscr{C}(E) = \mathscr{C}_1(E) + \mathscr{C}_2(E)$ holds for every expression $E \in \mathscr{E}$. In fact, most concrete unions considered in this paper have this property. However, additivity does not hold in general since a 1-steps may at the same time be a 2-steps. But it suffices to assume orthogonality for proving additivity.

Definition 3.5 (*Orthogonality*)

We call a union of calculi $(\mathscr{E}, \equiv, \rightarrow_1 \cup \ldots \cup \rightarrow_n)$ *orthogonal* if for any two expressions $E, E' \in \mathscr{E}$ there exists at most one integer $i \in \{1, \ldots n\}$ such that $E \rightarrow_i E'$.

In an orthogonal union, the length of a partial execution can be obtained by summing up the numbers of its i-steps for all $1 \leqslant i \leqslant n$. This additivity property for i-steps in partial executions can be lifted to an additivity property for the i-complexity $\mathscr{C}_i(E)$ of an expression E.

Lemma 3.6 (*Reduction decreases i-complexity*)
Let $(\mathscr{E}, \equiv, \rightarrow_1 \cup \ldots \cup \rightarrow_n)$ be an orthogonal union of uniformly confluent calculi with commuting reductions, $i, j \in \{1, \ldots, n\}$, $E, E' \in \mathscr{E}$, $E \rightarrow_i E'$, and $i \neq j$. Then:

$$\mathscr{C}_i(E) = 1 + \mathscr{C}_i(E') \qquad \text{and} \qquad \mathscr{C}_j(E) = \mathscr{C}_j(E')$$

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

Proof
If $E \to_i E'$ then every execution of E' can be extended to an execution of E by adding an $\to_i$ step in front. Hence, $\mathscr{C}_i(E) \geqslant 1 + \mathscr{C}_i(E')$ and $\mathscr{C}_j(E) \geqslant \mathscr{C}_j(E')$ follows from orthogonality and $i \neq j$. The converse follows from the following claim which can be proved by induction: For all $F, F' \in \mathscr{E}$ with $F \to_i F'$ and every finite partial executions $(F_k)_k$ of F there exists a finite partial execution $(F'_k)_k$ of F' such that $(F'_k)_k$ with fewer or equally many i-steps and the same number of j-steps. □

Lemma 3.7 (*Finite executions*)
Let $(\mathscr{E}, \equiv, \to_1 \cup \ldots \cup \to_n)$ be an orthogonal union of uniformly confluent calculi with commuting reductions. If $E \in \mathscr{E}$ satisfies $\mathscr{C}(E) < \infty$ and $1 \leqslant i \leqslant n$ then the number of i-steps coincides for all executions of E.

Proof
First note that Lemma 3.4 implies $\sum_{i=1}^{n} \mathscr{C}_i(E) \leqslant \sum_{i=1}^{n} \mathscr{C}(E) = n * \mathscr{C}(E) < \infty$. Lemma 3.7 follows from Lemma 3.6 which permits to show the following claim by induction on the value of $\sum_{i=1}^{n} \mathscr{C}_i(E)$ (which is distinct from ∞): For all $E \in \mathscr{E}$ and $1 \leqslant j \leqslant n$, if $\sum_{i=1}^{n} \mathscr{C}_i(E) < \infty$ then every executions of E contains the same number of j-steps. □

The reader should notice carefully that Lemma 3.7 fails for expressions E with infinite executions. For illustration, we consider the following calculus: Its expressions are pairs of natural numbers (n, m); its congruence is the equality of expressions, and its reduction is the union $\to_1 \cup \to_2$ where $(n, m) \to_1 (n+1, m)$ and $(n, m) \to_2 (n, m+1)$ for all n, m. This calculus is orthogonal but for each of its expressions there exists an execution with arbitrary numbers of 1-steps and 2-steps. For instance, the following executions are possible:

$$(0, 0) \to_1 (1, 0) \to_1 (2, 0) \to_1 \ldots$$
$$(0, 0) \to_2 (0, 1) \to_2 (0, 2) \to_2 \ldots$$

The first execution contains infinitely many 1-steps and no 2-steps, whereas the second execution contains no 1-step and infinitely many 2-steps. Both of these executions are unfair: For instance, every initial segment of the first execution could be continued with an 2-step but this never happens.

Proposition 3.8 (*Additivity*)

Let $(\mathscr{E}, \equiv, \to_1 \cup \ldots \cup \to_n)$ be an orthogonal union of uniformly confluent calculi with commuting reductions. For every expression E of $\mathscr{E}$, complexity is additive:

$$\mathscr{C}(E) = \mathscr{C}_1(E) + \ldots + \mathscr{C}_n(E)$$

Proof
If $\mathscr{C}(E) = \infty$ then there exists $1 \leqslant i \leqslant n$ and an execution of E which contains an infinite number of i-steps. Hence, $\mathscr{C}_i(E) = \infty$ such that $\sum_{i=1}^{n} \mathscr{C}_i(E) = \infty$. Otherwise, $\mathscr{C}(E) < \infty$. In this case, Lemma 3.7 implies for every $1 \leqslant i \leqslant n$ that every execution of E contains the same number of i-steps. Additivity for executions in orthogonal unions implies for every execution $(E_j)_j$ of E that the length of $(E_j)_j$ equals to the sum of the numbers of i-steps in $(E_j)_j$ where $1 \leqslant i \leqslant n$. Since all executions of E

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

contain the same numbers of i-steps (Lemma 3.7), additivity lifts from executions of E to the expression E itself. □

4 Embeddings and simulations

We define the notion of an embedding between two calculi and present a method for proving that an embedding preserves complexity. This method is based on simulations rather than bisimulation, but its applicability requires uniform confluence. Otherwise, simulations do not necessarily preserve preserve complexity, in contrast to bisimulations (Milner, 1992; Sangiorgi, 1996; Turner, 1996; Nestmann and Pierce, 1996).

Definition 4.1 (*Embedding*)
Let $(\mathscr{E}, \equiv_{\mathscr{E}}, \rightarrow_{\mathscr{E}})$ and $(\mathscr{F}, \equiv_{\mathscr{F}}, \rightarrow_{\mathscr{F}})$ be two calculi, $\Phi : \mathscr{E} \rightarrow \mathscr{F}$ a function, and $S \subseteq \mathscr{E} \times \mathscr{F}$ be a binary relation. We call Φ an *embedding of* $\mathscr{E}$ *into* $\mathscr{F}$ if $E_1 \equiv_{\mathscr{E}} E_2$ implies $\Phi(E_1) \equiv_{\mathscr{F}} \Phi(E_2)$ for all $E_1, E_2 \in \mathscr{E}$. The function Φ is an *embedding for* S if it is an embedding that satisfies (S1) below:

(S1) For all $E \in \mathscr{E}$: $(E, \Phi(E)) \in S$.

We formulate our theory for simulations between *unions of calculi*. Our results for simulations carry easily over to encodings provided that (S1) is assumed.

4.1 Complexity simulations

We introduce lengthening simulations which relate expressions with smaller complexity to expressions with higher complexity, and complexity simulations which preserve complexity. Lengthening simulations are of there own interest. For instance, the identity relation on λ-expressions can be seen as a simulation which lengthens call-by-need to call-by-value.

Definition 4.2 (*Lengthening and complexity simulation*)
Let $S \subseteq \mathscr{E} \times \mathscr{F}$ be a binary relation between the expressions of two calculi $(\mathscr{E}, \equiv_{\mathscr{E}}, \rightarrow_1 \cup \ldots \cup \rightarrow_n)$ and $(\mathscr{F}, \equiv_{\mathscr{F}}, \hookrightarrow)$. We define $\approx \subseteq \mathscr{F} \times \mathscr{F}$ such that

$$F \approx F' \qquad \text{iff} \qquad \mathscr{C}(F) = \mathscr{C}(F')$$

for all $F, F' \in \mathscr{F}$: Given natural numbers $m_1, \ldots, m_n \geqslant 0$, we call S a *lengthening simulation* with *indices* $m_1, \ldots, m_n$ if S satisfies (S2) and (S3) below.

(S2) For all $E, E' \in \mathscr{E}$, $F \in \mathscr{F}$, and $1 \leqslant i \leqslant n$: If $E \rightarrow_i E'$ and $(E, F) \in S$ then there exists an expression $F' \in \mathscr{F}$ such that $(E', F') \in S$ and $F \approx \circ\, (\hookrightarrow \circ \approx)^{m_i} F'$.

$$\begin{array}{ccc} E & \rightarrow_i & E' \\ S & & S \\ F & \approx \circ\, (\hookrightarrow \circ \approx)^{m_i} & \exists F' \end{array}$$

(S3) For every $E \in \mathscr{E}$ with $\mathscr{C}(E) = \infty$ there exists $1 \leqslant i \leqslant n$ such that $\mathscr{C}_i(E) = \infty$ and $m_i \geqslant 1$.

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

We call S a *complexity simulation* with *indices* $m_1,\ldots,m_n$ if S is a lengthening simulation with indices $m_1,\ldots,m_n$, which additionally satisfies the condition (S4).

(S4) For all $E \in \mathscr{E}$ and $F \in \mathscr{F}$: If E is irreducible with respect to $\to_1 \cup \ldots \cup \to_n$ and $(E, F) \in S$ then F is irreducible with respect to $\hookrightarrow$.

Property (S2) is most important. It requires that every $\to_i$ step can be simulated by m_i steps with $\hookrightarrow$ up to an equivalence $\approx$ which preserves complexity. If we would require $m_i \geqslant 1$ for all i then Property (S2) would clearly imply that a lengthening simulation relates expressions with smaller complexity to expressions with higher complexity. However, this property would be too restrictive. The slightly weaker Property (S3) turns out to be precisely what we need. Note that (S3) expresses a property of S even though S does not occur in it. But S depends on the indices m_i which occur in both (S2) and (S3) . Property (S4) requires that S preserves termination. If (S4) holds then S cannot strictly increase complexity.

Proposition 4.3 (*Lengthening or preserving complexity*)

Let $(\mathscr{E}, \equiv_{\mathscr{E}}, \to_1 \cup \ldots \to_n)$ be an orthogonal union with commuting reductions and $(\mathscr{F}, \equiv_{\mathscr{F}}, \hookrightarrow)$ another calculus. If $S \subseteq \mathscr{E} \times \mathscr{F}$ is a lengthening simulation with indices $m_1,\ldots,m_n$ then the following inequation holds for all $(E,F) \in S$:

$$\mathscr{C}(F) \;\geqslant\; \sum_{i=1}^{n} m_i * \mathscr{C}_i(E)$$

If S is also a complexity simulation with indices $m_1,\ldots,m_n$ then equality holds.

Lemma 4.4 (*Quotients*)

Let $(\mathscr{F}, \equiv_{\mathscr{F}}, \hookrightarrow_1 \cup \hookrightarrow_2)$ be an orthogonal union of uniformly confluent calculi with commuting reductions, $\equiv_2$ the relation $\overset{*}{\hookrightarrow_2} \circ \overset{*}{{}_2\!\hookleftarrow}$, and $\mathscr{G}$ the triple:

$$\mathscr{G} = (\mathscr{F}, \equiv_2, \equiv_2 \circ \hookrightarrow_1 \circ \equiv_2)$$

Then $\mathscr{G}$ is a uniformly confluent calculus whose complexity measure (denoted by $\mathscr{C}^{\mathscr{G}}$ in order to distinguish it from the complexity measure $\mathscr{C}$ of $\mathscr{F}$) satisfies $\mathscr{C}^{\mathscr{G}}(F) = \mathscr{C}_1(F)$ for all $F \in \mathscr{F}$.

Proof

The relation $\equiv_2$ is an equivalence relation since $\overset{*}{\hookrightarrow_2}$ is confluent (Lemma 3.2). In fact, $\mathscr{G}$ is a calculus since its reduction is modulo its congruence. It is also not difficult to verify that $\mathscr{G}$ is uniformly confluent. □

Proposition 4.5 (*Simple lengthening simulations*)

Let $(\mathscr{E}, \equiv_{\mathscr{E}}, \to_1 \cup \ldots \cup \to_n)$ be an orthogonal union of uniformly confluent calculi with pairwise commuting reductions and $S \subseteq \mathscr{E} \times \mathscr{E}$ a relation such that the following diagram can be completed for all $2 \leqslant i \leqslant n$ and $E, E', F \in \mathscr{E}$:

$$\begin{array}{lll} E & \to_1 & E' \\ S & & S \\ F & \to_1 & \exists F' \end{array} \qquad \begin{array}{lll} E & \to_i & E' \\ S & & S \\ F & \to_i^* & \exists F' \end{array}$$

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

Furthermore, we assume for all $E \in \mathscr{E}$ that $\mathscr{C}(E) = \infty$ implies $\mathscr{C}_1(E) = \infty$. In this case, the equation $\mathscr{C}_1(E) \leqslant \mathscr{C}_1(F)$ holds for all $(E, F) \in S$.

Proof

If $n = 1$ then S is a lengthening simulation between the calculus $\mathscr{E}$ and itself with index 1. According to Lemma 4.3, $\mathscr{C}_1(E) = \mathscr{C}_1(F)$ holds all $(E, F) \in S$. We next suppose $n \geqslant 2$ and define $\hookrightarrow_1 = \rightarrow_1$ and $\hookrightarrow_2 = \cup_{j=2}^{n} \rightarrow_j$. Since $n \geqslant 2$, the triple $(\mathscr{F}, \equiv_{\mathscr{F}}, \hookrightarrow_1 \cup \hookrightarrow_2)$ is an orthogonal union of uniformly confluent calculi with commuting reductions. Let $\equiv_2$ be the relation $\hookrightarrow_2^* \cup {}_2^*\!\hookleftarrow$ and $\mathscr{G}$ the auxiliary calculus:

$$\mathscr{G} = (\mathscr{E}, \equiv_2, \equiv_2 \circ \hookrightarrow_1 \circ \equiv_2)$$

It is not difficult to show that S is a lengthening simulation between the calculi $\mathscr{E}$ and $\mathscr{G}$ with indexes $1, 0 \ldots, 0$: Condition (S2) follows from the required diagrams and Lemma 3.6 which shows $\hookrightarrow_2 \subseteq \approx$, and (S3) follows from the assumption that $C(E) = \infty$ implies $C_1(E) = \infty$. Since $\mathscr{E}$ and $\mathscr{G}$ are uniformly confluent (Lemma 4.4), Lemma 4.3 yields for all $(E, F) \in S$ that $\mathscr{C}^{\mathscr{G}}(F) = \mathscr{C}_1(E)$. Lemma 4.4 implies $\mathscr{C}^{\mathscr{G}}(F) = \mathscr{C}_1(F)$ such that Proposition 4.5 follows. □

The following Proposition 4.6 is similar to Proposition 4.5 except that it does not require an assumption on infinite computations.

Proposition 4.6 (*Simple complexity simulations*)

Let $(\mathscr{E}, \equiv_{\mathscr{E}}, \rightarrow_1 \cup \ldots \cup \rightarrow_n)$ be an orthogonal union of uniformly confluent calculi with pairwise commuting reductions and $S \subseteq \mathscr{E} \times \mathscr{E}$ a relation. Then $\mathscr{C}_i(E) = \mathscr{C}_i(F)$ holds for all $1 \leqslant i \leqslant n$ and $(E, F) \in S$ if the following diagram can be completed for all $1 \leqslant i \leqslant n$ and $E, E', F \in \mathscr{E}$:

$$\begin{array}{ccc} E & \rightarrow_i & E' \\ S & & S \\ F & \rightarrow_i & \exists F' \end{array}$$

4.2 Administrative simulations

We finally consider a refined form of complexity simulation, which allow us to take administrative steps into account.

Definition 4.7 (*Administrative simulation*)

Let ϕ be an embedding between two calculi $(\mathscr{E}, \equiv_{\mathscr{E}}, \rightarrow)$ and $(\mathscr{F}, \equiv_{\mathscr{F}}, \hookrightarrow_1 \cup \hookrightarrow_2)$. Given natural numbers $n_1, n_2 \geqslant 0$, we call a relation S on $\mathscr{E} \times \mathscr{F}$ an *administrative simulation* for Φ with *administrative reduction* $\hookrightarrow_2$ and *administrative indices* n_1, n_2, if S satisfies the following properties (A1) to (A5):

(A1) For all $E \in \mathscr{E}$: $(E, \Phi(E)) \in S$.

(A2) For all $E, E' \in \mathscr{E}$ and $F \in \mathscr{F}$: If $E \rightarrow E'$ and $(E, F) \in S$ then there exists

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

$F' \in \mathscr{F}$ such that $(E', F') \in S$ and $F \hookrightarrow_2^{\leqslant n_2} \circ \hookrightarrow_1^{n_1} F'$.

$$\begin{array}{lll} E & \rightarrow & E' \\ S & & S \\ F & \hookrightarrow_2^{\leqslant n_2} \circ \hookrightarrow_1^{n_1} & \exists F' \end{array}$$

(A3) For the first administrative index n_1 it holds that $n_1 \geqslant 1$.

(A4) For all $E \in \mathscr{E}$ and $F \in \mathscr{F}$: If E is irreducible with respect to $\rightarrow$ and $(E, F) \in S$ then $\mathscr{C}_1(F) = 0$ and $\mathscr{C}_2(F) \leqslant n_2$.

(A5) For all $E \in \mathscr{E}$: $\Phi(E)$ is irreducible with respect to $\hookrightarrow_2$.

Lemma 4.8

Let Φ be an embedding between an uniformly confluent calculus $(\mathscr{E}, \equiv_{\mathscr{E}}, \rightarrow)$ and an orthogonal union of uniformly confluent calculi $(\mathscr{F}, \equiv_{\mathscr{F}}, \hookrightarrow_1 \cup \hookrightarrow_2)$ with commuting reductions. If there exists an administrative simulation for Φ with administrative reduction $\hookrightarrow_2$ and indices n_1 and n_2 then the two following properties hold:

1. For all $E \in \mathscr{E}$: $\mathscr{C}_1(\Phi(E)) = n_1 * \mathscr{C}(E)$.
2. For all $E \in \mathscr{E}$ with $\mathscr{C}(E) < \infty$: $\mathscr{C}_2(\Phi(E)) \leqslant n_2 * \mathscr{C}(E)$.

Proof

To establish the first statement of the lemma, we let $\equiv_2$ be $\hookrightarrow_2^* \circ {}_2^*\hookleftarrow$ and define an auxiliary calculus $\mathscr{G}$ by $(\mathscr{F}, \equiv_2, \equiv_2 \circ \hookrightarrow_1 \circ \equiv_2)$. According to Lemma 4.4, $\mathscr{G}$ is a uniformly confluent calculus. We consider Φ as an embedding from the calculus $\mathscr{E}$ into the auxiliary calculus $\mathscr{G}$ rather than the initial calculus $\mathscr{F}$. It is easy to show that S is a complexity simulation for this embedding with index n_1 since for all $1 \leqslant i \leqslant 4$, condition (Ai) easily implies (Si). Therefore, we can apply Proposition 4.3 which proves for all $E \in \mathscr{E}$:

$$\mathscr{C}^{\mathscr{G}}(\Phi(E)) = n_1 * \mathscr{C}(E)$$

Here again, we write $\mathscr{C}^{\mathscr{G}}$ instead of $\mathscr{C}$ in order to distinguish the complexity in the calculus $\mathscr{G}$ from that in $\mathscr{F}$. Lemma 4.4 implies $\mathscr{C}^{\mathscr{G}}(\Phi(E)) = \mathscr{C}_1(\Phi(E))$ such that property 1 of Lemma 4.8 follows.

We next prove $\mathscr{C}_2(F) \leqslant n_2 * \mathscr{C}(E) + n_2$ for all $(E, F) \in S$ with $\mathscr{C}(E) < \infty$ by induction on $\mathscr{C}(E)$. If $\mathscr{C}(E) = 0$ then E is irreducible. According to (A4), $\mathscr{C}_2(F) \leqslant n_2$ holds. We next consider the case $\mathscr{C}(E) \geqslant 1$ in which there exists E' with $E \rightarrow E'$. Lemma 3.6 implies $\mathscr{C}(E) = 1 + \mathscr{C}(E')$. Condition (A2) yields the existence of an expression $F' \in \mathscr{F}$ and $n \leqslant n_2$ which satisfies $(E', F') \in S$ and $F \hookrightarrow_2^n \circ \hookrightarrow_1^{n_1} F'$. Since $\mathscr{C}(E') < \mathscr{C}(E)$ we can apply the induction hypothesis to the pair (E', F'). This yields:

$$\begin{array}{lll} \mathscr{C}_2(F) = n + \mathscr{C}_2(F') & & \text{(Lemma 3.6)} \\ \leqslant n_2 + n_2 * \mathscr{C}(E') + n_2 & & \text{(Induction Hypothesis)} \\ = n_2 * \mathscr{C}(E) + n_2 & & \text{(Lemma 3.6)} \end{array}$$

We now prove $\mathscr{C}_2(\Phi(E)) \leqslant n_2 * \mathscr{C}(E)$ for all $E \in \mathscr{E}$. If $\mathscr{C}(E) = 0$ then $\mathscr{C}_2(\Phi(E)) = 0$ follows from (A5) and (A4). If $\mathscr{C}(E) \geqslant 1$ then there exists E' such that $E \rightarrow E'$. According to (A1), $(E, \Phi(E)) \in S$. Since $\Phi(E)$ is irreducible with respect to $\hookrightarrow_2$ due

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

to (A5), (A2) yields the existence of F' with $\Phi(E) \hookrightarrow_1^{n_1} F'$ and $(E', F') \in S$. Hence:

$$\begin{array}{rcll} \mathscr{C}_2(F) & = & \mathscr{C}_2(F') & \text{(Lemma 3.6)} \\ & \leqslant & n_2 * \mathscr{C}(E') + n_2 & \text{(as shown above)} \\ & = & n_2 * \mathscr{C}(E) & \text{(Lemma 3.6)} \end{array}$$

□

Proposition 4.9 (*Counting administrative steps*)

Let Φ be an embedding between a uniformly confluent calculus $(\mathscr{E}, \equiv_{\mathscr{E}}, \rightarrow)$ and an orthogonal union of uniformly confluent calculi $(\mathscr{F}, \equiv_{\mathscr{F}}, \hookrightarrow_1 \cup \hookrightarrow_2)$ with commuting reductions. If there exists an administrative simulation for Φ with administrative reduction $\hookrightarrow_2$ and indices n_1 and n_2 then Φ preserves complexity up to a constant factor. For all $E \in \mathscr{E}$:

$$n_1 * \mathscr{C}(E) \;=\; \mathscr{C}_1(\Phi(E)) \;\leqslant\; \mathscr{C}(\Phi(E)) \;\leqslant\; (n_1 + n_2) * \mathscr{C}(E)$$

Proof

The equation $n_1 * \mathscr{C}(E) = \mathscr{C}_1(\Phi(E))$ follows from Lemma 4.8 part 1 It remains to be shown for all $E \in \mathscr{E}$ that:

$$n_1 * \mathscr{C}(E) \;\leqslant\; \mathscr{C}(\Phi(E)) \;\leqslant\; (n_1 + n_2) * \mathscr{C}(E)$$

If $\mathscr{C}(E) = \infty$ then $\mathscr{C}_1(E) = \infty$ and thus $\mathscr{C}_1(\Phi(E)) = \infty$ as shown above. In this case, all three terms in the considered estimation evaluate to ∞ since $n_1 \geqslant 1$. For all $E \in \mathscr{E}$ with $\mathscr{C}(E) < \infty$ we have:

$$\begin{array}{rcll} \mathscr{C}(\Phi(E)) & = & \mathscr{C}_1(\Phi(E)) + \mathscr{C}_2(\Phi(E)) & \text{Additivity (Proposition 3.8)} \\ & \leqslant & n_1 * \mathscr{C}(E) + n_2 * \mathscr{C}(E) & \text{Lemma 4.8 parts 1 and 2} \\ & = & (n_1 + n_2) * \mathscr{C}(E) & \\ \mathscr{C}(\Phi(E)) & \geqslant & \mathscr{C}_1(\Phi(E)) & \text{Lemma 3.4} \\ & = & n_1 * \mathscr{C}(E) & \text{Lemma 4.8 part 1} \end{array}$$

□

5 Concurrent computation

For studying uniform confluence in concurrent computation, we investigate the applicative core of the π-calculus that we call π_0. We first define π_0 and its admissible expressions and show that the restriction of π_0 to admissible expressions is uniformly confluent. We then extend π_0 to the δ-calculus by adding two mechanisms for forwarding and triggering. We present a criterion for proving admissibility based on a linear type system.

5.1 The applicative core π_0 of the π-calculus

We recall the applicative core of the polyadic asynchronous π-calculus (Milner, 1991; Honda and Tokoro, 1991; Boudol, 1992).

As any other calculus, we define π_0 in terms of a set of expressions, a structural

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

Expressions	$E, F \ ::= \ x{:}\bar{y}/E \ \mid \ x\bar{y} \ \mid \ E \mid F \ \mid \ (\nu x)E$
Reduction	$x{:}\bar{y}/E \mid x\bar{z} \ \rightarrow_A \ x{:}\bar{y}/E \mid E[\bar{z}/\bar{y}]$
Contexts	$\dfrac{E \rightarrow_A E'}{E \mid F \rightarrow_A E' \mid F} \qquad \dfrac{E \rightarrow_A E'}{(\nu x)E \rightarrow_A (\nu x)E'}$
Congruence	$\dfrac{E_1 \equiv E_2 \quad E_2 \rightarrow_A F_2 \quad F_2 \equiv F_1}{E_1 \rightarrow_A F_1}$

Fig. 1. The applicative core π_0 of the π-calculus.

congruence, and a reduction relation. The definition is given in figure 1. Expressions of π_0 are built from variables ranged over by x, y, z. Possibly empty sequences of variables are denoted with $\bar{x}$, $\bar{y}$, $\bar{z}$. An expression E of π_0 is either an abstraction, an application, a concurrent composition, or a declaration.

An (*named*) *abstraction* $x{:}\bar{y}/E$ is named by x, has *formal arguments* $\bar{y}$ and *body* E. An *application* $x\bar{y}$ of x has *actual arguments* $\bar{y}$. A *composition* $E \mid F$ composes two concurrent processes E and F in interleaving manner. A *declaration* $(\nu x)E$ declares a new variable x with scope E. *Bound variables* are introduced as formal arguments of abstractions and by declaration. The set of free variables of an expression E is denoted by $\mathscr{V}(E)$.

The precedence of the syntactic constructors in expressions E is as follows: Declaration binds stronger than abstraction which binds stronger than composition. For instance, the expression $x{:}y/(\nu z)yz \mid z{:}y/yy$ reads as $(x{:}y/((\nu z)yz)) \mid z{:}y/yy$. We identify expressions up to consistent renaming of bound variables. When being very precise, we have to distinguish expressions and their representatives. We do so when needed only but are sloppy in most cases. However, we do apply Barendregt's hygiene condition (Barendregt, 1981) which requires that all considered representatives are α-standardized, i.e. that bound and free variables are always distinct.

As presented, the syntax of π_0 is borrowed from the ρ-calculus (Niehren and Müller, 1995) rather than from the π-calculus. In the terminology of the π-calculus as in Kobayashi *et al.* (1996), an abstraction $x{:}\bar{y}/E$ is called a replicated input agent $x?^*(\bar{y}).E$ and an application $x\bar{y}$ an output agent $x!(\bar{y})$. In comparison to the polyadic asynchronous π-calculus only once-only input agents $x?(\bar{y}).E$ are omitted.

The congruence of π_0 is well-known from the π-calculus. It is the least congruence on expressions of π_0 which renders composition associative and commutative and provides for the usual scoping rules for declaration:

$$E \mid F \ \equiv \ F \mid E \qquad E_1 \mid (E_2 \mid E_3) \ \equiv \ (E_1 \mid E_2) \mid E_3$$

$$(\nu x)(\nu y)E \ \equiv \ (\nu y)(\nu x)E \qquad (\nu x)E \mid F \ \equiv \ (\nu x)(E \mid F) \qquad \text{if } x \notin \mathscr{V}(F)$$

The *reduction* $\rightarrow_A$ of π_0 is essentially given by a single reduction rule for executing applications in figure 1. This rule is formulated in terms of the simultaneous substitution operator $[\bar{z}/\bar{y}]$ which replaces variables in $\bar{y}$ elementwise by variables in $\bar{z}$. When applying the operator $[\bar{z}/\bar{y}]$ we implicitly assume that the sequence $\bar{y}$ is linear and of the same length as $\bar{z}$. Reduction $\rightarrow_A$ can be performed in weak

contexts, i.e. inside of declarations and compositions but not inside of abstractions. Furthermore, reduction is modulo congruence, i.e. a reduction step of E can be performed on any expression congruent to E.

Example 5.1 (*Explicit recursion*)
The execution of the following expression is infinite since the application of x is recursive. The fact that we are able to express recursion in π_0 gives a first hint for that π_0 might be quite expressive.

$$xy \mid x{:}y/xy \;\rightarrow_A\; xy \mid x{:}y/xy \;\rightarrow_A\; \ldots$$

Definition 5.2 (*Consistency and admissibility*)

We call an expression E of π_0 *consistent* if it does not contain two non-congruent abstractions with the same name; more formally, for all subexpressions E' of some α-standardized representative of E and for all $x \in \mathscr{V}(E')$ if two abstractions are contained in E', say $x{:}\bar{y}/E_1$ and $x{:}\bar{z}/E_2$, then they are congruent:

$$x{:}\bar{y}/E_1 \equiv x{:}\bar{z}/E_2$$

We call E *admissible* if every expression E' such that $E \rightarrow^* E'$ is consistent.

Example 5.3
We assume $z_1 \neq z_2$. The expression E_1 equal to $x{:}y/z_1y \mid x{:}y/z_2y$ is not consistent since $x{:}y/z_1y \not\equiv x{:}y/z_2y$ due to $z_1 \neq z_2$. Similarly, $x{:}y/(z_1y \mid x{:}y/z_2y)$ is not consistent for $x \neq y$. In contrast, $x{:}y/z_1y \mid (\nu x)(x{:}y/z_2y)$ is consistent since we identify expressions modulo consistent renaming of bound variables. For every α-standardized representative of this expression (for instance $x{:}y/z_1y \mid (\nu z)(z{:}y/z_2y)$) contains a unique abstraction with name x. The expression E_2 equal to $z'{:}z/x{:}y/zy \mid z'z_1 \mid z'z_1$ is consistent by not admissible; The problem is that E_2 may reduce to E_1 which is not consistent. Note that all expressions of the form $x{:}\bar{y}/E \mid x{:}\bar{y}/E$ are consistent.

Example 5.4 (*Non-confluence*)
Typically, non-consistency may raise non-confluence. For illustration, we consider the expression $xz \mid E_1$ where E_1 is given in the previous example:

$$z_1z \mid E_1 \;{}_A\!\leftarrow\; xz \mid E_1 \rightarrow_A z_2z \mid E_1$$

The resulting expressions $z_1z \mid E_1$ and $z_2z \mid E_1$ are irreducible but not congruent if we assume $z_1 \neq z_2$.

The advantage of the notion of admissibility is that it is preserved by reduction and nevertheless very simple to define. It also allows for a simple reasoning about confluence. Unfortunately, the notion of admissibility is not always easy to deal with. For instance, it is undecidable whether a given expression E is admissible (since admissibility depends on termination in a Turing complete system). This failure is harmless for our purpose. The reason is that there exists a simple type

system that allows to test for admissibility for all expressions we are interested in. This system will be presented in section 5.3.

Theorem 5.5 (*Uniform confluence*)

The restriction of π_0 to admissible expressions is uniformly confluent.

Proof

We first reformulate reduction of π_0 based on the notion of reduction contexts. A *reduction context* D of the π_0 is given by the following abstract syntax:

$$D \ ::= \ [\,] \ \mid \ D \mid E \ \mid \ E \mid D \ \mid \ \nu(x)D$$

We write $D[E]$ for the expression obtained by replacing the hole $[\,]$ in the context D with E, and $D[D']$ for the context obtained by substitution $[\,]$ in D with D'. We can show for all α-standardized representatives E of expressions of π_0 and all E':

$$E \to_A E' \quad \text{iff} \quad \begin{cases} \text{exists } D, \bar{y} \text{ and } x{:}\overline{z}/F \in D \text{ such that} \\ E = D[x\bar{y}] \text{ and } E' \equiv D[F[\overline{z}/\bar{y}]] \end{cases}$$

Let E be an admissible and α-standardized expression of π_0 and E_1, E_2 such that $E_1 \ {}_A\!\leftarrow E \to_A E_2$. There exist D_1, $\overline{y_1}$, $x_1{:}\overline{z_1}/F_1 \in D_1$ and D_2, $\overline{y_2}$, $x_2{:}\overline{z_2}/F_2 \in D_2$ satisfying:

$$E_1 \equiv D_1[F_1[\overline{y_1}/\overline{z_1}]] \ {}_A\!\leftarrow D_1[x_1\overline{y_1}] = E = D_2[x_2\overline{y_2}] \to_A D_2[F_2[\overline{y_2}/\overline{z_2}]] \equiv E_2$$

We consider two cases depending on whether $D_1 = D_2$ holds or not.

1. For $D_1 = D_2$ the equation $D_1[x_1\overline{y_1}] = D_2[x_2\overline{y_2}]$ implies $x_1 = x_2$ and $\overline{y_1} = \overline{y_2}$. Since E is admissible and contains abstractions $x_1{:}\overline{z_1}/F_1$ and $x_2{:}\overline{z_2}/F_2$ we know that these abstractions are congruent: $x_1{:}\overline{z_1}/F_1 \equiv x_2{:}\overline{z_2}/F_2$. Hence $F_1[\overline{y_1}/\overline{z_1}] \equiv F_2[\overline{y_2}/\overline{z_2}]$ which proves $E_1 \equiv E_2$ as required.
2. If $D_1 \neq D_2$ then the condition $D_1[x_1\overline{y_1}] = D_2[x_2\overline{y_2}]$ implies that there exist D_0, D_1' and D_2' such that:

$$\begin{array}{lllll} & D_1 = D_0[D_1' \mid D_2'[x_2\overline{y_2}]] & \textit{and} & D_2 = D_0[D_1'[x_1\overline{y_1}] \mid D_2'] \\ \text{or} & D_1 = D_0[D_2'[x_2\overline{y_2}] \mid D_1'] & \textit{and} & D_2 = D_0[D_2' \mid D_1'[x_1\overline{y_1}]] \end{array}$$

   The second possibility above is symmetric up to commutativity of composition. We therefore only consider the first one. If F is the expression $D_0[D_1'[F_1[\overline{y_1}/\overline{z_1}]] \mid D_2'[F_2[\overline{y_2}/\overline{z_2}]]]$ then $E_1 \to_A F \ {}_A\!\leftarrow E_2$ follows from:

$$\begin{array}{l} E_1 \equiv D_0[D_1'[F_1[\overline{y_1}/\overline{z_1}]] \mid D_2'[x_2\overline{y_2}]] \to_A F \\ E_2 \equiv D_0[D_1'[x_1\overline{y_1}] \mid D_2'[F_2[\overline{y_2}/\overline{z_2}]]] \to_A F \end{array}$$

□

An alternative method for proving the uniform confluence of π_0 has been applied Niehren (1994). There reduction in π_0 is considered as rewriting modulo associativity and commutativity; the idea is simply to identify an expression of π_0 modulo congruence with a pair of a declaration prefix and a multisets of applications and abstractions, and then to redefine reduction for such pairs.

Expressions	$E,F \;::=\; x{:}\bar{y}/E \;\mid\; x\bar{y} \;\mid\; E \mid F \;\mid\; (\nu x)E \;\mid$
	$x{=}y \;\mid\; \mathbf{tr}(x) \;\mid\; x.E$
Reduction	$\rightarrow \;=\; \rightarrow_A \cup \rightarrow_F \cup \rightarrow_T$
	$x{:}\bar{y}/E \mid x\bar{z} \;\rightarrow_A\; x{:}\bar{y}/E \mid E[\bar{z}/\bar{y}]$
	$x{=}y \mid y{:}\bar{z}/E \;\rightarrow_F\; x{:}\bar{z}/E \mid y{:}\bar{z}/E$
	$\mathbf{tr}(x) \mid x.E \;\rightarrow_T\; \mathbf{tr}(x) \mid E$
Contexts	$\dfrac{E \rightarrow E'}{E \mid F \rightarrow E' \mid F} \qquad \dfrac{E \rightarrow E'}{(\nu x)E \rightarrow (\nu x)E'}$
Congruence	$\dfrac{E_1 \equiv E_2 \quad E_2 \rightarrow F_2 \quad F_2 \equiv F_1}{E_1 \rightarrow F_1}$

Fig. 2. The δ-calculus.

5.2 *The δ-calculus: forwarding and triggering*

We define the *δ-calculus* as an extension of π_0 with forwarding and triggering. Such an extension is not strictly needed from the viewpoint of expressiveness. The δ-calculus can be embedded into π_0 such that complexity is preserved (Niehren, 1999). Note that the embedding presented in Niehren (1996) does not have this property. The main purpose of the δ-calculus is to simplify reasoning on functional computation in π_0. Another purpose of the δ-calculus is to emphasize a concurrent constraint view on functional computation.

The δ-calculus extends π_0 with two mechanisms each of which is defined by a single reduction rule. The first mechanism provides a direct method for forwarding abstractions and the second one for triggering the execution of delayed expressions.

The δ-calculus is presented in figure 2. Its expressions extend those of π_0 with three new forms. A *forwarder* $x{=}y$ is used for forwarding an abstraction from y on the right to x on the left. A *delay expression* $x.E$ delays the execution of E until x is triggered. A *trigger expression* $\mathbf{tr}(x)$ triggers the execution of all expressions delayed by x.

The congruence of the δ-calculus is defined by the same equations as the congruence of π_0. The reduction $\rightarrow$ of the δ-calculus is the union of three relations, application $\rightarrow_A$, *forwarding* $\rightarrow_F$, and *triggering* $\rightarrow_T$. Each of these reductions is defined by a corresponding reduction rule. Similarly to reduction in π_0, reduction in the δ-calculus is closed under weak context and modulo congruence.

Example 5.6 (*Forwarding*)
The identity with name x can be written as $x{:}yz/z{=}y$. An execution of xxz' in the context of this expression turns z' into a name of the identity, too.

$$\begin{aligned} x{:}yz/z{=}y \mid xxz' \;&\rightarrow_A\; x{:}yz/z{=}y \mid z'{=}x \\ &\rightarrow_F\; x{:}yz/z{=}y \mid z'{:}yz/z{=}y \end{aligned}$$

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

Example 5.7 (*Triggering*)
The execution of the following expression illustrates how application $\rightarrow_A$ and triggering $\rightarrow_A$ may interact.

$$\begin{array}{lll} x{:}y/\mathbf{tr}(y) \mid xy \mid y.E & \rightarrow_A & x{:}y/\mathbf{tr}(y) \mid \mathbf{tr}(y) \mid y.E \\ & \rightarrow_T & x{:}y/\mathbf{tr}(y) \mid \mathbf{tr}(y) \mid E \end{array}$$

At the beginning the expression E is delayed. The application step creates a trigger expression $\mathbf{tr}(y)$ whose execution wakes up the delayed expression E.

Example 5.8 (*Multiple triggering*)
Multiple triggering has the same effect as once-only triggering. For instance, $\mathbf{tr}(x) \mid \mathbf{tr}(x) \mid x.E$ and $\mathbf{tr}(x) \mid x.E$ reduce in the same manner. Multiple triggering occurs naturally when expressing call-by-need control in a concurrent calculus. There, the execution of a functional argument is triggered once it is needed. Multiple requests of the same functional argument lead to multiple triggering.

The notions of *consistency* and *admissibility* (Definition 5.2) carry over literally from expressions of π_0 to expressions of the δ-calculus. For example $\mathbf{tr}(x) \mid \mathbf{tr}(x)$ and $x.x{:}y/\mathbf{tr}(y) \mid \mathbf{tr}(x)$ are admissible whereas $x{:}y/z{=}y \mid xx \mid z{:}y/zy$ is not admissible.

Proposition 5.9
The δ-calculus restricted to admissible expressions is an orthogonal union of uniformly confluent calculi with commuting reductions $\rightarrow_A$, $\rightarrow_F$, and $\rightarrow_T$.

Proof
Orthogonality follows from the following two observations: A forwarding step properly decreases the number of forwarders not nested into some body of an abstraction whereas all other steps do not. Hence, no forwarding step can be at the same time a triggering step or an application step. Every triggering step properly decreases the number of delay expressions whereas no other step does. Hence, no triggering step can be at the same time an application step. This show that the union $\rightarrow_A \cup \rightarrow_A \cup \rightarrow_F$ is orthogonal.

The uniform confluence of $\rightarrow_A$ has been proved in Theorem 5.5. With the same context-based technique as used there, the uniform confluence of $\rightarrow_F$ and $\rightarrow_T$ for admissible expressions can be checked easily. Also, the claims on commutativity follow in the same lines.

For illustration, we consider forwarding in more detail. Suppose $E_1 \ {}_F\!\leftarrow E \rightarrow_F E_2$ for an α-standardized representative E of some admissible expression and arbitrary E_1, E_2. There exist occurrences of forwarders $x_1{=}y_1$ and $x_2{=}y_2$ in E which have been reduced in one of the considered reduction steps respectively. If the two reduced occurrences of forwarders are distinct (which means that the contexts where the forwarding rule has been applied are distinct), then the existence of an expression F with $E_1 \rightarrow_F F \ {}_F\!\leftarrow E_2$ follows trivially by reducing the respective other occurrence. Otherwise, x_1 is equal to x_2 and y_1 is equal to y_2. Furthermore, there are abstractions $y_1{:}\overline{z_1}/F_1$ and $y_2{:}\overline{z_2}/F_2$ in E such that the considered forwarding steps have replaced $x_1{=}y_1$ (which is equal to $x_2{=}y_2$) with $x_1{:}\overline{z_1}/F_1$ and $x_2{:}\overline{z_2}/F_2$, respectively. Admissibility implies $y_1{:}\overline{z_1}/F_1 \equiv y_2{:}\overline{z_2}/F_2$. Thus $x_1{:}\overline{z_1}/F_2 \equiv x_2{:}\overline{z_2}/F_2$ follows and hence $E_1 \equiv E_2$. □

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

Theorem 5.10 (*Uniform confluence*)

The restriction of the δ-calculus to admissible expressions is uniformly confluent. If E is admissible then every execution of E contains the same number of application steps.

Proof

Uniform confluence follows from the uniform confluence of $\rightarrow_A$, $\rightarrow_T$, and $\rightarrow_F$ as stated in Proposition 5.9, in combination with Lemma 3.2. If $\mathscr{C}(E) < \infty$ then all executions of E contain the same number of application steps according to Propositions 5.9 and Lemma 3.7. Otherwise, $\mathscr{C}(E) = \infty$. The uniform confluence and Proposition 2.4 imply that every execution of E is infinite. Since $\rightarrow_F \cup \rightarrow_T$ terminates, every execution of E must contain an infinite number of application steps. □

If E is admissible and $\mathscr{C}(E) < \infty$ then Lemma 3.7 also implies that every execution of E contains the same number of forwarding steps and the same number of triggering steps. It might be surprising that this property fails without the assumption $\mathscr{C}(E) < \infty$. There exists an admissible expression of the δ-calculus with executions containing distinct numbers of forwarding steps. This phenomenon is of a quite general nature. It depends on fairness of infinite executions (and not on particularities of the δ-calculus), and has already been discussed in section 2 following Lemma 3.7.

Example 5.11 (*Unfair infinite executions*)

We consider the expression $x'{=}x \mid E_4$ where E_4 is the infinite loop of Example 5.1, i.e. $E_4 = xy \mid x{:}y/xy$.

$$\begin{array}{l} x'{=}x \mid E_4 \rightarrow_A x'{=}x \mid E_4 \rightarrow_A x'{=}x \mid E_4 \rightarrow_A \ldots \\ x'{=}x \mid E_4 \rightarrow_F x'{:}y/xy \mid E_4 \rightarrow_A x'{:}y/xy \mid E_4 \rightarrow_A \ldots \end{array}$$

The first execution above does not contain any forwarding step whereas the second one does. The first execution is not fair with respect to the forwarder $x'{=}x$ which could have been reduced at every time point, but remains untouched forever.

In the light of the above fairness concern, we recall the definition of complexity measures for the δ-calculus which can be obtained as instances of Definition 3.3:

$$\begin{array}{lcl} \mathscr{C}_A(E) & = & \sup\{m \mid \text{a finite partial execution of } E \text{ has } m \text{ application steps}\} \\ \mathscr{C}_F(E) & = & \sup\{m \mid \text{a finite partial execution of } E \text{ has } m \text{ forwarding steps}\} \\ \mathscr{C}_T(E) & = & \sup\{m \mid \text{a finite partial execution of } E \text{ has } m \text{ triggering steps}\} \end{array}$$

Only in the case of application steps, it would be sufficient to consider a single execution rather than a least upper bound over all executions. Nevertheless, Proposition 3.8 ensures additivity.

Proposition 5.12 (*Additivity*)

For all admissible expressions E of the δ-calculus: $\mathscr{C}(E) = \mathscr{C}_A(E) + \mathscr{C}_F(E) + \mathscr{C}_T(E)$.

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

5.3 Linear types for proving admissibility

We present a linear type system for π_0 which allows to prove admissibility by checking well-typedness. This type system was also presented in Niehren (1994, 1996). Type checking infers data flow information. The type system is linear in that it cares about how often a variable is used for naming an abstraction. In other words, a variable is a resource which is consumed when it is used for naming an abstraction. In the extended version of this paper (Niehren, 1999), a richer linear type systems is presented for which well-typed expressions of the δ-calculus can be encoded into well-typed expressions of π_0.

Sangiorgi (1997) has independently introduced a similar linear type system for the π-calculus in order to prove *uniform receptiveness* of channel names. The idea of uniform receptiveness is in fact the same as for admissibility, modulo a distinct concept of output. In the present article, functional output is done by side effect on logic variables, whereas Sangiorgi treats functional output by passing values. Another similar linear type system was proposed by Kobayashi, Pierce and Turner (1996) for the π-calculus. Their system is motivated by and applied to optimized code generation with the PICT compiler, in case that some channel in a PICT program is provably used exactly once for input and once for output.

Most typically, the inconsistent expression $x{:}y/E \mid x{:}uv/E'$ is not well-typed since it uses the variable x twice for naming an abstraction. For excluding multiple naming, our type system administrates a set of possible abstraction names, each variable of which can be used at most once.

We assume an infinite set of *type variables* denoted by α and use the following recursive *types* σ internally annotated with *modes* η (where $n \geqslant 0$).

$$\begin{array}{lcl} \sigma & ::= & \alpha \mid (\sigma_1{}^{\eta_1} \ldots \sigma_n^{\eta_n}) \mid \mu\alpha.\sigma \\ \eta & ::= & \mathsf{in} \mid \mathsf{out} \end{array}$$

A type σ is either a variable α, a *procedural type* $(\sigma_1^{\eta_1} \ldots \sigma_n^{\eta_n})$, or a *recursive type* $\mu\alpha.\sigma$. Note that we are interested in typability but not in principal types. In general, principal types do not exist since our mode language does not provide for a most general mode.

A *type assumption* $x{:}\sigma$ is a pair of a variable x and a type σ and reads as x has type σ. A *mode assumption* $x{:}\eta$ is a pair of a variable x and a mode η and reads as x has mode σ. For convenience, we will make use of the following sequence notation: Instead of $(\sigma_1^{\eta_1} \ldots \sigma_n^{\eta_n})$, we will write $(\overline{\sigma}^{\overline{\eta}})$ where $\overline{y}$ is the sequence $\sigma_1, \ldots, \sigma_n$ and $\overline{\eta}$ the sequence $\eta_1, \ldots, \eta_n$. We also write $\overline{y}{:}\overline{\sigma}$ for a sequence of *type assumptions* $y_i{:}\sigma_i$ and $\overline{y}{:}\overline{\eta}$ for a sequence of *mode assumptions* $y_i{:}\eta_i$.

A variable x has the *procedural types* $(\overline{\sigma}^{\overline{\eta}})$ in an abstraction $x{:}\overline{y}/E$ if the formal arguments $\overline{y}$ are typed by $\overline{\sigma}$ and moded by $\overline{\eta}$ in E. We call a formal argument with mode in an *input argument* and a formal argument with mode out an *output argument*. In the abstraction $x{:}yz/yz$ for example, the variable x can be given the procedural type $(\alpha^{\mathsf{in}}\ \alpha^{\mathsf{out}})$ which states that y is an input argument and z an output argument.

We call a sequence of mode assumptions $\overline{y}{:}\overline{\eta} = y_1{:}\eta_1, \ldots, y_n{:}\eta_n$ *output-linear* if

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

$$\text{(Com)}\quad \frac{\Gamma;\ W_1 \vdash E_1 \qquad \Gamma;\ W_2 \vdash E_2}{\Gamma;\ W \vdash E_1 \mid E_2} \qquad W \subseteq W_1 \uplus W_2$$

$$\text{(Dec)}\quad \frac{\Gamma, x{:}\sigma;\ W' \vdash E}{\Gamma;\ W \vdash (\nu x)E} \qquad W' \subseteq W \cup \{x\}$$

$$\text{(Abs)}\quad \frac{\Gamma, \bar{y}{:}\bar{\sigma};\ W \vdash E}{\Gamma;\ \{x\} \vdash x{:}\bar{y}/E} \qquad \begin{array}{l}\Gamma(x) = (\bar{\sigma}^{\bar{\eta}}) \\ W' \subseteq \mathsf{Out}(\bar{y}{:}\bar{\eta})\end{array}$$

$$\text{(App)}\quad \frac{}{\Gamma;\ W \vdash x\bar{y}} \qquad \begin{array}{l}\Gamma(x) = (\Gamma(\bar{y})^{\bar{\eta}}) \\ \mathsf{Out}(\bar{y}{:}\bar{\eta}) \subseteq W\end{array}$$

$$\text{(Forw)}\quad \frac{}{\Gamma;\{x\} \vdash x{=}y} \qquad \Gamma(x) = \Gamma(y)$$

$$\text{(Trig)}\quad \frac{}{\Gamma;\emptyset \vdash \mathbf{tr}(x)}$$

$$\text{(Del)}\quad \frac{\Gamma;W \vdash E}{\Gamma;W \vdash x.E}$$

Fig. 3. Linear type checking for proving admissibility.

there does not exist $1 \leqslant i < j \leqslant n$ such that $y_i = y_j$ and $\eta_i = \eta_j = \mathsf{out}$; for an output-linear sequence of mode assumptions, we define the set of its *output arguments*:

$$\mathsf{Out}(\bar{y}{:}\bar{\eta}) \;=\; \begin{cases} \text{undefined} & \text{if } \bar{y}{:}\bar{\eta} \text{ is not output-linear} \\ \{y_i \mid 1 \leqslant i \leqslant n,\ \eta_i = \mathsf{out}\} & \text{otherwise} \end{cases}$$

Finally, *recursive types* $\mu\alpha.\sigma$ are provided. These will be needed in order to deal with expressions stemming from recursively defined embeddings between calculi. As usual, we identify recursive types modulo the following identity:

$$\mu\alpha.(\bar{\sigma}^{\bar{\eta}}) \;=\; (\bar{\sigma}^{\bar{\eta}})[\mu\alpha.(\bar{\sigma}^{\bar{\eta}})/\alpha]$$

A *type environment* Γ is a sequence of *type assumptions* $x{:}\sigma$ with scoping to the right. We say that *a variable x has type σ in* Γ, written $\Gamma(x) = \sigma$, if there exists Γ_1 and Γ_2 such that $\Gamma = \Gamma_1, x{:}\sigma, \Gamma_2$ and x does not occur in Γ_2. A *type judgment for E* is a triple $\Gamma;\ W \vdash E$, where Γ is an environment and W are finite sets of variables. Such a type judgment means that E can be typed in the environment Γ whereby the variables in W may be consumed for naming an abstraction, but at most once.

An expression E is *well-typed* if there exists a judgment for E derivable with the rules in figure 3. There are rules for abstraction (Abs), application (App), composition (Com), and declaration (Dec); obvious additional rules for triggering (Trig), forwarding (Forw), and delay (Del) are also provided. The resources (the set of variables in a type judgment) are split by rule (Com) where $\uplus$ is the operator of disjoint union of sets. Recursive types are checked by the same rules as non-recursive ones. This works, since we identify recursive types with respect to their standard equality. For instance, the judgment $x{:}\mu\alpha.(\alpha^{\mathsf{in}});\{x\} \vdash x{:}y/xy$ can be derived with the rules (Abs) and (App).

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

Expressions	$M, N, P \ ::= \ x \ \mid \ V \ \mid \ MN$
Values	$V \ ::= \ \lambda x.M$
Reduction	$(\lambda x.M)V \ \rightarrow_{\text{val}} \ M[V/x]$
Contexts	$\dfrac{M \rightarrow_{\text{val}} M'}{MN \rightarrow_{\text{val}} M'N} \qquad \dfrac{N \rightarrow_{\text{val}} N'}{MN \rightarrow_{\text{val}} MN'}$

Fig. 4. The λ-calculus with weak call-by-value reduction.

Proposition 5.13

Every well-typed expression of the δ-calculus is admissible.

Rather than presenting the proof, we illustrate linear type checking at an example. We show how to derive $\Gamma_3; \{x\} \vdash E_3$ where:

$$\Gamma_3 \quad \equiv \quad x\colon(\alpha^{\mathsf{in}}\ \alpha^{\mathsf{out}}),\ u\colon\alpha, \qquad\qquad E_3 \quad \equiv \quad x\colon yz/(vv)(xuv \mid xvz)$$

The abstraction of name x in E_3 can be applied with an input and an output argument in first and second position respectively. The types of both arguments have to coincide with the type of the global variable u. This global variable plays the rôle of an additional input argument. The fact that the set of possible abstraction names in E_3 is $\{x\}$ shows that x is the only free variable in E_3 that may eventually be used for naming an abstraction during the reduction of E_3. Let Γ'_3 be the type environment $\Gamma_3, y\colon\alpha, z\colon\alpha, v\colon\alpha$. The rules in figure 3 yield:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma'_3; \{v\} \vdash xuv}\ (\text{App}) \qquad \overline{\Gamma'_3; \{z\} \vdash xuz}\ (\text{App})}{\Gamma'_3; \{z, v\} \vdash xuv \mid xvz}(\text{Com})}{\Gamma_3, y\colon\alpha, z\colon\alpha; \{z\} \vdash (vv)(xuv \mid xvz)}(\text{Dec})}{\Gamma_3; \{x\} \vdash x\colon yz/(vv)(xuv \mid xvz)}\ (\text{Abs})$$

We explain the above derivation bottom-up. First, (Abs) can be applied to the abstraction named x, since the set of possible abstraction names in the final judgment is $\{x\}$. The procedural type assumed for x in Γ_3 requires that the second argument z is the only output argument. We therefore continue with the set $\{z\}$ for possible abstraction names. An application of (Dec) adds the local variable v to this set. When applying (Com), the actual set of possible abstraction names $\{z, v\}$ is partitioned into two disjoint parts, the set $\{v\}$ for the possible abstraction names in xuv and the set $\{z\}$ for possible abstraction names in xvz. Finally, the rule (App) checks successfully that both second arguments in the considered applications of x (v and z, respectively) are a possible abstraction name.

6 Eager functional computation

We model eager functional computation in the λ-calculus with the weak call-by-value reduction strategy that we call λ_{val} and encode λ_{val} into the δ-calculus.

The definition of λ_{val} is recalled in figure 4. An expression M of λ_{val} is a usual λ-expression which is either a variable, an abstraction (ranged over by V), or an

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

$$\begin{aligned}
[\![MN]\!]^{\mathrm{val}}_z &\stackrel{\mathrm{def}}{\equiv} (\nu x)(\nu y)([\![M]\!]^{\mathrm{val}}_x \mid [\![N]\!]^{\mathrm{val}}_y \mid xyz)\\
[\![\lambda x.M]\!]^{\mathrm{val}}_z &\stackrel{\mathrm{def}}{\equiv} z{:}xy/[\![M]\!]^{\mathrm{val}}_y\\
[\![x]\!]^{\mathrm{val}}_z &\stackrel{\mathrm{def}}{\equiv} z{=}x
\end{aligned}$$

Fig. 5. Embedding λ_{val} into the δ-calculus.

application. Bound variables are introduced by λ-binders in abstractions. We identify λ-expressions up to consistent renaming of bound variables. The congruence of λ_{val} is the equality of λ-terms. Reduction $\rightarrow_{\mathrm{val}}$ in λ_{val} is given by a single reduction rule that is applicable in weak contexts (but not inside of abstractions).

We should note by an example that weak call-by-value reduction is not deterministic. Consider the expression $(II)(II)$ which allows for two executions:

$$\begin{aligned}
(II)\,(II) &\rightarrow_{\mathrm{val}} I\,(II) \rightarrow_{\mathrm{val}} II \rightarrow_{\mathrm{val}} I\\
(II)\,(II) &\rightarrow_{\mathrm{val}} (II)\,I \rightarrow_{\mathrm{val}} II \rightarrow_{\mathrm{val}} I
\end{aligned}$$

Proposition 6.1 (*Uniform confluence*)
The λ-calculus with weak call-by-value reduction λ_{val} is uniformly confluent.

The proof can be done by induction on the structure of λ-expressions.

Definition 6.2

We define the *call-by-value complexity* $\mathscr{C}^{\mathrm{val}}(M)$ of an expression M as the number of $\rightarrow_{\mathrm{val}}$ reduction steps in executions of M.

Note that this number coincides for all executions of M because of uniform confluence (Propositions 6.1 and 2.4).

6.1 Call-by-value translation

An embedding of λ_{val} into the δ-calculus is given in figure 5. A λ-expression M together with a variable z is mapped to an expression $[\![M]\!]^{\mathrm{val}}_z$ of the δ-calculus. The definition of $[\![M]\!]^{\mathrm{val}}_z$ is given in figure 5. It is modulo congruence and assumes that all variables introduced are fresh.

The translation of an application $[\![MN]\!]^{\mathrm{val}}_z$ with name z introduces new names x and y for naming the functor and the argument (in $[\![M]\!]^{\mathrm{val}}_x$ and $[\![N]\!]^{\mathrm{val}}_y$) and concurrently applies the functors name x to y and z (in xyz). The translation of an abstraction $[\![\lambda x.M]\!]^{\mathrm{val}}_z$ with name z is a binary abstraction of the δ-calculus named by z; its first argument x names the actual input whereas its second one y names the actual output (in $[\![M]\!]^{\mathrm{val}}_y$). The translation of a variable x with name z is simply a forwarder $z{=}x$.

Example 6.3
The call-by-value translation $[\![I]\!]^{\mathrm{val}}_z$ of I with name z is $z{:}xy/y{=}x$ and the translation of $[\![I\ (II)]\!]^{\mathrm{val}}_z$ is congruent to the following expression of the δ-calculus:

$$[\![I\ (II)]\!]^{\mathrm{val}}_z \equiv (\nu y_1)(\nu z_1)(\nu y_2)(\nu z_2)([\![I]\!]^{\mathrm{val}}_{y_1} \mid [\![I]\!]^{\mathrm{val}}_{y_2} \mid [\![I]\!]^{\mathrm{val}}_{z_2} \mid y_2 z_2 z_1 \mid y_1 z_1 z)$$

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

The application $y_2 z_2 z_1$ corresponds to the inner and $y_1 z_1 z$ to the outer redex of $I\ (I\,I)$. Although the call-by-value execution of $I\ (I\,I)$ is deterministic,

$$I\ (\,I\,I\,)\ \rightarrow_{\mathrm{val}}\ I\,I\ \rightarrow_{\mathrm{val}}\ I$$

there exist several executions of its call-by-value translation. Let D be the context of $y_2 z_2 z_1 \mid y_1 z_1 z$ in the expression $[\![I\ (I\,I)]\!]^{\mathrm{val}}_z$, i.e.:

$$D = (\nu y_1)(\nu z_1)(\nu y_2)(\nu z_2)([\![I]\!]^{\mathrm{val}}_{y_1} \mid [\![I]\!]^{\mathrm{val}}_{y_2} \mid [\![I]\!]^{\mathrm{val}}_{z_2} \mid [\ ])$$

Recall that we write $D[E]$ for the expression obtained by replacing the hole $[\ ]$ in D with E. With this notation we have:

$$[\![I\ (I\,I)]\!]^{\mathrm{val}}_z \ \equiv\ D[y_2 z_2 z_1 \mid y_1 z_1 z]$$

The following execution of $[\![I\ (I\,I)]\!]^{\mathrm{val}}_z$ corresponds to the unique execution of $I\ (I\,I)$:

$$\begin{array}{lll} D[\,y_2 z_2 z_1 \mid y_1 z_1 z] & \rightarrow_A & D[z_1{=}z_2 \mid y_1 z_1 z\,] \ \rightarrow_A\ D[\,z_1{=}z_2 \mid z{=}z_1] \\ & \rightarrow_F & D[[\![I]\!]^{\mathrm{val}}_{z_1} \mid z{=}z_1\,] \ \rightarrow_F\ D[[\![I]\!]^{\mathrm{val}}_{z_1}] \mid [\![I]\!]^{\mathrm{val}}_z \end{array}$$

Up to the closed expression $D[[\![I]\!]^{\mathrm{val}}_{z_1}]$ the outcome of the above execution is $[\![I]\!]^{\mathrm{val}}_z$. Every $\rightarrow_{\mathrm{val}}$ step in the unique execution of $I\ (I\,I)$ corresponds to one $\rightarrow_A$ step plus at most two $\rightarrow_F$ steps in the above execution of $[\![I\ (I\,I)]\!]^{\mathrm{val}}_z$. There also exist executions of $[\![I\ (I\,I)]\!]^{\mathrm{val}}_z$ corresponding to reducing the outer redex of $I\ (I\,I)$ first.

$$\begin{array}{lll} D[y_2 z_2 z_1 \mid y_1 z_1 z\,] & \rightarrow_A & D[\,y_2 z_2 z_1 \mid z{=}z_1] \ \rightarrow_A\ D[\,z_1{=}z_2 \mid z{=}z_1] \\ & \rightarrow_F & D[[\![I]\!]^{\mathrm{val}}_{z_1} \mid z{=}z_1\,] \ \rightarrow_F\ D[[\![I]\!]^{\mathrm{val}}_{z_1}] \mid [\![I]\!]^{\mathrm{val}}_z \end{array}$$

This shows that our embedding introduces new flexibility with respect to possible schedulings of application steps.

Proposition 6.4

For all z and closed M the expression $[\![M]\!]^{\mathrm{val}}_z$ is well-typed and hence admissible.

Proof

Given a set a variables $X = \{x_1, \ldots, x_n\}$ we define Γ_X to be the type environment:

$$\Gamma_X = x_1 : \mu\alpha.(\alpha^{\mathsf{in}}\ \alpha^{\mathsf{out}}), \ldots, x_n : \mu\alpha.(\alpha^{\mathsf{in}}\ \alpha^{\mathsf{out}})$$

For a fresh variable z, the judgment $\Gamma_{\mathscr{V}(M)}; \{z\} \vdash [\![M]\!]^{\mathrm{val}}_z$ can be derived. □

Proposition 6.5

Let z be an arbitrary variable and consider $[\![\ .\ ]\!]^{\mathrm{val}}_z$ as a mapping from closed λ-terms to admissible expressions of the δ-calculus. Then there exists an administrative simulation S for $[\![\ .\ ]\!]^{\mathrm{val}}_z$ with administrative reduction $\rightarrow_T \cup \rightarrow_F$ and indices $1, 2$.

Proof

The proof of Proposition 6.5 is delegated to section 6.2. There, an administrative simulation S is defined such that for every M, M' and E the following diagram can

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

be completed with some E':

$$\begin{array}{lll} M & \to_{\mathrm{val}} & M' \\ S & & S \\ E & \to_F^{\leqslant 2} \circ \to_A & \exists E' \end{array}$$

□

Theorem 6.6 (*Call-by-value translation preserves complexity*)

For every variable z and all closed λ-expressions M the following properties hold:

$$\mathscr{C}^{\mathrm{val}}(M) \;=\; \mathscr{C}_A([\![M]\!]_z^{\mathrm{val}}) \;\leqslant\; \mathscr{C}([\![M]\!]_z^{\mathrm{val}}) \;\leqslant\; 3 * \mathscr{C}^{\mathrm{val}}(M)$$

Proof

This follows directly from the existence of an administrative simulation with indexes 1 and 2 as claimed in Proposition 6.5 and Proposition 4.9. □

Theorem 6.6 shows that the the call-by-value translation of λ-calculus into the δ-calculus preserves complexity up to a factor of 3, whereby every β-reduction step in λ_{val} corresponds to exactly one application step in the δ-calculus.

6.2 A simulation for call-by-value

We now define an administrative simulation for the presented embedding of λ_{val} into the δ-calculus. As proposed by Milner (1991, 1992), we also make use of a notation for explicit substitution. Its syntax is given by:

$$\mathsf{subst}\ \ M_1/y_1\ \ldots\ M_n/y_n\ \ \mathsf{in}\ N \;\stackrel{\mathrm{def}}{\equiv}\; N[M_n/y_n]\ldots[M_1/y_1]$$

Example 6.7

We first illustrate the idea underlying our definition of an administrative simulation. We reduce the call-by-value translation $[\![I\ (II)]\!]_z^{\mathrm{val}}$ where we omit declaration prefixes for simplicity. The translation of the inner redex is applied first; the translation of the outer redex could also be reduced but without correspondence in λ_{val}.

$$\begin{array}{lll} [\![I\ (II)]\!]_z^{\mathrm{val}} & \to_A & [\![I]\!]_{y_0}^{\mathrm{val}} \mid [\![I]\!]_{z_0}^{\mathrm{val}} \mid [\![\,I\ z_0\,]\!]_z^{\mathrm{val}} \\ & \to_F & [\![I]\!]_{y_0}^{\mathrm{val}} \mid [\![I]\!]_{z_0}^{\mathrm{val}} \mid [\![\,I\ I\,]\!]_z^{\mathrm{val}} \\ & \to_A & [\![I]\!]_{y_0}^{\mathrm{val}} \mid [\![I]\!]_{z_0}^{\mathrm{val}} \mid [\![I]\!]_{y_1}^{\mathrm{val}} \mid [\![I]\!]_{z_1}^{\mathrm{val}} \mid [\![\,z_1\,]\!]_z^{\mathrm{val}} \\ & \to_F & [\![I]\!]_{y_0}^{\mathrm{val}} \mid [\![I]\!]_{z_0}^{\mathrm{val}} \mid [\![I]\!]_{y_1}^{\mathrm{val}} \mid [\![I]\!]_{z_1}^{\mathrm{val}} \mid [\![I]\!]_z^{\mathrm{val}} \end{array}$$

The call-by-value translation $[\![I\ (II)]\!]_z^{\mathrm{val}}$ introduces the variables y_0 and z_0 for naming the functor and argument of the inner redex, respectively. In the first step, the translated functor $[\![I]\!]_{y_0}^{\mathrm{val}}$ is applied with the arguments name z_0 which then is returned. In the second step, the variable z_0 is replaced by the abstraction it names. The similarity to an execution of $I\ (II)$ in λ_{val} shows up when using our new

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

substitution notion:

$$\begin{array}{rcl} I\ (\ II\ ) & \to_{\mathrm{val}} & \mathsf{subst}\ I/y_0\ I/z_0\ \mathsf{in}\ I\ z_0 \\ & \equiv & \mathsf{subst}\ I/y_0\ I/z_0\ \mathsf{in}\ I\ I \\ & \to_{\mathrm{val}} & \mathsf{subst}\ I/y_0\ I/z_0\ I/y_1\ I/z_1\ \mathsf{in}\ z_1 \\ & \equiv & \mathsf{subst}\ I/y_0\ I/z_0\ I/y_1\ I/z_1\ \mathsf{in}\ I \end{array}$$

In our formal treatment, we will freely make use of the following sequence notation. If $\bar{y} = (y_i)_{i=1}^n$ and $\overline{M} = (M_i)_{i=1}^n$ then we write:

$$\begin{array}{rcl} \mathsf{subst}\ \overline{M}/\bar{y}\ \mathsf{in}\ N & \equiv & \mathsf{subst}\ M_1/y_1\ \ldots\ M_n/y_n\ \mathsf{in}\ N \\ [\![\overline{M}]\!]_{\bar{y}}^{\mathrm{val}} & \equiv & [\![M_1]\!]_{y_1}^{\mathrm{val}} \mid \ldots \mid [\![M_n]\!]_{y_n}^{\mathrm{val}} \end{array}$$

If $1 \leqslant i \leqslant n$ then we write $\overline{M}^{<i}$ for the sequence $(M_j)_{j=1}^{i-1}$, $\overline{M}^{>i}$ for $(M_j)_{j=i+1}^{n}$, and similarly $\bar{y}^{<i}$ for $(y_j)_{j=1}^{i-1}$, and $\bar{y}^{>i}$ for $(y_j)_{j=i+1}^{n}$. The *concatenation* of two sequences is denoted by juxtaposition, for instance $\overline{M}\overline{N}$ or $\bar{y}\bar{z}$. We also write $\overline{M}N$, $N\overline{M}$, or $z\bar{y}$, $\bar{y}z$ for the concatenation of a single element to the left or right of a sequence.

We define *prefix equivalence* $\approx_3$ to be the smallest equivalence relation on expression of the δ-calculus that is modulo congruence, and satisfies the following property for all x, y, E and reduction contexts D:

$$D[(\nu x)E] \ \approx_3\ D[E[y/x]] \qquad y \text{ fresh}$$

Lemma 6.8
For all admissible E, F with $E \approx_3 F$: $\mathscr{C}(E) = \mathscr{C}(F)$ and $\mathscr{C}_A(E) = \mathscr{C}_A(F)$.

Proof
The lemma essentially follows from the fact that for all E, F, E' the following diagrams can be closed with some F':

$$\begin{array}{ccc} E & \to_A & E' \\ \wr_3 & & \wr_3 \\ F & \to_A & \exists F' \end{array} \qquad \begin{array}{ccc} E & \to_F & E' \\ \wr_3 & & \wr_3 \\ F & \to_F & \exists F' \end{array} \qquad \begin{array}{ccc} E & \to_T & E' \\ \wr_3 & & \wr_3 \\ F & \to_T & \exists F' \end{array}$$

Hence, Propositions 4.6 and Theorem 5.10 imply for all admissible E, F with $E \approx F$ that $\mathscr{C}_A(E) = \mathscr{C}_A(F)$, $\mathscr{C}_T(E) = \mathscr{C}_T(F)$, and $\mathscr{C}_F(E) = \mathscr{C}_F(F)$. If follows from the additivity Proposition 5.12 that $\mathscr{C}(E) = \mathscr{C}(F)$. □

Definition 6.9 (*Representation*)
A *representation* for (M, E) is a triple $(n, \bar{y}, \overline{M})$, where $\bar{y} = (y_i)_{i=1}^n$, $\overline{M} = (M_i)_{i=1}^n$, and such that the following properties hold for all $i \in \{1, \ldots, n\}$:

(R1) $\mathscr{V}(M_i) \subseteq \{y_1 \ldots y_{i-1}\}$ and $\bar{y}$ is linear.
(R2) $M \equiv \mathsf{subst}\ \overline{M}/\bar{y}\ \mathsf{in}\ y_n$.
(R3) $E \approx_3 [\![M_1]\!]_{y_1}^{\mathrm{val}} \mid \ldots \mid [\![M_n]\!]_{y_n}^{\mathrm{val}}$.
(R4) If $i < n$ then M_i is an abstraction.

Lemma 6.10 (*Closedness*)
If n, $\overline{M}$, $\bar{y}$, and M satisfy (R1) and (R2) then M is closed.

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

Definition 6.11
The relation S^{val} is the set of all pairs (M, E) for which a representation exists.

Proposition 6.12 (S^{val} *is an administrative simulation*)
For every z, the relation S^{val} is an administrative simulation for mapping of a closed λ-expression E to its call-by-value translation $[\![M]\!]_z^{\mathrm{val}}$. The administrative relation of this simulation is $\rightarrow_T \cup \rightarrow_F$ and its administrative indices are $1, 2$.

Proof
We check each of the conditions of Definition 4.7. Note that the proof of property (A2) requires Lemma 6.13 given below.

(A1) If M is closed then $(M, [\![M]\!]_z^{\mathrm{val}}) \in S^{\mathrm{val}}$ since $(n, (z), (M))$ is a representation of $(M, [\![M]\!]_z^{\mathrm{val}})$. Property (R1) follows from the closedness of M and (R2), (R3), (R4) are trivial in this case.

(A2) Let $(n, \bar{y}, \overline{M})$ be a representation of (M, E) and $M \rightarrow_{\mathrm{val}} M'$. Applying the following Lemma 6.13, there exists sequences $\bar{x}$ and $\overline{V}$ of length m and an expression E' such that $(n+m, \bar{y}^{<n}\bar{x}y_n, \overline{M}^{<n}\overline{V}M'_n)$ is a representation for (M', E') and $E \rightarrow_F^{\leqslant 2} \circ \rightarrow_A E'$.

(A3) The first administrative index is 1 thus greater than or equal to 1 as required.

(A4) Let M be closed and irreducible with respect to $\rightarrow_{\mathrm{val}}$ and assume $(M, E) \in S^{\mathrm{val}}$. Since M is irreducible and closed it is an abstraction. There exists a representation $(n, \bar{y}, \overline{M})$ for (M, E). Since M is an abstraction, either y_n is a variable or an abstraction. Hence E reduces in one $\rightarrow_F$ step to a composition of abstractions which is irreducible, i.e. $\mathscr{C}_A(E) = 0$ and $\mathscr{C}_F(E) + \mathscr{C}_T(E) \leqslant 1$.

(A5) Let M be closed. The expression $[\![M]\!]_z^{\mathrm{val}}$ is irreducible with respect to $\rightarrow_T$ since no trigger expressions are used by the translation. The expression $[\![M]\!]_z^{\mathrm{val}}$ is irreducible with respect forwarding since all forwarders introduced by translation belong to the bodies of some abstraction.

□

Lemma 6.13
Let $(n, \bar{y}, \overline{M})$ be a representation of (M, E) and $M \rightarrow_{\mathrm{val}} M'$. Then there exists fresh variables $\bar{x}$, abstractions $\overline{V}$, and a λ-expression M'_n such that $E \rightarrow_F^{\leqslant 2} \circ \rightarrow_A E'$, $\mathscr{V}(\overline{V}) \subseteq \mathscr{V}(\bar{y}^{<n})$, $\mathscr{V}(M'_m) \subseteq \mathscr{V}(\bar{y}^{<n}\bar{x})$, and:

$$\begin{array}{rcl} M' & \equiv & \mathsf{subst}\ \overline{M}^{<n}/\bar{y}^{<n}\ \overline{V}/\bar{x}\ M'_n/y_n\ \mathsf{in}\ y_n \\ E' & \approx_3 & [\![\overline{M}^{<n}]\!]_{\bar{y}^{<n}}^{\mathrm{val}} \mid [\![\overline{V}]\!]_{\bar{x}}^{\mathrm{val}} \mid [\![M'_n]\!]_{y_n}^{\mathrm{val}} \end{array}$$

Proof
Since $(n, \bar{y}, \overline{M})$ is an representation, we know $M \equiv \mathsf{subst}\ \overline{M}/\bar{y}\ \mathsf{in}\ y_n$ and $E \approx_3 [\![\overline{M}]\!]_{\bar{y}}^{\mathrm{val}}$. Since M can not be an abstraction, property (R4) implies that M_n is an application $N_1 N_2$ for some N_1 and N_2. Hence $M \equiv P_1 P_2$ and:

$$P_1 \equiv \mathsf{subst}\ \overline{M}^{<n}/\bar{y}^{<n}\ \mathsf{in}\ N_1\,, \qquad P_2 \equiv \mathsf{subst}\ \overline{M}^{<n}/\bar{y}^{<n}\ \mathsf{in}\ N_2$$

1. Case: $M \rightarrow_{\mathrm{val}} M'$ is an instance of the β-axiom, i.e. $P_1 \equiv \lambda x.\tilde{P}_1$ and:

$$M \equiv (\lambda x.\tilde{P}_1)P_2 \rightarrow_{\mathrm{val}} \tilde{P}_1[P_2/x] \equiv M'$$

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

Since P_1 and P_2 are abstractions, N_1 and N_2 have to be either variables or abstractions. This leads to four very similar subcases. We only consider the case where N_1 and N_2 are both variables. In this case there exists y_{l_1} and y_{l_2} such that $N_1 = y_{l_1}$ and $N_2 = y_{l_2}$. Furthermore:

$$P_1 \equiv \mathsf{subst}\ \overline{M}^{<n}/\overline{y}^{<n}\ \mathsf{in}\ M_{l_1}\,, \qquad P_2 \equiv \mathsf{subst}\ \overline{M}^{<n}/\overline{y}^{<n}\ \mathsf{in}\ M_{l_2}$$

If $M_{l_1} \equiv \lambda x.\tilde{M}_{l_1}$ then $\tilde{P}_1 \equiv \mathsf{subst}\ \overline{M}^{<n}/\overline{y}^{<n}\ \mathsf{in}\ \tilde{M}_{l_1}$. Let x_1 and x_2 be fresh:

$$\begin{array}{rcl} M' & \equiv & (\mathsf{subst}\ \overline{M}^{<n}/\overline{y}^{<n}\ \mathsf{in}\ \tilde{M}_{l_1})[P_2/x] \\ & \equiv & \mathsf{subst}\ \overline{M}^{<n}/\overline{y}^{<n}\ \mathsf{in}\ \tilde{M}_{l_1}[P_2/x] \\ & \equiv & \mathsf{subst}\ \overline{M}^{<n}/\overline{y}^{<n}\ \mathsf{in}\ \tilde{M}_{l_1}[y_{l_2}/x] \\ & \equiv & \mathsf{subst}\ \overline{M}^{<n}/\overline{y}^{<n}\ M_{l_1}/x_1\ M_{l_2}/x_2\ \tilde{M}_{l_1}[x_2/x]/y_n\ \mathsf{in}\ y_n \end{array}$$

Reduction of E may proceed with two forwarding steps followed by an application step.

$$\begin{array}{rcl} E & \approx_3 & [\![\overline{M}^{<n}]\!]^{\mathrm{val}}_{\overline{y}^{<n}} \mid [\![y_{l_1}\, y_{l_2}]\!]^{\mathrm{val}}_{y_n} \\ & \approx_3 & [\![\overline{M}^{<n}]\!]^{\mathrm{val}}_{\overline{y}^{<n}} \mid x_1{=}y_{l_1} \mid x_2{=}y_{l_2} \mid x_1 x_2 y_n \\ & \to_F^2 & [\![\overline{M}^{<n}]\!]^{\mathrm{val}}_{\overline{y}^{<n}} \mid [\![M_{l_1}]\!]^{\mathrm{val}}_{x_1} \mid [\![M_{l_2}]\!]^{\mathrm{val}}_{x_2} \mid x_1 x_2 y_n \\ & \to_A & [\![\overline{M}^{<n}]\!]^{\mathrm{val}}_{\overline{y}^{<n}} \mid [\![M_{l_1}]\!]^{\mathrm{val}}_{x_1} \mid [\![M_{l_2}]\!]^{\mathrm{val}}_{x_2} \mid [\![\tilde{M}_{l_1}]\!]^{\mathrm{val}}_{y_n}[x_2/x] \end{array}$$

This proves the inductive assertion with $M'_n \equiv \tilde{M}_{l_1}[x_2/x]$ and $\overline{V}$ equals the sequence (M_{l_1}, M_{l_2}).

2. Case: The last rule in the derivation of $M \to_{\mathrm{val}} M'$ allows for reduction in functional position:

$$\frac{P_1 \to_{\mathrm{val}} P_1'}{M \equiv P_1 P_2 \to_{\mathrm{val}} P_1' P_2 \equiv M'}$$

Let z_1 and z_2 be fresh variables and define:

$$E_1 \stackrel{\mathrm{def}}{\equiv} [\![\overline{M}^{<n}]\!]^{\mathrm{val}}_{\overline{y}^{<n}} \mid [\![N_1]\!]^{\mathrm{val}}_{z_1}$$

By induction hypothesis there exists fresh variables $\overline{x}$, abstractions $\overline{V}$, N_1', and E_1' such that $E_1 \to_F^{\leqslant 2} \circ \to_A E_1'$ and:

$$\begin{array}{rcl} P_1' & \equiv & \mathsf{subst}\ \overline{M}^{<n}/\overline{y}^{<n}\ \overline{V}/\overline{x}\ N_1'/y_n\ \mathsf{in}\ y_n \\ E_1' & \approx_3 & [\![\overline{M}^{<n}]\!]^{\mathrm{val}}_{\overline{y}^{<n}} \mid [\![\overline{V}]\!]^{\mathrm{val}}_{\overline{x}} \mid [\![N_1']\!]^{\mathrm{val}}_{y_n} \end{array}$$

Additionally, we obtain some conditions on variables occurrences which imply:

$$\begin{array}{rcl} M' \equiv P_1' P_2 & \equiv & (\mathsf{subst}\ \overline{M}^{<n}/\overline{y}^{<n}\ \overline{V}/\overline{x}\ \mathsf{in}\ N_1')\ (\mathsf{subst}\ \overline{M}^{<n}/\overline{y}^{<n}\ \mathsf{in}\ N_2) \\ & \equiv & \mathsf{subst}\ \overline{M}^{<n}/\overline{y}^{<n}\ \overline{V}/\overline{x}\ N_1' N_2/y_n\ \mathsf{in}\ y_n \end{array}$$

Furthermore:

$$\begin{array}{rcl} E & \approx_3 & [\![\overline{M}^{<n}]\!]^{\mathrm{val}}_{\overline{y}^{<n}} \mid [\![N_1 N_2]\!]^{\mathrm{val}}_{y_n} \\ & \approx_3 & [\![\overline{M}^{<n}]\!]^{\mathrm{val}}_{\overline{y}^{<n}} \mid [\![N_1]\!]^{\mathrm{val}}_{z_1} \mid [\![N_2]\!]^{\mathrm{val}}_{z_2} \mid z_1 z_2 y_n \\ & \to_F^{\leqslant 2} \circ \to_A & [\![\overline{M}^{<n}]\!]^{\mathrm{val}}_{\overline{y}^{<n}} \mid [\![\overline{V}]\!]^{\mathrm{val}}_{\overline{x}} \mid [\![N_1']\!]^{\mathrm{val}}_{z_1} \mid [\![N_2]\!]^{\mathrm{val}}_{z_2} \mid z_1 z_2 y_n \\ & \approx_3 & [\![\overline{M}^{<n}]\!]^{\mathrm{val}}_{\overline{y}^{<n}} \mid [\![\overline{V}]\!]^{\mathrm{val}}_{\overline{x}} \mid [\![N_1' N_2]\!]^{\mathrm{val}}_{y_n} \end{array}$$

This proves the inductive assertion with $M'_n \equiv N_1' N_2$.

3. Case: The last rule in the derivation of $M \to_{\mathrm{val}} M'$ allows for reduction in argument position. This case is symmetric to the previous one. □

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

Expressions	$L ::= x \mid V \mid LL' \mid \mathsf{let}\ x{=}L_2\ \mathsf{in}\ L_1 \quad \text{where } x \notin \mathscr{V}(L_2)$
	$V ::= \lambda x.L$
	$A ::= V \mid \mathsf{let}\ x{=}L\ \mathsf{in}\ A$
	$B ::= [\,] \mid BL \mid \mathsf{let}\ x{=}L\ \mathsf{in}\ B \mid \mathsf{let}\ x{=}B_2\ \mathsf{in}\ B_1[x]$
Reduction	$\to_{\text{need}} = \to_I \cup \to_V \cup \to_{Ans} \cup \to_C$
	$(\lambda x.L_1)L_2 \to_I \mathsf{let}\ x{=}L_2\ \mathsf{in}\ L_1$
	$\mathsf{let}\ x{=}V\ \mathsf{in}\ B[x] \to_V \mathsf{let}\ x{=}V\ \mathsf{in}\ B[V]$
	$\mathsf{let}\ y{=}(\mathsf{let}\ x{=}L\ \mathsf{in}\ A)\ \mathsf{in}\ B[y] \to_{Ans} \mathsf{let}\ x{=}L\ \mathsf{in}\ (\mathsf{let}\ y{=}A\ \mathsf{in}\ B[y])$
	$(\mathsf{let}\ x{=}L_1\ \mathsf{in}\ A)L_2 \to_C \mathsf{let}\ x{=}L_1\ \mathsf{in}\ AL_2$
Contexts	$\dfrac{L \to_{\text{need}} L'}{B[L] \to_{\text{need}} B[L']}$

Fig. 6. The call-by-need λ-calculus λ_{need} with standard reduction.

7 Lazy Functional computation

The call-by-need λ-calculus (Ariola *et al.*, 1995; Ariola and Felleisen, 1997; Maraist *et al.*, 1998) with standard reduction can be used to model complexity in lazy functional computation. We embed the call-by-need λ-calculus with standard reduction into the δ-calculus such that complexity is preserved.

The definition of the call-by-need λ-calculus with standard reduction λ_{need} is revisited in Figure 6. Its expressions L are variables, abstractions (denoted with V), applications, and let-expressions. The reduction $\to_{\text{need}}$ of the call-by-need λ-calculus is a union of four relations, $\to_I$, $\to_V$, $\to_{Ans}$, and $\to_C$. The relation $\to_I$ corresponds to β-reduction and the relation $\to_V$ to forwarding. The latter two relations are of administrative character.

Example 7.1

For illustrating we consider the λ-term $(II)\,I$. This examples shows that $\to_I$ steps correspond to β-reduction whereas $\to_V$ provides for forwarding abstractions.

$$\begin{aligned}(\,II\,)\,I \;&\to_I\; (\mathsf{let}\ y{=}I\ \mathsf{in}\ y)\,I \;\to_V\; (\mathsf{let}\ y{=}I\ \mathsf{in}\ I)\,I \;\to_C\; \mathsf{let}\ y{=}I\ \mathsf{in}\ II\\ &\to_I\; \mathsf{let}\ y{=}I\ \mathsf{in}\ (\mathsf{let}\ z{=}I\ \mathsf{in}\ z) \;\to_V\; \mathsf{let}\ y{=}I\ \mathsf{in}\ (\mathsf{let}\ z{=}I\ \mathsf{in}\ I)\end{aligned}$$

The rôle of $\to_C$ is to rearrange parenthesis introduced by let-expressions in function position, for instance after the evaluation of II in $(II)\,I$ as shown above. The rôle of $\to_{Ans}$ is to rearrange parenthesis after evaluation in argument position. This is

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

illustrated by the call-by-need reduction of $(\lambda x.xx)\ (II)$.

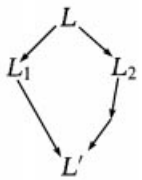
$$\begin{aligned}(\lambda x.xx)\ (II) &\to_I \text{let } x{=}II \text{ in } xx\\ &\to_I \text{let } x{=}(\text{let } y{=}I \text{ in } y) \text{ in } xx \to_V \text{let } x{=}(\text{let } y{=}I \text{ in } I) \text{ in } xx\\ &\to_{Ans} \text{let } y{=}I \text{ in } (\text{let } x{=}I \text{ in } x\,x) \to_V \text{let } y{=}I \text{ in } (\text{let } x{=}I \text{ in } Ix)\\ &\to_I \text{let } y{=}I \text{ in } (\text{let } x{=}I \text{ in } (\text{let } y{=}x \text{ in } y))\\ &\to_V^2 \text{let } y{=}I \text{ in } (\text{let } x{=}I \text{ in } (\text{let } y{=}I \text{ in } I))\end{aligned}$$

This example also illustrates sharing of computation. The evaluation of the functional argument II is shared between both uses of the value of this argument.

With respect to the call-by-need λ-calculus, we consider standard reduction rather than some form of weak reduction. The reason for this choice is purely technically motivated: The problem with weak reduction is that its administrative reduction steps spoil uniform confluence.

Example 7.2 (*Weak call-by-need reduction is not uniformly confluent*)
Weak reduction for the call-by-need λ-calculus allows to reduce in every weak context (i.e. everywhere but not in bodies of abstractions). Weak reduction for the call-by-need λ-calculus is not uniformly confluent. The problem depends on the number of steps needed for rearranging parenthesis. This number depends on the ordering in which parenthesis are rearranged. This can be illustrated with the following expression L:

$$L = \text{let } x{=}(\text{let } y{=}(\text{let } z{=}L_0 \text{ in } A_2) \text{ in } A_1) \text{ in } x$$



This expression contains two nested weak redexes where the $\to_{Ans}$ axiom applies. When reducing the outer redex first we obtain L_1, whereas we obtain L_2 when reducing the inner redex first:

$$\begin{aligned}L_1 &= \text{let } x{=}A_1 \text{ in } (\text{let } y{=}(\text{let } z{=}L_0 \text{ in } A_2) \text{ in } x)\\ L_2 &= \text{let } x{=}(\text{let } z{=}L_0 \text{ in } (\text{let } y{=}A_2 \text{ in } A_1)) \text{ in } x\end{aligned}$$

It is not possible to join L_1 and L_2 in exactly one step. It is possible however to join L_1 and L_2 into L' given below.

$$L' = \text{let } x{=}A_1 \text{ in } (\text{let } y{=}A_2 \text{ in } (\text{let } z{=}L_0 \text{ in } x))$$

The expression L_1 reduces in one weak answering step to L' whereas L_2 needs two weak answering steps.

In standard reduction, this non-uniformity problem does not occur. The inner weak redex of L can simply not be reduced since its context is not a reduction context with respect to standard reduction.

Proposition 7.3 (*Uniform confluence*)
The call-by-need λ-calculus with standard reduction λ_{need} is deterministic.

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

Proof
The context rule determines a unique position where reduction may happen. More precisely, whenever $B_1[L_1] = B_2[L_2]$ then either L_1 is a variable bound in B_1, or L_2 is a variable bound in B_2, or $B_1 = B_2$ and $L_1 = L_2$. This can be shown by induction on the size of the pair B_1, B_2. □

Proposition 7.3 implies in particular that λ_{need} is an orthogonal union of uniformly confluent calculi with commuting reductions $\to_I$, $\to_V$, $\to_{Ans}$ and $\to_C$. Hence, the theory developed in section 2 is applicable to λ_{need}.

We wish to define the call-by-need complexity of a λ-expression L. We have several choices in doing so. We might consider the number of all reduction steps of an execution of L in λ_{need}. This choice would be problematic (at least) for our purpose. The reason is that $\to_C$ and $\to_{Ans}$ do not have any correspondents in a concurrent calculus which we wish to compare with. This failure is illustrated by Example 7.2. It is also unclear in how far the $\to_C$ steps and $\to_{Ans}$ steps are realistic with respect to implementations of functional languages. We therefore ignore $\to_C$ steps and $\to_{Ans}$ completely and leave it to future research to lift this restriction.

Another question is whether we should count $\to_V$ steps. Doing so would not be too difficult since $\to_V$ steps nicely correspond to forwarding steps in the δ-calculus. It would also be possible to argue that the number of $\to_V$ steps is linearly bounded by the number of $\to_I$ steps. In favor of simplicity, we decide to count $\to_I$ steps only. These play the rôle of β-reduction in λ_{need}.

One might also argue against $\to_I$ steps claiming that a slightly reformulated version of the call-by-need λ-calculus in Ariola and Felleisen (1997) and Ariola *et al.* (1995) does not use $\to_I$ steps at all. The idea of this calculus is to identify a let expressions $\mathsf{let}\ x{=}L_2\ \mathsf{in}\ L_1$ with an application $(\lambda x.L_2)L_1$. In this approach $\to_I$ steps are no longer explicitly needed and can be replaced by a sequence of $\to_V$ steps. Since the number of $\to_V$ and $\to_I$ steps coincide up to a linear factor, the absence of $\to_I$ steps does not really affect our results.

Definition 7.4 (*Call-by-need complexity*)

We define the *call-by-need complexity* $\mathscr{C}^{\text{need}}(L)$ of a λ-term L as the number of $\to_I$ steps in the execution of L in λ_{need}.

7.1 Call-by-need translation

The call-by-need λ-calculus λ_{need} can be embedded into the δ-calculus such that complexity is preserved. In figure 7, for every expression L and variable z we define the call-by-need translation $[\![L]\!]^{\text{need}}_z$ into the δ-calculus. The call-by-need translation is fully analogous to the call-by-value translation, except that an additional control is added. In the translation of an application $[\![LL']\!]^{\text{need}}_z$ the translation of the functional argument $[\![L']\!]^{\text{need}}_y$ is delayed; whenever the value of a variable y is needed, its computation is trigger. This is encoded by the additional trigger expression in the translations of variables $[\![y]\!]^{\text{need}}_z$. Finally, notice that let-bound variables are translated to variables of the δ-calculus.

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

$$
\begin{array}{rcl}
[\![LL']\!]^{\text{need}}_z & \stackrel{\text{def}}{\equiv} & (vx)(vy)([\![L]\!]^{\text{need}}_x \mid y.[\![L']\!]^{\text{need}}_y \mid xyz) \\
[\![\lambda x.L]\!]^{\text{need}}_z & \stackrel{\text{def}}{\equiv} & z{:}xy/[\![L]\!]^{\text{need}}_y \\
[\![y]\!]^{\text{need}}_z & \stackrel{\text{def}}{\equiv} & z{=}y \mid \mathbf{tr}(y) \\
[\![\mathsf{let}\ y{=}L_2\ \mathsf{in}\ L_1]\!]^{\text{need}}_z & \stackrel{\text{def}}{\equiv} & (vy)(y.[\![L_2]\!]^{\text{need}}_y \mid [\![L_1]\!]^{\text{need}}_z)
\end{array}
$$

Fig. 7. Embedding λ_{need} into the δ-calculus.

Example 7.5

We consider the translation and execution of the λ-term $I\ (II)$.

$$
\begin{array}{rcl}
[\![I\ (II)]\!]^{\text{need}}_z & \equiv & (vy_1)(vz_1)([\![I]\!]^{\text{need}}_{y_1} \mid z_1.[\![II]\!]^{\text{need}}_{z_1} \mid y_1z_1z) \\
& \to_A & (vy_1)(vz_1)([\![I]\!]^{\text{need}}_{y_1} \mid z_1.[\![II]\!]^{\text{need}}_{z_1} \mid z{=}z_1 \mid \mathbf{tr}(z_1)) \\
& \to_T & (vy_1)(vz_1)([\![I]\!]^{\text{need}}_{y_1} \mid [\![II]\!]^{\text{need}}_{z_1} \mid z{=}z_1 \mid \mathbf{tr}(z_1))
\end{array}
$$

The translation of outer redex of $I\ (II)$ is reduced first (up to translation). The inner redex is delayed at beginning but triggered during the evaluation of the outer redex, such that further execution can be applied to the inner redex. Note that the execution of $[\![I\ (II)]\!]^{\text{need}}_z$ is deterministic.

Example 7.6 (*Non-needed arguments*)

We consider the λ-abstraction $C \equiv \lambda x.y$ where $x \neq y$. An application of C returns the constant y independently of (and without needing) the actual argument.

$$
\begin{array}{rcl}
[\![C\ (II)]\!]^{\text{need}}_z & \equiv & (vy_1)(vz_1)([\![C]\!]^{\text{need}}_{y_1} \mid z_1.[\![II]\!]^{\text{need}}_{z_1} \mid y_1z_1z) \\
& \to_A & (vy_1)(vz_1)([\![I]\!]^{\text{need}}_{y_1} \mid z_1.[\![II]\!]^{\text{need}}_{z_1} \mid z{=}y \mid \mathbf{tr}(y))
\end{array}
$$

Here, reduction terminates without having evaluated the translated functional argument $z_1.[\![II]\!]^{\text{need}}_{z_1}$ which is not needed and therefore delayed forever.

Example 7.7 (*Sharing of evaluation*)

Consider the λ expression $\mathsf{let}\ x{=}II\ \mathsf{in}\ xx$ where $x = II$ is needed twice but should be evaluated only once.

$$
\begin{array}{rcl}
[\![\mathsf{let}\ x{=}II\ \mathsf{in}\ xx]\!]^{\text{need}}_z & & \\
\quad \equiv & (vx)(x.[\![II]\!]^{\text{need}}_x \mid [\![xx]\!]^{\text{need}}_z) & \\
\quad \equiv & (vx)(vy_1)(vz_1)(x.[\![II]\!]^{\text{need}}_x \mid y_1{=}x \mid \mathbf{tr}(x) \mid z_1.(z_1{=}x \mid \mathbf{tr}(x)) \mid y_1z_1z) & \\
\to_T & (vx)(vy_1)(vz_1)([\![II]\!]^{\text{need}}_x \mid y_1{=}x \mid \mathbf{tr}(x) \mid z_1.(z_1{=}x \mid \mathbf{tr}(x)) \mid y_1z_1z) &
\end{array}
$$

Further execution on $[\![II]\!]^{\text{need}}_x$ results in $[\![x]\!]^{\text{need}}_I$ up to some garbage. Forwarding with $y_1{=}x$ yields $[\![y_1]\!]^{\text{need}}_I$ such that y_1z_1z reduces to $z{=}z_1 \mid \mathbf{tr}(z_1)$. This makes it possible to trigger $z_1.(z_1{=}x \mid \mathbf{tr}(x))$. This is the point where multiple triggers

$$\ldots \mid \mathbf{tr}(x) \mid \mathbf{tr}(x) \mid \ldots$$

have become active (one per need of x). Two final forwarding step yield $[\![I]\!]^{\text{need}}_{z_1}$ and hence $[\![I]\!]^{\text{need}}_z$.

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

Proposition 7.8
For all z and closed M the expression $[\![M]\!]^{\text{need}}_z$ is well-typed and hence admissible.

Proof
Given a set a variables $X = \{x_1, \ldots, x_n\}$ we define Γ_X to be the type environment:

$$\Gamma_X = x_1 : \mu\alpha.(\alpha^{\mathsf{in}}\ \alpha^{\mathsf{out}}), \ldots, x_n : \mu\alpha.(\alpha^{\mathsf{in}}\ \alpha^{\mathsf{out}})$$

For every λ-expression M and $z \notin \mathscr{V}(M)$ the following judgment is derivable:

$$\Gamma_{\mathscr{V}(L)}; \{z\} \vdash [\![L]\!]^{\text{need}}_z$$

This can be checked by induction on the structure of L. Hence every expression $[\![L]\!]^{\text{need}}_z$ is well-typed and hence admissible as shown by Corollary 5.13. □

Theorem 7.9 (*Call-by-need translation preserves complexity*)

For every z and closed λ-expression L the equation $\mathscr{C}^{\text{need}}(L) = \mathscr{C}_A([\![L]\!]^{\text{need}}_z)$ holds.

Proof
This follows from the existence of a complexity simulation as specified in Proposition 7.11 below, admissibility as stated (Proposition 7.8), uniform confluence (Propositions 7.3 and 5.9), and Proposition 4.3. □

7.2 A simulation for call-by-need

In this section, we present a complexity simulation for our call-by-need translation as required in the proof of Theorem 7.9. The idea for defining a complexity simulation S is to consider the relation $S = \{(L, [\![\![L]\!]\!]^{\text{need}}_z)\}$ (depending on a choice of z). However, we have to extend this relation S such that we can deal with technical details related to multiple triggering. Even worse, we cannot know statically, whether an expression has been needed at some time point. We therefore have to deal with similarities like between expressions E and $x.E \mid \mathbf{tr}(x) \mid \mathbf{tr}(x)$. To do so, we will consider sets $\mathscr{E}$ of expressions E and use the following notation.

$$\begin{array}{rclcrcl}
\mathscr{E} \mid E' & = & \{E \mid E' \mid E \in \mathscr{E}\} & \qquad & E \mid \mathscr{E}' & = & \{E \mid E' \mid E' \in \mathscr{E}'\} \\
x{:}\overline{y}/\mathscr{E} & = & \{x{:}\overline{y}/E \mid E \in \mathscr{E}\} & & (\nu x)\mathscr{E} & = & \{(\nu x)E \mid E \in \mathscr{E}\} \\
x.\mathscr{E} & = & \{x.E \mid E \in E\} & & E^* & = & \{\ |_{i=1}^{n} E \mid n \geqslant 0\}
\end{array}$$

Note that the auxiliary expression $\mathsf{0} \equiv |_{i=1}^{0} E$ is contained in E^* which satisfies $E \mid \mathsf{0} \equiv \mathsf{0} \mid E \equiv E$ for all E. We next define sets of expressions $[\![\![L]\!]\!]^{\text{need}}_x$ for all x and L such that $[\![L]\!]^{\text{need}}_x \in [\![\![L]\!]\!]^{\text{need}}_x$.

$$\begin{array}{rcl}
[\![\![LL']\!]\!]^{\text{need}}_z & = & (\nu x)(\nu y)([\![\![L]\!]\!]^{\text{need}}_x \mid y.[\![\![L']\!]\!]^{\text{need}}_y \mid xyz) \\
[\![\![\lambda x.L]\!]\!]^{\text{need}}_z & = & z{:}xy/[\![\![L]\!]\!]^{\text{need}}_y \\
[\![\![x]\!]\!]^{\text{need}}_z & = & \{z{=}x \mid \mathbf{tr}(x)\} \\
[\![\![\mathsf{let}\ y{=}L_2\ \mathsf{in}\ L_1]\!]\!]^{\text{need}}_z & = & \begin{cases} \text{if } L_2 \equiv V \\ \text{then } (\nu y)(y.[\![\![V]\!]\!]^{\text{need}}_y \mid \mathbf{tr}(y)^* \mid [\![\![L_1]\!]\!]^{\text{need}}_z) \\ \text{else } (\nu y)(y.[\![\![L_2]\!]\!]^{\text{need}}_y \mid [\![\![L_1]\!]\!]^{\text{need}}_z) \end{cases}
\end{array}$$

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

Definition 7.10
We define the relation S_z^{need} by $S_z^{\text{need}} = \{(L, E) \mid E \in [\![L]\!]_z^{\text{need}},\ L \text{ closed}\}$.

Proposition 7.11 (S_z^{need} *is a complexity simulation*)
The relation S_z^{need} is a complexity simulation for the embedding $[\![\ .\ ]\!]_z^{\text{need}}$ (restricted to closed terms) with indices 1, 0, 0.

Proof
We have to verify the properties (S1) , …, (S4) of a complexity simulation for an embedding according to Definitions 4.1 and 4.2.

(S1) For all closed L: $(L, [\![L]\!]_z^{\text{need}}) \in S_z^{\text{need}}$. This follows immediately from the definition of S_z^{need}.

(S2) We define the relation $\approx_A$ on expression of the δ-calculus (in analogy to π_0 before) such that $E \approx_A E'$ if and only if $\mathscr{C}_A(E) = \mathscr{C}_A(E')$. By Lemma 3.6, we know that $\to_T \cup \to_F \subseteq \approx_A$. For all L, L' and E there exists E' such that the following diagrams can be completed:

$$\begin{array}{ccc} L & \to_I & L' \\ S_z^{\text{need}} & & S_z^{\text{need}} \\ E & \approx_A \circ \to_A \circ \approx_A & \exists E' \end{array} \qquad \begin{array}{ccc} L & \to_V & L' \\ S_z^{\text{need}} & & S_z^{\text{need}} \\ E & \approx_A & \exists E' \end{array} \qquad \begin{array}{ccc} L & \to_{Ans} \cup \to_C & L' \\ S_z^{\text{need}} & & S_z^{\text{need}} \\ E & \equiv & \exists E' \end{array}$$

These diagrams will be proved by Lemmas 7.15, 7.16, and 7.17.

(S3) If $C(E) = \infty$ then $C_A(E) = \infty$ and the index for $\to_A$ is 1.

(S4) If a closed λ-term L is irreducible and $(L, E) \in S_z^{\text{need}}$ then $\mathscr{C}_A(E) = 0$. □

Lemma 7.12 (*Termination*)
If a closed λ-term L is irreducible and $E \in [\![L]\!]_z^{\text{need}}$ then there exist E' such that $E \to_T^* E'$ and $\mathscr{C}(E) = 0$.

Proof
If a closed λ-term L is irreducible then $L \equiv A$ for some answer A (It can be shown by induction on structure of L that either L is reducible or an answer). By induction on A we can show for all $E \in [\![A]\!]_z^{\text{need}}$ that E consists of a declaration scoping over composition of abstractions, delayed abstractions, triggers for delayed abstractions, and other delayed expressions without appropriate triggers. We obtain E' by triggering all delayed abstractions in E than can be triggered. □

A context B of the call-by-need λ-calculus can be considered as a function from λ-terms to λ-terms $\Lambda L.B[L]$. Given a variable z, we can translate a context B in call-by-need manner to a function $[\![B]\!]_z^{\text{need}}$ from λ-terms to sets of expression of the δ-calculus.

$$\begin{aligned}
[\![[\ ]]\!]_z^{\text{need}} &= \Lambda L.[\![L]\!]_z^{\text{need}} \\
[\![BL']\!]_z^{\text{need}} &= \Lambda L.(vx)(vy)([\![B]\!]_x^{\text{need}}(L) \mid [\![L']\!]_y^{\text{need}} \mid xyz) \\
[\![\text{let } y{=}L' \text{ in } B]\!]_z^{\text{need}} &= \Lambda L.(vy)(y.[\![L']\!]_y^{\text{need}} \mid [\![B_1]\!]_z^{\text{need}}(L)) \\
[\![\text{let } y{=}B_2 \text{ in } B_1[y]]\!]_z^{\text{need}} &= \Lambda L.\begin{cases} \text{if } B_2[L] \equiv V \\ \text{then } (vy)(y.[\![V]\!]_y^{\text{need}} \mid \mathbf{tr}(y)^* \mid [\![B_1[y]]\!]_z^{\text{need}}) \\ \text{else } (vy)(y.[\![B_2]\!]_y^{\text{need}}(L) \mid [\![B_1[y]]\!]_z^{\text{need}}) \end{cases}
\end{aligned}$$

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

Lemma 7.13 (*Translation and context application commute*)
For all B, L, and z the congruence $[\![B[L]]\!]^{\text{need}}_z = [\![B]\!]^{\text{need}}_z(L)$ holds.

Proof
By induction on the structure of contexts B.

1. In the base case, $B \equiv [\,]$, we have $[\![B[L]]\!]^{\text{need}}_z = [\![L]\!]^{\text{need}}_z = [\![B]\!]^{\text{need}}_z(L)$.
2. If $B \equiv B'L'$ then the induction hypothesis yields $[\![B'[L]]\!]^{\text{need}}_x = [\![B']\!]^{\text{need}}_x(L)$. Hence:
$$\begin{array}{lcl} [\![B[L]]\!]^{\text{need}}_z & = & (vx)(vy)([\![B'[L]]\!]^{\text{need}}_x \mid y.[\![L']\!]^{\text{need}}_y \mid xyz) \\ & = & (vx)(vy)([\![B']\!]^{\text{need}}_x(L) \mid y.[\![L']\!]^{\text{need}}_y \mid xyz) \\ & = & [\![B'L']\!]^{\text{need}}_z(L) \end{array}$$
3. The case $B \equiv \mathsf{let}\ y{=}L'\ \mathsf{in}\ B'$ is similar to the previous one.
4. In the case $B \equiv \mathsf{let}\ y{=}B_2\ \mathsf{in}\ B_1[y]$, the induction hypothesis implies $[\![B_2[L]]\!]^{\text{need}}_y = [\![B_2]\!]^{\text{need}}_y(L)$. If $B_2[L] \equiv V$ then:
$$\begin{array}{lcl} [\![\mathsf{let}\ y{=}B_2[L]\ \mathsf{in}\ B_1[y]]\!]^{\text{need}}_z & = & (vy)(y.[\![V]\!]^{\text{need}}_y \mid \mathbf{tr}(y)^* \mid [\![B_1[y]]\!]^{\text{need}}_z) \\ & = & [\![\mathsf{let}\ y{=}[\,]\ \mathsf{in}\ B_1[y]]\!]^{\text{need}}_z(V) \\ & = & [\![\mathsf{let}\ y{=}B_2\ \mathsf{in}\ B_1[y]]\!]^{\text{need}}_z(L) \end{array}$$
Otherwise $B_2[L] \not\equiv V$ for any V:
$$\begin{array}{lcl} [\![\mathsf{let}\ y{=}B_2[L]\ \mathsf{in}\ B_1[y]]\!]^{\text{need}}_z & = & (vy)(y.[\![B_2[L]]\!]^{\text{need}}_y \mid [\![B_1[y]]\!]^{\text{need}}_z) \\ & = & (vy)(y.[\![B_2]\!]^{\text{need}}_y(L) \mid [\![B_1[y]]\!]^{\text{need}}_z) \\ & = & [\![\mathsf{let}\ y{=}B_2\ \mathsf{in}\ B_1[y]]\!]^{\text{need}}_z(L) \end{array}$$

□

Given a binary relation $\to_R$ on expressions of the δ-calculus, we define a binary relation $\to_R$ on sets of expressions. If $\mathscr{E}_1$ and $\mathscr{E}_2$ are sets of expressions of the δ-calculus then $\mathscr{E}_1 \to_R \mathscr{E}_2$ holds if and only if for all $E_1 \in \mathscr{E}_1$ there exists $E_2 \in \mathscr{E}_2$ such that $E_1 \to_R E_2$.

$$\mathscr{E}_1 \to_R \mathscr{E}_2 \quad \text{iff} \quad \forall E_1 \in \mathscr{E}_1 \exists E_2 \in \mathscr{E}_2.\ E_1 \to_R E_2$$

For reasoning about call-by-need contexts B it is useful to introduce a notion of contexts for the δ-calculus. A corresponding notation for contexts D was already used in the proof of Theorem 5.5, i.e. $D ::= [\,] \mid D \mid E \mid E \mid D \mid v(x)D$. Note that $D[E] \mid E' \equiv D[E \mid E']$ for all E, E', D since we assume α-standardized expression.

Lemma 7.14 (*Translated needed arguments can be triggered*)
For all B, z there exists D, x such that $[\![B[L]]\!]^{\text{need}}_z \to^*_T D[[\![L]\!]^{\text{need}}_x]$ for all L.

Proof
By Lemma 7.13 it is sufficient to prove the existence of D and x such that for all L, $[\![B]\!]^{\text{need}}_z(L) \to^*_T D[[\![L]\!]^{\text{need}}_x]$. This can be done by induction on the structure of B. □

Lemma 7.15 (*Application*)
If $L \to_I L'$ then $[\![L]\!]^{\text{need}}_z \to^*_T \circ \to_A \circ \approx_1 \circ\ {}^*_T\!\leftarrow [\![L']\!]^{\text{need}}_z$.

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

In particular, if $L \to_I L'$ and $(L, E) \in S_z^{\text{need}}$ then there exists E' such that $E \approx_A \circ \to_A \circ \approx_A E'$ and $(L', E') \in S_z^{\text{need}}$.

Proof
We can assume that $L \equiv B[(\lambda y.L_2)L_1]$ and $L' \equiv B[\mathsf{let}\ y{=}L_2\ \mathsf{in}\ L_1]$ for some B, y, L_1 and L_2. Lemma 7.14 implies the existence of D and x such that for all L, $[\![B[L]]\!]_z^{\text{need}} \to_T^* D[[\![L]\!]_x^{\text{need}}]$. Hence:

$$\begin{array}{lll}
[\![L]\!]_z^{\text{need}} & \to_T^* & D[[\![(\lambda y.L_2)L_1]\!]_x^{\text{need}}] \\
& = & D[(\nu y_1)(\nu z_1)([\![\lambda y.L_2]\!]_{y_1}^{\text{need}} \mid z_1.[\![L_2]\!]_{z_1}^{\text{need}} \mid y_1 z_1 z)] \\
& \to_A & D[(\nu y_1)(\nu z_1)([\![\lambda y.L_2]\!]_{y_1}^{\text{need}} \mid z_1.[\![L_2]\!]_{z_1}^{\text{need}} \mid [\![L_2[z_1/y]]\!]_z^{\text{need}})] \\
& \approx_1 & D[[\![\mathsf{let}\ z_1{=}L_2\ \mathsf{in}\ L_2[z_1/y]]\!]_z^{\text{need}}] \\
& {}_T^*\!\leftarrow & [\![B[\mathsf{let}\ y{=}L_2\ \mathsf{in}\ L_2]]\!]_z^{\text{need}}
\end{array}$$

□

Lemma 7.16 (*Forwarding*)
If $L \to_V L'$ then $[\![L]\!]_z^{\text{need}} \to_T^* \circ \to_F \circ {}_T^*\!\leftarrow [\![L']\!]_z^{\text{need}}$.

In particular, if $L \to_V L'$ and $(L, E) \in S_z^{\text{need}}$ then there exists E' such that $E \approx_A E'$ and $(L', E') \in S_z^{\text{need}}$.

Proof
We can assume that $L \equiv B[\mathsf{let}\ y{=}V\ \mathsf{in}\ B'[y]]$ and $L' \equiv B[\mathsf{let}\ y{=}V\ \mathsf{in}\ B'[V]]$ for some B, B', y, and V. Lemma 7.14 implies the existence of D, x, D', and x' such that for all L, $[\![B[L]]\!]_z^{\text{need}} \to_T^* D[[\![L]\!]_x^{\text{need}}]$ and $[\![B'[L]]\!]_x^{\text{need}} \to_T^* D'[[\![L]\!]_{x'}^{\text{need}}]$. Hence:

$$\begin{array}{lll}
[\![L]\!]_z^{\text{need}} & \to_T^* & D[(\nu y)(y.[\![V]\!]_y^{\text{need}} \mid \mathbf{tr}(y)^* \mid [\![B'[y]]\!]_x^{\text{need}})] \\
& \to_T^* & D[(\nu y)(y.[\![V]\!]_y^{\text{need}} \mid \mathbf{tr}(y)^* \mid D'[x'{=}y \mid \mathbf{tr}(y)])] \\
& \to_T & D[(\nu y)([\![V]\!]_y^{\text{need}} \mid \mathbf{tr}(y)^* \mid D'[x'{=}y \mid \mathbf{tr}(y)])] \\
& \to_F & D[(\nu y)([\![V]\!]_y^{\text{need}} \mid \mathbf{tr}(y)^* \mid D'[[\![V]\!]_{x'}^{\text{need}} \mid \mathbf{tr}(y)])] \\
& {}_T\!\leftarrow & D[(\nu y)(y.[\![V]\!]_y^{\text{need}} \mid \mathbf{tr}(y)^* \mid D'[[\![V]\!]_{x'}^{\text{need}} \mid \mathbf{tr}(y)])] \\
& {}_T^*\!\leftarrow & D[(\nu y)(y.[\![V]\!]_y^{\text{need}} \mid [\![B'[V]]\!]_x^{\text{need}} \mid \mathbf{tr}(y)^*)] \\
& {}_T^*\!\leftarrow & [\![B[\mathsf{let}\ y{=}V\ \mathsf{in}\ B'[V]]]\!]_z^{\text{need}}
\end{array}$$

In the last line above, we have made use of the additional triggers introduced in $[\![L]\!]_z^{\text{need}}$ compared to $[\![L]\!]_z^{\text{need}}$. □

The *trigger equivalence* $\approx_4$ is the smallest equivalence on δ-expressions that is modulo congruence, closed under weak contexts, and that satisfies the following property for all x, y, E, and E':

$$y.(\nu x)(x.E \mid E') \approx_4 (\nu x)(x.E \mid y.E')$$

It is again not difficult to see that $E \approx_4 E'$ implies $E \approx_A E'$. The E' on the left hand side can only become active once y and x have been triggered. The same holds for E' on the right hand side since a trigger for y can only be computed in E which requires that x has to be triggered.

Lemma 7.17 (*Administration*)
If $L \to_{Ans} \cup \to_C L'$ then $[\![L]\!]_z^{\text{need}} \to_T^* \circ \approx_4 \circ {}_T^*\!\leftarrow [\![L']\!]_z^{\text{need}}$

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

In particular, if $L \to_{Ans} \cup \to_C L'$ and $(L, E) \in S_z^{\text{need}}$ then there exists E' such that $E \approx_A E'$ and $(L', E') \in S_z^{\text{need}}$.

Proof

We first consider $L \to_{Ans} L'$. We can assume that there exist y, y', L'', A such that $L \equiv B[\mathsf{let}\ y{=}(\mathsf{let}\ y'{=}L''\ \mathsf{in}\ A)\ \mathsf{in}\ B'[y]]$ and $L' \equiv B[\mathsf{let}\ y'{=}L''\ \mathsf{in}\ (\mathsf{let}\ y{=}A\ \mathsf{in}\ B'[y])]$. Lemma 7.14 implies the existence of D and x such that $[\![B[L]]\!]_z^{\text{need}} \to_T^* D[[\![L]\!]_x^{\text{need}}]$ for all L. In the case that L'' is not an abstraction, we have:

$$\begin{array}{rcl}
[\![L]\!]_z^{\text{need}} & = & [\![B[\mathsf{let}\ y{=}(\mathsf{let}\ y'{=}L''\ \mathsf{in}\ A)\ \mathsf{in}\ B'[y]]]\!]_z^{\text{need}} \\
& \to_T^* & D[(\nu y)(y.(\nu y')(y'.[\![L'']\!]_{y'}^{\text{need}} \mid [\![A]\!]_y^{\text{need}}) \mid [\![B'[y]]\!]_x^{\text{need}})] \\
& \approx_4 & D[(\nu y)(\nu y')(y'.[\![L'']\!]_{y'}^{\text{need}} \mid y.[\![A]\!]_y^{\text{need}}) \mid [\![B'[y]]\!]_x^{\text{need}}] \\
& = & D[(\nu y')(y'.[\![L'']\!]_{y'}^{\text{need}} \mid [\![\mathsf{let}\ y{=}A\ \mathsf{in}\ B'[y]]\!]_x^{\text{need}})] \\
& {}_T^*\!\leftarrow & [\![B[\mathsf{let}\ y'{=}L''\ \mathsf{in}\ (\mathsf{let}\ y{=}A\ \mathsf{in}\ B'[y])]]\!]_z^{\text{need}}
\end{array}$$

The consideration for $L'' \equiv V$ for some V is similar. We second consider $L \to_C L'$. We can assume that there exist y, y', L'', A such that $L \equiv B[(\mathsf{let}\ y{=}L_1\ \mathsf{in}\ A)L_2]$ and $L' \equiv B[\mathsf{let}\ y{=}L_1\ \mathsf{in}\ AL_2]$. Lemma 7.14 implies the existence of D and x such that $[\![B[L]]\!]_z^{\text{need}} \to_T^* D[[\![L]\!]_x^{\text{need}}]$ for all L. In the case that L_1 is not an abstraction, we have:

$$\begin{array}{rcl}
[\![L]\!]_z^{\text{need}} & = & [\![B[(\mathsf{let}\ y{=}L_1\ \mathsf{in}\ A)L_2]]\!]_z^{\text{need}} \\
& \to_T^* & D[(\nu x')(\nu y')((\nu y)(y.[\![L_1]\!]_y^{\text{need}} \mid [\![A]\!]_{x'}^{\text{need}}) \mid y'.[\![L_2]\!]_{y'}^{\text{need}} \mid x'y'x)] \\
& = & D[(\nu y)(y.[\![L_1]\!]_y^{\text{need}} \mid (\nu x')(\nu y')([\![A]\!]_{x'}^{\text{need}} \mid y'.[\![L_2]\!]_{y'}^{\text{need}} \mid x'y'x))] \\
& {}_T^*\!\leftarrow & [\![B[\mathsf{let}\ y{=}L_1\ \mathsf{in}\ AL_2]]\!]_z^{\text{need}}
\end{array}$$

The case that L_1 is not an abstraction is similar again. □

8 Call-by-need versus call-by-value complexity

We are now in the position to compare call-by-need complexity and call-by-value complexity. It is not difficult to find a variation of a simulation for the embedding $[\![M]\!]_z^{\text{need}} \mapsto [\![M]\!]_z^{\text{val}}$ since $[\![M]\!]_z^{\text{need}}$ coincides with $[\![M]\!]_z^{\text{val}}$ up to some delay and trigger expressions. Let $\mathsf{0}$ be the garbage expression $(\nu x)x$. Note that $E \mid \mathsf{0} \approx_1 E$ holds for all E where $\approx_1$ is the garbage collection equivalence (lifted from π_0 to the δ-calculus). We define a projection function π on δ-expressions which replaces all trigger expressions by $\mathsf{0}$ and removes all delays.

$$\begin{array}{lll}
\pi(x{:}\bar{y}/E) \equiv x{:}\bar{y}/\pi(E) & \pi(x\bar{y}) \equiv x\bar{y} & \pi(E \mid F) \equiv \pi(E) \mid \pi(F) \\
\pi((\nu x)E) \equiv (\nu x)\pi(E) & \pi(\mathbf{tr}(x)) \equiv \mathsf{0} & \pi(x.E) \equiv \pi(E)
\end{array}$$

Let $\equiv_1$ be the smallest congruence on expressions of the δ-calculus, which contains the garbage collection equivalence $\approx_1$. Again, it is not difficult to show that $E \equiv_1 E'$ implies $C_A(E) = C_A(E')$.

Definition 8.1

Let $\mathrm{S}_{\text{val}}^{\text{need}}$ be the relation $\{(E, E') \mid \pi(E) \equiv_1 E'\}$ on expressions of the δ-calculus.

https://doi.org/10.1017/S0956796800003762 Published online by Cambridge University Press

Lemma 8.2
The following diagrams can be completed for all E and E'.

$$\begin{array}{ccc} E & \to_A & E' \\ {\scriptstyle \mathrm{S}_{\mathrm{val}}^{\mathrm{need}}} & & {\scriptstyle \mathrm{S}_{\mathrm{val}}^{\mathrm{need}}} \\ \pi(E) & \to_A & \pi(E') \end{array} \qquad \begin{array}{ccc} E & \to_F & E' \\ {\scriptstyle \mathrm{S}_{\mathrm{val}}^{\mathrm{need}}} & & {\scriptstyle \mathrm{S}_{\mathrm{val}}^{\mathrm{need}}} \\ \pi(E) & \to_F & \pi(E') \end{array} \qquad \begin{array}{ccc} E & \to_T & E' \\ {\scriptstyle \mathrm{S}_{\mathrm{val}}^{\mathrm{need}}} & & {\scriptstyle \mathrm{S}_{\mathrm{val}}^{\mathrm{need}}} \\ \pi(E) & \equiv & \pi(E') \end{array}$$

Proposition 8.3
For every z and a closed λ-term M: $\quad \mathscr{C}_A([\![M]\!]_z^{\mathrm{need}}) \leqslant \mathscr{C}_A([\![M]\!]_z^{\mathrm{val}})$.

Proof
Lemma 8.2 and Proposition 4.5 yields $\mathscr{C}_A(E) \leqslant \mathscr{C}_A(F)$ for all pairs $(E,F) \in \mathrm{S}_{\mathrm{val}}^{\mathrm{need}}$ of admissible expressions of the δ-calculus. Let z be a variable and M a closed λ-expression. It is not difficult to verify $\pi([\![M]\!]_z^{\mathrm{need}}) \equiv_1 [\![M]\!]_z^{\mathrm{val}}$. Hence, $([\![M]\!]_z^{\mathrm{need}}, [\![M]\!]_z^{\mathrm{val}}) \in \mathrm{S}_{\mathrm{val}}^{\mathrm{need}}$ such that $\mathscr{C}_A([\![M]\!]_z^{\mathrm{need}}) \leqslant \mathscr{C}_A([\![M]\!]_z^{\mathrm{val}})$ follows. □

Corollary 8.4 (*Folk theorem*)

For every closed λ-term M the call-by-need complexity of M is smaller than its call-by-value complexity.

$$\mathscr{C}^{\mathrm{need}}(M) \leqslant \mathscr{C}^{\mathrm{val}}(M)$$

Proof
From Theorems 6.6 and 7.9 and Proposition 8.3. □

9 Conclusion

We have investigated uniform confluence in concurrent computation. We have embedded the λ-calculus with call-by-value and call-by-need reduction into the π-calculus such that complexity is preserved. We have worked out a powerful proof technique based on uniform confluence and simulations. We have proved that call-by-need complexity is smaller than call-by-value complexity.

Acknowledgements

I am deeply in debt to Gert Smolka, who initiated this work and contributed ideas during many discussions. It is an honor for me to thank Phil Wadler's for his engaged support. Without Phil, I would never have succeeded in revising the first version of this article according to the proposals of the referees. I would like to thank to the referees for careful reading and useful comments. It's my pleasure to thank Martin Müller for daily comments on concepts and related work and for extremely helpful discussions on notations and details. Denys Duchier deserves many thanks for reading and commenting on the final version. I would like to thank Kai Ibach, Peter Van Roy, Christian Schulte for their comments and the complete Oz team for continuous support and interest during the period of 5 years I needed for getting this work down to earth.

References

Abadi, M., Cardelli, L., Curien, P.-L. and Lévy, J.-J. (1991) Explicit substitutions. *J. Functional Programming*, **1**(4), 375–416.

Ariola, Z. M. and Felleisen, M. (1997) The call-by-need lambda calculus. *J. Functional Programming*, **7**(3).

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M. and Wadler, P. (1995) A call-by-need lambda calculus. *ACM Symposium on Principles of Programming Languages*, pp. 233–246.

Armstrong, J., Williams, R., Virding, M. and Wikstroem, C. (1996) *Concurrent Programing in Erlang*. Prentice-Hall. 2nd ed.

Arvind, Nikhil, R. S. and Pingali, K. K. (1989) I-structures: data-structures for parallel computing. *ACM Trans. Programming Languages and Systems*, **4**(11), 598–632.

Asperti, A. (1997) *P = NP, up to sharing*. Dipartimento di Scienze dell'Informatizione, Bologna. (Available at ftp://ftp.cs.unibo.it/pub/asperti/pnp.ps.gz.)

Barendregt, H. P. (1981) *The lambda calculus. Its syntax and semantics*. Studies in Logic and the Foundations of Mathematics, vol. 103. Elsevier.

Boudol, G. (1992) *Asynchrony and the π-calculus (note)*. Rapport de Recherche 1702. INRIA, Sophia Antipolis, France.

Brock, S. and Ostheimer, G. (1995) Process semantics of graph reduction. *6th International Conference on Concurrency Theory*, pp. 238–252.

Dershowitz, N. and Jouannaud, J.-P. (1990) *Rewrite systems*. Vol. B, pp. 243–320. MIT Press.

Fournet, C. and Gonthier, G. (1996) The Reflexive CHAM and the Join-Calculus. *23rd ACM Symposium on Principles of Programming Languages*, pp. 372–385.

Fournet, C. and Maranget, L. (1997) *Join Calculus Language*. INRIA Rocquencourt, `http://pauillac.inria.fr/join`.

Honda, K. and Tokoro, M. (1991) An object calculus for asynchronous communication. In: America, P. (ed.), *European Conference on Object-Oriented Programming*, pp. 133–147. *Lecture Notes in Computer Science 512*, Springer-Verlag.

Huet, G. (1980) Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, **27**(4), 797–821.

Jeffrey, A. (1994) A fully abstract semantics for concurrent graph reduction. *IEEE Symposium on Logic in Computer Science*, pp. 82–91.

Klop, J. W. (1987) Term rewriting systems: A tutorial. *Bulletin of the European Association on Theoretical Computer Science*, **32**, 143–182.

Kobayashi, N., Pierce, B. and Turner, D. N. (1996) Linearity and the pi-calculus. *ACM Symposium on Principles of Programming Languages*, pp. 358–371.

Launchbury, J. (1993) A natural semantics for lazy evaluation. *ACM Symposium on Principles of Programming Languages*, pp. 144–154.

Maher, M. J. (1987) Logic semantics for a class of committed-choice programs. In: Lassez, J.-L. (ed.), *4th International Conference on Logic Programming*, pp. 858–876.

Maraist, J., Odersky, M., Turner, D. and Wadler, P. (1995) Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. *11th International Conference on the Mathematical Foundations of Programming Semantics*.

Maraist, J., Odersky, M. and Wadler, P. (1998) The call-by-need lambda calculus. *J. Functional Programming*, **8**(3), 275–317.

Maranget, L. (1990) Optimal derivations in weak lambda-calculi and in orthogonal term rewriting systems. *ACM Symposium on Principles of Programming Languages*, pp. 255–269.

Maranget, L. (1992) *La Stategie Paresseuse*. Thèse doctorale, Université Paris VII.

Milner, R. (1991) *The polyadic π-calculus: A tutorial*. ECS-LFCS Report Series 91–180. Laboratory for Foundations of Computer Science, University of Edinburgh.

Milner, R. (1992) Functions as processes. *J. Mathematical Structures in Computer Science*, **2**(2), 119–141.

Moran, A. and Sands, D. (1999) Improvement in a lazy context: An operational theory of call-by-need. *26th ACM Symposium on Principles of Programming Languages*, pp. 43–56.

Nestmann, U. (1996) *On Determinacy and Nondeterminacy in Concurrent Programming*. Dissertation, technische Fakultät, Universität Erlangen.

Nestmann, U. and Pierce, B. (1996) Decoding choice encodings. In: Montanari, U. and Sassone, V. (eds.), *7th International Conference on Concurrency Theory: Lecture Notes in Computer Science 1119*, pp. 179–194.

Niehren, J. (1994) *Funktionale Berechnung in einem uniform nebenläufigen Kalkül mit logischen Variablen*. Dissertation, Universität des Saarlandes, Saarbrücken.

Niehren, J. (1996) Functional computation as concurrent computation. *23th ACM Symposium on Principles of Programming Languages*, pp. 333–343.

Niehren, J. (1999) *Uniform confluence in concurrent computation, unabridged*. Technical report, Programming Systems Lab, Universität des Saarlandes, Saarbrücken. (Available at `http://www.ps.uni-sb.de/Papers/abstracts/Uniform:99.html`.)

Niehren, J. and Müller, M. (1995) Constraints for Free in Concurrent Computation. In: Kanchanasut, K. and Lévy, J.-J. (eds.), *Asian Computing Science Conference: Lecture Notes in Computer Science 1023*, pp. 171–186.

Niehren, J. and Smolka, G. (1994) A confluent relational calculus for higher-order programming with constraints. *1st International Conference on Constraints in Computational Logics: Lecture Notes in Computer Science 845*, pp. 89–104.

Philippou, A. and Walker, D. (1997) On confluence in the π-calculus. *International Conference on Automata, Languages, and Programming: Lecture Notes in Computer Science 1256*, pp. 314–324. Springer-Verlag.

Pierce, B. C. and Turner, D. N. (1997) Pict: A Programming Language Based on the Pi-Calculus. *Milner Festschrift*. MIT Press.

Pingali, K. K. (1987) *Lazy Evaluation and the Logic Variable*. Technical report, Cornell University. Proceedings of the Institute on Declarative Programming.

Plotkin, G. D. (1975) Call-by-name, call-by-value and the λ-calculus. *J. Theor. Comput. Sci.*, **1**, 125–159.

Purushothaman, S. and Seaman, J. (1992) An adequate operational semantics of sharing in lazy evaluation. *European Symposium on Programming: Lecture Notes in Computer Science 582*. Springer-Verlag.

Reppy, J. H. (1992) Concurrent ML: Design, Application and Semantics. *Functional Programming, Concurrency, Simulation and Automated Reasoning: Lecture Notes in Computer Science 693*, pp. 165–198. Springer-Verlag.

Sangiorgi, D. (1996) A Theory of Bisimulation for π-Calculus. *Acta Informatica*, **33**.

Sangiorgi, D. (1997) The name discipline of uniform receptiveness. *International Conference on Automata, Languages, and Programming*, pp. 303–313.

Saraswat, V. A., Rinard, M. and Panangaden, P. (1991) Semantic foundations of concurrent constraint programming. *ACM Symposium on Principles of Programming Languages*, p. 333–352.

Smolka, G. (1994) A foundation for concurrent constraint programming. *1st International Conference on Constraints in Computational Logics: Lecture Notes in Computer Science 845*, pp. 50–72. Springer-Verlag.

Smolka, G. (1995) The Oz programming model. In: van Leeuwen, J. (ed.), *Computer Science Today: Lecture Notes in Computer Science 1000*, pp. 324–343.

Smolka *et al.* (1995) *The Oz Programming System*. Programming Systems Lab, Universität des Saarlandes. (Available at `www.mozart-oz.de`.)

Thomsen, B., Leth, L., Prasad, S., Kuo, T.-M., Kramer, A., Knabe, F. and Giacalone, A. (1993) *Facile Antigua release programming guide*. Technical Report ECRC-93-20. ECRC, European Computer-Industry Research Centre.

Turner, D. N. (1996) *The polymorphic pi-calculus: Theory and implementation*. PhD thesis, University of Edinburgh. (Available as LFCS report ECS-LFCS-96-345.)

Victor, B. and Parrow, J. (1996) Constraints as Processes. In: Montanari, U. and Sassone, V. (eds.), *7th International Conference on Concurrency Theory: Lecture Notes in Computer Science 1119*, p. 389–405.

Yoshida, N. (1993) Optimal reduction in weak λ-calculus with shared environments. *ACM Conference on Functional Programming Languages and Computer Architecture*.