Functional Computation as Concurrent Computation

Joachim Niehren*

German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
niehren@dfki.uni-sb.de

Abstract

We investigate functional computation as a special form of concurrent computation. As formal basis, we use a uniformly confluent core of the π-calculus, which is also contained in models of higher-order concurrent constraint programming. We embed the call-by-need and the call-by-value λ-calculus into the π-calculus. We prove that call-by-need complexity is dominated by call-by-value complexity. In contrast to the recently proposed call-by-need λ-calculus, our concurrent call-by-need model incorporates mutual recursion and can be extended to cyclic data structures by means of constraints.

1 Introduction

We investigate concurrency as unifying computational paradigm in the spirit of Milner [Mil92] and Smolka [Smo94, Smo95b]. Whereas the motivations for both approaches are quite distinct, the resulting formalisms are closely related: The π-calculus [MPW92] models communication and synchronisation via channels, whereas the ρ-calculus [NS94, Smo94, NM95][1] uses logic variables or more generally constraints as inspired by [Mah87, SRP91].

Our motivation in concurrent calculi lies in the design of programming languages. Concurrency enables us to integrate multiple programming paradigms such as functional [Mil92, Smo94, Nie94, Iba95, PT95b], object-oriented [Vas94, PT95a, HSW95, Wal95], and constraint programming [JH91, SSW94]. All these paradigms are supported by the programming language Oz [Smo95a, Smo95b].

In this paper, we model the time complexity of eager and lazy functional computation in a concurrent calculus. The importance of complexity is three-fold:

1. Every implementation-oriented model has to reflect complexity. In the case of lazy functional programming, the consideration of complexity leads to a call-by-need model in contrast to a call-by-name model.
2. A functional programmer has to reason about the complexity of his programs [San95]. Denotational semantics are too abstract in general.
3. Based on the notion of uniform confluence, complexity arguments provide for powerful proof techniques.

Our main technical result is that call-by-need complexity is dominated by call-by-value and call-by-name complexity, i.e. for all closed λ-expressions M:

$$\mathcal{C}_{\text{need}}(M) \;\leq\; \min\{\mathcal{C}_{\text{value}}(M),\, \mathcal{C}_{\text{name}}(M)\}$$

These two estimations can be interpreted as follows: Call-by-need reduction shares the evaluation of functional arguments and evaluates only needed arguments.

As a formal basis, we use a uniformly confluent applicative core of a concurrent calculus that we call δ_0-calculus. This is a proper subset of the polyadic asynchronous π-calculus [Mil91, HT91, Bou92] and of the ρ-calculus [NM95, Smo94], the latter being a foundation of higher-order concurrent constraint programming. The choice of δ_0 has the following advantages:

1. Delay and triggering mechanisms as needed for programming laziness are expressible within δ_0.
2. Mutually recursive definitions are expressible in a call-by-value and a call-by-need manner.
3. Cyclic data structures and the corresponding equality relations are expressible in an extension of δ_0 with constraints, the ρ-calculus.

The δ_0-calculus is defined via expressions, structural congruence, and reduction. Expressions are formed by abstraction, application, composition, and declaration:

$$E, F \;::=\; x{:}\overline{y}/E \;\mid\; x\overline{y} \;\mid\; E|F \;\mid\; (\nu x)E$$

In the terminology of the π-calculus, abstractions are replicated input-agents and applications are output-agents. Once-only input-agents as in the π-calculus are not provided, nor constraints or cells as in the ρ-calculus.

We identify expressions up to the structural congruence of the π-calculus. Reduction in δ_0 is defined by the following application axiom:

$$(x{:}\overline{y}/E) \mid x\overline{z} \;\rightarrow\; (x{:}\overline{y}/E) \mid E[\overline{z}/\overline{y}]$$

*The research reported herein has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FTZ-ITW-9105), the Esprit Project ACCLAIM (PE 7195), and the Esprit Working Group CCL (EP 6028)

To appear in the 23rd ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, January 21-24, 1996.

[1] Originally, Smolka's γ-calculus [Smo94] and the ρ-calculus [NS94] have been technically distinct. In [NM95], they have been combined in a refined version of the ρ-calculus. We note that Smolka's γ-calculus and Boudol's γ-calculus [Bou89] are completely unrelated.

We do not allow for reduction below abstraction. In terms of the λ-calculus, this means that we consider standard reduction only.

We embed the call-by-value and the call-by-name λ-calculus into δ_0, the latter with call-by-need complexity. This is done in two steps: We first extend δ_0 by adding mechanisms for single assignment, delay, and triggering. We obtain a new calculus that we call δ-calculus. Surprisingly δ can be embedded into δ_0 itself. The idea is to express single assignment by forwarders. In the second step, we encode the above mentioned λ-calculi into δ. Formulating these embeddings into δ rather than into δ_0 is motivated by our belief that the abstraction level of δ is relevant for programming, theory, and implementation.

The notion of single assignment we use in δ is known from a directed usage of logic variables [Pin87], as for instance in the data-flow language Id [ANP89, BNA91]. Alternatively, we could express single assignment via equational constraints, but these are not available in the π-calculus. In fact, the directed single assignment mechanism in this paper is motivated by a data-flow discussion for polymorphic typing a concurrent constraint language [Mül95].

The approach of this paper is based on the idea of uniform confluence [Nie94, NS94]. This is a simple criterion that ensures complexity is independent of the execution order. Unfortunately, we can not even expect confluence for δ_0. This is due to expressions such as $x{:}y/E \mid x{:}y/F$ that we consider inconsistent. Inconsistencies may arise dynamically. We can however exclude them statically by a linear type system. In fact, the restriction of δ_0 to well-typed expressions is uniformly confluent and sufficiently rich for embedding λ-calculi. We note that a well-typed first-order restriction of δ_0 has been proved confluent in [SRP91].

We base all our adequacy proofs for embeddings on a novel technique that combines uniform confluence and shortening simulations [Nie94, NS94]. Shortening simulations are more powerful than bisimulations, once uniform confluence is available. Nevertheless, the definitions of concrete shortening simulations in this paper are strongly inspired by Milner's bisimulations in [Mil92].

We are able to compare the complexity of call-by-need and call-by-value in δ, since up to our embeddings, every call-by-need step is also a call-by-value step. In particular, we do not require in δ that a call-by-value function evaluates its arguments before application. This additional freedom compared to the call-by-value λ-calculus does not affect complexity. This is a consequence of the uniform confluence of the well-typed restriction of δ.

Related Work. Many call-by-need models have been proposed over the last years but none of them has been fully satisfactory.

Our call-by-need model is closely related to the call-by-need λ-calculus of Ariola et al. [AFMOW95]. We show how to embed the call-by-need λ-calculus into δ such that complexity is preserved (but not vice versa). The main difference between both approaches is the level on which lazy control is defined. In the case of the call-by-need λ-calculus, laziness is defined on meta level, by evaluation contexts. In the case of the δ-calculus, laziness is expressible within the language itself. In other words, the call-by-need λ-calculus is more abstract, or, the δ-calculus is more general. The disadvantage of the abstraction level of the call-by-need λ-calculus is that mutual recursion and cyclic data structures are difficult to define. On the other hand side, δ is abstract enough for hiding most implementation details. We illustrated this fact by simple complexity reasoning based on shortening simulations and uniform confluence. This technique is again more general than the specialised λ-calculus technique in [AFMOW95].

The setting of the call-by-need λ-calculus is quite similar to Yoshida's λf-calculus [Yos93]. She proves that a call-by-need reduction strategy is optimal for weak reduction, but she does not compare call-by-need to call-by-name.

Embeddings of the call-by-value and the call-by-name λ-calculus into the π-calculus have been proposed and proved correct by Milner [Mil92]. A embedding of the call-by-need λ-calculus into the π-calculus is proved correct in [BO95]. The advantage of the here presented embeddings is that they do need not make use of once-only input channels, which are incompatible with uniform confluence.

Embeddings of the call-by-value and the call-by-name λ-calculus into the ρ-calculus are presented in [Smo94], the latter with call-by-need complexity. These embeddings motivated those presented here. The difference lies in the usage of constraints for single assignment and triggering. In [Smo94] no proofs are given, but the call-by-value embedding is proved correct in [Nie94]. There, most of the proof techniques presented in this paper have been introduced.

An abstract big-step semantics for call-by-need has been presented by Launchbury [Lau93]. It is complexity sensitive, since computation steps are reflected in proof trees. Launchbury's correctness result however does not cover complexity. This is a consequence of using a proof technique based on denotational semantics.

Many other attempts for call-by-need have been presented. To our knowledge, all of them are quite implementation oriented such that they suffer from low-level details. We note the approaches based on explicit substitutions [PS92, ACCL91] and on graph reduction [Jef94].

Structure of the Paper. As a first example we discuss the square function in a concurrent setting. We define δ_0 in Section 3. We then introduce the notion of uniform confluence and discuss its relationship to complexity and confluence. In Section 5, we prove uniform confluence for a subset of δ_0. In Section 6, we define the δ-calculus. Following, we discuss uniform confluence for δ. In Sections 8 and 9, we embed the call-by-value, the call-by-name, and the call-by-need λ-calculus into δ. We introduce a linear type system in Section 10 and prove that our embeddings fall into the uniformly confluent subset of δ. In Section 11, we show how to encode single assignment and triggering in δ_0. We introduce the simulation proof technique in Section 12 and apply it for proving the adequacy our calculus embeddings in Sections 13 and 14.

2 The Square Function: An Example

We informally introduce the δ-calculus by representing the square function in call-by-value and call-by-need manner. This motivates our embeddings of λ-calculi into δ and indicates the adequacy results we can expect.

We assume a infinite set of variables ranged over by x, y, z, s, and t. Sequences of variables are written as $\overline{x}$, $\overline{y}$, ... and integers are denoted with n, m, and k.

In a concurrent setting, we consider functions as relations

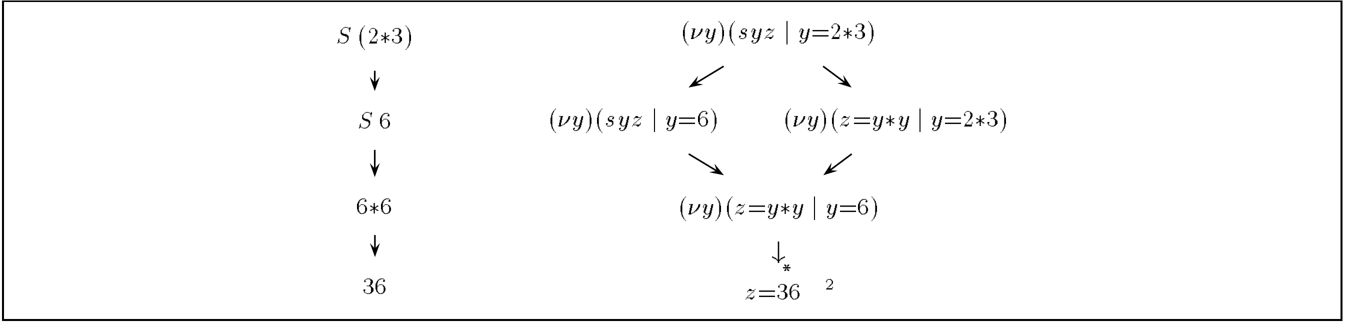

Figure 1: Square Function: Call-by-Value

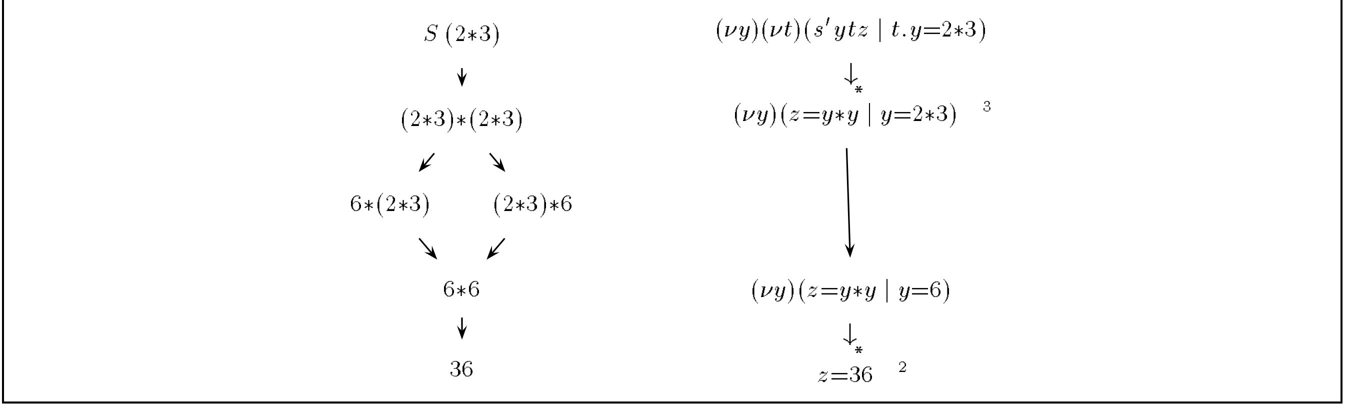

Figure 2: Square Function: Call-by-Name versus Call-by-Need

with an explicit output argument, for example:

$$S \;=\; \lambda x.x{*}x \qquad \text{versus} \qquad s{:}xz/z{=}x{*}x$$

The expression on the right-hand side is a call-by-value definition of the square function in the δ-calculus. The formal parameter z is the explicit output argument. The expression $z{=}x{*}x$ is syntactic sugar for an application of a predefined ternary relation $*$. We assume the following application axiom for all integers n, m, k and variables x:

$$x{=}n{*}m \;\rightarrow\; x{=}k \qquad \text{if } k = n * m$$

For forwarding values in equations $x{=}n$, we copy them into those positions where they are needed. This kind of administration is definable in many different manners, for instance:

$$(\nu y)(y{=}n \mid E) \;\rightarrow\; E[n/y]$$

Figure 1 illustrates the call-by value evaluation of the square of $2{*}3$ in the λ-calculus and the δ-calculus. If we ignore forwarding steps, then all possible computations in Figure 1 have length 3. In other words, our call-by-value embedding of the square function preserves time complexity measured in terms of application steps. Ignoring forwarding is correct in the sense that the number of forwarding steps in computations of functional expressions is linearly bounded by the number of application steps. We do no prove this claim formally.

It is interesting that call-by-value evaluation in δ is more flexible than in the λ-calculus, as shown by an additional call-by-value computation in our example. This is in the rightmost computation in Figure 1, where the square function is applied before its argument has been evaluated.

For defining a call-by-need square function in a concurrent setting, we need a delay and a triggering mechanism. For this purpose, we introduce two new expressions $t.E$ and $\mathbf{tr}(t)$. We say that E is delayed in $t.E$ until t is triggered. This behaviour can be provided by following triggering axiom:

$$t.E \mid \mathbf{tr}(t) \;\rightarrow\; E \mid \mathbf{tr}(t)$$

Note that multiple triggering is possible. A call-by-need version of the square function can be defined as follows:

$$s'{:}xtz/(z{=}x{*}x \mid \mathbf{tr}(t))$$

This function can be applied with a delayed argument x waiting on t to be triggered. Figure 2 presents call-by-name and call-by-need computations of the square of $2{*}3$. Both call-by-name computations have length 4, since the functional argument $2{*}3$ is evaluated twice. If we ignore triggering and forwarding steps, then our call-by-need computation has length 3. This illustrates that call-by-need complexity is dominated by call-by-name and by call-by-value complexity. In this example, the first estimation is proper (raised

[2] Here, $\rightarrow^*$ stands a forwarding followed by an application step:

$$(\nu y)(z{=}y{*}y \mid y{=}6) \;\rightarrow\; z{=}6{*}6 \;\rightarrow\; z{=}36$$

[3] Here, $\rightarrow^*$ consists of an application and a triggering step:

$$\begin{aligned}(\nu t)(s'ytz \mid t.y{=}2{*}3) \;&\rightarrow\; (\nu t)(z{=}y{*}y \mid \mathbf{tr}(t) \mid t.y{=}2{*}3)\\ &\rightarrow\; z{=}y{*}y \mid y{=}2{*}3 \mid (\nu t)(\mathbf{tr}(t))\end{aligned}$$

The garbage expression $(\nu t)(\mathbf{tr}(t))$ is omitted in Figure 2.

Variables	$x, y, z, s, t \ ::=$
Expressions	$E, F \ ::= \ x{:}\overline{y}/E \ \mid \ x\overline{y} \ \mid \ E \mid F \ \mid \ (\nu x)E$
Reduction	$x{:}\overline{y}/E \mid x\overline{z} \ \to_A \ x{:}\overline{y}/E \mid E[\overline{z}/\overline{y}]$

Figure 3: The δ_0-Calculus.

Structural Congruence	$E \mid F \ \equiv \ F \mid E$	$E_1 \mid (E_2 \mid E_3) \ \equiv \ (E_1 \mid E_2) \mid E_3$
	$(\nu x)(\nu y)E \ \equiv \ (\nu y)(\nu x)E$	$(\nu x)E \mid F \ \equiv \ (\nu x)(E \mid F)$ if $x \notin \mathcal{V}(F)$
	$E \equiv F$ if $E =_\alpha F$	
Contextual Rules	$\dfrac{E \to E'}{E \mid F \to E' \mid F}$ $\quad \dfrac{E \to E'}{(\nu x)E \to (\nu x)E'}$	$\dfrac{E_1 \equiv E_2 \quad E_2 \to F_2 \quad F_2 \equiv F_1}{E_1 \to F_1}$

Figure 4: Structural Congruence and Contextual Rules

by sharing), whereas the second is not (since the argument of the square function is needed).

We note that our call-by-need computation in Figure 2 has a direct relative in the call-by-value case, the rightmost computation in Figure 1. This statement holds in general and enables us to compare call-by-need and call-by-value complexity in the δ-calculus.

3 The Applicative Core of the π-Calculus

We define δ_0 as the applicative core of the polyadic asynchronous π-calculus [Mil91, HT91, Bou92] and the ρ-calculus [NM95, Smo94]. Interestingly, δ_0 as formulated here is part of the Oz computation model [Smo94] and the Pict computation model [PT95b], which have been developed independently.

We define the calculus δ_0 via expressions, structural congruence, and reduction. The definition is given in Figures 3 and 4. Expressions are abstractions, applications, compositions, or declarations. An *abstraction* $x{:}\overline{y}/E$ is named by x, has *formal arguments* $\overline{y}$ and *body* E. An *application* $x\overline{y}$ of x has *actual arguments* $\overline{y}$. In the standard π-notation, abstractions are replicated input-agents and applications asynchronous output-agents.

Bound variables are introduced as formal arguments of abstractions and by declaration. The set of free variables of an expression E is denoted by $\mathcal{V}(E)$. We write $E =_\alpha F$ if E and F are equal up to consistent renaming of bound variable. As usual for λ-calculi, we assume all expressions to be α-standardised and omit freeness conditions throughout the paper.

The *structural congruence* $\equiv$ of δ_0 coincides with that of the π-calculus. It is the least congruence on expressions satisfying the axioms in Figure 4. With respect to the structural congruence, bound variables can be renamed consistently, composition is associative and commutative, and declaration is equipped with the usual scoping rules.

The *reduction* $\to$ synonymously denoted by $\to_A$ is defined by a single axiom for *application*. The application axiom uses the simultaneous substitution operator $[\overline{z}/\overline{y}]$, which replaces the components of $\overline{y}$ elementwise by $\overline{z}$. We implicitly assume in case of application of $[\overline{z}/\overline{y}]$ that the sequence $\overline{y}$ is linear and of the same length as $\overline{z}$. Note that reduction is invariant under structural congruence and closed under weak contexts. This means that reduction is applicable below declaration and composition, but not inside of abstraction. In terms of λ-calculi, this means that consider standard reductions only.

Example 3.1 (Continuation Passing Style) The identify function $I = \lambda x.x$ can be defined in δ_0 in continuation passing style: $i{:}xy/yx$. An application let $i{=}I$ in ii referred to by z is definable as follows:

$$(\nu i)(i{:}xy/yx \mid (\nu y')(iiy' \mid y'{:}c/zc))$$

In composition with $i{:}xy/yx$ we obtain the following computation:

$$\begin{aligned}(\nu y')(\boxed{iiy'} \mid y'{:}c/zc) \ &\to_A \ (\nu y')(\boxed{y'i} \mid y'{:}c/zc) \\ &\to_A \ zi \mid (\nu y')(y'{:}c/zc)\end{aligned}$$

Example 3.2 Explicit Recursion The computation of the following recursive expression does not terminate:

$$\boxed{xy} \mid x{:}y/xy \ \to_A \ \boxed{xy} \mid x{:}y/xy \ \to_A \ \ldots$$

Compared to the asynchronous π-calculus [Mil91, Bou92, HT91], δ_0 does not provide for non-replicated input-agents. These are not needed for functional computation and are incompatible with uniform confluence if not restricted linearly [KPT96]. In absence of once-only inputs, it is not clear if the unary restriction of δ_0 is Turing complete.

4 Uniform Confluence

We formalise the notions of a calculus, complexity, and uniform confluence as in [Nie94, NS94] and discuss their relationships. These simple concepts will prove extremely useful in the sequel.

The notion of a calculus that we will define extends Klop's abstract rewrite systems [Klo87] by the concept of a congruence: A *calculus* is a triple $(\mathcal{E}, \equiv, \to)$, where $\mathcal{E}$ is a set, $\equiv$ an equivalence relation, and $\to$ a binary relation on $\mathcal{E}$. Elements of $\mathcal{E}$ are called *expressions*, $\equiv$ *congruence*, and $\to$ *reduction* of the calculus. We require that reduction is

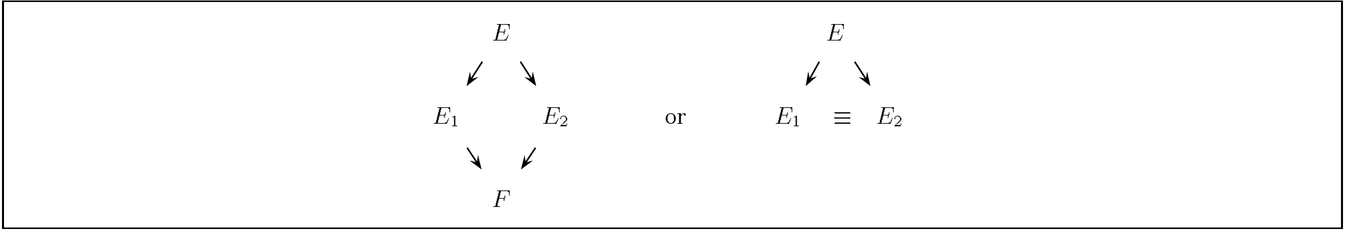

Figure 5: Uniform Confluence

invariant under congruence, i.e., $(\equiv \circ \rightarrow \circ \equiv) \subseteq \rightarrow$. Typical calculi are: δ_0, π, ρ, λ-calculi, abstract rewrite systems, Turing machines, etc.

A *derivation* in a calculus is a finite or infinite sequence of expressions such that $E_i \rightarrow E_{i+1}$ holds for all subsequent elements. A *derivation of an expression* E is a derivation, whose first element is congruent to E. A *computation of* E is a *maximal* derivation of E, i.e. an infinite derivation or a finite one, whose last element is irreducible. The least transitive relation containing $\rightarrow$ and $\equiv$ is denoted with $\rightarrow^*$.

The *length* of a finite derivation $(E_i)_{i=0}^n$ is n and the length of infinite derivation is ∞. We call an expression E *uniform* with respect to complexity (and termination), if all its computations have the same length. We define the *complexity* $\mathcal{C}(E)$ of a uniform expression E by the length of its computations. We call a calculus *uniform* if all its expressions are uniform.

We call a calculus *uniformly confluent*, if its reduction and congruence satisfy the following condition visualised in Figure 5:

$$(\leftarrow \circ \rightarrow) \subseteq ((\rightarrow \circ \leftarrow) \cup \equiv)$$

Typically, λ-calculi equipped with standard reductions are uniformly confluent, subject to weak contexts.

Proposition 4.1 *A uniformly confluent calculus is confluent and uniform with respect to complexity.*

Proof. By a standard inductive argument [Nie94] as for the notion of strong confluence [Hue80] (which is weaker than uniform confluence). □

5 Uniform Confluence for δ_0

In this Section, we distinguish a uniformly confluent subcalculus of δ_0 that is sufficient for functional computation. We call a δ_0-expression *inconsistent*, if it is of the form:

$$x{:}\overline{y}/E \mid x{:}\overline{y}/F$$

where $E \not\equiv F$. A typical example for non-confluence in the case of inconsistencies is to reduce the expression xz in composition with $x{:}y/sy \mid x{:}y/ty$:

$$sz \ {}_A\!\leftarrow\ xz \rightarrow_A tz$$

These results are irreducible but not congruent under the assumption $s \neq t$.

We call E *admissible*, if there exists no expression F containing an inconsistency and satisfying $E \rightarrow^* F$. The advantage of this condition is that it is very simple. Unfortunately, it is undecidable if a given expression E is admissible, since admissibility depends on the result of a Turing complete system. This failure is harmless, since we can prove admissibility for all functional expression of δ with the help of the linear type system in Section 10.

Theorem 5.1 *The restriction of δ_0 to admissible expressions is uniformly confluent.*

Together with Proposition 4.1 this implies that all admissible expressions E of δ_0 are uniform with respect to complexity such that $\mathcal{C}(E)$ is well-defined.

Proof of Theorem 5.1. Let E be an admissible δ_0-expression. Every application step on E can be performed on an arbitrary prenex normal form of E (compare [Nie94] for details). Since declarations are not involved during application, we can assume that E is a prenex normal form with an empty declaration prefix. On such E, reduction amounts to rewriting on multisets of abstractions and applications.

Let F_1 and F_2 be expressions such that $F_1 \ {}_A\!\leftarrow E \rightarrow_A F_2$. There exists an application $x_1\overline{z_1}$ reduced during the application step $E \rightarrow_A F_1$ and an application $x_2\overline{z_2}$ reduced during $E \rightarrow_A F_2$. If these applications are distinct, then we can join F_1 and F_2 by reducing the respective other one. If both applications coincide then $x_1 = x_2$. Hence, the applied abstractions have to be congruent by admissibility such that $F_1 \equiv F_2$. □

6 Single Assignment and Triggering

We extend δ_0 with directed single assignment and triggering. The resulting calculus is called δ. We do not exclude multiple assignment syntactically. This is a matter of the linear type system in Section 10.

For our extension, we need three new types of expressions and two additional reduction axioms. A *directed equation*[1] $x{=}y$ is used for single assignment directed from the right to the left. A *synchroniser* $x.E$ delays the computation of E until t is triggered. A *trigger expression* $\mathbf{tr}(t)$ triggers a delayed computation waiting on t.

The structural congruence of δ coincides with that of δ_0. Its reduction $\rightarrow$ is a union of three relations, application $\rightarrow_A$, *forwarding* $\rightarrow_F$, and *triggering* $\rightarrow_T$:

$$\rightarrow \ = \ \rightarrow_A \cup \rightarrow_F \cup \rightarrow_T$$

Each of these relations is define by the corresponding axiom in Figure 6 and the contextual rules in Figure 4.

Example 6.1 (Single Assignment Style) The identity function $I = \lambda x.x$ can be expressed in δ by $i{:}xy/y{=}x$. Compared to Example 3.1 we use single assignment instead of continuation passing. An application $\mathsf{let}\ i{=}I\ \mathsf{in}\ (ii)i$ referred to by z is represented in δ as follows:

$$(\nu i)(i{:}xy/y{=}x \mid (\nu y')(iiy' \mid y'iz))$$

[1] The original version of the δ-calculus [Nie94] uses symmetric equations instead of directed ones. This choice does not matter for well-typed expressions.

Expressions	$E, F \ ::= \ x{:}\overline{y}/E \ \mid \ x\overline{y} \ \mid \ E \mid F \ \mid \ (\nu x)E \ \mid \ x{=}y \ \mid \ \mathbf{tr}(t) \ \mid \ t.E$
Reduction	$x{:}\overline{y}/E \mid x\overline{z} \ \rightarrow_A \ x{:}\overline{y}/E \mid E[\overline{z}/\overline{y}]$ $\qquad x{=}y \mid y{:}\overline{z}/E \ \rightarrow_F \ x{:}\overline{z}/E \mid y{:}\overline{z}/E$
	$\mathbf{tr}(t) \mid t.E \ \rightarrow_T \ \mathbf{tr}(t) \mid E$

Figure 6: The δ-Calculus

In composition with $i{:}xy/y{=}x$ we obtain the following computation:

$$\begin{array}{rl} (\nu y')(\, iiy' \mid y'iz) & \rightarrow_A \ (\nu y')(\, y'{=}i \mid y'iz) \\ & \rightarrow_F \ (\nu y')(y'{:}xy/y{=}x \mid \, y'iz\,) \\ & \rightarrow_A \ z{=}i \mid (\nu y')(y'{:}xy/y{=}x) \\ & \rightarrow_F \ z{:}xy/y{=}x \mid (\nu y')(\ldots) \end{array}$$

Example 6.2 (Call-by-Need Selector Function) The call-by-need selector function $F = \lambda xy.x$ can be represented in δ by the abstraction $f{:}xt_xyt_yz/(z{=}x \mid \mathbf{tr}(t_x))$. The symbols t_x and t_y stand for ordinary variables. Their usage is for triggering the computations of x and y respectively. A call-by-need application $f(ii)(ii)$ has the form:

$$(\nu x)(\nu t_x)(\nu y)(\nu t_y)(fxt_xyt_yz \mid t_x.iix \mid t_y.iiy)$$

In composition with the abstractions named i and f, we obtain the following computation:

$$\begin{array}{ll} & (\nu x)(\nu t_x)(\nu y)(\nu t_y)(\, fxt_xyt_yz \mid t_x.iix \mid t_y.iiy) \\ \rightarrow_A & (\nu x)(\nu t_x)(z{=}x \mid \mathbf{tr}(t_x) \mid \, t_x.iix\,) \mid (\nu y)(\nu t_y)(t_y.iiy) \\ \rightarrow_T & (\nu x)(\nu t_x)(\, z{=}x \mid \mathbf{tr}(t_x) \mid \, iix\,) \mid (\nu y)(\nu t_y)(\ldots) \\ \rightarrow^* & z{:}xy/y{=}x \mid (\nu y)(\nu t_y)(\nu x)(\nu t_x)(\ldots) \end{array}$$

The resulting expression is irreducible. We note that only the needed first argument has been evaluated. The synchroniser $t_y.iiy$ for the second argument suspends forever.

7 Uniform Confluence for δ

For proving a uniform confluence result for δ, we have to consider how uniform confluence behaves with respect to a union of calculi. We first present a variation of the Hindley-Rosen Lemma [Bar84] for uniform confluence and then apply it to the δ-calculus. But the general results of this Section are also applicable to other unions of calculi such as the call-by-need λ-calculus [AFMOW95] and the ρ-calculus [NM95].

The *union* of two calculi $(\mathcal{E}, \equiv, \rightarrow_1)$ and $(\mathcal{E}, \equiv, \rightarrow_2)$ is defined by $(\mathcal{E}, \equiv, \rightarrow_1 \cup \rightarrow_2)$. We say that the relations $\rightarrow_1$ and $\rightarrow_2$ *commute*, if $({}_1\!\leftarrow \circ \rightarrow_2) \ \subseteq \ (\rightarrow_1 \circ {}_2\!\leftarrow)$.

Lemma 7.1 (Hindley-Rosen) *The union of two uniformly confluent calculi with commuting reductions is uniformly confluent.*

The proof is straightforward. Note that Lemma 7.1 implies the classical Hindley-Rosen Lemma, since a relation is confluent, if and only if its reflexive transitive closure is uniformly confluent. The next lemma allows us to ignore administrative steps such as forwarding and triggering in the case of δ:

Lemma 7.2 (Administrative Steps) *Let $(\mathcal{E}, \equiv, \rightarrow_1)$ be a uniformly confluent calculus and $(\mathcal{E}, \equiv, \rightarrow_2)$ a confluent and terminating calculus such that $\rightarrow_1$ and $\rightarrow_2$ commute. If E is an expression in $\mathcal{E}$, then every computation of E in the union $(\mathcal{E}, \equiv, \rightarrow_1 \cup \rightarrow_2)$ contains the same number of $\rightarrow_1$ steps.*

Proof. The idea is to apply Proposition 4.1 to $(\mathcal{E}, \equiv, \rightarrow_2^* \circ \rightarrow_1 \circ \rightarrow_2^*)$. This calculus is uniform but not uniformly confluent. This deficiency can be remedied by replacing $\equiv$ with $({}_2\!\leftarrow \cup \rightarrow_2)^*$. The details can be found in [Nie94]. □

Next, we apply the above results to the δ-calculus. We first note that the notion of admissibility carries over from δ_0 to δ without change.

Proposition 7.3 *The relations $\rightarrow_F$ and $\rightarrow_T$ terminate. The relation $\rightarrow_T$ is uniformly confluent and $\rightarrow_F$ is uniformly confluent when restricted to admissible expressions. The relations $\rightarrow_A$, $\rightarrow_F$, and $\rightarrow_T$ commute pairwise.*

Proof. Termination is trivial, since $\rightarrow_F$ decreases the number of directed equations and $\rightarrow_T$ the number of synchronisers. All other properties can be established by the normal form technique used in the proof of Theorem 5.1. □

Theorem 7.4 *The restriction of the δ-calculus to admissible expressions is uniformly confluent.*

Proof. Follows from Theorem 5.1, Proposition 7.3, and Lemma 7.1. □

Theorem 7.5 *If E is admissible, then all computations of E contain the same number of application steps.*

Proof. Follows from Theorem 5.1, Proposition 7.3, and Lemma 7.2. □

Definition 7.6 *We define the* A-complexity $\mathcal{C}^A(E)$ *of an admissible δ-expression E as the number of $\rightarrow_A$ steps in computations of E.*

Theorem 7.5 ensures that A-complexity is well defined. We consider forwarding and triggering steps as administrative steps and ignore them in favour of simpler complexity statements and adequacy proofs. However, we could prove for all functional expressions (but not in general) that the number of administrative steps is linearly bound by the number of $\rightarrow_A$ steps. This would require showing stronger invariants in adequacy proofs.

Expressions	$M,N ::= x \mid V \mid MN$		$V ::= \lambda x.M$
Reduction	$(\lambda x.M)V \to_{\mathrm{val}} M[V/x]$		$(\lambda x.M)N \to_{\mathrm{name}} M[N/x]$
Contextual Rules	$\dfrac{M \to_{\mathrm{val}} M'}{MN \to_{\mathrm{val}} M'N}$	$\dfrac{N \to_{\mathrm{val}} N'}{MN \to_{\mathrm{val}} MN'}$	$\dfrac{M \to_{\mathrm{name}} M'}{MN \to_{\mathrm{name}} M'N}$

Figure 7: The Call-by-Value and the Call-by-Name λ-Calculus

$$
\begin{array}{rcl@{\qquad}rcl}
z{=_v}MN & \equiv & (\nu x)(x{=_v}M \mid (\nu y)(xyz \mid y{=_v}N)) & z{=_n}MN & \equiv & (\nu x)(x{=_n}M \mid (\nu y)(\nu t_y)(xyt_yz \mid t_y.y{=_n}N)) \\
z{=_v}\lambda x.M & \equiv & z{:}xy/y{=_v}M & z{=_n}\lambda x.M & \equiv & z{:}xt_xy/y{=_n}M[x{\diamond}t_x/x] \\
z{=_v}x & \equiv & z{=}x & z{=_n}x{\diamond}t_x & \equiv & z{=}x \mid \mathbf{tr}(t_x)
\end{array}
$$

Figure 8: Call-by-Value and Call-by-Need in the δ-Calculus

8 Functional Computation in δ

We embed the call-by-value and the call-by-name λ-calculus into the δ-calculus, the latter with call-by-need complexity.

The call-by-value and the call-by-name λ-calculus are revisited in Figure 8. Note that we consider standard reduction only. A congruence allowing for consistent renaming of bound variables is left implicit as usual.

Proposition 8.1 *The call-by-value and the call-by-name λ-calculus are uniformly confluent.*

The statement for call-by-name is trivial, since call-by-name reduction is deterministic. The proof for call-by-value can be done by a simple induction on the structure of λ-expressions. Proposition 8.1 allows us to define the *call-by-value complexity* $\mathcal{C}_{\mathrm{val}}(M)$ and the *call-by-name complexity* $\mathcal{C}_{\mathrm{name}}(M)$ of a λ-expression M by the length of its computations in the respective λ-calculus.

Given an arbitrary variable z, Figure 8 presents an embedding $M \mapsto z{=_v}M$ of the call-by-value λ-calculus into δ. The definition of $z{=_v}M$ is given up to structural congruence. All variables introduced during this definition are supposed to be fresh.

Theorem 8.2 *For all closed λ-expressions M and variables z the call-by-value complexity of M and the A-complexity of $z{=_v}M$ coincide: $\mathcal{C}_{\mathrm{val}}(M) = \mathcal{C}^A(z{=_v}M)$.*

A complete proof based on the techniques of Section 12 is presented in [Nie94]. It makes heavy use of uniform confluence for covering the additional freedom provided by call-by-value reduction in δ.

An embedding $z \mapsto z{=_n}M$ of the call-by-name λ-calculus into δ is given in Figure 8. It is symmetric to our call-by-value embedding and provides for call-by-need complexity. Our definition of a δ-expression $x{=_n}M$ makes sense for closed M only and goes through intermediate λ-expressions containing pairs $y{\diamond}t_y$. For instance:

$$z{=_n}\lambda x.x \;\equiv\; z{:}xt_xy/y{=_n}x{\diamond}t_x \;\equiv\; z{:}xt_xy/(y{=}x \mid \mathbf{tr}(t_x))$$

As we will show in the next Section, our embedding of the call-by-name λ-calculus provides in fact for call-by-need complexity. In this sense, the next theorem states that call-by-need complexity is dominated by call-by-value and by call-by-name complexity.

Theorem 8.3 *Let M be a closed λ-expression and z a variable. Call-by-name reduction of M terminates if and only if δ-reduction of $z{=_n}M$ terminates. Furthermore:*

$$\mathcal{C}^A(z{=_n}M) \le \min\{\mathcal{C}_{\mathrm{val}}(M),\, \mathcal{C}_{\mathrm{name}}(M)\}\,.$$

The most difficult statements of Theorem 8.3 are proved in Sections 12, 13, and 14. These are the adequacy with respect to termination and the estimation $\mathcal{C}^A(z{=_n}M) \le \mathcal{C}_{\mathrm{name}}(M)$. The estimation $\mathcal{C}(z{=_n}M) \le \mathcal{C}_{\mathrm{val}}(M)$ can also be established by the simulation technique of Section 12. It is sufficient to relate the expressions $z{=_v}M$ and $z{=_n}M$. For applying the simulation technique, we need the admissibility of these expressions as proved in Section 10.

It is straightforward to express mutual recursion in δ, both in a call-by-value and in a call-by-need manner:

$$z{=_v}\mathsf{letrec}\ \overline{x}{=}\overline{M}\ \mathsf{in}\ N \;\equiv\; (\nu\overline{x})(\overline{x}{=_v}\overline{M} \mid z{=_v}N)$$

$$z{=_n}\mathsf{letrec}\ \overline{x}{=}\overline{M}\ \mathsf{in}\ N \;\equiv\; (\nu\overline{x})(\nu\overline{t})(\overline{t}.\overline{x}{=_n}\overline{M}\theta \mid z{=_n}N\theta)$$

where $\theta = [\overline{x}{\diamond}\overline{t}/\overline{x}]$. We do not claim a correctness result for mutual recursion in this paper.

9 Embedding the Call-by-Need λ-Calculus

We show that the A-complexity of $z{=_n}M$ equals to the complexity of M in the call-by-need λ-calculus.

The definition of the call-by-need λ-calculus [AFMOW95] is revisited in Figure 9. Again, we only consider standard reduction. The reduction $\to_{\mathrm{need}}$ of the call-by-need λ-calculus is a union of four relations:

$$\to_{\mathrm{need}} \;=\; \to_I \cup \to_V \cup \to_{Ans} \cup \to_C$$

The latter three relations are of administrative character, whereas $\to_I$ steps correspond to β-reduction steps.

Proposition 9.1 *The call-by-need λ-calculus is deterministic and hence uniformly confluent.*

The proof is simple. Applying Proposition 9.1, it makes sense to define the call-by-need complexity $\mathcal{C}_{\mathrm{need}}(L)$ of an expression the call-by-need λ-calculus by the number of $\to_I$ steps in the computation of L.

We extend the mapping $M \mapsto z{=_n}M$ to an embedding $L \mapsto z{=_n}L$ of the call-by-need λ-calculus into δ, defining:

$$z{=_n}\mathsf{let}\ \ x{=}L_2\ \ \mathsf{in}\ L_1 \;\equiv\; (\nu x)(x{=_n}L_2 \mid z{=_n}L_1)$$

Expressions	L ::= x | V | $L_1 L_2$ | let $x{=}L_2$ in L_1 $\quad V$ ::= $\lambda x.L$
Answers	A ::= V | let $x{=}L$ in A
Evaluation Contexts	E ::= [] | EL | let $x{=}L$ in E | let $x{=}E_2$ in $E_1[x]$
Reduction	$(\lambda x.L_1)L_2 \to_I$ let $x{=}L_2$ in L_1 $\quad$ let $x{=}V$ in $E[x] \to_V$ let $x{=}V$ in $E[V]$
	let $y{=}$(let $x{=}L$ in A) in $E[y] \to_{Ans}$ let $x{=}L$ in (let $y{=}A$ in $E[y]$)
	(let $x{=}L_1$ in $A)L_2 \to_C$ let $x{=}L_1$ in AL_2 $\quad \dfrac{L \to L'}{E[L] \to E[L']}$

Figure 9: The Call-by-Need λ-Calculus

The following Theorem states the adequacy of the extended embedding, and that our embedding of the call-by-name λ-calculus into δ yields in fact call-by-need complexity:

Theorem 9.2 *If L is a closed λ-expression and z a variable then $\mathcal{C}_{\text{need}}(L) = \mathcal{C}^A(z{=}_n L)$.*

This can be shown by a complexity simulation (Section 12).

10 Linear Types for Consistency

We define a linear type system for δ that statically excludes inconsistencies. It tests for single assignment and determines the data flow of a δ-expression via input and output modes.

We assume an infinite set of *type variables* denoted by α and use the following recursive *types* σ internally annotated with *modes* η:

$$\sigma \;::=\; (\overline{\tau}) \;\mid\; \mu\alpha.\tau \;\mid\; \alpha \;\mid\; \mathbf{tr}\,, \qquad \tau \;::=\; \sigma^\eta$$
$$\eta \;::=\; \mathsf{in} \;\mid\; \mathsf{out}$$

Our type systems distinguishes two classes of variables, trigger and single assignment variables. We use $\mathbf{tr}$ as type for trigger variables. A single assignment variable has a procedural type $(\overline{\tau})$, where $\overline{\tau}$ is a sequence of argument types. For instance, the variable z in $z{=}_v M$ is typed by $\mu\alpha.(\alpha^{\mathsf{in}}\,\alpha^{\mathsf{out}})$. This recursive type expresses that a call-by-value function is a binary relation, which inputs a call-by-value function in first position and outputs a call-by-value function in second position.

A type *environment* Γ is a sequence of *type assumptions* $x{:}\,\sigma$ with scoping to the right. A variable x *has type σ in* Γ, written $\Gamma(x) = \sigma$, if there exists Γ_1 and Γ_2 such that $\Gamma = \Gamma_1, x{:}\,\tau, \Gamma_2$ and x does not occur in Γ_2. The domain of an environment Γ is the set of all variables typed by Γ. We identify environments Γ_1 and Γ_2 if they have the same domain and $\Gamma_1(x) = \Gamma_2(x)$ for all x in this domain. The output variables $\mathcal{O}(\overline{y}{:}\,\overline{\sigma}^{\overline{\eta}})$ in a sequence of annotated types are defined as follows, where $\overline{y} = (y_i)_{i=1}^n$, $\overline{\sigma} = (\sigma_i)_{i=1}^n$, and $\overline{\eta} = (\eta_i)_{i=1}^n$:

$$\mathcal{O}(\overline{y}{:}\,\overline{\sigma}^{\overline{\eta}}) \;=\; \{y_i \;\mid\; \eta_i = \mathsf{out} \text{ and } \sigma_i \neq \mathbf{tr}\}$$

A *judgement for E* is a triple $\Gamma;\, O \,\triangleright\, E$, where Γ is an environment and O is a set of variables. An expression E is *well-typed*, if there exists a judgement for E derivable with the rules in Figure 10. If $\Gamma;\, O \,\triangleright\, E$ is derivable, then O contains those single assignment variables, to which an abstraction may be assigned during a computation of E.

Lemma 10.1 *If E is well-typed and $E \to^* F$, then F is well-typed. An inconsistent expression is not well-typed.*

The first property is often called subject reduction property. It can be checked by induction on derivations of judgements. The second property is straightforward. Lemma 10.1 immediately implies the following corollary:

Corollary 10.2 *Well-typed expressions are admissible.*

Proposition 10.3 *All expressions $z{=}_v M$ and $z{=}_n L$ are well-typed.*

Proof. We can check by induction on the structure of M that the following judgements are derivable with the rules in Figure 10, where η is arbitrary:

$$z{:}\,\mu\alpha.(\alpha^{\mathsf{in}}\,\alpha^{\mathsf{out}});\, \{z\} \;\triangleright\; z{=}_v M$$
$$z{:}\,\mu\alpha.(\alpha^{\mathsf{in}}\,\mathbf{tr}^\eta\,\alpha^{\mathsf{out}});\, \{z\} \;\triangleright\; z{=}_n L$$

11 Encoding δ in δ_0

Directed single assignment and triggering can be expressed in δ_0. For technical simplicity, we formalise this statement for *n-ary δ-expressions*, i.e. those containing n-ary abstractions and applications only. This is sufficient to carry over our λ-calculus embeddings from δ to δ_0, since $z{=}_v M$ and $z{=}_n L$ are binary and ternary respectively. An embedding of n-ary δ-expressions into δ_0 is given in Figure 11 and $\delta_0(E) \equiv E$ for all expressions E of δ_0.

Theorem 11.1 *For all well-typed n-ary δ-expressions E, $\delta_0(E)$ is admissible and terminates if and only if E terminates.*

The proof is omitted, but can be done with the simulation technique of Section 12. Adequacy with respect to termination follows from the fact that the embedding preserves the length of computations up to a factor of 2 (which is needed for triggering). This does however not imply the adequacy with respect to complexity measured in terms of application steps. This is the only point where ignoring $\to_F$ and $\to_T$ steps weakens our results.

12 Complexity Simulations and Uniformity

Milner [Mil92] uses bisimulations for proving the adequacy of λ-calculus embeddings into the π-calculus. We show that simulations are sufficient for uniform calculi.

$$\frac{\Gamma;\, O \;\triangleright\; E}{\Gamma;\, O\backslash\{x\} \;\triangleright\; (\nu x)E} \qquad \frac{\Gamma;\, O_1 \;\triangleright\; E_1 \quad \Gamma;\, O_2 \;\triangleright\; E_2}{\Gamma;\, O_1 \cup O_2 \;\triangleright\; E_1 \mid E_2} \; O_1 \cap O_2 = \emptyset$$

$$\frac{\Gamma, t{:}\,\mathbf{tr};\, O \;\triangleright\; E}{\Gamma, t{:}\,\mathbf{tr};\, O \;\triangleright\; t.E} \qquad \frac{\Gamma, \overline{y}{:}\,\overline{\sigma};\, O \;\triangleright\; E}{\Gamma, x{:}\,(\overline{\sigma}^{\overline{\eta}});\, \{x\} \;\triangleright\; x{:}\overline{y}/E} \; O \subseteq \mathcal{O}(\overline{y}{:}\,\overline{\sigma}^{\overline{\eta}})$$

$$\Gamma, t{:}\,\mathbf{tr};\, \emptyset \;\triangleright\; \mathbf{tr}(t) \qquad \Gamma, x{:}\,(\overline{\tau}), y{:}\,(\overline{\tau});\, \{x\} \;\triangleright\; x{=}y \qquad \Gamma, x{:}\,(\overline{\sigma}^{\overline{\eta}}), \overline{y}{:}\,\overline{\sigma};\, O \;\triangleright\; x\overline{y}\,, \quad \mathcal{O}(\overline{y}{:}\,\overline{\sigma}^{\overline{\eta}}) \subseteq O$$

Figure 10: Linear Type Checking

$$\delta_0(t.E) \;\equiv\; (\nu y)(ty \mid y{:}/\delta_0(E)) \qquad \delta_0(\mathbf{tr}(t)) \;\equiv\; t{:}y/y \qquad \delta_0(x{=}y) \;\equiv\; x{:}\overline{z}/y\overline{z}\,, \quad \mathsf{length}(\overline{z}) = n$$

Figure 11: Embedding n-ary δ-expressions in δ_0

Let $(\mathcal{E}, \equiv_{\mathcal{E}}, \rightarrow_{\mathcal{E}})$ and $(\mathcal{G}, \equiv_{\mathcal{G}}, \rightarrow_{\mathcal{G}})$ be two uniform calculi with expressions ranged over by E and G respectively. We omit the indices $\mathcal{E}$ and $\mathcal{G}$ whenever they are clear from the context. We call a function $\Phi : \mathcal{E} \rightarrow \mathcal{G}$ an *embedding of $\mathcal{E}$ into $\mathcal{G}$*, if Φ is invariant under congruence.

Definition 12.1 *A* shortening simulation *for an embedding $\Phi : \mathcal{E} \rightarrow \mathcal{G}$ is a relation S on $\mathcal{E} \times \mathcal{G}$ satisfying the following conditions for all E, E', and G:*

(Sho1) $(E, \Phi(E)) \in S$.

(Sho2) *If E is irreducible and $(E, G) \in S$, then G is irreducible.*

(Sho3) *If $E \rightarrow E'$ and $(E, G) \in S$, then exists E'' and G' with $\mathcal{C}(E') \geq \mathcal{C}(E'')$, $(E'', G') \in S$, and $G \rightarrow G'$.*

We call a shortening simulation complexity simulation *if it satisfies (Sho3) with $\mathcal{C}(E') = \mathcal{C}(E'')$ instead of $\mathcal{C}(E') \geq \mathcal{C}(E'')$.*

Theorem 12.2 *Let $\Phi : \mathcal{E} \rightarrow \mathcal{G}$ be an embedding between uniform calculi. If there exists a shortening simulation for Φ, then Φ preserves termination and improves complexity, i.e. $\mathcal{C}(\Phi(E)) \leq \mathcal{C}(E)$ for all E. If there exists a complexity simulation for Φ, then Φ preserves complexity: $\mathcal{C}(\Phi(E)) = \mathcal{C}(E)$ for all E.*

Proof. We assume a shortening simulation S for Φ and $(E, G) \in S$. First, we claim $\mathcal{C}(G) \leq \mathcal{C}(E)$ if $\mathcal{C}(E) \neq \infty$ by induction on $\mathcal{C}(E)$. Second, we claim $\mathcal{C}(G) \leq \mathcal{C}(E)$ if $\mathcal{C}(E) = \infty$. This can be shown by proving $\mathcal{C}(G) \geq n$ inductively for all $n \geq 0$. The theorem follows from these claims and condition *(Sho1)*. □

13 A Shortening Simulation for Call-by-Need

We sketch the adequacy proof for our embedding of the call-by-name λ-calculus into δ as stated in Theorem 8.3. We prove that the embedding $M \mapsto z{=}_n M$ preserves termination such that $\mathcal{C}^A(z{=}_n M) \leq \mathcal{C}_{\text{name}}(M)$ for all closed λ-expressions M.

For reflecting A-complexity, we consider the alternative reduction $\hookrightarrow \circ \rightarrow_A \circ \hookrightarrow$ where $\hookrightarrow = (\rightarrow_F \cup \rightarrow_T)^*$. Restricted to admissible expressions, this reduction yields a uniform calculus by Theorem 7.5. By Proposition 10.3, Corollary 10.2, and Proposition 8.1, it is sufficient to apply Theorem 12.2 once we have constructed a shortening simulation for the embedding $M \mapsto z{=}_n M$.

As syntactical convenience, we write $\mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ N$ for the λ-expression $N[M_n/y_n]\ldots[M_1/y_1]$ where $\overline{y} = (y_i)_{i=1}^n$ and $\overline{M} = (M_i)_{i=1}^n$. Before formally defining a shortening simulation (see Section 14) we illustrate it by a simple example. We first consider a call-by-name reduction step of $(II)\,I$ with $I = \lambda x.x$:

$$\begin{array}{ll} (II)\,I & \\ \quad = & \mathsf{let}\ y_1{=}I\ \ z_1{=}I\ \ \boxed{y_2{=}y_1 z_1}\ \ z_2{=}I\ \ y_3{=}y_2 z_2\ \ \mathsf{in}\ y_3 \\ \rightarrow_{\text{name}} & \mathsf{let}\ y_1{=}I\ \ z_1{=}I\ \ \boxed{y_2{=}z_1}\ \ z_2{=}I\ \ y_3{=}y_2 z_2\ \ \mathsf{in}\ y_3 \\ \quad = & \mathsf{let}\ y_1{=}I\ \ z_1{=}I\ \ y_2{=}I\ \ z_2{=}I\ \ y_3{=}y_2 z_2\ \ \mathsf{in}\ y_3 \end{array}$$

In the corresponding δ-reduction steps, we omit top-level declarations and write $E \approx F$ if $E \equiv (\nu\overline{x})F$.

$$\begin{array}{ll} y_3{=}_n(II)\,I & \\ \quad\approx & y_1{=}_n I \mid t_1.z_1{=}_n I \mid \boxed{y_1 z_1 t_1 y_2} \mid t_2.z_2{=}_n I \mid y_2 z_2 t_2 y_3 \\ \rightarrow_A & y_1{=}_n I \mid \boxed{t_1.z_1{=}_n I} \mid y_2{=}_n z_1 \diamond t_1 \mid t_2.z_2{=}_n I \mid y_2 z_2 t_2 y_3 \\ \rightarrow_T & y_1{=}_n I \mid z_1{=}_n I \mid \boxed{y_2{=}z_1} \mid \mathbf{tr}(t_1) \mid t_2.z_2{=}_n I \mid y_2 z_2 t_2 y_3 \\ \rightarrow_F & y_1{=}_n I \mid z_1{=}_n I \mid y_2{=}_n I \mid \mathbf{tr}(t_1) \mid t_2.z_2{=}_n I \mid y_2 z_2 t_2 y_3 \end{array}$$

The correspondence in this example is very close when ignoring $\rightarrow_F$ and $\rightarrow_T$ steps[2]. A more interesting example comes with sharing, when reducing $z{=}_n(\lambda x.(x\lambda y.x))(II)$. In this case, we can formulate the relationship via strong β-reduction: We write $M \Rightarrow_{\text{name}} M'$ if M reduces to M' by application of the β-axiom at any position in M.

Lemma 13.1 (The Invariant) *There exists a relation S between closed λ-expressions and admissible δ-expressions satisfying (Sho1) and (Sho2) and the following property: If $(M, E) \in S$ and $M \rightarrow_{\text{name}} M'$, then there exists M'' and E', such that $M' \Rightarrow^*_{\text{name}} M''$, $E \hookrightarrow \circ \rightarrow_A \circ \hookrightarrow E'$, and $(M'', E') \in S$.*

The proof is sketched in Section 14. Lemma 13.1 implies the existence of a shortening simulation for the embedding $M \mapsto z{=}_n M$. To verify condition (Sho3) we make use of Lemma 13.2:

Lemma 13.2 *If $M' \Rightarrow^*_{\text{name}} M''$, then $\mathcal{C}_{\text{name}}(M') \geq \mathcal{C}_{\text{name}}(M'')$.*

Proof. This is a reformulation of Plotkin's standardisation theorem [Plo75]. □

[2] The number of $\rightarrow_F$ and $\rightarrow_T$ steps in computations of $y_3{=}_n M$ is bounded by 3 times the number of $\rightarrow_A$ steps. This can be proved with a simulation for an amortised cost analysis.

14 Proof of Lemma 13.1

Our idea for defining a shorting simulation is to make substitutions explicit as in [Mil92, ACCL91] and to reflect lazy control by a notion of needed variables.

Explicit substitutions are already introduced in the definition of let $\overline{y}=\overline{M}$ in N. For defining needed variables, we write $\overline{\mu}^{<i}$ for the sequence $(\mu_j)_{j=1}^{i-1}$, if $\overline{\mu} = (\mu_j)_{j=1}^{n}$ is a sequence of variables or expressions.

Definition 14.1 (Needed Variables) *A variable x is* needed *in* let $\overline{y}=\overline{M}$ in N, *if the judgement* $\mathcal{N}(x, \text{let } \overline{y}=\overline{M} \text{ in } N)$ *is derivable by the following rules:*

$$\frac{}{\mathcal{N}(x, x)} \qquad \frac{\mathcal{N}(x, N_1)}{\mathcal{N}(x, N_1 N_2)} \qquad \frac{\mathcal{N}(x, N)}{\mathcal{N}(x, \text{let } \overline{y}=\overline{M} \text{ in } N)}$$

$$\frac{\mathcal{N}(x, \text{let } \overline{y}^{<i}=\overline{M}^{<i} \text{ in } M_i) \qquad \mathcal{N}(y_i, N)}{\mathcal{N}(x, \text{let } \overline{y}=\overline{M} \text{ in } N)}$$

For instance in let $y_1{=}I \;\; y_2{=}y_1 y_1 \;\; y_3{=}y_1 y_2$ in y_3, the variables y_3 and y_1 are needed whereas y_2 is not needed.

Definition 14.2 (Representation) *A* representation *for* (M, E) *is a five-tuple* $(n, \overline{M}, \overline{y}, \overline{t}, D)$, *where* $\overline{M} = (M_i)_{i=1}^n$, $\overline{y} = (y_i)_{i=1}^n$, $\overline{t} = (t_i)_{i=1}^n$, *and* $D \subseteq \{y_1, \ldots y_n\}$ *called the delay set, such that the following properties hold for all* $i \in \{1 \ldots n\}$:

(S1) $\mathcal{V}(M_i) \subseteq \{y_1 \ldots y_{i-1}\}$

(S2) $M \equiv$ let $\overline{y}=\overline{M}$ in y_n.

(S3) *There exists* $(E_i)_{i=1}^n$, θ, *and* ϕ *such that* ϕ *is a composition of trigger expressions* $\{\mathbf{tr}(j) \mid y_j \notin D\}$,

$$E \approx E_1 \mid \ldots \mid E_n \mid \phi,$$

θ is the substitution $[\overline{y}{\diamond}\overline{t}/\overline{y}]$, *and:*

$$E_i = \begin{cases} t_i.y_i{=}_n M_i\theta & \text{if } y_i \in D \\ y_j y_k t_k y_i & \text{if } y_i \notin D,\ M_i = y_j y_k \\ y_i{=}y_j & \text{if } y_i \notin D,\ M_i = y_j \\ y_i{=}_n M_i\theta & \text{if } y_i \notin D,\ M_i = \lambda z.N \end{cases}$$

(S4) *If* $y_i \notin D$ *and* M_i *is an application then* M_i *is an application of variables.*

(S5) *If* y_i *is needed in* let $\overline{y}=\overline{M}$ in y_n, *then* $y_i \notin D$.

(S6) *If* y_i *is not needed in* let $\overline{y}=\overline{M}$ in y_n, *then* $y_i \in D$ *or* M_i *is an abstraction.*

(S7) *The composed sequence* $\overline{y}\overline{t}$ *is linear.*

Definition 14.3 (Simulation S) *The relation* S *is the set of all pairs* (M, E) *for which a* representation *exists.*

Lemma 14.4 (Correctness of S) *The relation* S *satisfies the conditions of Lemma 13.1.*

15 Conclusion

We have presented a simple execution model for eager and lazy functional computation. We have applied concurrency for integration of programming paradigms. We have presented the concurrent δ-calculus, which features useful abstractions for programming, implementation, and theory. We have worked out a powerful proof technique based on uniform confluence and complexity simulations.

Acknowledgements. I am deeply in debt to Gert Smolka, who initiated this work and contributed ideas during many discussions. It's my pleasure to thank to Martin Müller for daily comments on concepts and related work, and for extremely helpful discussions on notations and details. I would like to thank Kai Ibach, Martin Müller, Peter Van Roy, Christian Schulte, and Gert Smolka, for their comments on the final version and the complete Oz team for continuous support and interest.

References

[ACCL91] Martín Abadi, Luca Cardelli, P.-L. Curien, and Jean-Jaques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[AFMOW95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *POPL*, pages 233–246. 1995.

[ANP89] Arvind, R.S. Nikhil, and K.K. Pingali. I-structures: Data-structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 4(11):598–632, 1989.

[Bar84] Henk P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1984.

[BNA91] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Functional Programming Languages and Computer Architecture*, number 523 in LNCS, pages 538–568. 1991.

[BO95] Simon Brook and Gerald Ostheimer. Process semantics of graph reduction. In *Sixth International Conference on Concurrency Theory*, pages 238–252, August 1995.

[Bou89] Gérard Boudol. Towards a λ-calculus for concurrent and communicating systems. In *Theory and Practice in Software Development*, number 351 in LNCS, pages 149–161. 1989.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA, Sophia Antipolis, France, 1992.

[HSW95] Martin Henz, Gert Smolka, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In V. Saraswat and P. Van Hentenryck, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 27–48. The MIT Press, 1995.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proceeding of the European Conference on Object-Oriented Programming*, number 512 in LNCS, pages 133–147, 1991.

[Hue80] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[Iba95] Kai Ibach. OzFun: Eine funktionale Spache für gemischte Eager- und Lazy-Programmierung, Universität des Saarlandes, Fachbereich Informatik. October 1995.

[Jef94] Alan Jeffrey. A fully abstract semantics for concurrent graph reduction. In *Proceedings of the Logic in Computer Science Conference*, pages 82–91, 1994.

[JH91] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In *Proceedings of the International Symposium on Logic Programming*, pages 167–186, 1991.

[Klo87] Jan Willem Klop. Term rewriting systems: A tutorial. *Bulletin of the European Association of Theoretical Computer Sience.*, 32:143–182, 1987.

[KPT96] Naoki Kobayashi, Benjamin Pierce, and David N. Turner. Linearity and the pi-calculus. In *POPL*. January 1996.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *POPL*, pages 144–154, 1993.

[Mah87] Michael J. Maher. Logic semantics for a class of committed-choice programs. In *Logic Programming, Proceedings of the Fourth International Conference*, pages 858–876. 1987.

[Mil91] Robin Milner. The polyadic π-calculus: A tutorial. ECS-LFCS Report Series 91–180, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.

[Mil92] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.

[Mül95] Martin Müller. Polymorphic types for concurrent constraints. DFKI Saarbrücken, Germany, `http://ps-www.dfki.uni-sb.de/`, 1996, submitted.

[Nie94] Joachim Niehren. *Funktionale Berechnung in einem uniform nebenläufigen Kalkül mit logischen Variablen.* Doctoral Dissertation. Universität des Saarlandes, Technische Fakultät, Saarbrücken, Germany, December 1994.

[NM95] Joachim Niehren and Martin Müller. Constraints for Free in Concurrent Computation. In *Asian Computing Science Conference*, LNCS, December 1995.

[NS94] Joachim Niehren and Gert Smolka. A confluent relational calculus for higher-order programming with constraints. In *1st International Conference on Constraints in Computational Logics*, volume 845 of *LNCS*, pages 89–104, 1994.

[Pin87] Keshav K. Pingali. Lazy Evaluation and the Logic Variable. Technical report, Cornell University, Proceedings of the Institute on Declarative Programming. 1987.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Journal of Theoretical Computer Science*, 1:125–159, 1975.

[PS92] S. Purushothaman and Jill Seaman. An adequate operational semantics of sharing in lazy evaluation. In *European Symposium on Programming*, volume 582 of *LNCS*. 1992.

[PT95a] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In *Theory and Practice of Parallel Programming*, number 907 in LNCS, pages 187–215. April 1995.

[PT95b] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical report in preparation; available electronically, 1996.

[San95] D. Sands. A Naïve Time Analysis and its Theory of Cost Equivalence. *The Journal of Logic and Computation*, page 48 pages, 1995+.

[Smo94] Gert Smolka. A foundation for concurrent constraint programming. In *Constraints in Computational Logics*, volume 845 of *LNCS*, pages 50–72. 1994.

[Smo95a] Gert Smolka. An Oz primer. Oz documentation series, DFKI Saarbrücken, Germany, 1995.

[Smo95b] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Current Trends in Computer Science*, LNCS, vol. 1000. 1995.

[SRP91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *POPL*, pages 333–352. 1991.

[SSW94] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In *Second Workshop on Principles and Practice of Constraint Programming*, volume 874 of *LNCS*, pages 134–150. 1994.

[Vas94] Vasco T. Vasconcelos. Typed concurrent objects. In *8th Proceedings of the European Conference on Object Oriented Programming*, volume 821 of *LNCS*, pages 100–117. 1994.

[Wal95] David Walker. Objects in the π-calculus. *Journal on Information and Computation*, 116:254–273, 1995.

[Yos93] Nobuko Yoshida. Optimal reduction in weak λ-calculus with shared environments. In *Conference on Functional Programming Languages and Computer Architecture*, 1993.