Functional Computation as Concurrent Computation

Joachim Niehren

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

15 November 1995

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.

Contents

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 programmer has to reason about the complexity of his programs. In particular for functional programs, denotational semantics are too coarse [San95].

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.

[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.

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:

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

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 [Mül96].

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 $x{:}\overline{y}/E \mid x{:}\overline{y}/F$ 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 complexities 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. We note that the call-by-let λ-calculus introduced in [MOTW95] provides the same kind of freedom.

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]. An embedding of the call-by-need λ-calculus into the π-calculus is proved correct in [BO95]. The advantage of the embeddings presented here 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

$$\begin{tikzcd}
S\,(2{*}3) \arrow[d] & & (\nu y)\,(syz \mid y{=}2{*}3) \arrow[dl] \arrow[dr] & \\
S\,6 \arrow[d] & (\nu y)\,(syz \mid y{=}6) \arrow[dr] & & (\nu y)\,(z{=}y{*}y \mid y{=}2{*}3) \arrow[dl] \\
6{*}6 \arrow[d] & & (\nu y)\,(z{=}y{*}y \mid y{=}6) \arrow[d, "*"'] & \\
36 & & z{=}36 \quad {}^{2} &
\end{tikzcd}$$

Figure 1: Square Function: Call-by-Value

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 Report. 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 – 18.

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 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 (commented by footnotemark [2]) 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 (commented by the footnotemarks [2] and [3]) 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

[2]Here, $\rightarrow^*$ stands a forwarding step followed by an application step: $(\nu y)(z{=}y{*}y \mid y{=}6) \;\rightarrow\; z{=}6{*}6 \;\rightarrow\; z{=}36$.

$$\begin{tikzcd}
 & S\ (2{*}3) \arrow[d] & & (\nu y)(\nu t)\,(s'ytz \mid t.y{=}2{*}3) \arrow[d, "*"] \\
 & (2{*}3){*}(2{*}3) \arrow[dl] \arrow[dr] & & (\nu y)\,(z{=}y{*}y \mid y{=}2{*}3)\ ^{3} \arrow[dd] \\
6{*}(2{*}3) \arrow[dr] & & (2{*}3){*}6 \arrow[dl] & \\
 & 6{*}6 \arrow[d] & & (\nu y)\,(z{=}y{*}y \mid y{=}6) \arrow[d, "*"] \\
 & 36 & & z{=}36\ ^{2}
\end{tikzcd}$$

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

[3]Here, $\rightarrow^*$ consists of an application and a triggering step: $(\nu t)(s'ytz \mid t.y{=}2{*}3) \rightarrow (\nu t)(z{=}y{*}y \mid \mathbf{tr}(t) \mid t.y{=}2{*}3) \rightarrow z{=}y{*}y \mid y{=}2{*}3 \mid (\nu t)(\mathbf{tr}(t))$. The garbage expression $(\nu t)(\mathbf{tr}(t))$ in 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	$x{:}\overline{y}/E \mid x\overline{z} \rightarrow_A x{:}\overline{y}/E \mid E[\overline{z}/\overline{y}]$

Figure 3: The δ_0-Calculus.

Structural Congruence

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

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

$$E \equiv F \quad \text{if } E =_\alpha F$$

Contextual Rules

$$\frac{E \rightarrow E'}{E \mid F \rightarrow E' \mid F} \qquad \frac{E \rightarrow E'}{(\nu x)E \rightarrow (\nu x)E'} \qquad \frac{E_1 \equiv E_2 \quad E_2 \rightarrow F_2 \quad F_2 \equiv F_1}{E_1 \rightarrow F_1}$$

Figure 4: Structural Congruence and Contextual Rules

and F are equal up to consistent renaming of bound variables. 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* $\rightarrow$, synonymously denoted by $\rightarrow_A$, 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 $\overline{z}$. In case of application of $[\overline{z}/\overline{y}]$, we implicitly assume 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 we consider standard reductions only.

Example 3.1 (Continuation Passing Style) *The identity 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:

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

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

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

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 $(\mathcal{E}, \equiv, \rightarrow)$, where $\mathcal{E}$ is a set, $\equiv$ an equivalence relation, and $\rightarrow$ a binary relation on $\mathcal{E}$. Elements of $\mathcal{E}$ are called *expressions*, $\equiv$ *congruence*, and $\rightarrow$ *reduction* of the calculus. We require that reduction is *invariant under congruence*, i.e., $(\equiv \circ \rightarrow \circ \equiv) \subseteq \rightarrow$, where $\circ$ stands for relational composition[4]. 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):

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

[4]If $\rightarrow_1$ and $\rightarrow_2$ are two binary relations on some set $\mathcal{E}$ and $E, E'' \in \mathcal{E}$, then $E \rightarrow_1 \circ \rightarrow_2 E''$ if and only if there exists $E' \in \mathcal{E}$ such that $E \rightarrow_1 E'$ and $E' \rightarrow_2 E''$.

$$\begin{tikzcd}
& E \arrow[dl] \arrow[dr] & & & & E \arrow[dl] \arrow[dr] & \\
E_1 \arrow[dr] & & E_2 \arrow[dl] & \text{or} & E_1 & \equiv & E_2 \\
& F & & & & &
\end{tikzcd}$$

Figure 5: Uniform Confluence

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

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). $\square$

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:

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

where $x{:}\overline{y}/E \not\equiv x{:}\overline{z}/F$[5]. 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 may depend on the termination 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.

[5]The flexibility provided by the side condition $x{:}\overline{y}/E \not\equiv x{:}\overline{z}/F$ is needed for encoding multiple triggering in δ_0. Consider for instance $[\![\mathbf{tr}(t) \mid \mathbf{tr}(t)]\!] \equiv t{:}y/y \mid t{:}y/y$ as introduced in Section 11.

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

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

$$x{=}y \mid y{:}\overline{z}/E \ \rightarrow_F \ x{:}\overline{z}/E \mid y{:}\overline{z}/E \qquad\qquad \mathbf{tr}(t) \mid t.E \ \rightarrow_T \ \mathbf{tr}(t) \mid E$$

Figure 6: The δ-Calculus

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 $F_1 \equiv F_2$. $\square$

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*[6] $x{=}y$ 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.

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

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 defined 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* let $i{=}I$ in $(ii)i$ *referred to by z is represented in δ as follows:*

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

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

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

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 $f{:}xt_x yt_y z/(z{=}x \mid \mathbf{tr}(t_x))$. 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:*

$$(\nu x)(\nu t_x)(\nu y)(\nu t_y)(fxt_x yt_y z \mid t_x.iix \mid t_y.iiy)$$

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

$$\begin{aligned}
&(\nu x)(\nu t_x)(\nu y)(\nu t_y)(fxt_x yt_y z \mid t_x.iix \mid t_y.iiy)\\
\rightarrow_A\;& (\nu x)(\nu t_x)(z{=}x \mid \mathbf{tr}(t_x) \mid t_x.iix) \mid (\nu y)(\nu t_y)(t_y.iiy)\\
\rightarrow_T\;& (\nu x)(\nu t_x)(z{=}x \mid \mathbf{tr}(t_x) \mid iix) \mid (\nu y)(\nu t_y)(\ldots)\\
\rightarrow^*\;& z{:}xy/y{=}x \mid (\nu y)(\nu t_y)(\nu x)(\nu t_x)(\ldots)
\end{aligned}$$

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

Proof. 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 $E \in \mathcal{E}$, 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. □

Expressions

$$M, N ::= x \mid V \mid MN \qquad V ::= \lambda x.M$$

Reduction

$$(\lambda x.M)V \rightarrow_{\text{value}} M[V/x] \qquad (\lambda x.M)N \rightarrow_{\text{name}} M[N/x]$$

Contextual Rules

$$\frac{M \rightarrow_{\text{value}} M'}{MN \rightarrow_{\text{value}} M'N} \qquad \frac{N \rightarrow_{\text{value}} N'}{MN \rightarrow_{\text{value}} MN'} \qquad \frac{M \rightarrow_{\text{name}} M'}{MN \rightarrow_{\text{name}} M'N}$$

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

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.

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 7. 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.*

$$\begin{array}{rcl} z{=_v}MN & \overset{\text{def}}{\equiv} & (\nu x)\,(x{=_v}M \mid (\nu y)\,(x\,y\,z \mid y{=_v}N)) \\ z{=_v}\lambda x.M & \overset{\text{def}}{\equiv} & z{:}x\,y/y{=_v}M \\ z{=_v}x & \overset{\text{def}}{\equiv} & z{=}x \end{array}$$

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

$$\begin{array}{rcl} z{=_n}MN & \overset{\text{def}}{\equiv} & (\nu x)\,(x{=_n}M \mid (\nu y)(\nu t_y)\,(x y t_y z \mid t_y.y{=_n}N)) \\ z{=_n}\lambda x.M & \overset{\text{def}}{\equiv} & z{:}x t_x\,y/y{=_n}M[x{\diamond}t_x/x] \\ z{=_n}x{\diamond}t_x & \overset{\text{def}}{\equiv} & z{=}x \mid \mathbf{tr}(t_x) \end{array}$$

Figure 9: Embedding the Call-by-Name λ-Calculus with Call-by-Need Complexity

Proof. 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* $\mathcal{C}_{\text{value}}(M)$ and the *call-by-name complexity* $\mathcal{C}_{\text{name}}(M)$ 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: $\mathcal{C}_{\text{value}}(M) = \mathcal{C}^A(z{=_v}M)$.*

Proof. A proof simplifies it's predecessor in [Nie94] is presented Section 14. It is based on a complexity simulation introduced in Section 12 and makes heavy use of uniform confluence for covering the additional freedom provided by call-by-value reduction in δ. We define our complexity simulation in the style of [Mil92] using explicit substitutions. □

An embedding $z \mapsto z{=_n}M$ of the call-by-name λ-calculus into δ is given in Figure 9. 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:

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

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 $z{=}_nM$ terminates. Furthermore:*

$$\mathcal{C}^A(z{=}_nM) \leq \min\{\mathcal{C}_{\text{value}}(M),\, \mathcal{C}_{\text{name}}(M)\}\,.$$

Proof. Preservation of termination and the estimation $\mathcal{C}^A(z{=}_nM) \leq \mathcal{C}_{\text{name}}(M)$ are proved in Section 15. These are the most difficult results to prove in this paper. The proof is based on a shortening simulation introduced in Section 12. It factorises into Theorem 12.2 and Corollary 15.2.

The proof of the estimation $\mathcal{C}^A(z{=}_nM) \leq \mathcal{C}_{\text{value}}(M)$ is given in Section 16. Applying Theorem 8.2 it is sufficient to compare the A-complexities of $z{=}_nM$ and $z{=}_vM$. This can be done with a lengthening simulation introduced in Section 12 and is stated in Corollary 16.5.

We note that our simulation technique makes use of uniform confluence such that we need the admissibility of embedded expressions as proved in Section 10. □

Extension 8.4 *It is straightforward to express mutual recursion in δ, both in a call-by-value and in a call-by-need manner:*

$$z{=}_v\mathsf{letrec}\ \overline{x}{=}\overline{M}\,\mathsf{in}\,N \overset{\text{def}}{\equiv} (\nu\overline{x})\,(\overline{x}{=}_v\overline{M} \mid z{=}_vN)$$

$$z{=}_n\mathsf{letrec}\ \overline{x}{=}\overline{M}\,\mathsf{in}\,N \overset{\text{def}}{\equiv} (\nu\overline{x})(\nu\overline{t})\,(\overline{t}.\overline{x}{=}_n\overline{M}\theta \mid z{=}_nN\theta)$$

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 $z{=}_nM$ equals 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 10. Again, we only consider standard reduction. The reduction $\rightarrow_{\text{need}}$ of the call-by-need λ-calculus is a union of four relations:

$$\rightarrow_{\text{need}} \;=\; \rightarrow_I \cup \rightarrow_V \cup \rightarrow_{Ans} \cup \rightarrow_C$$

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

Expressions

$$L ::= x \mid V \mid L_1 L_2 \mid \mathsf{let}\ x{=}L_2\ \mathsf{in}\ L_1 \qquad V ::= \lambda x.L$$

Answers

$$A ::= V \mid \mathsf{let}\ x{=}L\ \mathsf{in}\ A$$

Evaluation Contexts

$$E ::= [\,] \mid EL \mid \mathsf{let}\ x{=}L\ \mathsf{in}\ E \mid \mathsf{let}\ x{=}E_2\ \mathsf{in}\ E_1[x] \qquad \frac{L \to L'}{E[L] \to E[L']}$$

Reduction

$$(\lambda x.L_1)L_2 \to_I \mathsf{let}\ x{=}L_2\ \mathsf{in}\ L_1$$

$$\mathsf{let}\ x{=}V\ \mathsf{in}\ E[x] \to_V \mathsf{let}\ x{=}V\ \mathsf{in}\ E[V]$$

$$\mathsf{let}\ y{=}(\mathsf{let}\ x{=}L\ \mathsf{in}\ A)\ \mathsf{in}\ E[y] \to_{Ans} \mathsf{let}\ x{=}L\ \mathsf{in}\ (\mathsf{let}\ y{=}A\ \mathsf{in}\ E[y])$$

$$(\mathsf{let}\ x{=}L_1\ \mathsf{in}\ A)L_2 \to_C \mathsf{let}\ x{=}L_1\ \mathsf{in}\ AL_2$$

Figure 10: The Call-by-Need λ-Calculus

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

Proof. Evaluation context determine a unique term position where reduction may happen. □

By Proposition 9.1, it makes sense to define the call-by-need complexity $\mathcal{C}_{\text{need}}(L)$ 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:

$$z{=}_n \mathsf{let}\ x{=}L_2\ \mathsf{in}\ L_1 \equiv (\nu x)(\nu t)(t.x{=}_n L_2 \mid z{=}_n L_1[x{\diamond}t/x])$$

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 $\mathcal{C}_{\text{need}}(L) = \mathcal{C}^A(z{=}_n L)$.*

Proof. The proof is presented in Section 18, Corollary 18.3. If is based on a complexity simulation again. □

$$\frac{\Gamma;\ I\ \rhd\ E}{\Gamma;\ I\backslash\{x\}\ \rhd\ (\nu x)E} \qquad \frac{\Gamma;\ I_1\ \rhd\ E_1 \qquad \Gamma;\ I_2\ \rhd\ E_2}{\Gamma;\ I_1\cup I_2\ \rhd\ E_1 \mid E_2}\ \ I_1\cap I_2=\emptyset$$

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

$$\Gamma, x{:}\,\llparenthesis\overline{\tau}\rrparenthesis, y{:}\,\llparenthesis\overline{\tau}\rrparenthesis;\ \{x\}\ \rhd\ x{=}y \qquad\qquad \Gamma, t{:}\,\mathbf{tr};\ \emptyset\ \rhd\ \mathbf{tr}(t)$$

$$\Gamma, x{:}\,\llparenthesis\overline{\sigma}^{\overline{\eta}}\rrparenthesis, \overline{y}{:}\,\overline{\sigma};\ I\ \rhd\ x\overline{y}\ ,\quad \mathcal{O}(\overline{y}{:}\,\overline{\sigma}^{\overline{\eta}})\subseteq I$$

Figure 11: Linear Type Checking

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:

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

Our type systems distinguishes two classes of variables, trigger and single assignment variables. We use **tr** as type for trigger variables. A single assignment variable has a procedural type $\llparenthesis\overline{\tau}\rrparenthesis$, where $\overline{\tau}$ is a sequence of argument types. For instance, the variable z in $z{=}_vM$ is typed by $\mu\alpha.\llparenthesis\alpha^{\mathsf{in}}\ \alpha^{\mathsf{out}}\rrparenthesis$. 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.

If $\overline{y}=(y_i)_{i=1}^{n}$, $\overline{\sigma}=(\sigma_i)_{i=1}^{n}$, and $\overline{\eta}=(\eta_i)_{i=1}^{n}$, then we write $\overline{\sigma}^{\overline{\eta}}$ for the sequence of annotated types $({\sigma_i}^{\eta_i})_{i=1}^{n}$ and $\overline{y}{:}\,\overline{\sigma}$ for the sequence of type assumptions $y_1{:}\,\sigma_1,\ldots,y_n{:}\,\sigma_n$. The *output variables* $\mathcal{O}(\overline{y}{:}\,\overline{\sigma}^{\overline{\eta}})$ in a sequence of type assumptions are defined as follows:

$$\mathcal{O}(\overline{y}{:}\,\overline{\sigma}^{\overline{\eta}})\ =\ \{y_i \mid 1\le i\le n,\ \eta_i=\mathsf{out},\ \text{and}\ \sigma_i\ne\mathbf{tr}\}$$

A *judgement for* E is a triple $\Gamma;\ I\ \rhd\ E$, where Γ is an environment and I is a set of variables. An expression E is *well-typed*, if there exists a judgement for E derivable with the rules in Figure 11. If $\Gamma;\ I\ \rhd\ E$ is derivable, then I contains those single assignment variables, to which an abstraction may be assigned during a computation of E. Such variables correspond to input channels in the π-calculus.

Lemma 10.1 (Subject Reduction Property) *If E is well-typed and $E \to^* F$, then F is well-typed.*

Proof. By induction on derivations of judgements. $\square$

Lemma 10.2 *An inconsistent expression is not well-typed.*

Proof. An expression $x{:}\overline{y}/E \mid x{:}\overline{z}/F$ is not well-typed (even if $E \equiv F$). A potential type judgement would have to be of the following form:

$$\frac{\dfrac{\cdots}{\Gamma;\ \{x\}\ \triangleright\ x{:}\overline{y}/E} \qquad \dfrac{\cdots}{\Gamma;\ \{x\}\ \triangleright\ x{:}\overline{z}/F}}{\Gamma;\ \{x\}\ \triangleright\ x{:}\overline{y}/E \mid x{:}\overline{z}/F}$$

This is impossible by the side condition $\{x\} \cap \{x\} = \emptyset$ of the typing rule for composition. $\square$

Corollary 10.3 *A well-typed expression is admissible.*

Proof. Immediate from Lemmata 10.1 and 10.2 $\square$

Proposition 10.4 *All expressions $z{=}_vM$ and $z{=}_nL$ are well-typed and hence admissible.*

Proof. For all closed expressions M and L the following judgements are derivable with the rules in Figure 11, where η is arbitrary:

$$z{:}\,\mu\alpha.(\!(\alpha^{\mathsf{in}}\ \alpha^{\mathsf{out}})\!);\ \{z\}\ \triangleright\ z{=}_vM \qquad z{:}\,\mu\alpha.(\!(\alpha^{\mathsf{in}}\ \mathbf{tr}^{\eta}\ \alpha^{\mathsf{out}})\!);\ \{z\}\ \triangleright\ z{=}_nL$$

This can check by induction on the structure of M resp L. A slightly stronger invariant is needed for non-closed subexpressions, where all variables are substituted by pairs via $[x{\diamond}t/x]$. $\square$

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 $z{=}_vM$ and $z{=}_nL$ are binary and ternary respectively. An embedding of n-ary δ-expressions into δ_0 is given in Figure 12.

We have to be quite careful when formulating a correctness result for the embedding $E \mapsto [\![E]\!]$. The reason is that the translation of cyclic reference chains does not preserve termination. For instance, the expression $E \overset{\text{def}}{\equiv} x\overline{y} \mid x{=}x$ is terminating whereas $[\![E]\!] \equiv x\overline{y} \mid x{:}\overline{y}/x\overline{y}$ is not.

We call E *locally cyclic*, if there exists a sequence $(x_i)_{i=1}^{n}$ such that E contains a subexpression of the form $x_1{=}x_2 \mid \ldots \mid x_{n-1}{=}x_n$. We call E *cyclic* if there exists F, which is locally cyclic and satisfies $E \to^* F$, and *acyclic* otherwise.

$$
\begin{array}{rclrcl}
[\![t.E]\!] & \overset{\text{def}}{\equiv} & (\nu y)(ty \mid y{:}/[\![E]\!]) & [\![\mathbf{tr}(t)]\!] & \overset{\text{def}}{\equiv} & t{:}y/y \\
[\![x{=}y]\!] & \overset{\text{def}}{\equiv} & x{:}\overline{z}/y\overline{z}\ , \quad \mathsf{length}(\overline{z}) = n & & & \\
[\![E \mid F]\!] & \overset{\text{def}}{\equiv} & [\![E]\!] \mid [\![F]\!] & [\![(\nu x)E]\!] & \overset{\text{def}}{\equiv} & (\nu x)[\![E]\!] \\
[\![x{:}\overline{y}/E]\!] & \overset{\text{def}}{\equiv} & x{:}\overline{y}/[\![E]\!] & [\![x\overline{y}]\!] & \overset{\text{def}}{\equiv} & x\overline{y}
\end{array}
$$

Figure 12: Embedding n-ary δ-expressions in δ_0

Theorem 11.1 *If E is a well-typed, acyclic, and n-ary δ-expression, then $[\![E]\!]$ is admissible and terminates if and only if E terminates.*

Proof. This is proved in Section 17, Corollary 17.12. The simulation technique of Section 12 is applicable again. □

Proposition 11.2 *For all z, closed M and L, the expressions $z{=}_vM$ and $z{=}_nL$ are acyclic.*

Proof. We can show acyclicity by extending linear type checking in Figure 11. In the extended system, we derive judgements of the form $\Gamma;\, O;\, < \ \triangleright\ E$, where $<$ is some acyclic ordering on the set of variables. Typical examples for type checking rules of the extended system are:

$$\Gamma, x{:}\,(\!(\overline{\tau})\!), y{:}\,(\!(\overline{\tau})\!);\ \{x\};\ \prec\ \ \triangleright\ x{=}y \qquad x \prec y$$

$$\frac{\Gamma, \overline{y}{:}\,\overline{\sigma};\ O;\ \prec'\ \ \triangleright\ E}{\Gamma, x{:}\,(\!(\overline{\sigma}^{\overline{\eta}})\!);\ \{x\};\ \prec\ \ \triangleright\ x{:}\overline{y}/E} \qquad \begin{array}{l} \prec'\ = \prec \langle \mathcal{O}(\overline{y}{:}\,\overline{\sigma}^{\overline{\eta}})\rangle \\ O \subseteq \mathcal{O}(\overline{y}{:}\,\overline{\sigma}^{\overline{\eta}}) \end{array}$$

The ordering $\prec \langle \mathcal{O}(\overline{y}{:}\,\overline{\sigma}^{\overline{\eta}})\rangle$ consists of all pairs (y, z) such that $y \in \mathcal{O}(\overline{y}{:}\,\overline{\sigma}^{\overline{\eta}})$ and $z \notin \mathcal{O}(\overline{y}{:}\,\overline{\sigma}^{\overline{\eta}})\}$, or $y \prec z$ but not $y \in \mathcal{O}(\overline{y}{:}\,\overline{\sigma}^{\overline{\eta}})$. It is not difficult to verify that locally cyclic expressions are not well-typed in the extended system. Since the subject reduction property holds as before, cyclic expressions are not well-typed. On the other hand side the expressions $z{=}_vM$ and $z{=}_nL$ are well-typed and hence acyclic. □

We note that the embedding $E \mapsto [\![E]\!]$ does not preserve complexity in an obvious way. The main problem is about forwarding, which is illustrated by the following examples, where we assume that u_1, u_2, x, y denote distinct variables:

$$
\begin{array}{lll}
E_1 \overset{\text{def}}{\equiv} xu \mid x{=}y & \mathcal{C}(E_1) = 0 & \mathcal{C}([\![E_1]\!]) = 1 \\
E_2 \overset{\text{def}}{\equiv} x{=}y \mid y{:}z/zz & \mathcal{C}(E_2) = 1 & \mathcal{C}([\![E_2]\!]) = 0 \\
E_3 \overset{\text{def}}{\equiv} xu_1 \mid xu_2 \mid x{=}y \mid y{:}z/zz & \mathcal{C}(E_3) = 3 & \mathcal{C}([\![E_3]\!]) = 4
\end{array}
$$

12 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.

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 *Let S be a relation on $\mathcal{E} \times \mathcal{G}$ and Φ be an embedding from $\mathcal{E}$ into $\mathcal{G}$. We call S a* shortening simulation for Φ *if it satisfies the following conditions for all E, E', and G:*

(Sim1) $(E, \Phi(E)) \in S$.

(Sim2) If E is irreducible and $(E, G) \in S$, then G is irreducible.

(Sim3) 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 $G \rightarrow G'$.

$$\begin{array}{ccccc} E & \rightarrow & E' & \geq & E'' \\ S & & & & S \\ G & & \rightarrow & & G' \end{array}$$

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 shortens complexity, i.e. $\mathcal{C}(\Phi(E)) \leq \mathcal{C}(E)$ for all E.*

Proof. We assume a shortening simulation S for Φ and $(E, G) \in S$. At first, we claim $\mathcal{C}(G) \leq \mathcal{C}(E)$ if $\mathcal{C}(E) < \infty$. This can be proved by induction on $\mathcal{C}(E)$. If $\mathcal{C}(E) = 0$ then E is irreducible such that G is irreducible by (*Sim*2) . Hence $\mathcal{C}(G) = 0$. If $\mathcal{C}(E) = n \geq 1$ then there exists E' such that $E \rightarrow E'$. By uniformity $\mathcal{C}(E') = n - 1$ follows. Condition (*Sim*3) implies the existence of E'' and G' such that $G \rightarrow G'$, $\mathcal{C}(E') \geq \mathcal{C}(E'')$, and $(E'', G') \in S$. By induction hypothesis we obtain $\mathcal{C}(G') \leq \mathcal{C}(E'')$. The uniformity of both calculi implies:

$$\mathcal{C}(G) = \mathcal{C}(G') + 1 \leq \mathcal{C}(E'') + 1 \leq \mathcal{C}(E') + 1 = \mathcal{C}(E)$$

The theorem follows from both claims and condition (*Sim*1) . □

Definition 12.3 *Let S be a relation on $\mathcal{E} \times \mathcal{G}$ and Φ be an embedding from $\mathcal{E}$ into $\mathcal{G}$. We call S a* lengthening simulation for Φ *if it satisfies* (*Sim*1) *and the following condition for all E, E', and G:*

(Sim4) If $E \to E'$ and $(E, G) \in S$, then exists $G', G'' \in \mathcal{G}$ such that $(E', G'') \in S$, $G \to G'$ and $\mathcal{C}(G') \geq \mathcal{C}(G'')$.

$$\begin{tikzcd} E \arrow[rr] & & E' \\ S & & S \\ G \arrow[r] & G' \arrow[r, phantom, "\geq"] & G'' \end{tikzcd}$$

We call S a complexity simulation *for Φ if S is a shortening and a lengthening simulation for Φ.*

Proposition 12.4 *Let Φ be an embedding between uniform calculi. If there exists a lengthening simulation for Φ, then Φ lengthens complexity, i.e. $\mathcal{C}(E) \leq \mathcal{C}(\Phi(E))$ for all E.*

Proof. Let S be a lengthening simulation for Φ and $(E, G) \in S$. By induction on n we can show that if there exists a derivation of E of length n, then there exists a derivation of G of length $\geq n$. □

Corollary 12.5 *Let Φ be an embedding between uniform calculi. If there exists a complexity simulation for Φ, then Φ preserves complexity (and termination).*

Proof. Immediate from Theorem 12.2 and Proposition 12.4. □

13 Notation

We need several notations for defining simulations and proving them correct. We introduce notations for explicit substitutions, sequences, and specialised reduction relations.

We use the following notation for explicit substitutions ([Mil92, ACCL91]). If $\overline{y} = (y_i)_{i=1}^n$ and $\overline{L} = (L_i)_{i=1}^n$, then $\mathsf{let}\ \overline{y}{=}\overline{L}\ \mathsf{in}\ L'$ represents a λ-term:

$$\mathsf{let}\ \overline{y}{=}\overline{L}\ \mathsf{in}\ L' \overset{\text{def}}{\equiv} L'[L_n/y_n]\ldots[L_1/y_1]$$

We will freely make use of some further sequent notation. If furthermore $\overline{x} = (x_i)_{i=1}^n$, $\overline{t} = (t_i)_{i=1}^n$, $\overline{z} = (z_i)_{i=1}^n$, and $\overline{E} = (E_i)_{i=1}^n$, then we write:

$$\begin{array}{rcl@{\qquad}rcl}
\overline{z}{=}\overline{L} & \overset{\text{def}}{\equiv} & z_1{=}L_1\ \ldots\ z_n{=}L_n & \overline{x}\,\overline{y}\,\overline{t}\,\overline{z} & \overset{\text{def}}{\equiv} & x_1 y_1 t_1 z_1 \mid \ldots \mid x_n y_n t_n z_n \\
\overline{t}.\overline{E} & \overset{\text{def}}{\equiv} & t_1.E_1 \mid \ldots \mid t_1.E_n & (\nu\overline{y})E & \overset{\text{def}}{\equiv} & (\nu y_1)\ldots(\nu y_n)E \\
\overline{E} & \overset{\text{def}}{\equiv} & E_1 \mid \ldots \mid E_n & \overline{z}{=}_v\overline{M} & \overset{\text{def}}{\equiv} & z_1{=}_v M_1 \mid \ldots \mid z_n{=}_v M_n \\
\overline{z}{=}_n\overline{L} & \overset{\text{def}}{\equiv} & z_1{=}_n L_1 \mid \ldots \mid z_n{=}_n L_n & \mathcal{V}(\overline{x}) & \overset{\text{def}}{=} & \{x_1 \ldots x_n\}
\end{array}$$

If $\overline{\mu} = (\mu_j)_{j=1}^n$ is a sequence of variables or expressions then we write $\overline{\mu}^{<i}$ for the sequence $(\mu_j)_{j=1}^{i-1}$ and $\overline{\mu}^{>i}$ for the sequence $(\mu_j)_{j=i+1}^n$. The *concatenation* of two sequences $\overline{\mu}$ and $\overline{\nu}$ is denoted by $\overline{\mu}\overline{\nu}$.

Let $(\mathcal{E}, \equiv, \rightarrow)$ be a calculus, $E, E' \in \mathcal{E}$, and n a natural number. We write $E \rightarrow^n E'$ or $E \rightarrow^{\leq n} E'$, if E reduces in exactly (resp less than) n steps to E'. Formally, we define the relations $\rightarrow^n$ and $\rightarrow^{\leq n}$ as follows:

$$\rightarrow^0 \;=\; \equiv\,, \qquad \rightarrow^{n+1} \;=\; \rightarrow^n \circ \rightarrow\,, \qquad \rightarrow^{\leq n} \;=\; \cup\{\rightarrow^i \mid 0 \leq i \leq n\}$$

We note that $\rightarrow^* = \cup\{\rightarrow^i \mid 0 \leq i < \infty\}$.

For reflecting A-complexity, we define the relation $\hookrightarrow = (\rightarrow_F \cup \rightarrow_T)^*$. Let δ' be the variant of δ with the reduction $\hookrightarrow \circ \rightarrow_A \circ \hookrightarrow$ instead of $\rightarrow$.

Proposition 13.1 *The restriction of δ' to admissible expressions is uniform. For all admissible expressions E the complexity of E in δ' and the A-complexity of E (which is defined relative to δ) coincide: $\mathcal{C}_{\delta'}(E) = \mathcal{C}^A(E)$.*

Proof. This is an immediate consequence of Theorem 7.5. □

In expression of the δ-calculus, top-level declarations do not matter for complexity and termination considerations. We write $E \approx F$ if there exists $\overline{x}$ and $\overline{y}$ such that $(\nu\overline{x})E \equiv (\nu\overline{y})F$. The next two Lemmata justify ignoring top-level declarations in the sequel.

Lemma 13.2 *If $F \approx E \rightarrow_A E'$ then there exists F' such that $F \rightarrow_A F' \approx E'$. If $F \approx E \rightarrow_F E'$ then there exists F' such that $F \rightarrow_F F' \approx E'$. If $F \approx E \rightarrow_T E'$ then there exists F' such that $F \rightarrow_T F' \approx E'$.*

$$\begin{tikzcd}
E \arrow[r, "A"] \arrow[d, phantom, "\wr\wr"] & E' \arrow[d, phantom, "\wr\wr"] & E \arrow[r, "F"] \arrow[d, phantom, "\wr\wr"] & E' \arrow[d, phantom, "\wr\wr"] & E \arrow[r, "T"] \arrow[d, phantom, "\wr\wr"] & E' \arrow[d, phantom, "\wr\wr"] \\
F \arrow[r, "A"] & F' & F \arrow[r, "F"] & F' & F \arrow[r, "T"] & F'
\end{tikzcd}$$

Lemma 13.3 *The relation $\approx$ is closed under weak context and invariant under structural congruence, i.e. it satisfies the contextual rules in Figure 4 (with $\rightarrow$ replaced by $\approx$).*

14 A Complexity Simulation for Call-by-Value

We proof the adequacy of the embedding $M \mapsto z{=}_v M$ from the call-by-value λ-calculus into δ as stated in Theorem 8.2.

Our goal is to establish the equation $\mathcal{C}^A(z{=}_v M) = \mathcal{C}_{\mathrm{value}}(M)$ for all closed λ-expressions M. By Proposition 13.1 it is sufficient to show $\mathcal{C}_{\delta'}(z{=}_v M) = \mathcal{C}_{\mathrm{value}}(M)$. We will apply

Corollary 12.5 once we have constructed a complexity simulation for the above embedding considered into δ' instead of δ. The necessary application conditions for Theorem 12.2 are verified by Propositions 13.1, 10.4, and Proposition 8.1.

Example 14.1 *Before formally defining a complexity simulation, we illustrate it by a simple example. Let $C = \lambda x.xx$ be a λ-abstraction copying its argument and $I = \lambda x.x$ the identity.*

$$\begin{array}{rcl} C(CI) & \rightarrow_{\text{value}} & \mathsf{let}\ y_1{=}C\ \ z_1{=}I\ \ \mathsf{in}\ C(z_1 z_1) \\ & \equiv & \mathsf{let}\ y_1{=}C\ \ z_1{=}I\ \ \mathsf{in}\ C(II) \end{array}$$

In the first step, we have reduced the redex CI. Both involved abstractions have been moved into an environment. Note that only abstractions are moved into the environment. In the second step, we have forwarded abstractions into the next actual application. These two steps reflect the general scheme.

$$\begin{array}{rcl} z{=}_v C(CI) & \rightarrow_A \circ \approx & y_1{=}_v C \mid z_1{=}_v I \mid z{=}_v C(z_1 z_1) \\ & \rightarrow_F^2 & y_1{=}_v C \mid z_1{=}_v I \mid z{=}_v C(II) \end{array}$$

Reduction in the δ-calculus behaves very similar. The environment is represented by contexts built up with composition and declaration. Forwarding amounts to explicit $\rightarrow_F$ steps.

Definition 14.2 (v-Representation) *A* v-representation *for (M, E) is a triple $(n, \overline{y}, \overline{M})$, where $\overline{y} = (y_i)_{i=1}^n$ and $\overline{M} = (M_i)_{i=1}^n$. We require the following properties for all $i \in \{1 \ldots n\}$:*

(S_v1) $\mathcal{V}(M_i) \subseteq \{y_1 \ldots y_{i-1}\}$ *and $\overline{y}$ is linear.*

(S_v2) $M \equiv \mathsf{let}\ \ \overline{y}{=}\overline{M}\ \ \mathsf{in}\ y_n$.

(S_v3) $E \approx y_1{=}_v M_1 \mid \ldots \mid y_n{=}_v M_n$.

(S_v4) *If $i < n$ then M_i is an abstraction.*

Lemma 14.3 (Closedness) *If n, $\overline{M}$, $\overline{y}$, and M satisfy* (S_v1) *and* (S_v2)*, then M is closed.*

Proof. By induction on n. If $n = 1$ then $M \equiv \mathsf{let}\ \ y_1{=}M_1\ \ \mathsf{in}\ y_1$ such that $\mathcal{V}(M) \subseteq \mathcal{V}(M_1) \subseteq \emptyset$. If $n > 1$, then we can apply the induction hypothesis to the following representation of M:

$$M \equiv \mathsf{let}\ \overline{y}^{<n-1}{=}\overline{M}^{<n-1}\ \ y_n{=}M_n[M_{n-1}/y_{n-1}]\ \ \mathsf{in}\ y_n$$

□

Definition 14.4 (Relation S_v) *The relation S_v is the set of all pairs (M, E) for which a* v-representation *exists.*

Proposition 14.5 (S_v **is a Complexity Simulation)** *The relation* S_v *is a relation between closed* λ*-expressions and admissible* δ*-expressions. It satisfies the following properties for all* M, z, *and* E*:*

1. *If* M *is closed then* $(M,\ z{=}_vM) \in S$.
2. *If* M *is irreducible with respect to* $\to_{\text{value}}$ *and* $(M,\ E) \in S$, *then* E *is irreducible with respect to* $\to_A \cup \to_F \cup \to_T$.
3. *If* $(M,\ E) \in S$ *and* $M \to_{\text{value}} M'$, *then there exists* E' *such that* $E \to_F^{\leq 2} \circ \to_A E'$ *and* $(M',\ E') \in S$[7] .

$$\begin{array}{cccc} M & \to_{\text{value}} & & M' \\ S & & & S \\ E & \to_F^{\leq 2} & \to_A & E' \end{array}$$

Proof.

1. The triple $(n,\ (z),\ (M))$ is a v-representation of $(M,\ z{=}_vM)$. Property (S_v1) follows from the closedness of M and (S_v2)-(S_v4) are trivial.
2. Let M be closed and irreducible with respect to $\to_{\text{value}}$. Hence M is an abstraction such that $z{=}_vM$ is an abstraction and therefore irreducible with respect to $\to_A \cup \to_F \cup \to_T$.
3. Let $(n,\ \overline{y},\ \overline{M})$ be a v-representation of $(M,\ E)$ and $M \to_{\text{value}} M'$. Applying the following Lemma 14.7, there exists sequences $\overline{x}$ and $\overline{V}$ of length m, and an expression E' such that $(n+m,\ \overline{y}^{<n}\overline{x}y_n,\ \overline{M}^{<n}\overline{V}M'_n)$ is a v-representation for $(M',\ E')$ and $E \to_F^{\leq 2} \circ \to_A E'$.

□

Corollary 14.6 *The relation* S_v *is a complexity simulation for the mapping* $M \mapsto z{=}_vM$ *considered as embedding from the call-by-value* λ*-calculus restricted to closed expressions into* δ'.

Proof. Immediate from Proposition 14.5. □

[7]This invariant is strong enough for proving that the number of $\to_F$ steps in computations of expressions $z{=}_vM$ is bounded by 2 times the number of $\to_A$ steps. If we would embed a λ-calculus with n-ary instead of unary function, then we would obtain a factor of $n+1$ instead of 2.

Lemma 14.7 *Let $(n,\, \overline{y},\, \overline{M})$ be a v-representation of $(M,\, E)$ and $M \rightarrow_{\text{value}} M'$. Then there exists fresh variables $\overline{x}$, abstractions $\overline{V}$, and a λ-expression M'_n such that $E \rightarrow_F^{\leq 2} \circ \rightarrow_A E'$, $\mathcal{V}(\overline{V}) \subseteq \mathcal{V}(\overline{y}^{<n})$, $\mathcal{V}(M'_m) \subseteq \mathcal{V}(\overline{y}^{<n}\overline{x})$, and:*

$$\begin{array}{rcl} M' & \equiv & \mathsf{let}\ \overline{y}^{<n}{=}\overline{M}^{<n}\ \ \overline{x}{=}\overline{V}\ \ y_n{=}M'_n\ \ \mathsf{in}\ y_n \\ E' & \approx & \overline{y}^{<n}{=}_v\overline{M}^{<n} \mid \overline{x}{=}_v\overline{V} \mid y_n{=}_vM'_n \end{array}$$

Proof. Since $(n,\, \overline{y},\, \overline{M})$ is an v-representation, we know $M \equiv \mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_n$ and $E \approx \overline{y}{=}_v\overline{M}$. Since M can not be an abstraction, property (S$_v$4) implies that M_n is an application N_1N_2 for some N_1 and N_2. Hence $M \equiv P_1P_2$ and:

$$P_1 \ \equiv\ \mathsf{let}\ \overline{y}^{<n}{=}\overline{M}^{<n}\ \ \mathsf{in}\ N_1\,, \qquad P_2 \ \equiv\ \mathsf{let}\ \overline{y}^{<n}{=}\overline{M}^{<n}\ \ \mathsf{in}\ N_2$$

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

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

   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:

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

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

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

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

   $$\begin{array}{rl} E \ \approx & \overline{y}^{<n}{=}_v\overline{M}^{<n} \mid y_n{=}_vy_{l_1}y_{l_2} \\ \approx & \overline{y}^{<n}{=}_v\overline{M}^{<n} \mid x_1{=}y_{l_1} \mid x_2{=}y_{l_2} \mid x_1x_2y_n \\ \rightarrow_F^2 & \overline{y}^{<n}{=}_v\overline{M}^{<n} \mid x_1{=}M_{l_1} \mid x_2{=}M_{l_2} \mid x_1x_2y_n \\ \rightarrow_A & \overline{y}^{<n}{=}_v\overline{M}^{<n} \mid x_1{=}M_{l_1} \mid x_2{=}M_{l_2} \mid y_n{=}_v\tilde{M}_{l_1}[x_2/x] \end{array}$$

   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 $M \rightarrow_{value} M'$ allows for reduction in functional position:

   $$\frac{P_1 \rightarrow_{\text{value}} P'_1}{M \equiv P_1P_2 \rightarrow_{\text{value}} P'_1P_2 \equiv M'}$$

Let z_1 and z_2 be fresh variables and define:

$$E_1 \overset{\text{def}}{\equiv} \overline{y}^{<n} =_v \overline{M}^{<n} \mid z_1 =_v N_1$$

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

$$\begin{array}{lcl} P_1' & \equiv & \mathsf{let}\ \overline{y}^{<n} = \overline{M}^{<n}\ \ \overline{x} = \overline{V}\ \ y_n = N_1'\ \ \mathsf{in}\ y_n \\ E_1' & \approx & \overline{y}^{<n} =_v \overline{M}^{<n} \mid \overline{x} =_v \overline{V} \mid y_n =_v N_1' \end{array}$$

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

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

Furthermore:

$$\begin{array}{lcl} E & \approx & \overline{y}^{<n} =_v \overline{M}^{<n} \mid y_n =_v N_1 N_2 \\ & \approx & \overline{y}^{<n} =_v \overline{M}^{<n} \mid z_1 =_v N_1 \mid z_2 =_v N_2 \mid z_1 z_2 y_n \\ & \rightarrow_F^{\leq 2} \circ \rightarrow_A & \overline{y}^{<n} =_v \overline{M}^{<n} \mid \overline{x} =_v \overline{V} \mid z_1 =_v N_1' \mid z_2 =_v N_2 \mid z_1 z_2 y_n \\ & \approx & \overline{y}^{<n} =_v \overline{M}^{<n} \mid \overline{x} =_v \overline{V} \mid y_n =_v N_1' N_2 \end{array}$$

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

3. Case: The last rule in the derivation of $M \rightarrow_{value} M'$ allows for reduction in argument position:

$$\frac{P_2 \rightarrow_{\text{value}} P_2'}{M \equiv P_1 P_2 \rightarrow_{\text{value}} P_1 P_2' \equiv M'}$$

This case is symmetric to the previous one.

□

15 Shortening Call-by-Name to Call-by-Need

As stated in Theorem 8.3, we prove that the embedding $M \mapsto z =_n M$ from the call-by-name λ-calculus into δ preserves termination such that $\mathcal{C}^A(z =_n M) \leq \mathcal{C}_{\text{name}}(M)$ for all closed λ-expressions M.

By Proposition 13.1 the above complexity estimation is implied by the following one:

$$\mathcal{C}_{\delta'}(z =_n M)\ \leq\ \mathcal{C}_{\text{name}}(M)$$

for all closed M. For proof, we will apply Theorem 12.2 to a shortening simulation for the above embedding considered into δ' instead of δ. This is sufficient to establish our termination statement as well, since termination in δ' and δ are equivalent (since $\rightarrow_F$ and $\rightarrow_T$ terminate). As in the case of our call-by-value embedding, the necessary application conditions for Theorem 12.2 are verified by Propositions 13.1, 10.4, and Proposition 8.1.

15.1 Example

Before formally defining a shortening simulation, we illustrate it by a simple example. We first consider a call-by-name reduction step of $(II)\,I$ with $I \equiv \lambda x.x$:

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

First, the λ-term $(II)\,I$ is flattened. Second, an application is executed. Third, the value I is forwarding to the variable y_1. The corresponding δ-reduction sequence is quite similar:

$$\begin{array}{rcl}
y_3{=}_n(II)\,I & \approx & y_1{=}_nI \mid t_1.z_1{=}_nI \mid y_1z_1t_1y_2 \mid t_2.z_2{=}_nI \mid y_2z_2t_2y_3 \\
 & \rightarrow_A & y_1{=}_nI \mid t_1.z_1{=}_nI \mid y_2{=}_nz_1\diamond t_1 \mid t_2.z_2{=}_nI \mid y_2z_2t_2y_3 \\
 & \rightarrow_T & y_1{=}_nI \mid z_1{=}_nI \mid y_2{=}z_1 \mid \mathbf{tr}(t_1) \mid t_2.z_2{=}_nI \mid y_2z_2t_2y_3 \\
 & \rightarrow_F & y_1{=}_nI \mid z_1{=}_nI \mid y_2{=}_nI \mid \mathbf{tr}(t_1) \mid t_2.z_2{=}_nI \mid y_2z_2t_2y_3
\end{array}$$

The third step - triggering a needed argument - is not visible in the above λ-calculus derivation. Apart from this aspect, both computations are very similar.

15.2 Properties

An appropriate shortening simulation has to cover more aspects than illustrated in the previous example. In this subsection, we formulate sufficiently strong properties for an appropriate candidate.

An interesting example comes with sharing, when comparing call-by-name and call-by-need reduction for the expression $(\lambda x.(x\lambda y.x))(II)$. In this case, we can formulate the relationship via strong call-by-name reduction. We write $M \Rightarrow_{\text{name}} M'$ if M reduces to M' by application of the β-axiom at any position in M.

Proposition 15.1 (Shortening Call-by-Name to Call-by-Need) *There exists a relation S between closed λ-expressions and admissible δ-expressions satisfying the following properties for all M, z, and E:*

1. *If M is closed then $(M,\ z{=}_nM) \in S$.*
2. *If M is irreducible with respect to $\rightarrow_{\text{name}}$ and $(M,\ E) \in S$, then E is irreducible with respect to $\rightarrow_A$, $\rightarrow_F$, and $\rightarrow_T$.*
3. *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 $(M'', E') \in S$[8]*.*

$$\begin{tikzcd}
M \arrow[r, "\text{name}"] \arrow[d, phantom, "S"] & M' \arrow[r, Rightarrow, "*"', "\text{name}"] & M'' \arrow[d, phantom, "S"] \\
E \arrow[r, hook] & \cdot \arrow[r, "A"] & \cdot \arrow[r, hook] & E'
\end{tikzcd}$$

Proof. The relation S is defined in Section 15.3 and proved correct in Section 15.4. □

Corollary 15.2 *There exists a shortening simulation for the mapping $M \mapsto z{=}_n M$ considered as embedding from the call-by-name λ-calculus restricted to closed expressions into δ'.*

Proof. This is a consequence of Proposition 15.1. For proving property (Sho3) we additionally need Lemma 15.3. □

Lemma 15.3 (Reformulation of Plotkin's [Plo75] Standardisation Theorem) *If $M \Rightarrow^*_{\text{name}} M'$, then $\mathcal{C}_{\text{name}}(M) \geq \mathcal{C}_{\text{name}}(M')$.*

Proof. It is suffienct to consider $M \Rightarrow_{\text{name}} M'$. For proof, we define $M \bar{\Rightarrow}_{\text{name}} M'$ iff $M \Rightarrow_{\text{name}} M$ but not $M \rightarrow_{\text{name}} M'$. Trivially, $\Rightarrow_{\text{name}} = \bar{\Rightarrow}_{\text{name}} \cup \rightarrow_{\text{name}}$. In the case $M \rightarrow_{\text{name}} M'$ the lemma follows from uniform confluence of the call-by-name λ-calculus. If $M \bar{\Rightarrow}_{\text{name}} M'$, then it is implied by $\bar{\Rightarrow}^*_{\text{name}}$ being a shortening simulation for the identity embedding from the call-by-name λ-calculus into itself.

(*Sim*1) The relation $M \bar{\Rightarrow}^*_{\text{name}} M$ holds trivially.

(*Sim*2) An expression M is irreducible with respect to $\rightarrow_{\text{name}}$ iff it is an abstraction or an application of the form $((xQ_1)\ldots Q_n)$. The relation $\bar{\Rightarrow}^*_{\text{name}}$ preserves these forms of terms.

(*Sim*3) For all M, M', and N, there exists M'' and N' such that following diagram holds:

$$\begin{tikzcd}
M \arrow[r, "\text{name}"] \arrow[d, Rightarrow, "\text{name}"'] & M' \arrow[r, "*", "\text{name}"'] & M'' \arrow[d, Rightarrow, "*", "\text{name}"'] \\
N \arrow[rr, "\text{name}"] & & N'
\end{tikzcd}$$

[8]Ignoring $\rightarrow_F$ and $\rightarrow_T$ steps is correct in the sense that 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 by formulating a stronger invariant than in Proposition 15.1. As in the call-by-value case, an application invokes at most 2 forwarding steps. Additionally, every application step may raise the need for 1 triggering step.

For proving property (*Sim3*) , we need in fact a sligthly stronger property, where $M \overset{-}{\Rightarrow}_{\text{name}} N$ is replaced by $M \overset{-}{\Rightarrow}^{*}_{\text{name}} N$. This is implied by the above diagram and the inclusion $\overset{-}{\Rightarrow}_{\text{name}} \circ \rightarrow_{\text{name}} \subseteq \rightarrow_{\text{name}} \circ \overset{-}{\Rightarrow}^{*}_{\text{name}}$.
The above diagram can be shown by structural induction on M. For illustration, we consider the case $M \equiv (\lambda x.\tilde{M}_1)M_2$ where the $\overset{-}{\Rightarrow}_{\text{name}}$ step is applied inside of M_2. Hence, $M_2 \Rightarrow_{\text{name}} M_2^{\star}$, $N \equiv (\lambda x.\tilde{M}_1)M_2^{\star}$, and $M' \equiv \tilde{M}_1[M_2/x]$.
There are 4 possible subcases to consider: Either $\tilde{M}_1 \equiv ((xQ_1)\ldots Q_n)$ for some $Q_1,\ldots,Q_n$ or not, and either $M_2 \rightarrow_{\text{name}} M_2^{\star}$ or $M_2 \overset{-}{\Rightarrow}_{\text{name}} M_2^{\star}$. If we choose both times the first possibity, then we obtain:

$$\begin{tikzcd}
(\lambda x.\tilde{M}_1)M_2 \arrow[r, "\text{name}"'] \arrow[d, Rightarrow, "-"{marking}, "\text{name}"] & \tilde{M}_1[M_2/x] \arrow[r, "\text{name}"'] & ((M_2^{\star}Q_1)\ldots Q_n)[M_2/x] \arrow[d, Rightarrow, "-"{marking}, "\text{name}"', "*"] \\
(\lambda x.\tilde{M}_1)M_2^{\star} \arrow[rr, "\text{name}"'] & & \tilde{M}_1[M_2^{\star}/x]
\end{tikzcd}$$

Otherwise, we obtain the required diagram in the form:

$$\begin{tikzcd}
(\lambda x.\tilde{M}_1)M_2 \arrow[r, "\text{name}"'] \arrow[d, Rightarrow, "-"{marking}, "\text{name}"] & \tilde{M}_1[M_2/x] \arrow[d, Rightarrow, "-"{marking}, "\text{name}"', "*"] \\
(\lambda x.\tilde{M}_1)M_2^{\star} \arrow[r, "\text{name}"'] & \tilde{M}_1[M_2^{\star}/x]
\end{tikzcd}$$

□

15.3 Definition

We base our definition of a shortening simulation on the notion of needed variables.

Definition 15.4 (Needed Variables) *Let n be an integer, $\overline{y} = (y_i)_{i=1}^{n}$, $\overline{M} = (M_i)_{i=1}^{n}$ and $1 \leq j \leq n$. The variable y_j* is needed in $\mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ N$, *if the judgement $\mathcal{N}(y_j,\ \mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ N)$ is derivable by the following rules:*

$$\frac{}{\mathcal{N}(x,\ x)} \qquad \frac{\mathcal{N}(x,\ N_1)}{\mathcal{N}(x,\ N_1N_2)} \qquad \frac{\mathcal{N}(x,\ N)}{\mathcal{N}(x,\ \mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ N)}$$

$$\frac{\mathcal{N}(y_j,\ \mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ M_i) \qquad \mathcal{N}(y_i,\ N)}{\mathcal{N}(y_j,\ \mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ N)}\ j < i \leq n$$

Example 15.5 *The variables y_3 and y_1 are needed in* $\mathsf{let}\ y_1{=}I\ \ y_2{=}y_1y_1\ \ y_3{=}y_1y_2\ \ \mathsf{in}\ y_3$ *, whereas y_2 is not needed. The neededness of y_3 is shown by the following derivation:*

$$\frac{\dfrac{\dfrac{\dfrac{}{\mathcal{N}(y_1,\ y_1)}}{\mathcal{N}(y_1,\ y_1y_2)}}{\mathcal{N}(y_1,\ \mathsf{let}\ y_1{=}I\ \ y_2{=}y_1y_1\ \ y_3{=}y_1y_2\ \ \mathsf{in}\ y_1y_2)} \qquad \dfrac{}{\mathcal{N}(y_3,\ y_3)}}{\mathcal{N}(y_1,\ \mathsf{let}\ y_1{=}I\ \ y_2{=}y_1y_1\ \ y_3{=}y_1y_2\ \ \mathsf{in}\ y_3)}$$

Definition 15.6 (n-Representation) *A* n-representation *for* (M, E) *is a five-tuple* $(n, \overline{y}, \overline{M}, \overline{t}, D)$, *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. We require the following properties for all* $i \in \{1 \ldots n\}$*:*

(S_n*1)* $\mathcal{V}(M_i) \subseteq \{y_1 \ldots y_{i-1}\}$ *and the composed sequence* $\overline{y}\overline{t}$ *is linear.*

(S_n*2)* $M \equiv \mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_n$.

(S_n*3)* *There exists* $(E_i)_{i=1}^n$, ϕ, *and* θ *such that* $E \approx E_1 \mid \ldots \mid E_n \mid \phi$, *where* ϕ *is a possibly empty composition of trigger expressions in* $\{\mathbf{tr}(t_j) \mid y_j \notin D\}$, $\theta = [\overline{y}\diamond\overline{t}/\overline{y}]$, *and:*

$$E_i = \begin{cases} t_i.y_i{=}_n M_i\theta & \textit{if } y_i \in D \\ y_j y_k t_k y_i & \textit{if } y_i \notin D \textit{ and } M_i = y_j y_k \textit{ for some } j,k \\ y_i{=}y_j & \textit{if } y_i \notin D \textit{ and } M_i = y_j \textit{ for some } j \\ y_i{=}_n M_i\theta & \textit{if } y_i \notin D \textit{ and } M_i \textit{ is an abstraction} \end{cases}$$

(S_n*4)* *If* $y_i \notin D$ *and* M_i *is an application then* M_i *is an application of variables.*

(S_n*5)* *If* y_i *is needed in* $\mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_n$, *then* $y_i \notin D$.

(S_n*6)* *If* y_i *is not needed in* $\mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_n$, *then* $y_i \in D$ *or* M_i *is an abstraction.*

Definition 15.7 (Relation S_n) *We define the relation* S_n *as the set of all pairs* (M, E) *for which a* n-representation *exists.*

Proposition 15.8 (S_n is a Shortening Simulation) *The relation* S_n *satisfies the conditions of Proposition 15.1.*

Proof. This is the content of the Lemmata 15.11, 15.15, and 15.17. □

15.4 Correctness Proof

We prove Proposition 15.8, which states the correctness of our shortening simulation S_n. We have to validate three properties reconsidered in Propositions 15.11, 15.15, and 15.17.

15.4.1 Property (*Sim*1)

Lemma 15.9 *For every* M *there exists* $m \geq 0$, $(P_i)_{i=1}^m$ *and* Q *such that* $M \equiv (\ldots(Q P_m)\ldots)P_1$ *and* Q *is not an application.*

Proof. By structural induction on M. If M is an abstraction or a variable, then we choose $m = 0$ and $Q \equiv M$. If $M \equiv M_1 M_2$ then there exists $m \geq 0$ and $(P_i)_{i=2}^m$ such that:

$$M_1 \equiv (\ldots(Q P_m)\ldots)P_2$$

If we set $P_1 \equiv M_2$, then we obtain $M \equiv M_1 M_2 \equiv (\ldots(Q P_m)\ldots)P_1$. □

Lemma 15.10 (Flattening) *If $M \equiv (\ldots(QP_m)\ldots)P_1$ for some $m \geq 0$ and $\overline{u} = (u_i)_{i=1}^{m+1}$, $\overline{v} = (v_i)_{i=1}^{m}$, $\overline{s} = (s_i)_{i=1}^{m}$ are variables not contained in $\mathcal{V}(M)$, then the following representations are valid:*

$$\begin{array}{rcl} M & \equiv & \mathsf{let}\ u_1{=}Q\ \ \overline{v}{=}\overline{P}\ \ \overline{u}^{>1}{=}\overline{u}^{<m+1}\overline{v}\ \ \mathsf{in}\ u_{m+1} \\ u_{m+1}{=}_nM & \approx & u_1{=}_nQ \mid \overline{s}.\overline{v}{=}_n\overline{P} \mid \overline{u}^{<m+1}\overline{v}\overline{s}\overline{u}^{>1} \end{array}$$

Proof. By induction on m. In the case $m = 0$ there is nothing to show. If $m > 0$ then $M \equiv M_1M_2$ where $M_1 \equiv (\ldots(QP_1)\ldots)P_{m-1}$, and $M_2 \equiv P_m$. Applying the induction hypothesis to $u_m{=}_nM_1$ we obtain:

$$\begin{array}{rcl} M_1 & \equiv & \mathsf{let}\ u_1{=}Q\ \ \overline{v}^{<m}{=}\overline{P}^{<m}\ \ u_2{=}u_1v_1\ \ldots\ u_m{=}u_{m-1}v_{m-1}\ \ \mathsf{in}\ u_m \\ u_m{=}_nM_1 & \equiv & u_1{=}_nQ \mid \overline{s}^{<m}.\overline{v}^{<m}{=}_n\overline{P}^{<m} \mid u_1v_1s_1u_2 \mid \ldots \mid u_{m-1}v_{m-1}s_{m-1}u_m \end{array}$$

Since $M = M_1P_m$, this implies:

$$\begin{array}{rcl} M & \equiv & \mathsf{let}\ u_1{=}Q\ \ \overline{v}^{<m}{=}\overline{P}^{<m}\ \ u_2{=}u_1v_1\ \ldots\ u_m{=}u_{m-1}v_{m-1}\ \ \mathsf{in}\ u_mP_m \\ & \equiv & \mathsf{let}\ u_1{=}Q\ \ \overline{v}{=}\overline{P}\ \ u_2{=}u_1v_1\ \ldots\ u_{m+1}{=}u_mv_m\ \ \mathsf{in}\ u_{m+1} \\ & \equiv & \mathsf{let}\ u_1{=}Q\ \ \overline{v}{=}\overline{P}\ \ \overline{u}^{>1}{=}\overline{u}^{<m+1}\overline{v}\ \ \mathsf{in}\ u_{m+1} \end{array}$$

The expression $u_{m+1}{=}_nM$ satisfies:

$$u_{m+1}{=}_nM \ \approx\ u_m{=}_nM_1 \mid s_m.v_m{=}_nP_m \mid u_mv_ms_mu_{m+1}$$

Replacing $u_m{=}_nM_1$ in $u_{m+1}{=}_nM$ by its above representation yields:

$$\begin{array}{rcl} u_{m+1}{=}_nM & \approx & u_1{=}_nQ \mid \overline{s}^{<m}.\overline{v}^{<m}{=}_n\overline{P}^{<m} \mid u_1v_1s_1u_2 \mid \ldots \mid u_{m-1}v_{m-1}s_{m-1}u_m \\ & & \mid s_m.v_m{=}_nP_m \mid u_mv_ms_mu_{m+1} \\ & \equiv & u_1{=}_nQ \mid \overline{s}.\overline{v}{=}_n\overline{P} \mid \overline{u}^{<m+1}\overline{v}\overline{s}\overline{u}^{>1} \end{array}$$

□

Proposition 15.11 *The relation S_n satisfies (Sim1) .*

Proof. Let M be a closed λ-expression and z a variable. We have to construct a n-representation for $(M,\ z{=}_nM)$. Lemma 15.9 yields the existence of m and $(P_i)_{i=1}^{m}$ such that $M = (\ldots(QP_m)\ldots)P_1$. Let $(u_i)_{i=1}^{m}$, $(v_i)_{i=1}^{m}$, $(s_i)_{i=1}^{m}$ be sequence of fresh variables and define $u_{m+1} = z$. Applying the flattening Lemma 15.10 yields:

$$\begin{array}{rcl} M & \equiv & \mathsf{let}\ u_1{=}Q\ \ \overline{v}{=}\overline{P}\ \ \overline{u}^{>1}{=}\overline{u}^{<m+1}\overline{v}\ \ \mathsf{in}\ u_{m+1} \\ z{=}_nM & \approx & u_1{=}_nQ \mid \overline{s}.\overline{v}{=}_n\overline{P} \mid \overline{u}^{<m+1}\overline{v}\overline{s}\overline{u}^{>1} \end{array}$$

These properties essentially verify (S_n2) and (S_n3) where $E \equiv z{=}_nM$. In order to formalise this statement, we have to define a n-representation $(n,\ \overline{y},\ \overline{M},\ \overline{t},\ D)$ for $(M,\ E)$ appropriately.

$$\begin{array}{rclll} \overline{y} & = & u_1 & \overline{s} & \overline{u}^{>1}\,, \\ \overline{M} & = & Q & \overline{P} & (\overline{u}^{<m+1}\overline{v})\,, \\ \overline{t} & = & \_ & \overline{v} & \_ \end{array} \qquad \begin{array}{rcl} n & = & 2m+1 \\ D & = & \mathcal{V}(\overline{v}) \end{array}$$

In these definitions, each occurences of the symbol _ stands for a fresh variable. We have to verify the conditions of Definition 15.6. Property (S_n1) follows from the closedness of M. (S_n2) and (S_n3) have already been discussed. (S_n4) holds trivially. For (S_n5) we note that the needed variables in $\mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_n$ are those in $\mathcal{V}(\overline{u})$. For ($\mathrm{S}_n6$) we note that the not needed variables are those in $\mathcal{V}(\overline{v})$. □

15.4.2 Property (*Sim*2)

Lemma 15.12 (Forwarding) *If* $(M, E) \in \mathrm{S}_n$ *then there exists* E' *with* $E \to_F^* E'$ *and there exists a n-representation* $(n, \overline{y}, \overline{M}, \overline{t}, D)$ *of* (M, E') complete under forwarding, *i.e. satisfying the property:*

(S_n7) If $j, k \in \{1 \dots n\}$, $y_j \notin D$, *and* $M_j = y_k$, *then* M_k *is not an abstraction.*

Proof. Let $(n, \overline{y}, \overline{M}, \overline{t}, D)$ be a n-representation of (M, E). We have to construct a n-represent ion of (M, E) satisfying (S_n7) . Suppose there exists a pair of indices $j, k \in \{1 \dots n\}$ such that $y_j \notin D$, $M_j = y_k$, and M_k is an abstraction. We show how to eliminate this index pair by forwarding $\to_F$. Our elimination procedure terminates, since it decreases the number of such index pairs. By assumption and (S_n1) we obtain:

$$\begin{array}{rcl} M & \equiv & \mathsf{let} \dots y_k{=}M_k \dots\ y_j{=}y_k \dots\ \mathsf{in}\ y_n \\ & \equiv & \mathsf{let} \dots y_k{=}M_k \dots\ y_j{=}M_k \dots\ \mathsf{in}\ y_n \end{array}$$

Property (S_n6) implies that y_j is needed in $\mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_n$ (since $y_j \notin D$ and M_j is not an abstraction). By definition of neededness, y_k is also needed in $\mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_n$ such that (S_n5) implies $y_k \notin D$. Hence:

$$\begin{array}{rcl} E & \approx & \dots \mid y_k{=}_n M_k \mid \dots \mid y_j{=}y_k \mid \dots \\ & \to_F & \dots \mid y_k{=}_n M_k \mid \dots \mid y_j{=}_n M_k \mid \dots \end{array}$$

□

Definition 15.13 *Let* $(n, \overline{M}, \overline{y})$ *and* M *satisfy* (S_n1) *and* (S_n2). *A* reference chain from y_n to $y_{\nu(1)}$ *is a sequence* $(y_{\nu(i)})_{i=1}^{p}$, *if* $p \geq 1$ *is an integer, the* $\nu(i)$*'s are indices such that* $1 \leq \nu(1) < \dots < \nu(p) = n$, *and* $M_{\nu(i)} = y_{\nu(i-1)}$ *for all* $1 < i \leq p$. *In this case, we write:*

$$M \equiv \mathsf{let} \dots\ y_{\nu(1)}{=}M_{\nu(1)}\ \dots\ y_{\nu(2)}{=}y_{\nu(1)}\ \dots\ y_n{=}y_{\nu(p-1)}\ \mathsf{in}\ y_n$$

Lemma 15.14 (Reference Chains) *Let* $(n, \overline{y}, \overline{M})$ *and* M *satisfy* (S_n1) *and* (S_n2). *Then there exists* $1 \leq j \leq n$ *and a reference chain from* y_n *to* y_j *such that* M_j *is not a variable.*

Proof. By induction on n. If $n = 1$, then M_1 may not be a variable since M is closed (Lemma 14.3). If $n > 0$ and M_n is not a variable then there is nothing to prove. Otherwise, we use $M \equiv \mathsf{let}\ \overline{y}^{<n}{=}\overline{M}^{<n}\ \mathsf{in}\ M_n$ and apply the induction hypothesis. □

Proposition 15.15 *The relation* S_n *satisfies* (*Sim*2) .

Proof. Let $(M, E) \in \mathrm{S}_n$ and M be irreducible with respect to $\rightarrow_{\mathrm{name}}$. We have to show that E is irreducible with respect to $\hookrightarrow \circ \rightarrow_A \circ \hookrightarrow$. Instead, we prove that E is irreducible with respect to $\rightarrow_A$, $\rightarrow_F$, and $\rightarrow_T$.

Without less of generality, we can assume that S_n is complete under forwarding (Lemma 15.12). Since M is closed (Lemma 14.3) it has to be an abstraction (Lemma 15.9). Lemma 15.14 implies of the existence of $1 \le j \le n$ such that there exists a reference chain from y_n to y_j and M_j is not an variable. Since M is an abstraction M_j has to be an abstraction. Completeness under forwarding (S_n7) implies $j = n$ such that:

$$M \;\equiv\; \mathsf{let} \ldots y_n{=}M_j \;\;\mathsf{in}\; y_n$$

Hence, y_n is a unique needed variable in $\mathsf{let} \ldots y_n{=}M_j \;\mathsf{in}\; y_n$ such that (S_n6) implies for all $i \in \{1 \ldots n{-}1\}$ that $y_i \in D$ or M_i is an abstraction.

Let $E_1, \ldots, E_n$, and ϕ be defined as in (S_n3). This implies $E \approx E_1 \mid \ldots \mid E_n \mid \phi$. Since none of the E_i's may be an application, E is irreducible with respect to $\rightarrow_A$. It is irreducible with respect to $\rightarrow_F$ because none of the E_i's may an directed equation, and irreducible with respect to $\rightarrow_T$ because none of the delayed E_i's is triggered in ϕ. □

15.4.3 Proof of the Invariant

Lemma 15.16 (Shared Redexes) *Let* $(n, \overline{y}, \overline{M}, \overline{t}, D)$ *be a n-representation of* (M, E) *satisfying* (S_n7) *and* $\overline{y} = (y_i)_{i=1}^n$, $\overline{M} = (M_i)_{i=1}^n$, $\overline{t} = (t_i)_{i=1}^n$. *For all* M' *with* $M \rightarrow_{\mathrm{name}} M'$, *there exists* j, k, l *and* x, $\tilde{M}_k$ *such that* $M_j = y_k y_l$, y_j *is needed in* $\mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_n$, $M_k = \lambda x.\tilde{M}_k$, *and:*

$$M' \;\Rightarrow^*_{\mathrm{name}}\; \mathsf{let}\, \overline{y}^{<j}{=}\overline{M}^{<j}\;\; y_j{=}\tilde{M}_k[y_l/x]\;\; \overline{y}^{>j}{=}\overline{M}^{>j}\;\; \mathsf{in}\; y_n$$

Proof. By induction on derivations of $M \rightarrow_{\mathrm{name}} M'$. We have to consider two cases:

1. In the first case, the $M \rightarrow_{\mathrm{name}} M'$ is an instance of the β-axiom: There exists P_1, P_2, x such that:
$$M \equiv (\lambda x.\tilde{P}_1)P_2 \rightarrow_{\mathrm{name}} \tilde{P}_1[P_2/x] \equiv M'$$
Applying Lemma 15.14, there exists a $1 \le j \le n$ and a reference chain from y_n to y_j such that M_j is not a variable. Since M is an application, M_j is an application. (S_n6) implies $M_j = y_k y_l$ for some k, l and (S_n1) yields $k, l < j$. Hence:
$$\lambda x.\tilde{P}_1 \;\equiv\; \mathsf{let}\;\; \overline{y}^{<j}{=}\overline{M}^{<j}\; \mathsf{in}\; y_k\,, \qquad P_2 \;\equiv\; \mathsf{let}\;\; \overline{y}^{<j}{=}\overline{M}^{<j}\; \mathsf{in}\; y_l$$
Applying Lemma 15.14 there exists $1 \le k' \le k$ and a reference chain from y_k to $y_{k'}$ in $\mathsf{let}\;\; \overline{y}^{<k}{=}\overline{M}^{<k}\; \mathsf{in}\; y_k$ there such that $M_{k'}$ is not a variable, i.e. $M_{k'}$ is an abstraction.

The variables y_k and $y_{k'}$ are needed in $\mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_n$ (by induction on the length of reference chains) such that $k = k'$ follows from completeness with respect to forwarding ($\mathrm{S}_n 7$) . Hence, M_k is an abstraction such that there exists M'_k with $M_k \equiv \lambda x.M'_k$. Furthermore:

$$\tilde{P}_1 \equiv \mathsf{let}\ \overline{y}^{<j}{=}\overline{M}^{<j}\ \mathsf{in}\ \tilde{M}_k$$

The following equality justifies the Lemma with $\equiv$ instead of $\Rightarrow^*_{\text{name}}$.

$$\begin{array}{lll} M' & \equiv & \tilde{P}_1[P_2/x] \\ & \equiv & \mathsf{let}\ \overline{y}^{<j}{=}\overline{M}^{<j}\ \mathsf{in}\ \tilde{M}_k[P_2/x] \\ & \equiv & \mathsf{let}\ \overline{y}^{<j}{=}\overline{M}^{<j}\ \mathsf{in}\ \tilde{M}_k[y_l/x] \\ & \equiv & \mathsf{let}\ \overline{y}^{<j}{=}\overline{M}^{<j}\ \ y_j{=}\tilde{M}_k[y_l/x]\ \ \overline{y}^{>j}{=}\overline{M}^{>j}\ \ \mathsf{in}\ y_j \\ & \equiv & \mathsf{let}\ \overline{y}^{<j}{=}\overline{M}^{<j}\ \ y_j{=}\tilde{M}_k[y_l/x]\ \ \overline{y}^{>j}{=}\overline{M}^{>j}\ \ \mathsf{in}\ y_n \end{array}$$

The last step uses the reference chain from y_n to y_j backwards.

2. In the second case, the β-axiom is applied in functional position. There exists P_1, P'_1, P_2 such that the last step in the derivation of $M \rightarrow_{\text{name}} M'$ has the following form:

$$\frac{P_1 \rightarrow_{\text{name}} P'_1}{M \equiv P_1 P_2 \rightarrow_{\text{name}} P'_1 P_2 \equiv M'}$$

Yet another argumentation with reference chains implies the existence of j', k', l' such that:

$$\begin{array}{lll} M & \equiv & \mathsf{let}\ \overline{y}^{<j'}{=}\overline{M}^{<j'}\ \ y_{j'}{=}y_{k'}y_{l'}\ \ \overline{y}^{>j'}{=}\overline{M}^{>j'}\ \ \mathsf{in}\ y_{j'} \\ P_1 & \equiv & \mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ j_{k'} \\ P_2 & \equiv & \mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_{l'} \end{array}$$

By induction hypothesis applied to $P_1 \rightarrow_{\text{name}} P'_1$ there exists j, k, l, and x, $\tilde{M}_k$ such that: $M_j = y_k y_l$, y_j is needed in $\mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_{k'}$, $M_k = \lambda x.\tilde{M}_k$, and:

$$P'_1 \ \Rightarrow^*_{\text{name}}\ \mathsf{let}\ \overline{y}^{<j}{=}\overline{M}^{<j}\ \ y_j{=}\tilde{M}_k[y_l/x]\ \ \overline{y}^{j}{=}\overline{M}^{>j}\ \ \mathsf{in}\ y_{k'}$$

P_2 reduces to a similar expression than P'_1 does:

$$\begin{array}{lll} P_2 & \equiv & \mathsf{let}\ \overline{y}^{<j}{=}\overline{M}^{<j}\ \ y_j{=}y_k y_l\ \ \overline{y}^{>j}{=}\overline{M}^{>j}\ \ \mathsf{in}\ y_{l'} \\ & \Rightarrow^*_{\text{name}} & \mathsf{let}\ \overline{y}^{<j}{=}\overline{M}^{<j}\ \ y_j{=}\tilde{M}_k[y_l/x]\ \ \overline{y}^{>j}{=}\overline{M}^{>j}\ \ \mathsf{in}\ y_{l'} \end{array}$$

Sticking both reductions together concludes the Lemma:

$$\begin{array}{lll} M' & \equiv & P_1 P_2 \\ & \Rightarrow^*_{\text{name}} & \mathsf{let}\ \overline{y}^{<j}{=}\overline{M}^{<j}\ \ y_j{=}\tilde{M}_k[y_l/x]\ \ \overline{y}^{>j}{=}\overline{M}^{>j}\ \ \mathsf{in}\ y_{k'} y_{l'} \\ & \equiv & \mathsf{let}\ \overline{y}^{<j}{=}\overline{M}^{<j}\ \ y_j{=}\tilde{M}_k[y_l/x]\ \ \overline{y}^{>j}{=}\overline{M}^{>j}\ \ \mathsf{in}\ y_k \\ & \equiv & \mathsf{let}\ \overline{y}^{<j}{=}\overline{M}^{<j}\ \ y_j{=}\tilde{M}_k[y_l/x]\ \ \overline{y}^{>j}{=}\overline{M}^{>j}\ \ \mathsf{in}\ y_n \end{array}$$

The second step uses $M_k = y_k y_l$ and the last step a reference chain from y_n to y_k backwards that we left implicit at the beginning of this case. □

Proposition 15.17 (The Invariant) *Let $(M,\, E) \in \mathrm{S}_n$ 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 $(M'',\, E') \in \mathrm{S}_n$.*

Proof. Let $(n,\, \overline{y},\, \overline{M},\, \bar{t},\, D)$ be a n-representation of $(M,\, E)$. We assume without loss of generality that E is complete under forwarding (Lemma 15.12). Let $\overline{y} = (y_i)_{i=1}^n$, $\overline{M} = (M_i)_{i=1}^n$, $\bar{t} = (t_i)_{i=1}^n$, and $D \subseteq \mathcal{V}(\overline{y})$. Let $(E_i)_{i=1}^n$ and ϕ be defined as in $(\mathrm{S}_n 3)$ and $\theta = [\overline{y}\diamond\bar{t}/\overline{y}]$. Since $M \rightarrow_{\text{name}} M'$, we can apply Lemma 15.16 such that there exists j, k, l and x, $\tilde{M}_k$, M'' with the following properties:

(1) $M_j \equiv y_k y_l$.

(2) y_j is needed in $\mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_n$.

(3) $M_k = \lambda x.\tilde{M}_k$.

(4) $M' \Rightarrow^*_{\text{name}} M''$

(5) $M'' \equiv \mathsf{let}\, \overline{y}^{<j}{=}\overline{M}^{<j}\ \ y_j{=}\tilde{M}_k[y_l/x]\ \ \overline{y}^{>j}{=}\overline{M}^{>j}\ \ \mathsf{in}\ y_n$

Applying Lemma 15.9 there exists $m \geq 0$, $\overline{P} = (P_i)_{i=1}^m$, and Q such that:

(6) $\tilde{M}_k = (\ldots(Q P_m)\ldots) P_1$

(7) Q is not an application.

For all $i \in \{1 \ldots m\}$ let u_i, v_i, s_i be fresh variables. We define $u_{m+1} = y_j$, $\overline{u} = (u_i)_{i=1}^{m+1}$, $\overline{v} = (v_i)_{i=1}^m$, and $\overline{s} = (s_i)_{i=1}^m$. Flattening $\tilde{M}_k$ (Lemma 15.10) yields:

(8) $\tilde{M}_k \ \equiv\ \mathsf{let}\ u_1{=}Q\ \ \overline{v}{=}\overline{P}\ \ \overline{u}^{>1}{=}\overline{u}^{<m+1}\overline{v}\ \ \mathsf{in}\ y_j$

(9) $y_j{=}_n\tilde{M}_k \ \approx\ u_1{=}_nQ \mid \overline{s}.\overline{v}{=}_n\overline{P} \mid \overline{u}^{<m+1}\overline{v}\,\overline{s}\,\overline{u}^{>1}$

Since y_j is needed in $\mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_n$ (2), y_k is needed in $\mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_n$ as well. $(\mathrm{S}_n 5)$ implies $y_j, y_k \notin D$ such that:

(10) $E \ \approx\ E_1 \mid \ldots \mid E_n \mid \phi$

(11) $E_j \ \equiv\ y_k y_l t_l y_j$

(12) $E_k \ \equiv\ y_k{=}_n\lambda x.\tilde{M}_k\theta \qquad (3)$

For fresh variables t and z this implies:

(13) $E_k \ \equiv\ y_k{:}xtz/z{=}_n\tilde{M}_k\theta[x\diamond t/x]$

If $\eta = [y_l/x]$, then applying y_k in the context of E_k yields:

$$\begin{array}{rcll} E_j & \rightarrow_A & (y_j{=}_n\tilde{M}_k\theta[x{\diamond}t/x])[y_l/x][t_l/t] & (13) \\ & \equiv & y_j{=}_n\tilde{M}_k\eta\theta & \\ & \equiv & u_1{=}_nQ\eta\theta \mid \overline{s}.\overline{v}{=}_n\overline{P}\eta\theta \mid \overline{u}^{<m+1}\overline{v}\overline{s}\overline{u}^{>1} & (9) \end{array}$$

Combining this result with (10) we obtain:

(14) $E \rightarrow_A E'$

(15) $E' \overset{\text{def}}{\equiv} \overline{E}^{<j} \mid u_1{=}_nQ\eta\theta \mid \overline{s}.\overline{v}{=}_n\overline{P}\eta\theta \mid \overline{u}^{\leq m}\,\overline{v}\overline{s}\overline{u}^{\geq 2} \mid \overline{E}^{>j} \mid \phi$

Next, we construct a five-tuple $\mathcal{R} = (n', \overline{y'}, \overline{M'}, \overline{t'}, D')$, which satisfies all properties of the Lemma except one.

$$\begin{array}{rclllllcrcl} \overline{y'} & = & \overline{y}^{<j} & u_1 & \overline{v} & \overline{u}^{\geq 2} & \overline{y}^{>j}\,, & \qquad & n' & = & n+2m \\ \overline{M'} & = & \overline{M}^{<j} & Q\eta & \overline{P}\eta & (\overline{u}^{\leq m}\,\overline{v}) & \overline{M}^{>j}\,, & & D' & = & D \cup \mathcal{V}(\overline{v}) \\ \overline{t'} & = & \overline{t}^{<j} & t_j & \overline{s} & \_ & \overline{t}^{>j}\,, & & & & \end{array}$$

Property (4) implies $M' \Rightarrow^*_{\text{name}} M''$. We even obtain $E \rightarrow^*_F \circ \rightarrow_A E'$ from (14) and the fact that we completed E under forwarding at the beginning. It remains to show that $\mathcal{R}$ is a n-representation for (M'', E'). $\mathcal{R}$ satisfies all required properties except (S_n5): (S_n1) is simple, (S_n2) follows from (10), (S_n3) is covered by (15). (S_n4) follows from (8) (the variables in $\mathcal{V}(\overline{y'}) \setminus D'$ are those in $\mathcal{V}(\overline{u})$). Property ($\mathrm{S}_n$6) holds, since $\mathcal{V}(\overline{v}) \subseteq D'$ and all other non-needed variables have also been non-needed in the original n-representation.

The tuple $\mathcal{R}$ does not necessary satisfy (S_n5), because Q might be a variable, say y_p. In this case, $u_1{=}_ny_p\theta \equiv u_1{=}y_p \mid \mathbf{tr}(t_p)$. This means that the expression E_p is delayed, even if y_p is needed. We have to use $\rightarrow_T$ for triggering the computation in E_p waiting on t_p. Since M_p may again be a variable, more triggering steps may be needed.

The failure of $\mathcal{R}$ being a n-representation for (M'', E') is harmless, since $\mathcal{R}$ is a least an uncompletely triggered n-representation for (M'', E') in the sense of Definition 15.18. This is sufficient to accomplish the actual proof by applying Lemma 15.19. □

Definition 15.18 *A five-tuple* $(n, \overline{y}, \overline{M}, \overline{t}, D)$ *is called* uncompletely triggered *n*-representation *of* (M, E)*, if it satisfies* (S_n1)-(S_n4), (S_n6), *and* (S'_n5) *, where:*

(S'_n5) If y_i is needed in let $\overline{y}{=}\overline{M}$ in y_n*, then either $y_i \notin D$ or there exists a reference chain $(y_{\nu(j)})_{j=1}^{p}$ such that $y_{\nu(1)} = y_i$, $\{y_{\nu(i)} \mid 1 \leq i < p\} \subseteq D$ and $\mathbf{tr}(t_{\nu(p)})$ is contained in E.*

Lemma 15.19 (Triggering) *If there exists an uncompletely triggered n-representation of (M, E), then there exists E' such that $E \rightarrow^*_T E'$ and $(M, E') \in \mathrm{S}_n$.*

Proof. Let $\mathcal{R} = (n, \overline{y}, \overline{M}, \overline{t}, D)$ be a uncompletely triggered n-representation on (M, E). We call a variable y_i *critical for* $\mathcal{R}$ *and* (M, E), if y_i is needed in $\mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\, y_n$, and $y_i \in D$.

If there exists no critical variable for $\mathcal{R}$ and (M, E), then $\mathcal{R}$ is a n-representation for (M, E). Hence it is sufficient to define a procedure that given a uncompletely triggered n-representation $\mathcal{R}$ for (M, E) computes some E' and $\mathcal{R}'$ such that:

1. $\mathcal{R}'$ is an uncompletely triggered n-representation for (M, E') and $E \rightarrow_T E'$.
2. The number of critical variables for $\mathcal{R}'$ and (M, E') is strictly smaller than the number of critical variables for $\mathcal{R}$ and (M, E).

Let $\mathcal{R} = (n, \overline{y}, \overline{M}, \overline{t}, D)$ be a uncompletely triggered n-representation on (M, E). If there exists a critical variable for $\mathcal{R}$ and (M, E) then by condition ($S_n'5$) there also exists a critical variable $y_i \in D$ such that $\mathbf{tr}(t_i)$ is contained in E. Let $E_1, \ldots, E_n$, and ϕ be defined as in (S_n3). Since $\mathbf{tr}(t_i)$ is contained in E, there exists ϕ' such that $\phi \equiv \mathbf{tr}(t_i) \mid \phi'$. We can reduce E and define E' as follows:

$$\begin{array}{rcl} E & \approx & E_1 \mid \ldots \mid t_i.y_i{=}_n M_i \mid \ldots \mid \mathbf{tr}(t_i) \mid \phi' \\ & \rightarrow_T & E_1 \mid \ldots \mid y_i{=}_n M_i \mid \ldots \mid \mathbf{tr}(t_i) \mid \phi' \\ & \overset{\text{def}}{\equiv} & E' \end{array}$$

If we set $D' = D \setminus \{y_i\}$ then $(n, \overline{y}, \overline{M}, \overline{t}, D')$ is a uncompletely triggered n-representation of (M, E') in which the variable y_i no more critical. □

16 Relating Call-by-Value to Call-by-Need

In this Section, we prove the estimation $\mathcal{C}^A(z{=}_n M) \leq \mathcal{C}^A(z{=}_v M)$ for all closed λ-expressions M as stated in Theorem 8.3. For proof, we will define a lengthening simulation for the embedding $z{=}_n M \mapsto z{=}_v M$ and apply Proposition 12.4.

The correspondence between an expression $z{=}_n M$ and an expression $z{=}_v M$ is very simple. We define a projection function p between ternary and binary δ-expressions, which eliminates all triggering information in expressions such as $z{=}_n M$:

$$\begin{array}{llll} p(x{:}ytz/E) \overset{\text{def}}{\equiv} x{:}yz/E & p(xytz) \overset{\text{def}}{\equiv} xyz & p(E \mid F) \overset{\text{def}}{\equiv} p(E) \mid p(F) \\ p((\nu x)E) \overset{\text{def}}{\equiv} (\nu x)p(E) & p(\mathbf{tr}(t)) \overset{\text{def}}{\equiv} \mathsf{0} & p(t.E) \overset{\text{def}}{\equiv} p(E) \end{array}$$

In this definition, we use a new expression $\mathsf{0}$ that we require to be nilpotent in the sense $\mathsf{0} \mid E \equiv E$ for all E. Being a little bit less restrictive we could also define $\mathsf{0}$ in δ itself, for example by $\mathsf{0} \overset{\text{def}}{\equiv} (\nu x)(xx)$.

Let $\approx_1$ be the smallest congruence on δ-expressions (with $\mathsf{0}$) containing the structural congruence and satisfying the axiom:

$$(\nu x)E \approx_1 E \qquad \text{if } x \notin \mathcal{V}(E)$$

Lemma 16.1 *For all closed M and variables z the relation* $p(z{=}_nM) \approx_1 z{=}_vM$ *holds.*

Proof. By induction on the structure of M. $\square$

Lemma 16.2 *Let R be one of the letters in $\{A, F, T\}$. If $E \rightarrow_R E'$ and $E \approx_1 F$ then there exists F' such that $F \rightarrow_R F'$ and $E' \approx_1 F'$.*

$$\begin{tikzcd}
E \arrow[r, phantom, "\rightarrow_A"] \arrow[d, phantom, sloped, "\approx_1"] & E' \arrow[d, phantom, sloped, "\approx_1"] & E \arrow[r, phantom, "\rightarrow_F"] \arrow[d, phantom, sloped, "\approx_1"] & E' \arrow[d, phantom, sloped, "\approx_1"] & E \arrow[r, phantom, "\rightarrow_T"] \arrow[d, phantom, sloped, "\approx_1"] & E' \arrow[d, phantom, sloped, "\approx_1"] \\
F \arrow[r, phantom, "\rightarrow_A"] & F' & F \arrow[r, phantom, "\rightarrow_F"] & F' & F \arrow[r, phantom, "\rightarrow_T"] & F'
\end{tikzcd}$$

Proof. By induction on derivations of $E \rightarrow_A E'$, $E \rightarrow_F E'$, and $E \rightarrow_T E'$ respectively. $\square$

Lemma 16.3 *Let E ternary, E' a δ-expression, and R one of the letters in $\{A, F\}$. If $E \rightarrow_R E'$ then $p(E) \rightarrow_R p(E')$. If $E \rightarrow_T E'$ then $p(E) \rightarrow_T p(E')$.*

$$\begin{tikzcd}
E \arrow[r, phantom, "\rightarrow_A"] \arrow[d, phantom, "p"] & E' \arrow[d, phantom, "p"] & E \arrow[r, phantom, "\rightarrow_F"] \arrow[d, phantom, "p"] & E' \arrow[d, phantom, "p"] & E \arrow[r, phantom, "\rightarrow_T"] \arrow[d, phantom, "p"] & E' \arrow[d, phantom, "p"] \\
p(E) \arrow[r, phantom, "\rightarrow_A"] & p(E') & p(E) \arrow[r, phantom, "\rightarrow_F"] & p(E') & p(E) \arrow[r, phantom, "\equiv"] & p(E')
\end{tikzcd}$$

Proof. By induction on derivation of $E \rightarrow_A E'$, $E \rightarrow_F E'$, and $E \rightarrow_T E'$ respectively. $\square$

Let S_v^n be a the binary relation on δ-expressions that contains all pairs (E, F) such that $F \approx_1 p(E)$ and E ternary and admissible.

Proposition 16.4 *The relation S_v^n is a lengthening simulation for the mapping $z{=}_nM \mapsto z{=}_vM$ considered as embedding from the restriction of δ' to admissible, closed, and ternary expressions into itself.*

Proof. Lemma 16.1 implies (*Sim*1) and the Lemmata 16.2 and 16.3 ensure (*Sim*4) . $\square$

Corollary 16.5 *The estimation $\mathcal{C}^A(z{=}_nM) \leq \mathcal{C}^A(z{=}_vM)$ is valid for all closed M and variable z.*

Proof. Immediate from Propositions 12.4 and 16.4. $\square$

17 Adequacy of the Embedding of δ into δ_0

We prove that the embedding $E \mapsto [\![E]\!]$ restricted to well-typed expressions preserves termination as stated in Theorem 11.1. Of course, we again apply the simulation technique.

It is however not possible to use a simulation immediately. One reason is that reference chains are shortened in different order when expressing $\to_F$ via $\to_A$. Forwarding $\to_F$ shortens reference chains from the right to the left. For instance:

$$\begin{array}{rcl} xu \mid x{=}y \mid y{:}z/E & \to_F & xu \mid x{:}z/E \mid y{:}z/E \\ & \to_A & E[u/z] \mid x{:}z/E \mid y{:}z/E \end{array}$$

After encoding, chains are traversed from the left to the right:

$$\begin{array}{rcl} [\![xu \mid x{=}y \mid y{:}z/E]\!] & \equiv & xu \mid x{:}z/yz \mid y{:}z/[\![E]\!] \\ & \to_A & yu \mid x{:}z/yz \mid y{:}z/[\![E]\!] \\ & \to_A & [\![E]\!][u/z] \mid x{:}z/yz \mid y{:}z/[\![E]\!] \end{array}$$

Note that $\to_F$ provides for path compression, which is not preserved by encoding. We formally handle the effect of path compression to complexity by an appropriate shortening simulation (compare Lemma 17.4).

Instead of simulating single forwarding steps, we will simulate sequences of forwarding steps followed by application. For all $n \geq 0$ we define the relation $\to_{F^nA}$ by the following axiom and the contextual rules in Figure 4:

$$x_1\overline{z} \mid x_1{=}x_2 \mid \ldots \mid x_{n-1}{=}x_n \mid x_n{:}\overline{y}/E \;\to_{F^nA}\; E[\overline{z}/\overline{y}] \mid x_1{:}\overline{y}/E \mid \ldots \mid x_n{:}\overline{y}/E$$

where we assume the sequence $(x_i)_{i=1}^n$ to be linear.

Lemma 17.1 *If $E \to_{F^nA} E'$, then $E \to_F^n \circ \to_A E'$.*

Proof. By induction on derivations of $E \to_{F^nA} E'$. The axiom case is by induction on n. □

Lemma 17.2 *If E is reducible with respect to $\to_F^n \circ \to_A$, then there exists $m \leq n$ such that E is reducible with respect to $\to_{F^mA}$.*

Proof. By induction on n. □

We define $\supseteq$ to be the smallest binary relation on δ-expressions, which is reflexive and transitive, satisfies the contextual rules of Figure 4, and the axiom:

$$(x_1{:}\overline{y}/x_2\overline{y}) \mid x_2{:}\overline{y}/E \;\supseteq\; (x_1{:}\overline{y}/E) \mid x_2{:}\overline{y}/E$$

Lemma 17.3 *If $E \rightarrow_{F^nA} E'$ then $\llbracket E \rrbracket \rightarrow_A^{n+1} \circ \supseteq \llbracket E' \rrbracket$.*

$$\begin{array}{ccc} E & \rightarrow_{F^nA} & E' \\ \llbracket - \rrbracket & & \llbracket - \rrbracket \\ \llbracket E \rrbracket & \rightarrow_A^{n+1} \quad \supseteq & \llbracket E' \rrbracket \end{array}$$

Proof. By induction on derivations of $E \rightarrow_{F^nA} E'$. We only consider the axiom case:

$$\begin{array}{rl} F_1 & \overset{\text{def}}{\equiv} \quad x_1\overline{z} \mid x_1{=}x_2 \mid \ldots \mid x_{n-1}{=}x_n \mid x_n{:}\overline{y}/E \\ \rightarrow_{F^nA} & E[\overline{z}/\overline{y}] \mid x_1{:}\overline{y}/E \mid \ldots \mid x_n{:}\overline{y}/E \;\overset{\text{def}}{\equiv}\; F_2 \end{array}$$

After translation, we obtain:

$$\begin{array}{rcl} \llbracket F_1 \rrbracket & \equiv & x_1\overline{z} \mid x_1{:}\overline{y}/x_2\overline{y} \mid \ldots \mid x_{n-1}{:}\overline{y}/x_n\overline{y} \mid x_n{:}\overline{y}/\llbracket E \rrbracket \\ & \rightarrow_A^{n+1} & \llbracket E \rrbracket[\overline{z}/\overline{y}] \mid x_1{:}\overline{y}/x_2\overline{y} \mid \ldots \mid x_{n-1}{:}\overline{y}/x_n\overline{y} \mid x_n{:}\overline{y}/\llbracket E \rrbracket \\ & \supseteq & \llbracket E \rrbracket[\overline{z}/\overline{y}] \mid x_1{:}\overline{y}/E \mid \ldots \mid x_{n-1}{:}\overline{y}/E \mid x_n{:}\overline{y}/\llbracket E \rrbracket \\ & \equiv & \llbracket F_2 \rrbracket \end{array}$$

□

Lemma 17.4 *The relation $\supseteq$ is a shortening simulation for the identity function on δ' restricted to admissible expressions.*

Proof.

(*Sim*1) The relation $\supseteq$ is required to be reflexive.

(*Sim*2) We show that $\supseteq$ preserves termination with respect to $\rightarrow_A$, $\rightarrow_F$, $\rightarrow_T$, which implies that it also preserves termination in δ'. It is sufficient to prove the previous statement for expressions E without top-level declarations. For $\rightarrow_A$, we note that the set of variables naming abstractions in E is invariant under $\supseteq$. The same holds for the set of applications in E. For $\rightarrow_F$, note that the set of directed equations in E is preserved under $\supseteq$. Triggering is completely unaffected by $\supseteq$.

(*Sim*3) We can establish the following diagrams. For all E, E', and F there exists E'' and F such that:

$$\begin{array}{ccccc} E & \rightarrow_A & E' & \rightarrow_A^* & E'' \\ \rotatebox{-90}{$\supseteq$} & & & & \rotatebox{-90}{$\supseteq$} \\ F & & \rightarrow_A & & F' \end{array} \qquad \begin{array}{ccc} E & \rightarrow_F & E' \\ \rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} \\ F & \rightarrow_F & F' \end{array} \qquad \begin{array}{ccc} E & \rightarrow_T & E' \\ \rotatebox{-90}{$\supseteq$} & & \rotatebox{-90}{$\supseteq$} \\ F & \rightarrow_T & F' \end{array}$$

The proofs are rather simple for expressions without top-level declarations and this is sufficient. □

Lemma 17.5 *The relation $\supseteq$ restricted to admissible expressions preserves termination and shortens complexity. If E and E' are admissible and $E \supseteq E'$, then $\mathcal{C}(E) \geq \mathcal{C}(E')$.*

Proof. This is an immediate consequence of Lemma 17.4 and Theorem 12.2. $\Box$

We next consider the encoding of triggering. We consider the following example:

$$\begin{array}{rcl} [\![t.E \mid \mathbf{tr}(t)]\!] & \equiv & (\nu y)(ty \mid y{:}/[\![E]\!]) \mid t{:}y/y \\ & \rightarrow_A & (\nu y)(y \mid y{:}/[\![E]\!]) \mid t{:}y/y \\ & \rightarrow_A & [\![E]\!] \mid t{:}y/y \mid (\nu y)(y{:}/E) \\ & \equiv & [\![E \mid \mathbf{tr}(t)]\!] \mid (\nu y)(y{:}/E) \end{array}$$

This illustrates that every triggering step is encoded by two application steps. The correspondence is quite direct up to garbage expressions such as $(\nu y)(y{:}/E)$. To keep track of these, we define the relation $\approx_2$ as the least congruence on δ-expressions, which is invariant under congruence and satisfies the following axiom:

$$E \mid (\nu x)(x{:}\overline{y}/E) \approx_2 E$$

Lemma 17.6 *If $E \rightarrow_T E'$ then $[\![E]\!] \rightarrow_A^2 \circ \approx_2 [\![E']\!]$.*

$$\begin{tikzcd} E \arrow[rr, phantom, "\rightarrow_T"] \arrow[d, phantom, "{[\![\,\text{-}\,]\!]}"] & & E' \arrow[d, phantom, "{[\![\,\text{-}\,]\!]}"] \\ {[\![E]\!]} \arrow[r, phantom, "\rightarrow_A^2"] & {} \arrow[r, phantom, "\approx_2"] & {[\![E']\!]} \end{tikzcd}$$

Lemma 17.7 *The relation $\approx_2$ is a complexity simulation for the identity function on δ.*

Proof. Omitted, but not difficult. $\Box$

Lemma 17.8 *The relation $\approx_2$ restricted to admissible expressions preserves complexity and termination.*

Proof. This is an immediate consequence of Lemma 17.7 and Corollary 12.5. $\Box$

In the last part of this Section, we combine the above results in order to prove the adequacy of the embedding $E \mapsto [\![E]\!]$ restricted to well-typed expressions.

Lemma 17.9 *If E is well-typed then $[\![E]\!]$ is admissible.*

Proof. We can introduce new typing rules that type $[\![E]\!]$ symmetrically to E. With respect to this new system $[\![E]\!]$ is well-typed. This implies the admissibility of $[\![E]\!]$ in the same manner than for the original type system. We note that $[\![\mathbf{tr}(t) \mid \mathbf{tr}(t)]\!]$ is not inconsistent by definition. In other words, multiple triggering does not lead to an inconsistency. $\square$

Lemma 17.10 *If E is well-typed, acyclic, and irreducible with respect to $\rightarrow_T$ and $\rightarrow_F^* \circ \rightarrow_A$, then $[\![E]\!]$ terminates with respect to $\rightarrow_A$.*

Proof. Since E is irreducible with respect to $\rightarrow_T$, applications of a variable t of type $\mathbf{tr}^{\eta}$ can not be executed in $[\![E]\!]$. Otherwise, there would exist any application of t in $[\![E]\!]$ which in not derived form $[\![\mathbf{tr}(t)]\!]$. This would contradict well-typedness of E.

An applications $[\![x\overline{y}]\!]$ can be executed in $[\![E]\!]$, if a translated equation $[\![x{=}y]\!]$ is available in $[\![E]\!]$. This can happen finitely many times, since E is acyclic. Applying an abstraction not derived from a directed equation is never possible, since E is irreducible with respect to $\rightarrow_F^* \circ \rightarrow_A$. $\square$

Proposition 17.11 *If E is well-typed and acyclic, then E terminates if and only if $[\![E]\!]$ terminates.*

Proof. First, we consider the case that E terminates in δ and proof that $[\![E]\!]$ terminates in δ_0. This proof is by induction on $\mathcal{C}^A(E) < \infty$.

If $\mathcal{C}^A(E) = 0$, then E is irreducible with respect to $\hookrightarrow \circ \rightarrow_A \circ \hookrightarrow$. Applying Lemmata 17.6 and 17.8, we can assume that E is irreducible with respect to $\rightarrow_T$. This implies that E is irreducible with respect to $\rightarrow_F^* \circ \rightarrow_A$. Well-typedness of E and Lemma 17.10 yields termination of $[\![E]\!]$ in δ_0.

Let $\mathcal{C}^A(E) > 0$. Applying the Lemmata 17.6 and 17.8, we can assume that E is irreducible with respect to $\rightarrow_T$. This implies that E is reducible with respect to $\rightarrow_F^* \circ \rightarrow_A$. Applying Lemma 17.2, there exists $n \geq 0$ and E' such that $E \rightarrow_{F^nA} E'$. Lemma 17.1 implies $\mathcal{C}^A(E) = \mathcal{C}^A(E') + 1$ such that $\mathcal{C}^A(E') < \mathcal{C}^A(E)$. Applying the induction hypothesis to E' yields termination of $[\![E]\!]$. From Lemma 17.3, we obtain $[\![E]\!] \rightarrow_A^* \circ \supseteq [\![E']\!]$. Termination of $[\![E']\!]$ implies termination of $[\![E]\!]$ by Lemma 17.5.

It remains show that if E does not terminate then $[\![E]\!]$ does not terminate. This can be done with a similar inductive argument, which proves that $\mathcal{C}^A(E) \geq n$ implies $\mathcal{C}([\![E]\!]) \geq n$ for all $n \geq 0$. $\square$

Corollary 17.12 *If E is well-typed, then $[\![E]\!]$ is admissible and terminates if and only if E terminates.*

Proof. Immediate from Lemmata 17.11 and 17.9. $\square$

18 Simulating the Call-by-Need λ-Calculus

In this Section, we sketch the proof that our embedding of the call-by-need λ-calculus into δ preserves complexity as stated in Theorem 9.2.

Syntactically, the call-by-need λ-calculus and the δ-calculus differ in flattening λ-terms. We define a flattening functions f mapping an expression L of the call-by-need λ-calculus to an expression of the form $\mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ N$ with explicit substitutions:

$$f(x) \;=\; \mathsf{let}\ z{=}x\ \mathsf{in}\ z \qquad f(\lambda x.L) \;=\; \mathsf{let}\ z{=}\lambda x.L\ \mathsf{in}\ z$$

$$\frac{f(L_1) \;=\; \mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_n \qquad f(L_2) \;=\; \mathsf{let}\ \overline{y'}{=}\overline{M'}\ \mathsf{in}\ y'_n}{f(L_1 L_2) \;=\; \mathsf{let}\ \overline{y}{=}\overline{M}\ \ \overline{y'}{=}\overline{M'}\ \ z{=}y_n y'_n\ \ \mathsf{in}\ z}$$

$$\frac{f(L_1) \;=\; \mathsf{let}\ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_n \qquad f(L_2) \;=\; \mathsf{let}\ \overline{y'}{=}\overline{M'}\ \mathsf{in}\ y'_n}{f(\mathsf{let}\ x{=}L_1\ \mathsf{in}\ L_2) \;=\; \mathsf{let}\ \overline{y}{=}\overline{M}\ \ \overline{y'}{=}\overline{M'}\ \ \mathsf{in}\ y'_n}$$

Definition 18.1 *We define the relation S_n^λ as the set of all pairs $(L,\, E)$ such that there exists a pair $(M,\, F)$ and a n-representation $(n,\, \overline{y},\, \overline{M},\, \overline{t},\, D)$ for $(M,\, F)$ such that $f(L) \equiv M$, $E \approx_2 F$, and $f(L) \;=\; \mathsf{let}\ \ \overline{y}{=}\overline{M}\ \mathsf{in}\ y_n$.*

Proposition 18.2 *The relation S_n^λ is a complexity simulation for the embedding $L \mapsto z{=}_n L$ from the call-by-need λ-calculus restricted to closed expressions into δ'.*

Proof. The conditions of a complexity simulation will be checked by the following Lemmata. Property (*Sim*1) is implied by Lemma 18.4, (*Sim*2) by Lemma 18.5, and (*Sim*3) and (*Sim*4) by Lemmata 18.6, 18.7, and 18.8. □

Corollary 18.3 *For all closed L the equality $\mathcal{C}_{\mathrm{need}}(L) \;=\; \mathcal{C}^A(z{=}_n L)$ is valid.*

Proof. Immediate consequence of Proposition 18.2 and Corollary 12.5. □

Lemma 18.4 *If L is closed, then $(L,\, z{=}_n L) \in \mathrm{S}_n^\lambda$.*

Proof. By induction on the structure of L. □

Lemma 18.5 *If L is irreducible in the call-by-need λ-calculus and $(L,\, E) \in \mathrm{S}_n^\lambda$, then E is irreducible in δ'.*

Proof. If L is irreducible in δ and $(M,\, F)$ justifies $(L,\, E) \in \mathrm{S}_n^\lambda$. Since $M \equiv f(L)$, M is an abstraction and hence irreducible with respect to $\rightarrow_{\mathrm{name}}$. Since S_n is a shortening simulation (Proposition 15.8) F is irreducible in δ'. Since $\approx_2$ is a complexity simulation (Lemma 17.7), E is also irreducible in δ'. □

Lemma 18.6 *If $L \rightarrow_I L'$ and $(L, E) \in \mathrm{S}^{\lambda}_{n}$, then there exists E' such that $E \rightarrow_A \circ \approx_2 \circ \rightarrow^{*}_{T} E'$ and $(L', E') \in \mathrm{S}^{\lambda}_{n}$.*

$$\begin{array}{ccccc} L & & \rightarrow_I & & L' \\ \mathrm{S}^{\lambda}_{n} & & & & \mathrm{S}^{\lambda}_{n} \\ E & \rightarrow_A & \approx_2 & \rightarrow^{*}_{T} & E' \end{array}$$

Proof. By induction on derivations of $L \rightarrow_I L'$. □

Lemma 18.7 *If $L \rightarrow_V L'$ and $(L, E) \in \mathrm{S}^{\lambda}_{n}$, then there exists E' such that $E \rightarrow_F \circ \rightarrow^{*}_{T} E'$ and $(L, E') \in \mathrm{S}^{\lambda}_{n}$.*

$$\begin{array}{cccc} L & \multicolumn{2}{c}{\rightarrow_V} & L' \\ \mathrm{S}^{\lambda}_{n} & & & \mathrm{S}^{\lambda}_{n} \\ E & \rightarrow_F & \rightarrow^{*}_{T} & E' \end{array}$$

Proof. By induction on derivations of $L \rightarrow_V L'$ □

Lemma 18.8 *If $L \rightarrow_{Ans} L'$ or $L \rightarrow_C L'$, then $f(L) \equiv f(L')$.*

$$\begin{array}{ccc} L & \rightarrow_{Ans} & L' \\ f & & f \\ f(L) & \equiv & f(L') \end{array} \qquad\qquad \begin{array}{ccc} L & \rightarrow_{C} & L' \\ f & & f \\ f(L) & \equiv & f(L') \end{array}$$

Proof. By induction on derivations of $L \rightarrow_V L'$ and $L \rightarrow_C L'$ respectively. □

19 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 simulations. We have formally related the complexities of call-by-value, call-by-need, and call-by-name.

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 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 *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 233–246. The ACM Press, 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/North Holland, Amsterdam - New York - Oxford, 1984.

[BNA91] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In John Hughes, editor, *Functional Programming Languages and Computer Architecture - 5th ACM Conference*, number 523 in Lecture Notes in Computer Science, pages 538–568. Springer-Verlag, August 1991.

[BO95] Simon Brock 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 Lecture Notes in Computer Science, pages 149–161. Springer-Verlag, October 1989.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA, Sophia Antipolis, France, May 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, Cambridge, Massachusetts, 1995. A previous version is published as [**?**].

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

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

[Iba95] Kai Ibach. *OzFun: Eine funktionale Spache für gemischte Eager- und Lazy-Programmierung.* Diploma Thesis, Universität des Saarlandes, Fachbereich Informatik, Stuhlsatzenhausweg, 66041 Saarbrücken, Germany., 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 Vijay Saraswat and Kazunori Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 167–186, San Diego, California, October 1991.

[Klo87] Jan Willem Klop. Term rewriting systems: A tutorial. *EATACS*, 32:143–182, 1987. Bull. European Ass. Theoretical Computer Sience.

[KPT96] Naoki Kobayashi, Benjamin Pierce, and David N. Turner. Linearity and the pi-calculus. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. The ACM Press, January 1996.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 144–154. The ACM Press, 1993.

[Mah87] Michael J. Maher. Logic semantics for a class of committed-choice programs. In Jean-Louis Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, pages 858–876. The MIT Press, 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, Edinburgh EH9 3JZ, October 1991.

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

[MOTW95] John Maraist, Martin Odersky, David Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. In *11'th International Conference on the Mathematical Foundations of Programming Semantics*, New Orleans, Lousiana, April 1995.

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

[Mül96] Martin Müller. Polymorphic types for concurrent constraints. submitted, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, {mmueller}@dfki.uni-sb.de, 1996.

[Nie94] Joachim Niehren. *Funktionale Berechnung in einem uniform nebenläufigen Kalkül mit logischen Variablen.* Doctoral Dissertation. Universität des Saarlandes, Technische Fakultät, 66041 Saarbrücken, Germany, December 1994.

[Nie96] Joachim Niehren. Functional computation as concurrent computation. In *Proceedings of the ACM Symposium on Principles of Programming Languages.* The ACM Press, January 1996.

[NM95] Joachim Niehren and Martin Müller. Constraints for Free in Concurrent Computation. In *Asian Computing Science Conference*, Lecture Notes in Computer Science, pages 171-186, Springer-Verlag. Pathumthani, Thailand, December 1995.

[NS94] Joachim Niehren and Gert Smolka. A confluent relational calculus for higher-order programming with constraints. In Jean-Pierre Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 89–104, München, Germany, September 1994.

[Pin87] Keshav K. Pingali. Lazy Evaluation and the Logic Variable. Technical report, Cornell University, October 1987. Proceedings of the Institute on Declarative Programming. Austin, Texas. August 24-29, 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 (ESOP)*, volume 582 of *Lecture Notes in Computer Science.* Springer-Verlag, 1992.

[PT95a] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan (Nov. 1994)*, number 907 in Lecture Notes in Computer Science, pages 187–215. Springer-Verlag, 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, 1995.

[San95] D. Sands. A Naïve Time Analysis and its Theory of Cost Equivalence. *The Journal of Logic and Computation*, page 48 pages, 1995+. Accepted, to appear (Preliminary version available as TOPPS report D-173, 1993, Copenhagen).

[Smo94] Gert Smolka. A foundation for concurrent constraint programming. In *Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 50–72. Springer-Verlag, September 1994.

[Smo95a] Gert Smolka. An Oz primer. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1995.

[Smo95b] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Current Trends in Computer Science*, Lecture Notes in Computer Science, vol. 1000. Springer-Verlag, 1995. To appear.

[SRP91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 333–352. The ACM Press, 1991.

[SSW94] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In A.H. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 134–150. Springer-Verlag, 2-4 May 1994.

[Vas94] Vasco T. Vasconcelos. Typed concurrent objects. In *8th Proceedings of the European Conference on Object Oriented Programming*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, July 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 *ACM Conference on Functional Programming Languages and Computer Architecture*, 1993.