Monadic Type Systems: Pure Type Systems for Impure Settings (Preliminary Report)

Gilles Barthe [1]

Chalmers Tekniska Högskola, Institutionen för Datavetenskap, Eklandagatan 86, Göteborg, S-412 96 Sweden, gillesb@cs.chalmers.se

John Hatcliff [2]

Oklahoma State University, Department of Computer Science, 219 Math Sciences, Stillwater, OK, USA, 74078, hatcliff@a.cs.okstate.edu

Peter Thiemann

Dept. of Computer Science, University of Nottingham, University Park, Nottingham NG7 2RD, England, pjt@cs.nott.ac.uk

Abstract

Pure type systems and *computational monads* are two parameterized frameworks that have proved to be quite useful in both theoretical and practical applications. We join the foundational concepts of both of these to obtain *monadic type systems*. Essentially, monadic type systems inherit the parameterized higher-order type structure of pure type systems and the monadic term and type structure used to capture computational effects in the theory of computational monads. We demonstrate that monadic type systems nicely characterize previous work and suggest how they can support several new theoretical and practical applications.

A technical foundation for monadic type systems is laid by recasting and scaling up the main results from pure type systems (confluence, subject reduction, strong normalisation for particular classes of systems, *etc.*) and from operational presentations of computational monads (notions of operational equivalence based on applicative similarity, co-induction proof techniques).

We demonstrate the use of monadic type systems with case studies of several call-by-value and call-by-name systems. Our framework allows to capture the restriction to value polymorphism in the type structure and is flexible enough to accommodate extensions of the type system, e.g., with higher-order polymorphism. The theoretical foundations make monadic type systems well-suited as a typed intermediate language for compilation and specialization of higher-order,

Preprint submitted to Elsevier Preprint *18 February 1998*

strict and non-strict functional programs. The monadic structure guarantees sound compile-time optimizations and the parameterized type structure guarantees sufficient expressiveness.

Key words: Functional programming, polymorphism, pure type systems, monads, operational semantics.

---

[1] Research supported by TMR Fellowship FMBICT972065.
[2] Research supported by NSF under grant CCR-9701418.

Contents

1 Introduction

The compilation of typed higher-order programming languages requires the use of intermediate languages. This is due to the large conceptual gap between source and target languages in such a translation. In addition, the use of an intermediate language yields a synergistic effect since common backends and program transformations may be shared among several source languages. Therefore, a large amount of flexibility in terms of operational properties as well as typing properties is required from an intermediate language.

Recent work in compilation has produced a number of compile-time frameworks that either stress the operational properties or the typing properties. But none of the frameworks considered so far offers parametricity in both respects. Our aim is to reconcile these hitherto separate concerns by joining in a single framework the fundamental concepts of pure type systems and those of computational monads.

Computational monads:

Moggi's theory of computational monads as embodied by his *computational metalanguage* parameterizes semantic specifications by a *notion of computation* (state transformation, I/O, *etc.*) [36]. Terms in the metalanguage can be reduced without regard to computational effects — even without specifying their nature — provided that they can be expressed with a monad. In addition, the metalanguage explicitly orders all operations with computational effects. A wide range of evaluation orders (call-by-name, call-by-value, and mixed strategies) can be encoded by choosing a suitable translation into the metalanguage. Thus, the metalanguage provides an *evaluation-order independent* framework for describing evaluation and reasoning in the presence of computational effects.

The computational metalanguage has been used successfully in a number of applications including compiling transformations, intermediate languages for partial evaluation, denotational foundations for languages with I/O, and structuring functional programs with effects (*e.g.*, [14, 16, 21, 27, 28, 32, 40, 50]). The benefits of the metalanguage are as follows.

Increased modularity: Compilation, partial evaluation, and semantics can be specified in terms of the metalanguage: all computational aspects of the source language are captured in the metalanguage; no extra information is necessary.

Correctness: The framework guarantees the soundness of compile-time reductions in the presence of computational effects.

Genericity: The metalanguage is general enough to encode the structure of a variety of source languages by choosing a suitable translation. The resulting translated programs are in *monadic normal form* [28] — a generalization of A-normal forms [16], where each intermediate result is named. This form is particularly suited to compiling.

However, most applications deal with untyped or simply-typed versions of the metalanguage and do not support parametricity of the type structure.

Pure type systems:

Pure type systems (PTS) [4] are a generic framework to define type systems and logics. Examples are F_2, F_ω, LF, and the Calculus of Constructions. They provide the theoretical foundation for proof assistants and modern programming language type systems. Subtle differences between the systems can be described by varying the *specification* upon which the framework is parameterized.

Compactness and parametricity make PTSs an attractive framework for typed intermediate languages [31]. This builds on a well-established practice of using subsystems of F_ω in compiling polymorphic programming languages like ML or Haskell [23,26,35,39,46,47]. It is by now an accepted fact that type information is beneficial in all phases of compilation and program transformation. PTSs are exceptionally well suited because of the following features.

Economy: Since terms and types are conflated in one syntactic category, the same functions can be used for their manipulation.

Extensibility: The intermediate language is robust with respect to changes to the type system of the source language: extensions to the type system usually amount to changing only the specification of the PTS.

Checkability Being explicitely typed, many PTSs have a decidable type-checking. Therefore all compile-time transformations can be type checked and thus debugged.

In many of these applications, PTSs (whose semantics are *free from computational effects*) are being used in settings where computational effects are ubiquitous. Whereas the PTS framework provides adequate treatment of the type structure, it does not ensure sound treatment of effects. This stems from the fact that β-conversion is the foundational notion of equality in PTSs. As a consequence, a PTS-based intermediate language (such as one might use for treating ML) must impose ad hoc restrictions to ensure soundness. Furthermore, it seems difficult to use a PTS-based language for languages that explicitly control the evaluation order. This situation arises in a Haskell compiler after strictness and/or totality analysis.

Monadic type systems:

Monadic type systems (MTSs) join the concepts of the computational metalanguage and of pure type systems. They inherit the parameterized higher-order type structure of pure type systems and the monadic term and type structure used to capture computational effects in the theory of computational monads. The combination shares the above-listed advantages of both frameworks—except for type-checking which is undecidable for languages that include dependent types—and provides some additional features. For example, an MTS specification can enforce the restriction to value polymorphism [34] by type checking the intermediate language. Furthermore, it is possible to encode other approaches to the sound treatment of effects in the presence of polymorphism (e.g., polymorphism by name [33]).

1.1 Contributions

This paper takes a first step towards combining the fundamental concepts of pure type systems and computational monads and towards developing operational theories for λ-calculi with dependent types.

- MTSs combine the fundamental concepts of pure type systems and computational monads, thus providing a typed intermediate framework that offers parametricity with respect to type disciplines and computational effects. [3]
- The main technical properties of PTSs (confluence, subject reduction, strong normalisation for particular classes of systems, *etc.*) are recast and proved for MTSs.
- The operational properties of the computational metalanguage (notions of operational equivalence based on applicative similarity, co-induction proof techniques) are recast and proved for MTSs where the notion of computation is *non-termination* as expressed by the *lifting monad.*
- A detailed case study (SML'97 with value polymorphism) demonstrates a specific instance of the MTS framework which captures the operational and typing properties that are desirable in a typed intermediate representation for SML'97. This particular language can serve as a basis for compiling ML and also for specialization of ML. It is noteworthy that specialization with respect to values and specialization with respect to types is indistinguishable in the language, which leads to some simplification inside a compiler/specializer. On the other hand, the monadic structure is indispensable to ensure the soundness of transformations inside a compiler/specializer.

1.2 Overview

The rest of the paper is organized as follows. Section 2 presents the basic definitions of monadic type systems. Section 3 provides intuition and illustrates how the MTS framework can be used to describe many systems and concepts appearing in the literature. Section 4 reviews technical properties of monadic type systems. Section 5 is devoted to the operational semantics of monadic type systems.

Section 6 assesses the applicability of MTS and compares them with related work. Section 7 gives conclusions and future work.

Throughout the paper, we assume some basic knowledge of computational monads and pure type systems.

2 Syntax of Monadic Type Systems

In PTSs, different typed λ-calculi are generated by varying PTS parameters called *specifications* that consist of *sorts*, *axioms* and *rules.* Sorts provide the type universes of the calculus. Axioms provide a "membership" relation between universes, while rules determine which function types may be formed and in which universe they live.

[3] Both pure type systems and the metalanguage have achieved a status of 'standards'. We do not claim that MTSs inherit this status and expect alternative or refined syntaxes to arise.

Logical Pure Type Systems [10] form a class of specifications that feature a distinguished sort $*$ of propositions or programs, that is required to comply with several requirements. Examples of Logical Pure Type Systems include systems in Barendregt's λ-cube [3, 4] which are obtained from specifications that use the sorts $*$ and $\Box$, with the axiom $* : \Box$. Intuitively, inhabitants of $*$ are types and inhabitants of types are programs, whereas inhabitants of $\Box$ are kinds and inhabitants of kinds are type constructors.

Monadic Type Systems are based on the same idea. The difference is that MTSs *also* include a fundamental concept from the theory of computational monads: they distinguish between types for *values* (completely evaluated terms, or terms with no computational effects) and types for *computations* (terms with remaining effectful computational steps). Technically, the distinction is achieved by distinguishing a sort of values and a sort of computations. As a result of our choice, the framework does not encompass every meaningful monadic type system—in the same way as the class of logical specifications does not encompass every meaningful pure type system—yet it allows for many interesting systems to be defined.

Definition 1 (specifications) A (MTS) *specification* is a tuple $\mathbf{S} = (\mathcal{S}, *^v, *^c, \mathcal{A}, \mathcal{R})$ where

- $\mathcal{S}$ is a set of *sorts*;
- $*^v \in \mathcal{S}$ is the sort of *values* and $*^c \in \mathcal{S}$ is the sort of *computations*;
- $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of *axioms*;
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a set of *product rules*.

The above Definition does not impose any requirement of the sorts, axioms or rules and thus allows for specifications that violate the monadic characteristics. These requirements (which yield what we call "strictly monadic specifications") are postponed until Section 4 and Section 5.

Monadic type systems have a single category of expressions, which are called pseudo-terms. Pseudo-terms are built from the standard constructors for pure type systems—including abstraction, application and dependent product—and for monads—including unit and let-expressions. In addition, pseudo-terms include a data type of natural numbers with some basic operations and a fixpoint construct that give a "PCF-like" presentation of monadic type systems. A more general presentation, which falls out of the scope of this paper, would be to include a scheme for inductive types, e.g. based on recent work by T. Coquand [9].

Definition 2 (pseudo-terms) The set $\mathcal{T}$ of *pseudo-terms* is defined by the abstract syntax

$$\begin{aligned}\mathcal{T} = {} & \mathcal{V} \mid \mathcal{S} \mid \mathcal{T}\ \mathcal{T} \mid \lambda\mathcal{V}{:}\mathcal{T}.\ \mathcal{T} \mid \Pi\mathcal{V}{:}\mathcal{T}.\ \mathcal{T} \mid \mathsf{let}\ \mathcal{V}{:}\mathcal{T} \Leftarrow \mathcal{T}\ \mathsf{in}\ \mathcal{T} \mid \mathsf{unit}\ \mathcal{T} \mid \mathsf{M}\ \mathcal{T} \mid \\ & \mathsf{fix}\ \mathcal{V}{:}\mathcal{T}.\ \mathcal{T} \mid \mathsf{nat} \mid \lceil n \rceil \mid \mathsf{succ}\ \mathcal{T} \mid \mathsf{pred}\ \mathcal{T} \mid \mathsf{if0}\ \mathcal{T}\ \mathcal{T}\ \mathcal{T}\end{aligned}$$

For technical convenience, we assume:

(i) variables to be *sorted*, i.e. $\mathcal{V} = \bigcup_{s \in \mathcal{S}} \mathcal{V}^s$ where the $\mathcal{V}^s$s are pairwise disjoint countably infinite sets;

(ii) Barendregt's variable conventions [2].

Substitution, free and bound variables, *etc.* are defined as usual. The notions of reduction for monadic type systems are drawn from those of pure type systems and those of the computational metalanguage.

Definition 3 (notions of reduction)

- β-reduction $\rightarrow_\beta$ is defined as the compatible closure of the rule

$$(\lambda x\colon A.\ M)\ N \rightarrow_\beta M\{x := N\}$$

- μ-reduction $\rightarrow_\mu$ is defined as the compatible closure of the rules

$$\begin{array}{c}\mathsf{let}\ x\colon A \Leftarrow (\mathsf{unit}\ N)\ \mathsf{in}\ M \rightarrow_{\mu_1} M\{x := N\}\\ \mathsf{let}\ x\colon A \Leftarrow M\ \mathsf{in}\ (\mathsf{unit}\ x) \rightarrow_{\mu_2} M\\ \mathsf{let}\ x_2\colon A_2 \Leftarrow (\mathsf{let}\ x_1\colon A_1 \Leftarrow M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3 \rightarrow_{\mu_3} \mathsf{let}\ x_1\colon A_1 \Leftarrow M_1\ \mathsf{in}\ (\mathsf{let}\ x_2\colon A_2 \Leftarrow M_2\ \mathsf{in}\ M_3)\end{array}$$

  where in the last rule it is assumed that $x_2 \notin \mathrm{FV}(A_1) \cup \mathrm{FV}(M_1)$.

- ϕ-reduction $\rightarrow_\phi$ is defined as the compatible closure of the rule

$$\mathsf{fix}\ x\colon M.\ N \rightarrow_\phi N\{x := \mathsf{fix}\ x\colon M.\ N\}$$

- ι-reduction is defined as the compatible closure of the rule

$$\begin{aligned} \mathsf{if0}\ \lceil 0 \rceil\ M\ N &\rightarrow_\iota M\\ \mathsf{if0}\ \lceil n+1 \rceil\ M\ N &\rightarrow_\iota N\\ \mathsf{succ}\ \lceil n \rceil &\rightarrow_\iota \lceil n+1 \rceil\\ \mathsf{pred}\ \lceil n+1 \rceil &\rightarrow_\iota \lceil n \rceil\\ \mathsf{pred}\ \lceil 0 \rceil &\rightarrow_\iota \lceil 0 \rceil\\ \mathsf{pred}\ (\mathsf{succ}\ x) &\rightarrow_\iota x \end{aligned}$$

- The notion of *ml*-reduction $\rightarrow_{ml}$ is defined as the union of β, ι, ϕ and μ-reductions.

We do not consider η-reduction since it is unsound for CPS translations and would complicate the meta-theory. The typing rules for MTSs are drawn from the rules of pure type systems and computational monads. In addition, there are rules for data types and fixpoints.

Definition 4 (derivability)

- A *pseudo-context* is a finite (ordered) list of pairs $x : A$ where $x \in \mathcal{V}$ and $A \in \mathcal{T}$. The set of pseudo-contexts is denoted by $\mathcal{C}$ and the empty context is denoted by $\langle\rangle$. The domain of a context Γ is

$$\mathrm{dom}(\Gamma) = \{x \mid \exists t \in \mathcal{T}.\ x : t \in \Gamma\}$$

  We use Γ, Δ as metavariables ranging $\mathcal{C}$.

Structural rules

$$\text{(axiom)} \quad \frac{(s_1, s_2) \in \mathcal{A}}{\langle\rangle \vdash s_1 : s_2}$$

$$\text{(conversion)} \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_{ml} B'}{\Gamma \vdash A : B'}$$

$$\text{(start)} \quad \frac{\Gamma \vdash A : s \quad x \in \mathcal{V}^s \setminus \mathrm{dom}(\Gamma)}{\Gamma, x : A \vdash x : A}$$

$$\text{(weakening)} \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s \quad x \in \mathcal{V}^s \setminus \mathrm{dom}(\Gamma)}{\Gamma, x : C \vdash A : B}$$

Product rules

$$\text{(product)} \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash (\Pi x{:}A.\, B) : s_3}$$

$$\text{(abstraction)} \quad \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x{:}A.\, B) : s}{\Gamma \vdash \lambda x{:}A.\, b : \Pi x{:}A.\, B}$$

$$\text{(application)} \quad \frac{\Gamma \vdash F : (\Pi x{:}A.\, B) \quad \Gamma \vdash a : A}{\Gamma \vdash F\, a : B\{x := a\}}$$

Fig. 1. Rules for Monadic Type Systems (structural and product rules)

- A *judgement* is a triple $\Gamma \vdash M : A$ where $\Gamma \in \mathcal{C}$ and $M, A \in \mathcal{T}$.
- The derivability relation $\vdash$ is induced by the rules of Figures 1 and 2. If $\Gamma \vdash M : A$ then Γ, M, A are *legal*.
- The MTS generated by $\mathbf{S}$ is the quadruple $\lambda\mathbf{S} = (\mathcal{T}, \mathcal{C}, \rightarrow_{ml}, \vdash)$.

The next Section illustrates the use of the framework through several instances of monadic type systems.

3 Instances of Monadic Type Systems

Each specification defines a particular instance of a monadic type system. In this section, we explore specifications that characterize the images of various translations that encode evaluation strategies in the computational metalanguage. The crucial point, in each case, is the distinction between values and computations. We illustrate this for some well-studied simply-typed languages, then—in more detail—for a fragment of ML, and finally for call-by-name and call-by-value versions of F_ω. The latter one is particularly interesting because it is quite close to the intermediate language used in the FLINT/ML compiler project [45].

All systems considered in this section are related to the λ-cube. They share the set of sorts and the axioms:

- $\mathcal{S} = \{*^v, *^c, \Box^v, \Box^c\}$;
- $\mathcal{A} = \{*^v : \Box^v, *^c : \Box^c\}$.

The intuition here is that the elements of $*^v$ are types of values, which are inhabited by effect-free terms, whereas the elements of $*^c$ are types of computations, which are inhabited by terms that may have computational effects: nontermination, in the most simple case. This distinction is used to enforce the value restriction of SML'97. Also certain operational properties of a term can be proved just from determining whether its type has kind $*^v$ or $*^c$. Similarly, the elements of $\Box^v$ are value kinds, which are inhabited by value type constructors, and likewise for $\Box^c$.

The rule sets of the specifications considered in this section do not involve $\Box^c$ at all. Abstractions involving $\Box^v$ are found to be sufficient, even to characterize the images of call-by-name translations, where we expect to perform abstraction over computations. Since we do not have a proof that $\Box^c$ and $\Box^v$ can be conflated to $\Box$, in general, we keep the more precise distinction for the time being.

3.1 Simply-typed systems

Many systems of interest only have product rules of the form $(s_1, s_2, *^v)$, which we write as $[s_1, s_2]$. They reflect the expected monadic typing of abstractions as values. Since simply-typed systems do not allow abstraction of types, the rules do not contain $\Box^v$ and $\Box^c$.

Monad rules

$$\text{(monad)}\quad \frac{\Gamma \vdash A : *^x}{\Gamma \vdash \mathsf{M}\ A : *^c}$$

$$\text{(unit)}\quad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash \mathsf{M}\ A : s}{\Gamma \vdash \mathsf{unit}\ M : \mathsf{M}\ A}$$

$$\text{(let)}\quad \frac{\Gamma \vdash M : \mathsf{M}\ A \qquad \Gamma, x : A \vdash N : \mathsf{M}\ B}{\Gamma \vdash \mathsf{let}\ x{:}A \Leftarrow M\ \mathsf{in}\ N : \mathsf{M}\ B} \quad x \notin \mathrm{FV}(B)$$

Rules for natural numbers

$$\text{(nat)}\quad \vdash \mathsf{nat} : *^v$$

$$\text{(num)}\quad \vdash \lceil n \rceil : \mathsf{nat}$$

$$\text{(succ)}\quad \frac{\Gamma \vdash M : \mathsf{nat}}{\Gamma \vdash \mathsf{succ}\ M : \mathsf{nat}}$$

$$\text{(pred)}\quad \frac{\Gamma \vdash M : \mathsf{nat}}{\Gamma \vdash \mathsf{pred}\ M : \mathsf{nat}}$$

$$\text{(test)}\quad \frac{\Gamma \vdash M : \mathsf{nat} \qquad \Gamma \vdash N_1 : \mathsf{M}\ B \qquad \Gamma \vdash N_2 : \mathsf{M}\ B}{\Gamma \vdash \mathsf{if0}\ M\ N_1\ N_2 : \mathsf{M}\ B}$$

Fixpoint rule

$$\text{(fixpoint)}\quad \frac{\Gamma, x : \mathsf{M}\ A \vdash M : \mathsf{M}\ A}{\Gamma \vdash \mathsf{fix}\ x{:}\mathsf{M}\ A.\, M : \mathsf{M}\ A}$$

Fig. 2. Rules for Monadic Type Systems (monads, naturals and fixpoint rules)

Call-by-name:

Translations that encode call-by-name evaluation in the metalanguage yield functions that map computations to computations [21, 28, 50]. The intuition is that arguments are suspended computations. Each variable access initiates evaluation of the suspended computation. There is only one rule:

$$\mathcal{R}_{cbn} = \{[*^c, *^c]\}.$$

Call-by-value:

In contrast to call-by-name, call-by-value functions take values as arguments. This is enforced by allowing only abstractions with value types in the domain position [28, 50]:

$$\mathcal{R}_{cbv} = \{[*^v, *^c]\}.$$

It is also possible to encode call-by-value in the system generated by the call-by-name specification above. The idea is to force the evaluation of the argument first, then pass the resulting value as a thunk (which corresponds to a computation, see [21, 28]).

Call-by-name and call-by-value:

This language provides a generic intermediate language for encoding both call-by-name and call-by-value programs [28] and also for strictness analyzed call-by-name programs [7, 12]. The rule set consists of the rules from both preceding cases:

$$\mathcal{R}_{mixed} = \{[*^c, *^c], [*^v, *^c]\}.$$

Full value types:

Some applications [13, 21] need to distinguish functions that are *trivial* [42] in the sense that their application causes no computational effects. With the lifting monad, such functions would be guaranteed to be *total* given any appropriately typed argument. The following specification allows for such functions by letting value types occur in the codomain position.

$$\mathcal{R}_{full} = \{[*^v, *^v], [*^c, *^v], [*^c, *^c], [*^v, *^c]\}.$$

3.2 Polymorphic systems (ML-style)

In this section, we investigate the translation of a fragment of ML into a suitable MTS. We then prove that the translation preserves typing and that ML evaluation is compatible with MTS-equality ($=_{ml}$, the reflexive, transitive, symmetric closure of $\rightarrow_{ml}$).

The starting point for a specification of an ML-style language is the specification $\mathcal{R}_{cbv}$. For concreteness, we will concentrate on a fragment of Standard ML'97 [34]. SML is a call-by-value language with impure features like references and I/O. In the fragment that we consider initially, the only effect is nontermination. Hence the monad must include the lifting monad. Later on, we add stores and require (at least) a combination of the state monad with the lifting monad. The full SML language also includes exceptions, I/O, and

types	τ	$::= \alpha \mid \mathsf{nat} \mid \tau \to \tau$
type schemes	σ	$::= \tau \mid \forall\alpha.\sigma$
terms	e	$::= v \mid n$
values	v	$::= x \mid f \mid i \mid \lambda x.e \mid \mathsf{fix}\ f.e$
non-values	n	$::= e@e \mid \mathsf{let}\ x = v\ \mathsf{in}\ e \mid \mathsf{if0}\ e\ e\ e \mid \mathsf{succ}\ e \mid \mathsf{pred}\ e$

Fig. 3. A Fragment of SML'97

continuations (in the New Jersey dialect of ML). These features are not considered here, but are nevertheless tractable in the framework.

ML abstractions are modeled by the rule $[*^v, *^c]$. To express type abstractions, we add the rule $[\Box^v, *^v]$ which basically says that a type scheme is also a type. This setting ignores the fact that ML's polymorphism is predicative: in ML, type abstractions may be nested inside of type abstractions, but not inside of other type constructors.

Predicativity can be recovered in at least two ways:

(i) Add a new sort $\sharp$ and use the rules $(\Box^v, *^v, \sharp)$ and $(\Box^v, \sharp, \sharp)$ for forming type abstractions.

(ii) Add an inclusion $*^v \subseteq \Box^v$ in the style of Harper and Mitchell [25]. Now type abstractions are governed by the rule $(\Box^v, \Box^v, \Box^v)$.

Both approaches do not fit into the present framework: the specification involving $\sharp$ is not value-adhering (see Definition 26) and inclusion is not considered.[4] But still, there is no harm in admitting impredicative polymorphism in an explicitly typed intermediate language as long as type checking remains decidable.

The rule $[\Box^v, *^v]$ forces the body of a type abstraction to have a value type. Thus the MTS specification enforces in a natural way the restriction to value polymorphism present in SML'97 (this restriction ensures type soundness in the presence of side effects, see e.g., [22, 23, 49]).

In summary, here are the rules we consider.

$$\mathcal{R}_{ML} = \{[*^v, *^c], [\Box^v, *^v]\}$$

The source language for the encoding in the MTS given by $\mathcal{R}_{ML}$ is the fragment of SML'97 given in Fig. 3. A minor difference to SML'97 is the syntactic distinction between variables f bound by $\mathsf{fix}\ f.e$ and other variables x. Both are considered values, but the translation treats them differently. Furthermore, SML's $\mathsf{letrec}\ x = v\ \mathsf{in}\ e$ is syntactic sugar for $\mathsf{let}\ x \Leftarrow \mathsf{fix}\ f.v\{x := f\}\ \mathsf{in}\ e$. We have chosen the syntax with fix to simplify the translation.

The intended operational semantics is call-by-value, which we formalize by defining

[4] Actually, Pure Type Systems with Universe Inclusion have been studied and formalized in [41] and it would make sense to use them as a basis.

$$
\begin{array}{lll}
E & ::= & [\,] \mid E@e \mid v@E \mid \mathsf{if0}\ E\ e\ e \mid \mathsf{succ}\ E \mid \mathsf{pred}\ E \\
\\
(\lambda x.e)@v & \mapsto & e\{x := v\} \\
(\mathsf{fix}\ f.e)@v & \mapsto & e\{f := \mathsf{fix}\ f.e\}@v \\
\mathsf{let}\ x = v\ \mathsf{in}\ e & \mapsto & e\{x := v\} \\
\mathsf{if0}\ 0\ e_1\ e_2 & \mapsto & e_1 \\
\mathsf{if0}\ (i+1)\ e_1\ e_2 & \mapsto & e_2 \\
\mathsf{succ}\ i & \mapsto & (i+1) \\
\mathsf{pred}\ (i+1) & \mapsto & i \\
\mathsf{pred}\ 0 & \mapsto & 0 \\
\\
l \mapsto r & \Rightarrow & E[l] \to E[r]
\end{array}
$$

Fig. 4. Evaluation Relation for SML'97 Fragment

a small-step evaluation relation $\to$ using the evaluation contexts E and reduction rules shown in Fig. 4. The reduction for $\mathsf{fix}\ f.e$ is slightly unusual, but it nicely captures a call-by-value fixpoint computation. Figure 5 recalls the typing rules of ML for reference. We use the notation $\tau \leq \sigma$ if $\sigma = \forall\alpha_1 \ldots \forall\alpha_n.\tau'$ and there exist $\tau_1, \ldots, \tau_n$ such that $\tau = \tau'\{\alpha_i \mapsto \tau_i\}$, and $\mathsf{gen}(\Gamma, \tau) = \forall\alpha_1 \ldots \forall\alpha_n.\tau$ where $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{FV}(\tau) \setminus \mathrm{FV}(\Gamma)$ with FV denoting the set of free variables in types and type assumptions.

The translation of types and type schemes has three layers (see Fig. 6). The translation $\mathcal{C}^M$ for top-level terms generates computation types. The translation $\mathcal{V}^M$ generates value types and the translation $\mathcal{S}^M$ applies to type schemes, which also abstract over value types. This translation generates well-formed types.

Lemma 1 *Let $G_\sigma = \alpha_1 : *^v, \ldots \alpha_n : *^v$ where $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{FV}(\sigma)$. Then*

(i) $G_\tau \vdash \mathcal{V}^M[\![\tau]\!] : *^v$;

(ii) $G_\tau \vdash \mathcal{C}^M[\![\tau]\!] : *^c$;

(iii) $G_\sigma \vdash \mathcal{S}^M[\![\sigma]\!] : *^v$.

The translation on terms is structured after the translation on types. It is split into the translations $\mathcal{V}^M$ of syntactic values and $\mathcal{C}^M$ of non-values. Since the translation really maps type derivations to type derivations, we assume access to a syntax-directed SML type derivation (as generated by the typing rules in Fig. 5), namely at variable accesses to $x : \forall\alpha_1 \ldots \alpha_n.\tau$ we assume access to the instantiation $\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n$, and at let expressions we assume we know the generalized type $\forall\alpha_1 \ldots \alpha_n.\tau$. We indicate the type information by superscripts and the instantiation by subscripts. Figure 7 defines the

$$\Gamma, x:\sigma \vdash_{SML} x:\tau \quad \tau \leq \sigma \qquad \Gamma, f:\tau \vdash_{SML} f:\tau \qquad \Gamma \vdash_{SML} i:\mathsf{nat}$$

$$\frac{\Gamma, x:\tau_2 \vdash_{SML} e:\tau_1}{\Gamma \vdash_{SML} \lambda x.e:\tau_2 \to \tau_1} \qquad \frac{\Gamma \vdash_{SML} e_1:\tau_2 \to \tau_1 \quad \Gamma \vdash_{SML} e_2:\tau_2}{\Gamma \vdash_{SML} e_1 @ e_2:\tau_1}$$

$$\frac{\Gamma \vdash_{SML} v:\tau_1 \quad \Gamma, x:\mathsf{gen}(\Gamma,\tau_1) \vdash_{SML} e:\tau_2}{\Gamma \vdash_{SML} \mathsf{let}\ x = v\ \mathsf{in}\ e:\tau_2}$$

$$\frac{\Gamma \vdash_{SML} e_1:\mathsf{nat} \quad \Gamma \vdash_{SML} e_2:\tau \quad \Gamma \vdash_{SML} e_3:\tau}{\Gamma \vdash_{SML} \mathsf{if0}\ e_1\ e_2\ e_3:\tau}$$

$$\frac{\Gamma \vdash_{SML} e:\mathsf{nat}}{\Gamma \vdash_{SML} \mathsf{succ}\ e:\mathsf{nat}} \qquad \frac{\Gamma \vdash_{SML} e:\mathsf{nat}}{\Gamma \vdash_{SML} \mathsf{pred}\ e:\mathsf{nat}}$$

$$\frac{\Gamma, f:\tau_2 \to \tau_1 \vdash_{SML} e:\tau_2 \to \tau_1}{\Gamma \vdash_{SML} \mathsf{fix}\ f.e:\tau_2 \to \tau_1}$$

Fig. 5. Typing Rules for SML'97 Fragment

$$\begin{aligned}
\mathcal{C}^M[\![\tau]\!] &= \mathsf{M}\ \mathcal{V}^M[\![\tau]\!] \\
\mathcal{V}^M[\![\alpha]\!] &= \alpha \\
\mathcal{V}^M[\![\mathsf{nat}]\!] &= \mathsf{nat} \\
\mathcal{V}^M[\![\tau_1 \to \tau_2]\!] &= \mathcal{V}^M[\![\tau_1]\!] \to \mathcal{C}^M[\![\tau_2]\!] \\
\mathcal{S}^M[\![\tau]\!] &= \mathcal{V}^M[\![\tau]\!] \\
\mathcal{S}^M[\![\forall\alpha.\sigma]\!] &= \Pi\alpha{:}*^v.\, \mathcal{S}^M[\![\sigma]\!]
\end{aligned}$$

Fig. 6. Translation of SML'97 Types

$$
\begin{aligned}
\mathcal{C}^M[\![v]\!] &= \mathsf{unit}\ \mathcal{V}^M[\![v]\!] \\
\mathcal{C}^M[\![e_1^{\tau_2\to\tau_1} @ e_2]\!] &= \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathcal{C}^M[\![e_1]\!]\ \mathsf{in}\ \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathcal{C}^M[\![e_2]\!]\ \mathsf{in}\ y_1\ y_2 \\
\mathcal{C}^M[\![\mathsf{let}\ x^{\sigma=\forall\alpha_1\ldots\alpha_n.\tau} = v^\tau\ \mathsf{in}\ e]\!] &= (\lambda x : \mathcal{S}^M[\![\sigma]\!].\mathcal{C}^M[\![e]\!])\ (\lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!]) \\
\mathcal{C}^M[\![\mathsf{if0}\ e_1\ e_2\ e_3]\!] &= \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^M[\![e_1]\!]\ \mathsf{in}\ \mathsf{if0}\ y\ \mathcal{C}^M[\![e_2]\!]\ \mathcal{C}^M[\![e_3]\!] \\
\mathcal{C}^M[\![\mathsf{succ}\ e]\!] &= \mathsf{let}\ y : \mathsf{int} \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (\mathsf{succ}\ y) \\
\mathcal{C}^M[\![\mathsf{pred}\ e]\!] &= \mathsf{let}\ y : \mathsf{int} \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (\mathsf{pred}\ y)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}^M[\![i]\!] &= \lceil i \rceil \\
\mathcal{V}^M[\![x^{\forall\alpha_1\ldots\alpha_n.\tau}_{\alpha_i\mapsto\tau_i}]\!] &= x\ \mathcal{V}^M[\![\tau_1]\!]\ \ldots\ \mathcal{V}^M[\![\tau_n]\!] \\
\mathcal{V}^M[\![f^{\tau_2\to\tau_1}]\!] &= \lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow f\ \mathsf{in}\ y_1\ y_2 \\
\mathcal{V}^M[\![\lambda x^\tau.e]\!] &= \lambda x : \mathcal{V}^M[\![\tau]\!].\mathcal{C}^M[\![e]\!] \\
\mathcal{V}^M[\![\mathsf{fix}\ f^{\tau_2\to\tau_1}.e]\!] &= \lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2
\end{aligned}
$$

Fig. 7. Translation of the SML'97 Fragment

translation. It generates well-formed types and is type-sound. Let

$$G_{y_1:\sigma_1,\ldots,y_n:\sigma_n} = y_1 : G_{\sigma_1}, \ldots, y_n : G_{\sigma_n}$$

where y ranges over all variables (fix-bound and other variables, $y ::= x \mid f$).

Lemma 2 *Let $\Gamma = \{y_1 : \sigma_1, \ldots, y_n : \sigma_n\}$. Let further $\Gamma' = \{y_1 : A'_1, \ldots, y_n : A'_n\}$ where $A'_i = \mathcal{S}^M[\![A_i]\!]$ if y_i is not* fix*-bound and $A'_i = \mathcal{C}^M[\![A_i]\!]$ otherwise. Then*

(1) $$\Gamma \vdash_{SML} e : \tau \quad \Rightarrow \quad G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e]\!] : \mathcal{C}^M[\![\tau]\!]$$

using $\mathcal{R}_{ML}$.

Proof See Appendix C. ■

Next, we prove that the translation is compatible with conversion.

Theorem 5 *Suppose $e_1 \to e_2$ in ML. Then $\mathcal{C}^M[\![e_1]\!] =_{ml} \mathcal{C}^M[\![e_2]\!]$.*

The proof and some auxiliary lemmas may be found in Appendix D.

Unfortunately, the result cannot be strengthened: it would be more satisfactory to relate to single-step MTS evaluation (see Definition 25), e.g., to have $\mathcal{C}^M[\![e_1]\!] \longmapsto_{ml} \mathcal{C}^M[\![e_2]\!]$ but that is not possible due to the translation of the let and fix constructs. In addition, the primitives succ and pred require a reduction step inside of unit. The latter could be avoided by giving the primitives a computation type.

The MTS formulation admits various extensions to the SML'97 fragment either without variation of the specification or with only slight additions.

3.2.1 References

We extend the syntax of our ML fragment by types and terms related to references and the operations thereof in the usual way.

$$
\begin{aligned}
\tau &::= \ldots \mid \mathsf{ref}\ \tau \mid () \\
n &::= \ldots \mid \mathsf{ref}\ e \mid !\ e \mid e := e
\end{aligned}
$$

That is, there are a new type constructor ref and non-value terms for creating a reference $\mathsf{ref}\ e$, reading a reference $!\ e$, and updating the contents of a reference $e := e$. The new type constant $()$ corresponds to the one-element type unit of ML. The typing rules are just the standard ML rules and are therefore omitted. The semantics of the operators can be defined using the monad of state transformers. However, we give a standard operational (small-step) semantics which maps a state σ and an expression to the next state and the reduced expression. Let L be a denumerable set of *locations* with $l \in L$ and let V be the set of syntactic values v.

$$
\begin{array}{lll}
E & ::= \ldots \mid \mathsf{ref}\ E \mid !\ E \mid E := e \mid v := E & \\
v & ::= \ldots \mid l \mid () & \textit{locations, unit value} \\
\sigma & \in\ L \hookrightarrow V &
\end{array}
$$

$$
\begin{array}{lll}
\sigma, E[\mathsf{ref}\ v] & \to\ \sigma[l \mapsto v], E[l] & l \notin \mathrm{dom}(\sigma) \\
\sigma, E[!\ l] & \to\ \sigma, E[\sigma(l)] & \\
\sigma, E[l := v] & \to\ \sigma[l \mapsto v], E[()] &
\end{array}
$$

The remaining reductions do not affect the store and are simple variations of the ones defined in Fig. 4.

To construct the translation we need to add three new constants to the metalanguage. These constants are the metalanguage's operators to perform the operations on references. Their types are given as follows.

$$
\begin{array}{ll}
\mathsf{ref}_M & : (\Pi\alpha : *^v.\alpha \to \mathsf{M}\ (\mathsf{ref}\ \alpha)) \\
!_M & : (\Pi\alpha : *^v.\mathsf{ref}\ \alpha \to \mathsf{M}\ \alpha) \\
:=_M & : (\Pi\alpha : *^v.\mathsf{ref}\ \alpha \to \alpha \to \mathsf{M}\ ())
\end{array}
$$

The typing makes it clear that only values can be stored in references. The translation on types extends as follows:

$$
\begin{array}{ll}
\mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] & = \mathsf{ref}\ \mathcal{V}^M[\![\tau]\!] \\
\mathcal{V}^M[\![()]\!] & = ()
\end{array}
$$

The translation on terms extends correspondingly.

$$\begin{aligned}
\mathcal{C}^M[\![\mathsf{ref}\ e_{\alpha\mapsto\tau}]\!] &= \mathsf{let}\ y : \mathcal{V}^M[\![\tau]\!] \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ (\mathsf{ref}_M\ \mathcal{V}^M[\![\tau]\!])\ y \\
\mathcal{C}^M[\![!\ e_{\alpha\mapsto\tau}]\!] &= \mathsf{let}\ y : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ (!_M\ \mathcal{V}^M[\![\tau]\!])\ y \\
\mathcal{C}^M[\![e_1 := e_{2\alpha\mapsto\tau}]\!] &= \mathsf{let}\ y_1 : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] \Leftarrow \mathcal{C}^M[\![e_1]\!]\ \mathsf{in} \\
&\qquad \mathsf{let}\ y_2 : \mathcal{C}^M[\![\tau]\!] \Leftarrow \mathcal{C}^M[\![e_2]\!]\ \mathsf{in}\ (:=_M\ \mathcal{V}^M[\![\tau]\!])\ y_1 y_2
\end{aligned}$$

The appendix A shows an example of a sound compile time transformation with references and types.

Lemma 3 *Lemma 1 extends to the new type constructors.*

Lemma 4 *Lemma 2 extends to the new constructs.*

Proof See Appendix E. ■

3.2.2 Higher-order Polymorphism

Semi-explicit Polymorphism

Recently, there have been several proposals to extend ML with higher-order polymorphism [17, 37]. The idea is that type abstractions are no longer restricted to the top level — they may appear everywhere in types — and in the type $(\Pi\alpha : *^v.\alpha) : *^v$ the variable α may range over all terms $M : *^v$ (impredicative polymorphism).

Garrigue and Remy [17] have proposed such an extension of ML with explicit type annotations, called semi-explicit polymorphism. They extend the term and type language by

$$\begin{aligned}
v &::= \ldots \mid [v : \sigma] \\
e &::= \ldots \mid \langle e \rangle \\
\tau &::= \ldots \mid [\sigma]
\end{aligned}$$

The notation $[\sigma]$ wraps a type scheme into a monotype which can instantiate type variables and which can occur nested within function types [5]. The basic idea is that $[v : \sigma]$ generalizes the inferred type of v to σ and wraps the type into a monotype so that $[v : \sigma]$ has type $[\sigma]$. The expression $\langle v \rangle$ unwraps such a canned type scheme: if $v : [\sigma]$ then $\langle v \rangle : \sigma$. Roughly, the $[v : \sigma]$ expression corresponds to type abstraction and the $\langle v \rangle$ expression corresponds to type application, but without giving the instantiation explicitly. The restriction to polymorphic values is inherited from SML'97.

The translation of this extended language into our instance of MTS is straightforward.

$$\begin{aligned}
\mathcal{V}^M[\![\,[\sigma]\,]\!] &= \mathcal{S}^M[\![\sigma]\!] \\
\mathcal{V}^M[\![\,[v^\tau : \forall\alpha_1 \ldots \alpha_n.\tau]\,]\!] &= \lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!] \\
\mathcal{C}^M[\![\langle e \rangle^{[\sigma = \forall\alpha_1 \ldots \alpha_n.\tau]}_{\alpha_i \mapsto \tau_i}]\!] &= \mathsf{let}\ y : \mathcal{S}^M[\![\sigma]\!] \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (y\ \mathcal{V}^M[\![\tau_1]\!]\ \ldots\ \mathcal{V}^M[\![\tau_n]\!])
\end{aligned}$$

[5] This glosses over the main technical point of their work, which makes type inference decidable for their system. We can safely ignore it here because we start from a given type derivation in their system.

For this translation, we can show again that typing is preserved with respect to the syntax directed system given in the type inference algorithm of Garrigue and Remy [17].

Second order polymorphic lambda calculus

In the same way, we can encode an ML-style variant of the second order polymorphic lambda calculus F_2 where SML'97 types and expressions (Fig. 3) are extended by type abstraction and type application:

$$\begin{aligned} \sigma &::= \alpha \mid \mathsf{nat} \mid \sigma \to \sigma \mid \forall\alpha.\sigma \\ v &::= \ldots \mid \Lambda\alpha.v \\ e &::= \ldots \mid \Lambda\alpha.v \mid e\{\sigma\} \end{aligned}$$

with the standard typing rules and the additional ML-style call-by-value reduction rule

$$(\Lambda\alpha.v)\{\sigma\} \mapsto v\{\alpha := \sigma\}$$

The translation of terms is extended in essentially the same way as for semi-explicit polymorphism.

$$\begin{aligned} \mathcal{V}^M[\![\Lambda\alpha.v]\!] &= \lambda\alpha : *^v.\mathcal{V}^M[\![v]\!] \\ \mathcal{C}^M[\![e^{\forall\alpha.\sigma_1}\{\sigma_2\}]\!] &= \mathsf{let}\ y : \mathcal{V}^M[\![\forall\alpha.\sigma_1]\!] \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (y\ \mathcal{V}^M[\![\sigma]\!]) \end{aligned}$$

with the translation of value types changed to:

$$\begin{aligned} \mathcal{V}^M[\![\alpha]\!] &= \alpha \\ \mathcal{V}^M[\![\mathsf{nat}]\!] &= \mathsf{nat} \\ \mathcal{V}^M[\![\sigma_1 \to \sigma_2]\!] &= \mathcal{V}^M[\![\sigma_1]\!] \to \mathcal{C}^M[\![\sigma_2]\!] \\ \mathcal{V}^M[\![\forall\alpha.\sigma]\!] &= \Pi\alpha{:}*^v.\ \mathcal{V}^M[\![\sigma]\!] \end{aligned}$$

3.2.3 Polymorphism by name

Leroy [33] has shown that the unsoundness of the naive approach to polymorphic references in ML can be attributed to the fact that type generalization evaluates its body once and for all. The restriction to syntactic values (as enforced by our ML rule $[\Box^v, *^v]$) forces this evaluation step to be a trivial step. Another sound option, that Leroy [33] calls polymorphism by name, is to equip type generalization with a non-strict semantics and to repeat the computation whenever the corresponding type abstraction is applied. This corresponds to replacing $*^v$ by $*^c$ in the rules for type $\Box^v$ abstractions.

$$\mathcal{R}_{pbn} = \{[*^v, *^c], [*^c, *^c], [\Box^v, *^c], [\Box^v, *^v]\}$$

Let's go briefly over the uses of the four kinds of abstractions (cf. Fig. 8):

- $[*^c, *^c]$ image of a source abstraction;
- $[\Box^v, *^c]$ image of let: typing the innermost abstraction around the image of the header e_1;
- $[\Box^v, *^v]$ image of let: typing the subsequent abstractions around the image of the header e_1;

- $[*^v, *^c]$ image of let: typing the abstraction around the image of the body e_2 in case the header is polymorphic.

The modification of the translation and the proof of the preservation of typing are straightforward. Here is the translation on types:

$$
\begin{aligned}
\mathcal{C}^P[\![\tau]\!] &= \mathsf{M}\ \mathcal{V}^P[\![\tau]\!] \\
\mathcal{V}^P[\![\alpha]\!] &= \alpha \\
\mathcal{V}^P[\![\mathsf{nat}]\!] &= \mathsf{nat} \\
\mathcal{V}^P[\![\tau_1 \to \tau_2]\!] &= \mathcal{C}^P[\![\tau_1]\!] \to \mathcal{C}^P[\![\tau_2]\!] \\
\mathcal{S}^P[\![\tau]\!] &= \mathcal{C}^P[\![\tau]\!] \\
\mathcal{S}^P[\![\forall\alpha.\sigma]\!] &= \Pi\alpha{:}*^v.\ \mathcal{S}^P[\![\sigma]\!]
\end{aligned}
$$

For the translation of terms we need to observe the subtle change that variables (regardless whether fix-bound or not) will not be regarded as values, anymore. The encoding of call-by-value application employs the method mentioned in Sec. 3.1. The value restriction is no longer required. The revised source language is thus

$$
\begin{array}{lrl}
\text{terms} & e & ::= v \mid n \\
\text{values} & v & ::= i \mid \lambda x.e \mid \mathsf{fix}\ x.e \\
\text{non-values} & n & ::= x \mid e@e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{if0}\ e\ e\ e \mid \mathsf{succ}\ e \mid \mathsf{pred}\ e
\end{array}
$$

Evaluation is now defined by:

$$
\begin{array}{ll}
E_{pbn} & ::= [\,] \mid E_{pbn}@e \mid v@E_{pbn} \mid \mathsf{if0}\ E_{pbn}\ e\ e \mid \mathsf{succ}\ E_{pbn} \mid \mathsf{pred}\ E_{pbn} \\
\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \ \mapsto\ e_2\{x := e_1\} & \\
 & \ldots
\end{array}
$$

with the remaining reductions as in the SML'97 fragment (see Fig. 4). Figure 8 defines the translation. Again, the translation preserves typing (we omit the statement of a lemma analogous to Lemma 1).

Lemma 5 *Let PBN be the syntax-directed ML type system with polymorphism by name defined by Leroy [33, Fig. 4, p.226].*

$$\Gamma \vdash_{PBN} e : \tau \quad \Rightarrow \quad G_\Gamma, \mathcal{S}^P[\![\Gamma]\!] \vdash_{\mathcal{R}_{pbn}} \mathcal{C}^P[\![e]\!] : \mathcal{C}^P[\![\tau]\!]$$

3.2.4 Call-by-name ML

As an example of a call-by-name language, we investigate a call-by-name variant of ML. In this variant, also non-functional fixpoints make sense (at least, in an extended language

$$
\begin{aligned}
\mathcal{C}^P[\![v]\!] &= \mathsf{unit}\ \mathcal{V}^P[\![v]\!]\\
\mathcal{C}^P[\![x^{\forall\alpha_1\ldots\alpha_n.\tau}_{\alpha_i\mapsto\tau_i}]\!] &= x\ \mathcal{V}^P[\![\tau_1]\!]\ \ldots\ \mathcal{V}^P[\![\tau_n]\!]\\
\mathcal{C}^P[\![e_1^{\tau_2\to\tau_1}@e_2]\!] &= \mathsf{let}\ y_1 : \mathcal{V}^P[\![\tau_2\to\tau_1]\!] \Leftarrow \mathcal{C}^P[\![e_1]\!]\ \mathsf{in}\ \mathsf{let}\ y_2 : \mathcal{V}^P[\![\tau_2]\!] \Leftarrow \mathcal{C}^P[\![e_2]\!]\ \mathsf{in}\ y_1\ (\mathsf{unit}\ y_2)\\
\mathcal{C}^P[\![\mathsf{let}\ x^{\sigma=\forall\alpha_1\ldots\alpha_n.\tau} = e_1^\tau\ \mathsf{in}\ e_2]\!] &= (\lambda x : \mathcal{S}^P[\![\sigma]\!].\mathcal{C}^P[\![e_2]\!])\ (\lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{C}^P[\![e_1]\!])\\
\mathcal{C}^P[\![\mathsf{if0}\ e_1\ e_2\ e_3]\!] &= \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^P[\![e_1]\!]\ \mathsf{in}\ \mathsf{if0}\ y\ \mathcal{C}^P[\![e_2]\!]\ \mathcal{C}^P[\![e_3]\!]\\
\mathcal{C}^P[\![\mathsf{succ}\ e]\!] &= \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^P[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (\mathsf{succ}\ y)\\
\mathcal{C}^P[\![\mathsf{pred}\ e]\!] &= \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^P[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (\mathsf{pred}\ y)\\
\\
\mathcal{V}^P[\![i]\!] &= i\\
\mathcal{V}^P[\![\lambda x^\tau.e]\!] &= \lambda x : \mathcal{C}^P[\![\tau]\!].\mathcal{C}^P[\![e]\!]\\
\mathcal{V}^P[\![\mathsf{fix}\ x^{\tau_2\to\tau_1}.e]\!] &= \lambda y_2 : \mathcal{C}^P[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^P[\![\tau_2\to\tau_1]\!] \Leftarrow \mathsf{fix}\ x : \mathcal{C}^P[\![\tau_2\to\tau_1]\!].\mathcal{C}^P[\![e]\!]\ \mathsf{in}\ y_1\ y_2
\end{aligned}
$$

Fig. 8. Translation for Polymorphism by Name

with sum types) so $\mathsf{fix}\ x.e$ is no longer a value, but it is a redex by itself.

$$
\begin{array}{lll}
\text{terms} & e & ::= v \mid n\\
\text{values} & v & ::= i \mid \lambda x.e\\
\text{non-values} & n & ::= x \mid \mathsf{fix}\ x.e \mid e@e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{if0}\ e\ e\ e \mid \mathsf{succ}\ e \mid \mathsf{pred}\ e
\end{array}
$$

Evaluation changes in the expected way.

$$
\begin{array}{lll}
E_{cbn} & ::= & [\,] \mid E_{cbn}@e \mid \mathsf{if0}\ E_{cbn}\ e\ e \mid \mathsf{succ}\ E_{cbn} \mid \mathsf{pred}\ E_{cbn}\\
\\
(\lambda x.e_2)@e_1 & \mapsto & e_2\{x := e_1\}\\
\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 & \mapsto & e_2\{x := e_1\}\\
\mathsf{fix}\ x.e & \mapsto & e\{x := \mathsf{fix}\ x.e\}\\
 & \ldots & \\
l \mapsto r & \Rightarrow & E_{cbn}[l] \to_{cbn} E_{cbn}[r]
\end{array}
$$

It turns out that we can use the same rule set as for polymorphism by name:

$$\mathcal{R}_{cbn} = \mathcal{R}_{pbn} = \{[*^v, *^c], [*^c, *^c], [\Box^v, *^c], [\Box^v, *^v]\}$$

The translation of type schemes is identical, too.

$$
\begin{aligned}
\mathcal{C}^N[\![\tau]\!] &= \mathcal{C}^P[\![\tau]\!]\\
\mathcal{V}^N[\![\tau]\!] &= \mathcal{V}^P[\![\tau]\!]\\
\mathcal{S}^N[\![\sigma]\!] &= \mathcal{S}^P[\![\sigma]\!]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}^N[\![v]\!] &= \mathsf{unit}\ \mathcal{V}^N[\![v]\!] \\
\mathcal{C}^N[\![x^{\forall\alpha_1\ldots\alpha_n.\tau}_{\alpha_i\mapsto\tau_i}]\!] &= x\ \mathcal{V}^N[\![\tau_1]\!]\ \ldots\ \mathcal{V}^N[\![\tau_n]\!] \\
\mathcal{C}^N[\![\mathsf{fix}\ x^\tau.e]\!] &= \mathsf{fix}\ x : \mathcal{C}^N[\![\tau]\!].\mathcal{C}^N[\![e]\!] \\
\mathcal{C}^N[\![e_1^{\tau_2\to\tau_1}@e_2]\!] &= \mathsf{let}\ y_1 : \mathcal{V}^N[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathcal{C}^N[\![e_1]\!]\ \mathsf{in}\ y_1\ \mathcal{C}^N[\![e_2]\!] \\
\mathcal{C}^N[\![\mathsf{let}\ x^{\sigma=\forall\alpha_1\ldots\alpha_n.\tau} = e_1^\tau\ \mathsf{in}\ e_2]\!] &= (\lambda x : \mathcal{S}^N[\![\sigma]\!].\mathcal{C}^N[\![e_2]\!])\ (\lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{C}^N[\![e_1]\!]) \\
\mathcal{C}^N[\![\mathsf{if0}\ e_1\ e_2\ e_3]\!] &= \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^N[\![e_1]\!]\ \mathsf{in\ if0}\ y\ \mathcal{C}^N[\![e_2]\!]\ \mathcal{C}^N[\![e_3]\!] \\
\mathcal{C}^N[\![\mathsf{succ}\ e]\!] &= \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^N[\![e]\!]\ \mathsf{in\ unit}\ (\mathsf{succ}\ y) \\
\mathcal{C}^N[\![\mathsf{pred}\ e]\!] &= \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^N[\![e]\!]\ \mathsf{in\ unit}\ (\mathsf{pred}\ y) \\
\mathcal{V}^N[\![i]\!] &= i \\
\mathcal{V}^N[\![\lambda x^\tau.e]\!] &= \lambda x : \mathcal{C}^N[\![\tau]\!].\mathcal{C}^N[\![e]\!]
\end{aligned}
$$

Fig. 9. Translation for call-by-name ML

The translation of terms may be found in Fig. 9. In fact, it is identical to the translation in Fig. 8, up to the cases for fix $x.e$ and $e@e$.

We can prove the following statements about the translation.

Lemma 6 *Let* $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ *and* $\Gamma' = \{x_1 : \mathcal{S}^N[\![\sigma_1]\!], \ldots, x_n : \mathcal{S}^N[\![\sigma_n]\!]\}$. *Then*

$$\Gamma \vdash_{SML} e : \tau \quad \Rightarrow \quad G_\Gamma, \Gamma' \vdash \mathcal{C}^N[\![e]\!] : \mathcal{C}^N[\![\tau]\!] \tag{2}$$

using $\mathcal{R}_{cbn}$.

Theorem 6 *Suppose* $e_1 \to_{cbn} e_2$. *Then* $\mathcal{C}^N[\![e_1]\!] =_{ml} \mathcal{C}^N[\![e_2]\!]$.

The proofs are omitted, because they are very similar to the ones presented for the call-by-value variant. Again, we only obtain convertibility of the translated terms, which is again due to the translations of let, fix, and the primitives.

3.3 Encodings of F_ω

Finally, we give an outline of call-by-name and call-by-value translations for F_ω [19], also known as the higher-order lambda calculus. Figure 10 defines the syntax of the source language. We omit the standard formation rules for constructor contexts Δ, term contexts Θ, constructors $\Delta \rhd \sigma : \kappa$, constructor conversion $\Delta \rhd \sigma_1 = \sigma_2 : \kappa$, and terms $\Delta; \Theta \rhd e : \sigma$.

3.3.1 Standard Call-by-Name Encoding

Figure 11 shows a call-by-name encoding of F_ω. In order to prove that this translation preserves typing, we need the following specification:

$$\mathcal{R}_{Fcbn} = \{(\Box^v, \Box^v, \Box^v), [*^c, *^c], [\Box^v, *^c]\}$$

$$
\begin{array}{ll}
\text{kinds} & \kappa ::= \Omega \mid \kappa \Rightarrow \kappa \\
\text{constructors} & \sigma ::= \alpha \mid \sigma \to \sigma \mid \forall \alpha : \kappa.\sigma \mid \lambda \alpha : \kappa.\sigma \mid \sigma\{\sigma\} \\
\text{terms} & e ::= x \mid \lambda x : \sigma.e \mid e \mathtt{@} e \mid \Lambda \alpha : \kappa.e \mid e\{\sigma\}
\end{array}
$$

Fig. 10. Syntax of F_ω

Translation of kinds

$$
\begin{aligned}
\mathcal{K}[\![\Omega]\!] &= *^v \\
\mathcal{K}[\![\kappa_1 \Rightarrow \kappa_2]\!] &= \mathcal{K}[\![\kappa_1]\!] \to \mathcal{K}[\![\kappa_2]\!]
\end{aligned}
$$

Call-by-name translation of constructors

$$
\begin{array}{llll}
\mathcal{T}[\![\alpha^{\Omega}]\!] = \mathsf{M}\ \alpha & & & \\
\mathcal{T}[\![\alpha^{\kappa_1 \Rightarrow \kappa_2}]\!] = \alpha & & \mathcal{V}[\![\alpha]\!] = \alpha & \\
\mathcal{T}[\![\sigma_1 \to \sigma_2]\!] = \mathsf{M}\ (\mathcal{T}[\![\sigma_1]\!] \to \mathcal{T}[\![\sigma_2]\!]) & & \mathcal{V}[\![\sigma_1 \to \sigma_2]\!] = \mathcal{T}[\![\sigma_1]\!] \to \mathcal{T}[\![\sigma_2]\!] & \\
\mathcal{T}[\![\forall \alpha : \kappa.\sigma]\!] = \mathsf{M}\ (\Pi \alpha{:}\mathcal{K}[\![\kappa]\!].\ \mathcal{T}[\![\sigma]\!]) & & \mathcal{V}[\![\forall \alpha : \kappa.\sigma]\!] = \Pi \alpha{:}\mathcal{K}[\![\kappa]\!].\ \mathcal{T}[\![\sigma]\!] & \\
\mathcal{T}[\![\lambda \alpha : \kappa.\sigma]\!] = \lambda \alpha : \mathcal{K}[\![\kappa]\!].\mathcal{T}[\![\sigma]\!] & & \mathcal{V}[\![\lambda \alpha : \kappa.\sigma]\!] = \lambda \alpha : \mathcal{K}[\![\kappa]\!].\mathcal{V}[\![\sigma]\!] & \\
\mathcal{T}[\![\sigma_1\{\sigma_2\}]\!] = \mathcal{T}[\![\sigma_1]\!]\ (\mathcal{V}[\![\sigma_2]\!]) & & \mathcal{V}[\![\sigma_1\{\sigma_2\}]\!] = \mathcal{V}[\![\sigma_1]\!]\ (\mathcal{V}[\![\sigma_2]\!]) &
\end{array}
$$

Call-by-name translation of terms

$$
\begin{aligned}
\mathcal{O}[\![x]\!] &= x \\
\mathcal{O}[\![\lambda x : \sigma.e]\!] &= \mathsf{unit}\ (\lambda x : \mathcal{T}[\![\sigma]\!].\mathcal{O}[\![e]\!]) \\
\mathcal{O}[\![e_1^{\sigma_2 \to \sigma_1} \mathtt{@} e_2]\!] &= \mathsf{let}\ y : \mathcal{V}[\![\sigma_2 \to \sigma_1]\!] \Leftarrow \mathcal{O}[\![e_1]\!]\ \mathsf{in}\ y\ (\mathcal{O}[\![e_2]\!]) \\
\mathcal{O}[\![\Lambda \alpha : \kappa.e]\!] &= \mathsf{unit}\ (\lambda \alpha : \mathcal{K}[\![\kappa]\!].\mathcal{O}[\![e]\!]) \\
\mathcal{O}[\![e^{\forall \alpha : \kappa.\sigma_1}\{\sigma_2\}]\!] &= \mathsf{let}\ y : \mathcal{V}[\![\forall \alpha : \kappa.\sigma_1]\!] \Leftarrow \mathcal{O}[\![e]\!]\ \mathsf{in}\ y\ (\mathcal{V}[\![\sigma]\!])
\end{aligned}
$$

Fig. 11. Standard call-by-name encoding of F_ω

Objects

$$O \ ::= x \mid \mathsf{unit}\ W \mid \mathsf{let}\ x : \hat{W} \Leftarrow O \ \mathsf{in}\ O \mid x\ O \mid x\ \hat{W} \mid W\ O \mid W\ \hat{W}$$

Values

$$W \ ::= \lambda x : \hat{T}.O \mid \lambda \alpha : \hat{K}.O$$

Value constructors

$$\hat{W} \ ::= \alpha \mid \hat{T} \to \hat{T} \mid \Pi\alpha{:}\hat{K}.\,\hat{T} \mid \lambda\alpha : \hat{K}.\hat{W} \mid \hat{W}\ \hat{W}$$

Computation constructors

$$\hat{T} \ ::= \alpha^{\hat{K}_1 \to \hat{K}_2} \mid \mathsf{M}\ \hat{W} \mid \lambda\alpha : \hat{K}.\hat{T} \mid \hat{T}\ \hat{W}$$

Kinds

$$\hat{K} \ ::= *^v \mid \hat{K} \to \hat{K}$$

Fig. 12. Image of the standard call-by-name translation

Lemma 7 *For a constructor context* $\Delta = \alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n$ *let* $G_\Delta = \alpha_1 : \mathcal{K}[\![\kappa_1]\!], \ldots, \alpha_n : \mathcal{K}[\![\kappa_n]\!]$. *For a term context* $\Theta = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ *let* $G_\Theta = x_1 : \mathcal{T}[\![\sigma_1]\!], \ldots, x_n : \mathcal{T}[\![\sigma_n]\!]$.

(i) If $\Delta \vartriangleright \sigma : \kappa$ *then* $G_\Delta \vdash \mathcal{V}[\![\sigma]\!] : \mathcal{K}[\![\kappa]\!]$.

(ii) If $\Delta; \Theta \vartriangleright e : \sigma$ *then* $G_\Delta, G_\Theta \vdash \mathcal{O}[\![e]\!] : \mathcal{T}[\![\sigma]\!]$.

Harper and Lillibridge [22, 24] have defined a standard call-by-name semantics for a superset of the calculus considered here. For their small-step operational semantics $\to_{std-cbn}$ (see Appendix F.1), we can show that the translation is compatible with single-step reduction for MTS (see Definition 25).

Theorem 7 *Suppose* $\Delta; \Theta \vartriangleright e_1 : \sigma_1$ *and* $e_1 \to_{std-cbn} e_2$ *then* $\mathcal{O}[\![e_1]\!] \longmapsto_{ml} \mathcal{O}[\![e_2]\!]$.

In order to characterize the connection between the source calculus and the image precisely, we work towards the definition of an inverse translation. The first step is the characterization of the image of the translation.

Lemma 8 *The grammar in Figure 12 characterizes the closure of the image of the standard call-by-name encoding under* $=_{ml}$.

Figure 13 defines the inverse translation.

Lemma 9 *Suppose* $\Gamma \vdash O : \hat{T}$. *Then there exists a constructor context* Δ *and a term context* Θ *such that* $\Delta; \Theta \vartriangleright \mathcal{O}^{-1}[\![O]\!] : \mathcal{T}^{-1}[\![\hat{T}]\!]$.

Theorem 8 *Suppose* $\Gamma \vdash O_1 : \hat{T}$ *and* $O_1 \longmapsto_{ml} O_2$ *then* $\mathcal{O}^{-1}[\![O_1]\!] =_\beta \mathcal{O}^{-1}[\![O_2]\!]$.

The reduction rule μ_3 is the culprit for having $=_\beta$ instead of $=_{std-cbn}$ or even $\to_{std-cbn}$.

Objects

$$
\begin{aligned}
\mathcal{O}^{-1}[\![x : \hat{T} : *^c]\!] &= x \\
\mathcal{O}^{-1}[\![\mathsf{unit}\ W]\!] &= \mathcal{W}^{-1}[\![W]\!] \\
\mathcal{O}^{-1}[\![\mathsf{let}\ x : \hat{W} \Leftarrow O_1\ \mathsf{in}\ O_2]\!] &= (\lambda x : \mathcal{V}^{-1}[\![\hat{W}]\!].\mathcal{O}^{-1}[\![O_2]\!])\texttt{@}\mathcal{O}^{-1}[\![O_1]\!] \\
\mathcal{O}^{-1}[\![x\ O]\!] &= x\texttt{@}\mathcal{O}^{-1}[\![O]\!] \\
\mathcal{O}^{-1}[\![x\ \hat{W}]\!] &= x\texttt{@}\mathcal{V}^{-1}[\![\hat{W}]\!] \\
\mathcal{O}^{-1}[\![W\ O]\!] &= \mathcal{W}^{-1}[\![W]\!]\texttt{@}\mathcal{O}^{-1}[\![O]\!] \\
\mathcal{O}^{-1}[\![W\ \hat{W}]\!] &= \mathcal{W}^{-1}[\![W]\!]\texttt{@}\mathcal{V}^{-1}[\![\hat{W}]\!]
\end{aligned}
$$

Values

$$
\begin{aligned}
\mathcal{W}^{-1}[\![\lambda x : \hat{T}.O]\!] &= \lambda x : \mathcal{T}^{-1}[\![\hat{T}]\!].\mathcal{O}^{-1}[\![O]\!] \\
\mathcal{W}^{-1}[\![\lambda \alpha : \hat{K}.O]\!] &= \lambda \alpha : \mathcal{K}^{-1}[\![\hat{K}]\!].\mathcal{O}^{-1}[\![O]\!]
\end{aligned}
$$

Value constructors

$$
\begin{aligned}
\mathcal{V}^{-1}[\![\alpha]\!] &= \alpha \\
\mathcal{V}^{-1}[\![\hat{T}_1 \to \hat{T}_2]\!] &= \mathcal{T}^{-1}[\![\hat{T}_1]\!] \to \mathcal{T}^{-1}[\![\hat{T}_2]\!] \\
\mathcal{V}^{-1}[\![\Pi\alpha{:}\hat{K}.\,\hat{T}]\!] &= \forall \alpha : \mathcal{K}^{-1}[\![\hat{K}]\!].\mathcal{T}^{-1}[\![\hat{T}]\!] \\
\mathcal{V}^{-1}[\![\lambda \alpha : \hat{K}.\hat{W}]\!] &= \lambda \alpha : \mathcal{K}^{-1}[\![\hat{K}]\!].\mathcal{V}^{-1}[\![\hat{W}]\!] \\
\mathcal{V}^{-1}[\![\hat{W}_1\ \hat{W}_2]\!] &= \mathcal{V}^{-1}[\![\hat{W}_1]\!]\texttt{@}\mathcal{V}^{-1}[\![\hat{W}_2]\!]
\end{aligned}
$$

Computation constructors

$$
\begin{aligned}
\mathcal{T}^{-1}[\![\alpha]\!] &= \alpha \\
\mathcal{T}^{-1}[\![\mathsf{M}\ \hat{W}]\!] &= \mathcal{V}^{-1}[\![\hat{W}]\!] \\
\mathcal{T}^{-1}[\![\lambda \alpha : \hat{K}.\hat{T}]\!] &= \lambda \alpha : \mathcal{K}^{-1}[\![\hat{K}]\!].\mathcal{T}^{-1}[\![\hat{T}]\!] \\
\mathcal{T}^{-1}[\![\hat{T}\ \hat{W}]\!] &= \mathcal{T}^{-1}[\![\hat{T}]\!]\texttt{@}\mathcal{V}^{-1}[\![\hat{W}]\!]
\end{aligned}
$$

Kinds

$$
\begin{aligned}
\mathcal{K}^{-1}[\![*^v]\!] &= \Omega \\
\mathcal{K}^{-1}[\![\hat{K}_1 \to \hat{K}_2]\!] &= \mathcal{K}^{-1}[\![\hat{K}_1]\!] \Rightarrow \mathcal{K}^{-1}[\![\hat{K}_2]\!]
\end{aligned}
$$

Fig. 13. Inverse of the standard call-by-name translation

Call-by-value translation of constructors

$$\mathcal{T}^V[\![\sigma_1 \to \sigma_2]\!] = \mathsf{M}\ (\mathcal{V}^V[\![\sigma_1]\!] \to \mathcal{T}^V[\![\sigma_2]\!]) \qquad \mathcal{V}^V[\![\sigma_1 \to \sigma_2]\!] = \mathcal{V}^V[\![\sigma_1]\!] \to \mathcal{T}^V[\![\sigma_2]\!]$$

Call-by-value translation of terms

$$\begin{aligned}
\mathcal{O}^V[\![x]\!] &= \mathsf{unit}\ x \\
\mathcal{O}^V[\![\lambda x:\sigma.e]\!] &= \mathsf{unit}\ (\lambda x:\mathcal{V}^V[\![\sigma]\!].\mathcal{O}^V[\![e]\!]) \\
\mathcal{O}^V[\![e_1^{\sigma_2 \to \sigma_1} @ e_2]\!] &= \mathsf{let}\ y_1 : \mathcal{V}^V[\![\sigma_2 \to \sigma_1]\!] \Leftarrow \mathcal{O}^V[\![e_1]\!]\ \mathsf{in}\ \mathsf{let}\ y_2 : \sigma_2 \Leftarrow \mathcal{O}^V[\![e_2]\!]\ \mathsf{in}\ y_1\ y_2
\end{aligned}$$

Fig. 14. Standard call-by-value encoding of F_ω

3.3.2 Standard Call-by-Value Encoding

Next, we consider a call-by-value translation. The changes with respect to the call-by-name version are minimal: the translation of the function constructor changes because call-by-value functions take values as arguments. This causes a reclassification of x as a value and changes in the translations of (value) abstraction and application. Figure 14 gives those parts that have changed with respect to Fig. 11. The MTS specification for the image of the call-by-value translation changes only the rule for value abstractions (with respect to the call-by-name specification $\mathcal{R}_{Fcbn}$).

$$\mathcal{R}_{Fcbv} = \{(\Box^v, \Box^v, \Box^v), [*^v, *^c], [\Box^v, *^c]\}$$

Again, we can prove our two standard results.

Lemma 10 *For a constructor context $\Delta = \alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n$ let $G_\Delta = \alpha_1 : \mathcal{K}[\![\kappa_1]\!], \ldots, \alpha_n : \mathcal{K}[\![\kappa_n]\!]$. For a term context $\Theta = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ let $G_\Theta = x_1 : \mathcal{V}^V[\![\sigma_1]\!], \ldots, x_n : \mathcal{V}^V[\![\sigma_n]\!]$.*

(i) If $\Delta \rhd \sigma : \kappa$ then $G_\Delta \vdash \mathcal{V}^V[\![\sigma]\!] : \mathcal{K}[\![\kappa]\!]$.

(ii) If $\Delta; \Theta \rhd e : \sigma$ then $G_\Delta, G_\Theta \rhd \mathcal{O}^V[\![e]\!] : \mathcal{T}^V[\![\sigma]\!]$.

Theorem 9 *Suppose $\Delta; \Theta \rhd e_1 : \sigma_1$ and $e_1 \to_{std-cbv} e_2$ then $\mathcal{O}^V[\![e_1]\!] \longmapsto_{ml} \mathcal{O}^V[\![e_2]\!]$.*

See Appendix F.2 for the definition of $\to_{std-cbv}$.

3.3.3 ML-style Call-by-Value Encoding

It is also possible to consider an ML-style variant of the call-by-value semantics. The difference to the standard call-by-value semantics lies in the fact that ML performs computations under type abstraction. The ML-style translation presents only a few changes with respect to the standard call-by-value encoding in Fig. 14. Therefore, Figure 15 presents only the changes, which amount to the cases involving the constructor $\forall\alpha : \kappa.\sigma$.

We can prove our two standard results.

Lemma 11 *For a constructor context $\Delta = \alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n$ let $G_\Delta = \alpha_1 : \mathcal{K}[\![\kappa_1]\!], \ldots, \alpha_n : \mathcal{K}[\![\kappa_n]\!]$. For a term context $\Theta = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ let $G_\Theta = x_1 : \mathcal{V}^{MV}[\![\sigma_1]\!], \ldots, x_n : \mathcal{V}^{MV}[\![\sigma_n]\!]$.*

ML-style call-by-value translation of constructors

$$\mathcal{T}^{MV}[\![\forall\alpha:\kappa.\sigma]\!] = \Pi\alpha{:}\mathcal{K}[\![\kappa]\!].\,\mathcal{T}^{MV}[\![\sigma]\!] \qquad \mathcal{V}^{MV}[\![\forall\alpha:\kappa.\sigma]\!] = \Pi\alpha{:}\mathcal{K}[\![\kappa]\!].\,\mathcal{T}^{MV}[\![\sigma]\!]$$

ML-style call-by-value translation of terms

$$\begin{aligned} \mathcal{O}^{MV}[\![\Lambda\alpha:\kappa.e]\!] &= \lambda\alpha:\mathcal{K}[\![\kappa]\!].\mathcal{O}^{MV}[\![e]\!] \\ \mathcal{O}^{MV}[\![e^{\forall\alpha:\kappa.\sigma_1}\{\sigma_2\}]\!] &= \mathcal{O}^{MV}[\![e]\!]\ (\mathcal{T}^{MV}[\![\sigma]\!]) \end{aligned}$$

Fig. 15. ML-style call-by-value encoding of F_ω

(i) If $\Delta \triangleright \sigma : \kappa$ *then* $G_\Delta \vdash \mathcal{V}^{MV}[\![\sigma]\!] : \mathcal{K}[\![\kappa]\!]$.
(ii) If $\Delta;\Theta \triangleright e : \sigma$ *then* $G_\Delta, G_\Theta \triangleright \mathcal{O}^{MV}[\![e]\!] : \mathcal{T}^{MV}[\![\sigma]\!]$.

Theorem 10 *Suppose* $\Delta;\Theta \triangleright e_1 : \sigma_1$ *and* $e_1 \rightarrow_{ml-cbv} e_2$ *then* $\mathcal{O}^{MV}[\![e_1]\!] =_{ml} \mathcal{O}^{MV}[\![e_2]\!]$.

See Appendix F.3 for the definition of $\rightarrow_{ml-cbv}$.

We can strengthen the result if we restrict the source language to only admit value polymorphism as in

$$e ::= \ldots \mid \Lambda\alpha:\kappa.v$$

In this case, the reductions are compatible, again.

Theorem 11 *Suppose* e_1 *adheres to the value restriction,* $\Delta;\Theta\triangleright e_1 : \sigma_1$*, and* $e_1 \rightarrow_{ml-cbv} e_2$ *then* $\mathcal{O}^{MV}[\![e_1]\!] \longmapsto_{ml} \mathcal{O}^{MV}[\![e_2]\!]$.

3.3.4 ML-style Call-by-Name Encoding

It is possible to define a call-by-name variant of the ML-style call-by-value encoding considered above. Harper and Lillibridge [24] show that this semantics actually coincides with the standard call-by-name semantics on closed terms of base type. Therefore, we do not consider it here in any depth.

3.3.5 FLINT

Shao [45] has considered an extension of F_ω with the ML-style call-by-value semantics as an intermediate language for compilation. The main differences to our system (Sec. 3.3.3) are the following.

(i) The term language of FLINT is restricted to A-normal form. In A-normal form, the results of all non-trivial computatations have to be named [16]. Therefore, FLINT restricts function application to the form $\mathsf{let}\ x = v @ v\ \mathsf{in}\ e$ (this construct does not generate polymorphism) and relies on a preceding transformation to A-normal form. This restriction slightly simplifies the translation of applications:

$$\mathcal{O}^{MV}[\![\mathsf{let}\ x = v_1^{\sigma_2\to\sigma_1} @ v_2\ \mathsf{in}\ e]\!] = \mathsf{let}\ x : \mathcal{V}^{MV}[\![\sigma_1]\!] \Leftarrow \mathcal{O}'^{MV}[\![v_1]\!]\ (\mathcal{O}'^{MV}[\![v_2]\!])\ \mathsf{in}\ \mathcal{O}^{MV}[\![e]\!]$$

where $\mathcal{O}'^{MV}[\![\cdot]\!]$ drops the leading $\mathsf{unit}\ \cdot$ from $\mathcal{O}^{MV}[\![\cdot]\!]$ restricted to values.

(ii) FLINT has several built-in primitives. These can be dealt with just as demonstrated for succ and pred in the ML encoding (see Sec. 3.2).

(iii) FLINT's type system has predicative polymorphism. It achieves this by distinguishing between constructors and types and by having an explicit type forming operator T that maps a constructor of kind Ω to a type (as introduced by Harper and Morrisett [26]).

One way to model this would be to extend the set of sorts by a sort $\Diamond$ with $\Diamond : \Box^v$ for the constructors and add a rule like

$$\frac{\Gamma \vdash A : \Diamond}{\Gamma \vdash T(A) : *^v}$$

to the MTS framework. This way, the specification would be

$$\{[*^v, *^c], [\Box^v, *^c], (\Diamond, \Diamond, \Diamond)\}$$

which is well-behaved in the current framework.

(iv) There is a new kind of sequence kinds $\kappa \otimes \kappa$ in FLINT (to express the type of a module) which is not present in MTS, but could be added without much difficulty.

4 Properties of Monadic Type Systems

The meta-theory of Monadic Type Systems is complicated by two major hurdles, namely the non-termination of ϕ-reduction and the non-confluence of μ-reduction. Nevertheless, MTSs enjoy some important properties such as Subject Reduction and Classification for injective specifications. Besides, a large class of MTSs is strongly normalizing w.r.t. $\beta\iota\mu$-reduction and has decidable type-checking.

4.1 Basic properties

Basic properties such as the Substitution Lemma or the Generation Lemma, which do not rely on confluence nor normalization, are proved in exactly the same way as for Pure Type Systems.

The first Lemma states that judgements are closed under substitution—substitution is extended to pseudo-contexts in the obvious way.

Lemma 12 (Substitution) *Assume* $\Gamma, x : A, \Delta \vdash B : C$ *and* $\Gamma \vdash a : A$. *Then also* $\Gamma, \Delta\{x := a\} \vdash B\{x := a\} : C\{x := a\}$.

Proof By induction on the derivation of $\Gamma, x : A, \Delta \vdash B : C$. ■

The second Lemma states that introducing new assumptions do not affect derivability—we write $\Delta \supseteq \Gamma$ if $(x : A) \in \Delta$ whenever $(x : A) \in \Gamma$.

Lemma 13 (Thinning) *If* $\Gamma \vdash A : B$ *and* $\Delta \supseteq \Gamma$ *is legal then* $\Delta \vdash A : B$.

Proof By induction on the derivation of $\Gamma \vdash A : B$. ■

Conversely, one can remove unused assumptions without affecting derivability. However, this result is surprinsingly difficult to prove and is postponed until the end of this Section.

$$
\begin{array}{lll}
\Gamma \vdash s : C & \Rightarrow & \exists (s, s') \in \mathcal{A}.\ C =_{ml} s' \\
\Gamma \vdash x : C & \Rightarrow & \exists s \in \mathcal{S}, D \in \mathcal{T}.\ C =_{ml} D \quad \wedge \quad (x : D) \in \Gamma \\
 & & \quad \wedge \quad \Gamma \vdash D : s \quad \wedge \quad x \in \mathcal{V}^s \\
\Gamma \vdash \lambda x{:}A.\, b : C & \Rightarrow & \exists s \in \mathcal{S}, B \in \mathcal{T}.\ C =_{ml} \Pi x{:}A.\, B \quad \wedge \quad \Gamma, x : A \vdash b : B \\
 & & \quad \wedge \quad \Gamma \vdash \Pi x{:}A.\, B : s \\
\Gamma \vdash \Pi x{:}A.\, B : C & \Rightarrow & \exists (s_1, s_2, s_3) \in \mathcal{R}.\ C =_{ml} s_3 \quad \wedge \quad \Gamma \vdash A : s_1 \\
 & & \quad \wedge \quad \Gamma, x : A \vdash B : s_2 \\
\Gamma \vdash F\ a : C & \Rightarrow & \exists x \in \mathcal{V}, A, B \in \mathcal{T}.\ C =_{ml} B\{x := a\} \\
 & & \quad \wedge \quad \Gamma \vdash F : \Pi x{:}A.\, B \quad \wedge \quad \Gamma \vdash a : A \\
\Gamma \vdash \mathsf{fix}\ x{:}A.\, b : C & \Rightarrow & \exists B \in \mathcal{T}. C =_{ml} A \quad \wedge \quad A \equiv \mathsf{M}\ B \quad \wedge \quad \Gamma, x : A \vdash b : A \\
\Gamma \vdash \mathsf{let}\ x{:}A \Leftarrow M\ \mathsf{in}\ N : C & \Rightarrow & \exists B \in \mathcal{T}.\ C =_{ml} \mathsf{M}\ B \quad \wedge \quad \Gamma, x : A \vdash N : \mathsf{M}\ B \\
 & & \quad \wedge \quad \Gamma \vdash M : \mathsf{M}\ A \\
\Gamma \vdash \mathsf{unit}\ M : C & \Rightarrow & \exists B \in \mathcal{T}.\ C =_{ml} \mathsf{M}\ B \quad \wedge \quad \Gamma \vdash M : B \\
 & & \quad \wedge \quad \Gamma \vdash \mathsf{M}\ B : *^c \\
\Gamma \vdash \mathsf{M}\ A : C & \Rightarrow & C =_{ml} *^c \quad \wedge \quad \exists s \in \{*^c, *^v\}.\ \Gamma \vdash A : s \\
\Gamma \vdash \mathsf{nat} : C & \Rightarrow & C =_{ml} *^v \\
\Gamma \vdash \lceil n \rceil : C & \Rightarrow & C =_{ml} \mathsf{nat} \\
\Gamma \vdash \mathsf{succ}\ M : C & \Rightarrow & C =_{ml} \mathsf{nat} \quad \wedge \quad \Gamma \vdash M : \mathsf{nat} \\
\Gamma \vdash \mathsf{pred}\ M : C & \Rightarrow & C =_{ml} \mathsf{nat} \quad \wedge \quad \Gamma \vdash M : \mathsf{nat} \\
\Gamma \vdash \mathsf{if0}\ M\ M_1\ M_2 : C & \Rightarrow & \exists B \in \mathcal{T}.\ C =_{ml} \mathsf{M}\ B \quad \wedge \quad \Gamma \vdash M : \mathsf{nat} \\
 & & \quad \wedge \quad \Gamma \vdash M_1 : B \quad \wedge \quad \Gamma \vdash M_2 : B
\end{array}
$$

Fig. 16. Generation

The next Lemma provides an analysis of the possible derivations of a judgement. The lemma is used extensively throughout the paper.

Lemma 14 (Generation) *See Figure 16.*

Proof By induction on the structure of derivations using the Thinning Lemma. ■

The last result states that types are correct, in the sense that if $A : B$ then either B is a sort or $B : s$ for some sort s.

Lemma 15 (Correctness of types) *If* $\Gamma \vdash A : B$ *then either* $B \in \mathcal{S}$ *or* $\exists s \in \mathcal{S}.\ \Gamma \vdash B : s$.

Proof By induction on $\Gamma \vdash A : B$. ∎

4.2 Strong normalization

Fixpoint reduction, i.e. ϕ-reduction is not normalizing hence we cannot expect a MTS to be *ml*-strongly normalizing. However MTSs may still be $\beta\iota\mu$-strongly normalizing. The purpose of this Subsection is to define such a class of MTSs. The actual definition of the class is determined by the technique used to prove strong normalisation rather than by any intrinsic criterion. If we view MTS-specifications as a subclass of PTS-specifications, this class corresponds to those specifications which may be embedded in Barendregt's λ-cube [3, 4].

Definition 12 Let $\mathbf{S} = (\mathcal{S}, *^v, *^c, \mathcal{A}, \mathcal{R})$ be a MTS-specification.

(i) The set $\mathcal{S}^\top$ of *top-sorts* is defined as $\{s \in \mathcal{S} \mid \forall s' \in \mathcal{S}.\ (s, s') \notin \mathcal{A}\}$.

(ii) The set $\mathcal{S}^\perp$ of *bottom-sorts* is defined as $\{s \in \mathcal{S} \mid \forall s' \in \mathcal{S}.\ (s,' s) \notin \mathcal{A}\}$.

(iii) $\mathbf{S}$ is *cubic* if
 (a) $\mathcal{S} = \mathcal{S}^\top \cup \mathcal{S}^\perp$,
 (b) $*^v, *^c \in \mathcal{S}^\perp$,
 (c) for every $(s_1, s_2, s_3) \in \mathcal{R}$, $s_2 \in \mathcal{S}^\perp \Leftrightarrow s_3 \in \mathcal{S}^\perp$.

Our method to prove strong normalization is to define a reduction-preserving translation from cubic MTSs to the Calculus of Constructions with Fixpoints λC_{fix} [1, 5]. The translation could be generalized to specifications that do not comply with requirement (a) of the above definition; in that case the target language would be a Logical Pure Type System with Fixpoints at the $*$ level. However, we would need to prove such systems to be strongly normalizing in order to conclude—a method is suggested in [5] but is still in need to be carried out in detail. Following standard practice, we write $\lambda\mathbf{S} \models \mathrm{SN}(\beta\iota\mu)$ iff every legal term in $\lambda\mathbf{S}$ is $\beta\iota\mu$-strongly normalizing.

Theorem 13 *If* $\mathbf{S}$ *is a cubic specification, then* $\lambda\mathbf{S} \models \mathrm{SN}(\beta\iota\mu)$.

Proof For the sake of simplicity, we consider a slightly modified syntax where $\mathsf{if0}$ expressions are labelled by their result types, i.e. are of the form $\mathsf{if0}_A\ M\ M_1\ M_2$. This enables us to encode $\mathsf{if0}$ impredicatively. For the same reason, we only consider the rule

$\mathsf{pred}\ (\mathsf{succ}\ M) \to_\iota M$ for M a numeral. The translation $\lceil . \rceil$ defined inductively as follows:

$$
\begin{aligned}
\lceil x \rceil &= x \\
\lceil s \rceil &= s \\
\lceil \lambda x : A.t \rceil &= \lambda x : \lceil A \rceil . \lceil t \rceil \\
\lceil \Pi x : A.B \rceil &= \Pi x : \lceil A \rceil . \lceil B \rceil \\
\lceil t\ u \rceil &= \lceil t \rceil\ \lceil u \rceil \\
\lceil \mathsf{let}\ x{:}\, A \Leftarrow N\ \mathsf{in}\ M \rceil &= (\lambda x{:}\, \lceil A \rceil .\ \lceil M \rceil)\ \lceil N \rceil \\
\lceil \mathsf{unit}\ t \rceil &= \lceil t \rceil \\
\lceil \mathsf{M}\ A \rceil &= \lceil A \rceil \\
\lceil \mathsf{fix}\ x{:}\, A.\ M \rceil &= \mathsf{fix}\ x{:}\, \lceil A \rceil .\ \lceil M \rceil \\
\lceil \mathsf{nat} \rceil &= \Pi \alpha{:}\, *.\ \alpha \to (\alpha \to \alpha) \to \alpha \\
\lceil \lceil n \rceil \rceil &= \underline{n} \\
\lceil \mathsf{succ}\ M \rceil &= \lambda \alpha{:}\, *.\ \lambda x{:}\, \alpha.\ \lambda f{:}\, \alpha \to \alpha.\ f\ (\lceil M \rceil\ \alpha\ x\ f) \\
\lceil \mathsf{pred}\ M \rceil &= \underline{\mathsf{pred}}\ \lceil M \rceil \\
\lceil \mathsf{if0}_A\ M\ M_1\ M_2 \rceil &= \lceil M \rceil\ \lceil A \rceil\ \lceil M_1 \rceil\ (\lambda \_{:}\, \lceil A \rceil .\ \lceil M_2 \rceil)
\end{aligned}
$$

where $\underline{n}$ and $\underline{\mathsf{pred}}$ are the impredicative encodings of n and predecessor respectively. [20, page 90]. The translation $\lceil . \rceil$ preserves conversion and maps legal terms of $\lambda\mathbf{S}$ to legal terms of λC_{fix}. In addition $\lceil . \rceil$ maps infinite $\beta\mu\iota$-reduction sequences with infinitely many $\beta\mu$-steps to infinite $\beta\beta'$-reduction sequences, where β'-reduction is defined by the contraction rule:

$$
P\ ((\lambda x{:}\, A.\ Q)\ R) \quad \to_{\beta'} \quad (\lambda x{:}\, A.\ P\ Q)\ R \quad \text{provided } x \notin \mathrm{FV}(P)
$$

Conclude by proving that $\mathrm{SN}(\beta\beta') = \mathrm{SN}(\beta)$, see Appendix, and by noting that an infinite $\beta\mu\iota$-reduction sequence must contain infinitely many $\beta\mu$-steps. ■

4.3 Confluence

μ-reduction is not confluent on pseudo-terms, as illustrated by the following example: [6]

$$
\begin{aligned}
\mathsf{let}\ x{:}\, A \Leftarrow M\ \mathsf{in}\ x &=_\mu \mathsf{let}\ x{:}\, A \Leftarrow (\mathsf{let}\ y{:}\, B \Leftarrow M\ \mathsf{in}\ (\mathsf{unit}\ y))\ \mathsf{in}\ x \\
&=_\mu \mathsf{let}\ y{:}\, B \Leftarrow M\ \mathsf{in}\ (\mathsf{let}\ x{:}\, A \Leftarrow (\mathsf{unit}\ y)\ \mathsf{in}\ x) \\
&=_\mu \mathsf{let}\ x{:}\, B \Leftarrow M\ \mathsf{in}\ x
\end{aligned}
$$

[6] However, we believe that the failure of confluence for monadic reduction is a consequence of the choice of primitives rather than of the intrinsic nature of monadic languages and that alternative formulations of the monadic syntax would not suffer from non-confluence.

The failure of confluence for ml-reduction requires a careful elaboration of the meta-theoretical results. The solution adopted in this paper is to 'complete' ml-reduction. Put it otherwise, we consider an extension $ml+$ of ml-reduction that is confluent and that has the same reflexive-symmetric-transitive closure as ml-reduction. An alternative would be to follow [18] and define a notion of erased pseudo-term and of erased reduction, but our approach has the advantage of being more concise.

Definition 14 The notion of κ-reduction is defined by the contraction rules

$$\mathsf{let}\ x{:}A \Leftarrow M\ \mathsf{in}\ N \rightarrow_{\kappa} \mathsf{let}\ x{:}\bullet \Leftarrow M\ \mathsf{in}\ N$$

The notion of $ml+$-reduction is defined as the union of ml and κ.

The notions of reduction ml and $ml+$ yield the same convertibility relation.

Lemma 16 $M =_{ml+} N \Leftrightarrow M =_{ml} N$

Proof Only the direct implication is interesting. To prove it, first show by induction on the structure of M that $M \rightarrow_{\kappa} N$ implies $M =_{ml} N$. Then proceed by induction on the derivation of $M =_{ml+} N$. ∎

Next we prove that $ml+$ is confluent. We start with some preliminary results.

Theorem 15 $\mathrm{SN}(\mu) = \mathcal{T}$.

Proof Using Finiteness of Developments. More precisely, define the set $\mathcal{U}$ of *underlined λ-terms* by the abstract syntax:

$$\mathcal{U} = \mathcal{V} \mid \bullet \mid \mathcal{U}\,\mathcal{U} \mid \lambda\mathcal{V}.\,\mathcal{U} \mid (\underline{\lambda}\mathcal{V}.\,\mathcal{U})\,\mathcal{U}$$

Then define the notion $\underline{\beta}$ of *underlined β-reduction* by the contraction rule

$$(\underline{\lambda}x.\,M)\,N \quad \rightarrow_{\underline{\beta}} \quad M\{x := N\}$$

and the notion of reduction $\underline{\beta'}$ by the contraction rule

$$P\,((\underline{\lambda}x.\,Q)\ R) \quad \rightarrow_{\underline{\beta'}} \quad (\underline{\lambda}x.\,P\,Q)\ R$$

provided $x \notin \mathrm{FV}(P)$. One possible formulation of Finiteness of Developments states $\mathcal{U} = \mathrm{SN}(\underline{\beta})$. Using an analysis of the possible infinite $\underline{\beta\beta'}$-reduction sequences, one can prove $\mathcal{U} = \mathrm{SN}(\underline{\beta\beta'})$, see Appendix. To prove our theorem, define a translation $[.] : \mathcal{T} \rightarrow \mathcal{U}$

as follows:

$$\begin{array}{lll}
[x] & = x & x \in \mathcal{V} \\
[s] & = \bullet & s \in \mathcal{S} \\
[\lambda x{:}\, A.\ t] & = \mathsf{K}\ (\lambda x.\ [t])\ [A] & \\
[\Pi x{:}\, A.\ B] & = (\lambda x.\ [B])\ [A] & \\
[t\ u] & = \bullet\ [t]\ [u] & \\
[\mathsf{let}\ x{:}\, A \Leftarrow N\ \mathsf{in}\ M] & = (\underline{\lambda} x.\ \mathsf{K}\ [M]\ [A])\ [N] & \\
[\mathsf{unit}\ t] & = [t] & \\
[\mathsf{M}\ A] & = [A] & \\
[\mathsf{fix}\ x{:}\, A.\ M] & = \mathsf{K}\ (\lambda x.\ [M])\ [A] & \\
[\mathsf{nat}] & = \bullet & \\
[\lceil n \rceil] & = \bullet & \\
[\mathsf{succ}\ M] & = [M] & \\
[\mathsf{pred}\ M] & = [M] & \\
[\mathsf{if0}\ M\ M_1\ M_2] & = \bullet\ [M]\ [M_1]\ [M_2] &
\end{array}$$

To conclude, we only need to show that $[.]$ maps infinite sequences of μ-reductions are mapped into infinite sequences of $\underline{\beta}\beta'$-reductions, i.e. for every $P, Q \in \mathcal{T}$,

$$P \rightarrow_\mu Q \quad \Rightarrow \quad [P] \twoheadrightarrow^+_{\underline{\beta}\beta'} [Q]$$

The latter is proved by induction on $M \in \mathcal{T}$. We treat three cases:

- $P \equiv \mathsf{let}\ x{:}\, A \Leftarrow (\mathsf{unit}\ N)\ \mathsf{in}\ M$ and $Q \equiv M\{x := N\}$.

$$\begin{array}{rl}
[P] & \equiv\ (\underline{\lambda} x.\ \mathsf{K}\ [M]\ [A])\ [N] \\
& \rightarrow_{\underline{\beta}} (\underline{\lambda} x.\ [M])\ [N] \\
& \rightarrow_{\underline{\beta}} [M]\{x := [N]\} \\
& \equiv\ [M\{x := N\}] \\
& \equiv\ [Q]
\end{array}$$

- $P \equiv \mathsf{let}\ x{:}\, A \Leftarrow M\ \mathsf{in}\ (\mathsf{unit}\ x)$ and $Q \equiv M$.

$$\begin{array}{rl}
[P] & \equiv\ (\underline{\lambda} x.\ \mathsf{K}\ [x]\ [A])\ [M] \\
& \rightarrow_{\underline{\beta}} (\underline{\lambda} x.\ [x])\ [M] \\
& \rightarrow_{\underline{\beta}} [M] \\
& \equiv\ [Q]
\end{array}$$

- $P \equiv \mathsf{let}\ x_2{:}\, A_2 \Leftarrow (\mathsf{let}\ x_1{:}\, A_1 \Leftarrow M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3$ and $Q \equiv \mathsf{let}\ x_1{:}\, A_1 \Leftarrow M_1\ \mathsf{in}\ (\mathsf{let}\ x_2{:}\, A_2 \Leftarrow M_2\ \mathsf{in}\ M_3)$ assuming $x_1 \notin \mathrm{FV}(M_3) \cup \mathrm{FV}(A_2)$. We have

$$[P] \ \equiv\ (\underline{\lambda} x_2.\ \mathsf{K}\ [M_3]\ [A_2])\ ((\underline{\lambda} x_1.\ \mathsf{K}\ [M_2]\ [A_1])\ [M_1])$$

$$
\begin{aligned}
&\to_{\underline{\beta}'} (\underline{\lambda}x_1.\, (\underline{\lambda}x_2.\, \mathsf{K}\ [M_3]\ [A_2])\ (\mathsf{K}\ [M_2]\ [A_1]))\ [M_1] \\
&\to_{\underline{\beta}'} (\underline{\lambda}x_1.\, \mathsf{K}\ ((\underline{\lambda}x_2.\, \mathsf{K}\ [M_3]\ [A_2])\ [M_2])\ [A_1])[M_1] \\
&\equiv\ (\underline{\lambda}x_1.\, \mathsf{K}\ [\mathsf{let}\ x_2{:}A_2 \Leftarrow M_2\ \mathsf{in}\ M_3]\ [A_1])\ [M_1] \\
&\equiv\ [\mathsf{let}\ x_1{:}A_1 \Leftarrow M_1\ \mathsf{in}\ (\mathsf{let}\ x_2{:}A_2 \Leftarrow M_2\ \mathsf{in}\ M_3)] \\
&\equiv\ [Q]
\end{aligned}
$$

Other cases follow directly from the induction hypothesis. ■

Proposition 16 *$\mu\kappa$-reduction is confluent.*

Proof $\mu\kappa$-reduction is locally confluent, hence by Newman's Lemma it is enough to show that $\mathrm{SN}(\mu\kappa) = \mathcal{T}$. This may be proved by noting that:

(i) [.] maps κ-redexes to κ'-redexes where κ' is the reduction rule

$$M \to_{\kappa'} \bullet \qquad \text{if } M \not\equiv \bullet$$

(ii) κ' may be postponed and is strongly normalising.

■

Theorem 17 (Confluence) *$ml+$ is confluent.*

Theorem 17 is useful in proving several subsequent results, especially Subject Reduction. However, there does not seem to be any direct method from which to derive confluence of ml-reduction on legal terms, even if we assume the specification to be cubic. The situation is to be contrasted with that of PTSs with $\beta\eta$-conversion [18], where confluence of $\beta\eta$-reduction on legal terms is derived from normalisation and confluence. In our case we do not have ml-normalization so we cannot conclude.

4.4 Subject Reduction

Subject Reduction is a fundamental property of type systems: from a practical point of view, it ensures that types are closed under reduction. From a theoretical point of view, it is used throughout the metatheory of type systems, e.g. in consistency proofs and in operational semantics. As expected, the failure of confluence for ml-reduction yields slight complications in the proof of Subject Reduction. Nevertheless the proof of Subject Reduction for $\iota\phi$-reduction proceeds as usual.

Theorem 18 ($\iota\phi$-subject reduction) $\Gamma \vdash A : B,\ A \to_{\iota\phi} A' \quad \Rightarrow \quad \Gamma \vdash A' : B$

Proof Prove by simultaneous induction on the derivation of $\Gamma \vdash M : A$:

- $M \to_{\iota\phi} N \quad \Rightarrow \quad \Gamma \vdash N : A$
- if $\Gamma \to_{\iota\phi} \Delta \quad \Rightarrow \quad \Delta \vdash M : B$

(Recall that $\to_{\iota\phi}$ is extended to contexts by the clause $A \to_{\iota\phi} B \quad \Rightarrow \quad \Gamma, x : A, \Delta \to_{\iota\phi} \Gamma, x{:}B, \Delta$.) ■

The proof of Subject Reduction for $\beta\mu$-reduction requires a preliminary result, known as the Key Lemma, which follows directly from Theorem 17.

Lemma 17 (Key Lemma)

- *If* $\Pi x{:}A.\,B =_{ml} \Pi x{:}A'.\,B'$ *then* $A =_{ml} A'$ *and* $B =_{ml} B'$.
- *If* $\mathsf{M}\ A =_{ml} \mathsf{M}\ A'$ *then* $A =_{ml} A'$.

Proof The second step of both proofs follows from confluence of $\rightarrow_{ml+}$.

$$\begin{array}{rcl} \Pi x{:}A.\,B =_{ml} \Pi x{:}C.\,D & \Rightarrow & \Pi x{:}A.\,B =_{ml+} \Pi x{:}C.\,D \\ & \Rightarrow & A =_{ml+} C \ \wedge\ B =_{ml+} D \\ & \Rightarrow & A =_{ml} C \ \wedge\ B =_{ml} D \end{array}$$

$$\begin{array}{rcl} \mathsf{M}\ A =_{ml} \mathsf{M}\ A' & \Rightarrow & \mathsf{M}\ A =_{ml+} \mathsf{M}\ A' \\ & \Rightarrow & A =_{ml+} A' \\ & \Rightarrow & A =_{ml} A' \end{array}$$

■

Theorem 19 ($\beta\mu$-subject reduction) $\Gamma \vdash A : B,\ A \rightarrow_{\beta\mu} A' \quad \Rightarrow \quad \Gamma \vdash A' : B$

Proof Prove by simultaneous induction on the derivation of $\Gamma \vdash M : A$:

- $M \rightarrow_{\beta\mu} N \quad \Rightarrow \quad \Gamma \vdash N : A$
- if $\Gamma \rightarrow_{\beta\mu} \Delta \quad \Rightarrow \quad \Delta \vdash M : B$

(Recall that $\rightarrow_{\beta\mu}$ is extended to contexts by the clause $A \rightarrow_{\beta\mu} B \quad \Rightarrow \quad \Gamma, x : A, \Delta \rightarrow_{\beta\mu} \Gamma, x{:}B, \Delta$.) ■

The next result is a mild strengthening of the Key Lemma which is useful in the proof of strengthening.

Lemma 18

- *If* $s =_{ml} s'$ *then* $s \equiv s'$.
- *If* $A =_{ml} \mathsf{M}\ B$ *and* A *legal then* $A \twoheadrightarrow_{\beta} \mathsf{M}\ C$ *for some* $C =_{ml} B$.
- *If* $A =_{ml} s$ *and* A *legal then* $A \twoheadrightarrow_{\beta} s$.
- *If* $A =_{ml} \Pi x{:}B.\,C$ *and* A *legal then* $A \twoheadrightarrow_{\beta} \Pi x{:}D.\,E$ *with* $B =_{ml} D$ *and* $C =_{ml} E$.

An important consequence of Subject Reduction is strengthening, which is proved along the same lines as in [18].

Definition 20 $\mathbf{S}$ preserves sorts if

$$\Gamma \vdash A : s,\ \Gamma \vdash A' : s',\ A =_{ml} A' \quad \Rightarrow \quad s \equiv s'$$

The above technical assumption ensures that strengthening holds.

Theorem 21 *If* $\mathbf{S}$ *preserves sorts, then* $\mathbf{S}$ *satisfies strenghtening, i.e.*

$$\Gamma_1, x : A, \Gamma_2 \vdash b : B,\ x \notin \mathrm{FV}(\Gamma_2) \cup \mathrm{FV}(b) \cup \mathrm{FV}(B) \quad \Rightarrow \quad \Gamma_1, \Gamma_2 \vdash b : B$$

Proof First show that

$$\Gamma_1, x : A, \Gamma_2 \vdash b : B,\ x \notin \mathrm{FV}(\Gamma_2) \cup \mathrm{FV}(b) \cup \mathrm{FV}(B) \quad \Rightarrow \quad \exists B' \in \mathcal{T}.\ B =_{ml} B' \wedge \Gamma_1, \Gamma_2 \vdash b : B' \quad (*)$$

Then assume

$$\Gamma_1, x : A, \Gamma_2 \ \vdash\ b : B\ (\&),\ \ x \notin \mathrm{FV}(\Gamma_2) \cup \mathrm{FV}(b) \cup \mathrm{FV}(B)$$

Hence there exists $C =_{ml} B$ such that

$$\Gamma_1, \Gamma_2 \ \vdash\ b : C$$

By Correctness of Types on (&) we find that $B \in \mathcal{S}$ or $\Gamma_1, x : A, \Gamma_2 \ \vdash\ B : s$ for some $s \in \mathcal{S}$. In the second case, apply $(*)$ to find $E =_{ml} s$ such that

$$\Gamma_1, \Gamma_2 \ \vdash\ B : E$$

By Lemma 18 and Subject Reduction,

$$\Gamma_1, \Gamma_2 \ \vdash\ B : s$$

By (conversion)

$$\Gamma_1, \Gamma_2 \ \vdash\ b : B$$

In the first case, apply the Lemma 18 to conclude. In both cases, we obtain

$$\Gamma_1, \Gamma_2 \ \vdash\ b : B$$

To conclude the proof, we prove $(*)$ by induction on the structure of derivations.

- Assume the last rule is

$$\frac{\Gamma, y : C \ \vdash\ N : D \quad \Gamma \ \vdash\ (\Pi x{:}C.\ D) : s}{\Gamma \ \vdash\ \lambda y{:}C.\ N : \Pi y{:}C.\ D}$$

  with $\Gamma \equiv \Gamma_1, x : A, \Gamma_2$ and $x \notin \mathrm{FV}(\Gamma_2) \cup \mathrm{FV}(\lambda y{:}C.\ N) \cup \mathrm{FV}(\Pi y{:}C.\ D)$. By induction hypothesis, there exists E, F in $\mathcal{T}$ such that $E =_{ml} D$ and $F =_{ml} s$ and such that

$$\Gamma_1, \Gamma_2, y : C \ \vdash\ M : E$$
$$\Gamma_1, \Gamma_2 \ \vdash\ \Pi y{:}C.\ D : F$$

  By Generation, there exists $(s_1, s_2, s_3) \in \mathcal{R}$ such that

$$\Gamma_1, \Gamma_2 \ \vdash\ C : s_1$$
$$\Gamma_1, \Gamma_2, y : C \ \vdash\ D : s_2$$

  By Correctness of Types, there exists $s' \in \mathcal{S}$ such that

$$\Gamma_1, \Gamma_2, y : C \ \vdash\ E : s'$$

By Preservation of Sorts, $s' \equiv s_2$ and hence by (product)

$$\Gamma_1, \Gamma_2 \vdash \Pi y{:}C.\, D : s_3$$

By (abstraction),

$$\Gamma_1, \Gamma_2 \vdash \lambda y{:}C.\, M : \Pi y{:}C.\, D$$

- Assume the last rule is

$$\frac{\Gamma \vdash M : \mathsf{M}\ C \quad \Gamma, y : C \vdash N : \mathsf{M}\ D}{\Gamma \vdash \mathsf{let}\ y{:}C \Leftarrow M\ \mathsf{in}\ N : \mathsf{M}\ D}$$

with $\Gamma \equiv \Gamma_1, x : A, \Gamma_2$ and $x \notin \mathrm{FV}(\Gamma_2) \cup \mathrm{FV}(\mathsf{let}\ y{:}C \Leftarrow M\ \mathsf{in}\ N) \cup \mathrm{FV}(D)$. By induction hypothesis, there exists E, F in $\mathcal{T}$ such that $E =_{ml} \mathsf{M}\ C$ and $F =_{ml} \mathsf{M}\ D$ and such that

$$\Gamma_1, \Gamma_2 \vdash M : E$$

$$\Gamma_1, \Gamma_2, y : C \vdash N : F$$

By Correctness of types, $\Gamma_1, \Gamma_2 \vdash C : s$. By Extended Uniqueness of Types $s =_{ml} *^x$. By Lemma 18, $s \equiv *^x$ and hence by (monad) $\Gamma_1, \Gamma_2 \vdash \mathsf{M}\ C : *^c$. By (conversion),

$$\Gamma_1, \Gamma_2 \vdash M : E$$

By Lemma 18, there exists $G \in \mathcal{T}$ such that $F \twoheadrightarrow_\beta \mathsf{M}\ G$. By β-Subject Reduction,

$$\Gamma_1, \Gamma_2, y : C \vdash N : \mathsf{M}\ G$$

By (let),

$$\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ y{:}C \Leftarrow M\ \mathsf{in}\ N : \mathsf{M}\ G$$

- Assume the last rule is

$$\frac{\Gamma, y : \mathsf{M}\ B \vdash M : \mathsf{M}\ B}{\Gamma \vdash \mathsf{fix}\ y{:}\mathsf{M}\ B.\, M : \mathsf{M}\ B}$$

with $\Gamma \equiv \Gamma_1, x : A, \Gamma_2$ and $x \notin \mathrm{FV}(\Gamma_2) \cup \mathrm{FV}(\mathsf{fix}\ y{:}\mathsf{M}\ B.\, M)$. By induction hypothesis, there exists $E \in \mathcal{T}$ such that $E =_{ml} \mathsf{M}\ B$ and

$$\Gamma_1, \Gamma_2, y : \mathsf{M}\ B \vdash M : E$$

By Correctness of Types and Weakening,

$$\Gamma_1, \Gamma_2 \vdash \mathsf{M}\ B : *^c$$

By (conversion),

$$\Gamma_1, \Gamma_2, y : \mathsf{M}\ B \vdash M : \mathsf{M}\ B$$

and by (fixpoint),

$$\Gamma_1, \Gamma_2 \vdash \mathsf{fix}\ y{:}\mathsf{M}\ B.\, M : \mathsf{M}\ B$$

∎

As we shall see later, most cubic specifications preserve sorts.

4.5 Uniqueness of Types and Classification

This Subsection is concerned with properties that hold for some specific classes of MTSs. The first class we consider is that of functional specifications.

Definition 22 A specification $\mathbf{S} = (\mathcal{S}, *^v, *^c, \mathcal{A}, \mathcal{R})$ is *functional* if for every $s_1, s_2, s_2', s_3, s_3' \in \mathcal{S}$,

$$\begin{array}{l} (s_1, s_2) \in \mathcal{A} \quad\quad \wedge\ (s_1, s_2') \in \mathcal{A} \quad\quad \Rightarrow s_2 \equiv s_2' \\ (s_1, s_2, s_3) \in \mathcal{R} \wedge (s_1, s_2, s_3') \in \mathcal{R} \Rightarrow s_3 \equiv s_3' \end{array}$$

Functional specifications enjoy uniqueness of types.

Lemma 19 *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash M : B$ *then* $A =_{ml} B$.

Proof By induction on the structure of derivations. ■

Uniqueness of Types may be used to prove preservation of sorts.

Lemma 20 *If* $\mathbf{S}$ *is cubic and functional then it preserves sorts.*

Proof It is enough to show that for every A, A' in β-normal form, we have

$$\Gamma \vdash A : s,\ \Gamma \vdash A' : s',\ A =_{ml} A' \quad \Rightarrow \quad s \equiv s'$$

Types in β-normal form are given by the syntax

$$\mathcal{U} = \mathcal{S} \mid \mathcal{B} \mid \Pi\mathcal{V}{:}\mathcal{U}.\,\mathcal{U} \mid \mathsf{M}\ \mathcal{U} \mid \mathsf{nat}$$

where

$$\mathcal{B} = \mathcal{V} \mid \mathcal{V}\ \mathcal{T}$$

We now reason by induction on the definition of $\mathcal{U}$, using Uniqueness of Types. ■

We now turn to Classification. For PTSs the result is traditionally expressed for the so-called injective specifications and states—among other things—that for every $M \in \mathcal{T}$ and $s, s' \in \mathcal{S}$

$$\Gamma \vdash M : A : s \quad \wedge \quad \Gamma' \vdash M : A' : s' \quad \Rightarrow \quad s \equiv s'$$

One can prove a similar result for cubic MTSs—using preservation of sorts. However several MTSs of interest are not cubic and the statement of Classification itself is overly strong for the purpose of our operational semantics. What is needed is a stratification of terms in the usual three layers: objects, types and kinds—we only focus on cubic specifications. [7]

[7] Note that one may probably relax the assumption $\mathcal{S}^{\perp} = \{*^v, *^c\}$ in the statement of the Lemma.

Lemma 21 (Classification) *Let* $\mathbf{S} = (\mathcal{S}, *^v, *^c, \mathcal{A}, \mathcal{R})$ *be a cubic specification such that* $\mathcal{S}^\perp = \{*^v, *^c\}$. *Let* $*$ *range over* $\mathcal{S}^\perp$ *and* $\Box$ *range over* $\mathcal{S}^\top$. *Define*

$$\begin{aligned}
\mathcal{O} &= \mathcal{V}^* \mid \mathcal{O}\ \mathcal{O} \mid \mathcal{O}\ \mathcal{C} \mid \lambda\mathcal{V}^*{:}\mathcal{C}.\ \mathcal{O} \mid \lambda\mathcal{V}^\Box{:}\mathcal{K}.\ \mathcal{O} \mid \lceil 0 \rceil \mid \mathsf{succ}\ \mathcal{O} \mid \mathsf{pred}\ \mathcal{O} \mid \mathsf{if0}\ \mathcal{O}\ \mathcal{O}\ \mathcal{O} \mid \\
&\qquad \mathsf{unit}\ \mathcal{O} \mid \mathsf{let}\ \mathcal{V}^*{:}\mathcal{C} \Leftarrow \mathcal{O}\ \mathsf{in}\ \mathcal{O} \mid \mathsf{fix}\ \mathcal{V}^*{:}\mathcal{C}.\ \mathcal{O} \\
\mathcal{C} &= \mathcal{V}^\Box \mid \mathcal{C}\ \mathcal{O} \mid \mathcal{C}\ \mathcal{C} \mid \lambda\mathcal{V}^*{:}\mathcal{C}.\ \mathcal{C} \mid \lambda\mathcal{V}^\Box{:}\mathcal{K}.\ \mathcal{C} \mid \Pi\mathcal{V}^*{:}\mathcal{C}.\ \mathcal{C} \mid \Pi\mathcal{V}^\Box{:}\mathcal{K}.\ \mathcal{C} \mid \mathsf{nat} \mid \mathsf{M}\ \mathcal{C} \\
\mathcal{K} &= * \mid \Pi\mathcal{V}^*{:}\mathcal{C}.\ \mathcal{K} \mid \Pi\mathcal{V}^\Box{:}\mathcal{K}.\ \mathcal{K}
\end{aligned}$$

Then $\mathcal{O}, \mathcal{C}, \mathcal{K}$ *are pairwise disjoint. Moreover,*

(i) If $\Gamma \vdash M : A : *$ *then* $M \in \mathcal{O}$*;*

(ii) If $\Gamma \vdash M : A : \Box$ *then* $M \in \mathcal{C}$*;*

(iii) If $\Gamma \vdash M : \Box$ *then* $M \in \mathcal{K}$.

Proof By using properties of $\lceil . \rceil$ and a similar classification result for the Calculus of Constructions with fixpoints. ■

It follows from Subject Reduction that for every M, N in $\mathcal{T}$ such that M is legal and $M \twoheadrightarrow_{ml} N$ that

$$M \in \mathcal{X} \quad \Rightarrow \quad N \in \mathcal{X}$$

for $\mathcal{X}$ ranging over $\mathcal{O}, \mathcal{C}, \mathcal{K}$.

4.6 Type-checking

Type-checking is in general undecidable because we cannot check convertibility of types. However, type-checking becomes decidable if types terminate.

Definition 23 A cubic specification $\mathbf{S} = (\mathcal{S}, *^v, *^c, \mathcal{A}, \mathcal{R})$ is *non-dependent* if for every $(s_1, s_2, s_3) \in \mathcal{R}$,

$$s_1 \in \mathcal{S}^\perp \quad \Rightarrow \quad s_2, s_3 \in \mathcal{S}^\perp$$

Non-dependent cubic specifications have normalising types, which is the key to decidable type-checking.

Proposition 24 *Non-dependent functional cubic specifications have decidable type-checking provided* $\mathcal{S}, \mathcal{A}, \mathcal{R}$ *are recursive.*

5 Operational Theory

In this section, we give an operational theory for the class of MTS specifications that we consider to be *strictly monadic.* The goal is to obtain an appropriate notion of MTSprogram equivalence. A general operational theory that allowed for an arbitrary monad would quite difficult to develop. As a first step, we consider only the computational effect of non-termination as expressed by the lifting monad.

The foundation of this material is Gordon's work [21] on an operational theory for a simply-typed version of the computational with inductive and co-inductive types, and many of our definitions and properties are minor adaptations of Gordon's. The main technical challenges in scaling up Gordon's work are dealing with conversion in types in kinds and handling dependent types.

- *Conversion in types:* In Gordon's presentation, the notion of a type-indexed relation is used in many places. With conversion in types, this becomes more complicated. For example, one must typically show that the type-indexed relations in our setting are closed under conversion of types, and substitution of terms in types.
- *Dependent types:* With dependent type, objects may occur in types and thus many notions regarding substitution and contexts become more complicated.

5.1 Program evaluation

With the single syntactic category of pseudo-terms in MTSs (as adapted from PTSs), it is not immediately clear how to define a notion of evaluation. Classes of terms that we intuitively understand to be objects, types, and kinds all belong to the same set. Normally, only objects (which express computation) should be evaluated.

Given this mixing of terms with clearly different roles, we feel that is it important to impose order by considering systems arising from cubic specifications. Lemma 21 (classification) guarantees that terms in cubic specifications can be separated into disjoint classes of objects, type constructors, and kinds.

Even with this restriction, dependent types introduce complications. With dependent types, objects can occur in types, and thus type-checking inevitably requires some notion of evaluation. This well-known phenomenon has been described as a loss of distinction between phases of type-checking and evaluation [8].

Our approach to handling this situation is to simply define a notion of evaluation for well-typed objects resulting from cubic specifications. Specifically, we assume that type-checking (by whatever means) has already been performed. Thus, for example, evaluation never reduces type arguments to polymorphic abstractions, nor does it evaluate terms that appear in types. Any such manipulation would be performed by type-checking. The classification implied by cubic specifications along with subject reduction guarantees that the class of objects is closed under evaluation — that is, evaluation of objects need only involve objects.

We now turn to the following definition specifying the notions of *program* and *program reduction* and *evaluation.*

Definition 25

- A *program* is a closed object M with a closed type, *i.e.*, it is an object M where there exists an A such that $\cdot \vdash M : A : *^x$ for $x \in \{v, c\}$.
- A *value* is an object of given by the following grammar:

  $$V ::= \lceil n \rceil \;\mid\; \lambda x{:}A.\, M \;\mid\; \mathsf{unit}\ M.$$
- An *experiment* is an object context (with exactly one hole) given by the following grammar:

  $$E ::= \mathsf{succ}\,\bullet \;\mid\; \mathsf{pred}\,\bullet \;\mid\; \mathsf{if0}\;\bullet\;M\;N \;\mid\; \bullet\;N \;\mid\; \mathsf{let}\ x{:}A \Leftarrow \bullet\ \mathsf{in}\ M$$

$$\begin{aligned}
\mathsf{succ}\ \lceil n\rceil &\longmapsto_{ml} \lceil n+1\rceil \\
\mathsf{pred}\ \lceil n+1\rceil &\longmapsto_{ml} \mathsf{unit}\ \lceil n\rceil \\
\mathsf{pred}\ \lceil 0\rceil &\longmapsto_{ml} \mathsf{unit}\ \lceil 0\rceil \\
\mathsf{if0}\ \lceil 0\rceil\ M_2\ M_3 &\longmapsto_{ml} M_2 \\
\mathsf{if0}\ \lceil n+1\rceil\ M_2\ M_3 &\longmapsto_{ml} M_3 \\
(\lambda x{:}\,A.\ M_0)\ M_1 &\longmapsto_{ml} M_0\{x := M_1\} \\
\mathsf{fix}\ x{:}\,A.\ M &\longmapsto_{ml} M\{x := \mathsf{fix}\ x{:}\,A.\ M\} \\
\mathsf{let}\ x{:}\,A \Leftarrow \mathsf{unit}\ M_1\ \mathsf{in}\ N_2 &\longmapsto_{ml} M_2\{x := M_1\}
\end{aligned}$$

$$\frac{e \longmapsto_{ml} e'}{E[e] \longmapsto_{ml} E[e']}$$

Fig. 17. Program reduction rules

$$\lceil n\rceil \Downarrow \lceil n\rceil \qquad \lambda x{:}\,A.\ M \Downarrow \lambda x{:}\,A.\ M \qquad \mathsf{unit}\ M \Downarrow \mathsf{unit}\ M$$

$$\mathsf{succ}\ \lceil n\rceil \Downarrow \lceil n+1\rceil \qquad \mathsf{pred}\ \lceil n+1\rceil \Downarrow \lceil n\rceil \qquad \mathsf{pred}\ \lceil 0\rceil \Downarrow \lceil 0\rceil$$

$$\frac{M_2 \Downarrow V}{\mathsf{if0}\ \lceil 0\rceil\ M_2\ M_3 \Downarrow V} \qquad \frac{M_3 \Downarrow V}{\mathsf{if0}\ \lceil n+1\rceil\ M_2\ M_3 \Downarrow V}$$

$$\frac{M_0 \Downarrow \lambda x{:}\,A.\ M_0' \qquad M_0'\{x := M_1\} \Downarrow V}{M_0\ M_1 \Downarrow V} \qquad \frac{M\{x := \mathsf{fix}\ x{:}\,A.\ M\} \Downarrow V}{\mathsf{fix}\ x{:}\,A.\ M \Downarrow V}$$

$$\frac{M \Downarrow \mathsf{unit}\ M' \qquad N\{x := M'\} \Downarrow V}{\mathsf{let}\ x{:}\,A \Leftarrow M\ \mathsf{in}\ N \Downarrow V}$$

Fig. 18. Program evaluation rules

- A *confined object* is a triple (Γ, M, A) such that M is an object and $\Gamma \vdash M : A$ is derivable
- *Single-step program reduction* $\cdot \longmapsto_{ml} \cdot$ is the smallest relation on programs satisfying the rules of Figure 17.
- *Big-step program evaluation* $\cdot \Downarrow \cdot$ is the smallest relation on programs satisfying the rules of Figure 18.

We write $M \Downarrow$ when there exists a V such that $M \Downarrow V$ and $M \Uparrow$ when there does not exist a V such that $M \Downarrow V$.

5.2 *Strictly monadic specifications*

A fundamental property of monadic frameworks is that canonical programs have value type. Above we noted that canonical programs include abstractions — and thus we desire that they have a value type (*i.e.*, they belong to El^{*^v}). To ensure this, one needs to consider a restriction on specifications that forces object abstractions to be typed as values.

Definition 26 A specification $\mathcal{R}$ is *value adhering* if $s' = *^v$ for every $(s, *^x, s') \in \mathcal{R}$.

We have considered several restrictions on specifications, and we now give a final restriction that captures what we feel are the natural requirements for a specification to have truly monadic character.

Definition 27 A specification $\mathcal{R}$ is *strictly monadic* if it is

- functional,
- cubic and $\mathcal{S}^\perp = \{*^v, *^c\}$, and
- value-adhering.

Thus, a strictly monadic specification guarantees uniqueness of types, ensures a three-level stratefication into objects, constructors, and kinds, and forces object abstractions to be values (observe that if $\mathcal{R}$ is a strictly monadic specification and $M\ N \in \mathsf{El}^{*^v}$, then $M \in \mathsf{El}^{*^v}$)

Throughout the remainder of this section, we will let $*$ range over bottom sorts $\mathcal{S}^\perp$ and $\Box$ range over top sorts $\mathcal{S}^\top$.

We now give several basic properties of evaluation under strictly monadic specifications.

Proposition 28 *Assume that $\mathcal{R}$ is strictly monadic. Let L, M, N be programs and U, V be values.*

(i) Program reduction is deterministic: *if $L \longmapsto_{ml} M$ and $L \longmapsto_{ml} N$ then $M \equiv N$.*

(ii) Program evaluation is deterministic: *if $L \Downarrow U$ and $L \Downarrow V$ then $U \equiv V$.*

(iii) Values are the canonical forms of program reduction: *if $\neg\exists N$ such $M \longmapsto_{ml} N$, then M is a value.*

(iv) Suppose $M \longmapsto_{ml} N$. Then for any V, $N \Downarrow V$ implies $M \Downarrow V$.

(v) $M \Downarrow V$ iff $M \longmapsto_{ml}^* V$

Proof Follows the same structure as the proof by Gordon [21]. ■

Proposition 29 *Assume that $\mathcal{R}$ is strictly monadic.*

(i) If $\vdash A : *^v$ then $A \twoheadrightarrow_{ml} \mathsf{nat}$, or $A \twoheadrightarrow_{ml} \Pi x{:}B.\, C$ for some $B, C \in \mathcal{T}$.

(ii) If $\vdash A : *^c$ then $A \twoheadrightarrow_{ml} \mathsf{M}\ B$ for some $B \in \mathcal{T}$.

Proof Proof of (1); Necessarily A is β-strongly normalizing. Let A' be its β-normal form. By Subject Reduction $\vdash A' : *^v$. By Generation and strict monadicity, A' can only be nat or a product so we are done. The proof for (2) is similar. ∎

The following property formalizes the notion that programs of value type must converge to a canonical program.

Proposition 30 *Assume that $\mathcal{R}$ is strictly monadic.*

(i) *If* $\vdash M : A : *^v$ *then* $M \Downarrow$.

(ii) *If* $\vdash M : \mathsf{nat}$, *then* $M \Downarrow \lceil n \rceil$.

(iii) *If* $\vdash M : \Pi x{:}A.\, B : *^v$ *then* $M \Downarrow \lambda x{:}A'.\, N$.

Proof (1) By a case analysis on the possible shapes of M, conclude that every weak-head reduction sequence starting from M is a $\beta\iota$-reduction sequence. Hence (by strong normalization of $\beta\iota$-reduction) the weak-head reduction sequence starting from M must end. (2) and (3) Again, case analysis on the possible shapes of M. ∎

Proposition 31 (program type inhabitation) *Assume that $\mathcal{R}$ is strictly monadic.*

(i) *There is no pseudo-term M s.t. $\cdot \vdash M : \Pi X{:}*^v.\, X$. Thus, some value program types are uninhabited.*

(ii) *For any A such that $\cdot \vdash A : *^c$, there is a B such that $\cdot \vdash \mathsf{fix}\ x{:}\mathsf{M}\ B.\, x : A$. Thus, all computation program types are inhabited.*

Proof (1) Without loss of generality we may assume that M is in β-normal form. We proceed by a case analysis on the possible shapes of M.

(2) We know $A \twoheadrightarrow_{ml} \mathsf{M}\ B$ for some $B \in \mathcal{T}$. By Subject Reduction, $\vdash \mathsf{M}\ B : *^c$. By (start), $x : \mathsf{M}\ B \vdash x : \mathsf{M}\ B$. By (fixpoint), $\vdash \mathsf{fix}\ x{:}\mathsf{M}\ B.\, x : \mathsf{M}\ B$. By (conversion), $\vdash \mathsf{fix}\ x{:}\mathsf{M}\ B.\, x : A$. ∎

Let $\bot_A$ represent the program $\mathsf{fix}\ x{:}\mathsf{M}\ B.\, x$ of type A that exists (as shown above) for any $A \in \mathcal{T}$ such that $\vdash A : *^c$. We write $\bot$ when the type A is clear from the context.

5.3 Operational equality

We now establish a notion of operational equivalence *applicative bisimilarity* for strictly monadic MTSs following Gordon's elegant presentation of a similar notion for a simply-typed version of the computational metalanguage with inductive/co-inductive types [21]. Given the definitions below, Gordon's proofs [21] carry over in a reasonably straightforward manner. Only issues regarding substitution and type equivalence are more complicated, and we indicate some of the interesting points below. In the end, we can show that the MTS calculus is sound w.r.t. our notion of operational equivalence.

Definition 32

- A *ground relation* R is a binary relation between programs of the same type.
- A *confined relation* is a binary relation between confined objects of the same type in context Γ. Write $\Gamma \vdash M\, \mathcal{R}\, N : A$ to mean that $(\Gamma \vdash M : A)\, \mathcal{R}\, (\Gamma \vdash N : A)$.

Definition 33 A ground relation $\mathcal{R}$ is *operationally adequate* iff for all programs M,N, and canonical programs V where $\cdot \vdash M, N, V : A$:

(i) If $M \longmapsto_{ml} N$ then $N\,\mathcal{R}\,M$.

(ii) If $M \Downarrow V$ then $V\,\mathcal{R}\,M$.

(iii) $M \Uparrow$ iff $M\,\mathcal{R}\,\bot_A$.

(iv) $M \Downarrow$ iff for some V, $V\,\mathcal{R}\,M$.

We often need to consider confined relations that are *compatible* with the syntactic structure of objects. Figure 19 introduces a special notion for this property of confined relations called *compatible closure*. All the rules have an implicit side condition that any sentence denoting a pair of confined terms is well-formed. Specifically, a sentence $\Gamma \vdash M\,\mathcal{R}\,N : A$ is well-formed if $\Gamma \vdash M, N : A$. Thus, the rule

$$\frac{\Gamma \vdash M_1\,\mathcal{R}\,N_1 : \Pi x{:}\,A.\,B \qquad \Gamma \vdash M_2\,\mathcal{R}\,N_2 : A}{\Gamma \vdash (M_1\,M_2)\,\widehat{\mathcal{R}}\,(N_1\,N_2) : B\{x := M_2\}}$$

should be read as follows: if $\Gamma \vdash M_1, N_1 : \Pi x{:}\,A.\,B$ and $\Gamma \vdash M_1\,\mathcal{R}\,N_1 : \Pi x{:}\,A.\,B$ and $\Gamma \vdash M_2, N_2 : A$, then if $\Gamma \vdash M_1\,M_2, N_1\,N_2 : B\{x := M_2\}$ then $\Gamma \vdash (M_1\,M_2)\,\widehat{\mathcal{R}}\,(N_1\,N_2) : B\{x := M_2\}$. Note that the MTS typing rules also imply $\Gamma \vdash N_1\,N_2 : B\{x := N_2\}$. Then, by uniqueness of types we have $B\{x := M_2\} =_{ml} B\{x := N_2\}$. Thus, by the conversion rule for compatible closure we have $\Gamma \vdash (M_1\,M_2)\,\widehat{\mathcal{R}}\,(N_1\,N_2) : B\{x := N_2\}$.

To clarify such properties, the we now give a proposition regarding compatible closure that is analogous to the Generation Lemma (Lemma 14) for MTS typing judgements.

Proposition 34 *Let $\mathcal{R}$ be a confined relation, and consider judgements of the form $\Gamma \vdash M\,\widehat{\mathcal{R}}\,N : A$.*

- *If* $\Gamma \vdash \lceil n \rceil\,\widehat{\mathcal{R}}\,\lceil n \rceil : A$,
  then $\Gamma \vdash \lceil n \rceil\,\widehat{\mathcal{R}}\,\lceil n \rceil : \mathsf{nat}$ *and* $A =_{ml} \mathsf{nat}$
- *If* $\Gamma \vdash (\mathsf{succ}\,M)\,\widehat{\mathcal{R}}\,(\mathsf{succ}\,N) : A$,
  then $\Gamma \vdash (\mathsf{succ}\,M)\,\widehat{\mathcal{R}}\,(\mathsf{succ}\,N) : \mathsf{nat}$, *and* $A =_{ml} \mathsf{nat}$
  and $\Gamma \vdash M\,\mathcal{R}\,N : \mathsf{nat}$
- *If* $\Gamma \vdash (\mathsf{pred}\,M)\,\widehat{\mathcal{R}}\,(\mathsf{pred}\,N) : A$,
  then $\Gamma \vdash (\mathsf{pred}\,M)\,\widehat{\mathcal{R}}\,(\mathsf{pred}\,N) : \mathsf{nat}$, *and* $A =_{ml} \mathsf{nat}$
  and $\Gamma \vdash M\,\mathcal{R}\,N : \mathsf{nat}$
- *If* $\Gamma \vdash (\mathsf{if0}\ M_1\ M_2\ M_3)\,\widehat{\mathcal{R}}\,(\mathsf{if0}\ N_1\ N_2\ N_3) : A$,
  then $\Gamma \vdash (\mathsf{if0}\ M_1\ M_2\ M_3)\,\widehat{\mathcal{R}}\,(\mathsf{if0}\ N_1\ N_2\ N_3) : \mathsf{M}\ B$ *for some* B
  and $A =_{ml} \mathsf{M}\ B$
  and $\Gamma \vdash M_1\,\mathcal{R}\,N_1 : \mathsf{nat}$, $\Gamma \vdash M_2\,\mathcal{R}\,N_2 : \mathsf{M}\ B$, *and* $\Gamma \vdash M_3\,\mathcal{R}\,N_3 : \mathsf{M}\ A$.

$$\Gamma \vdash x \,\widehat{\mathcal{R}}\, x : A \qquad\qquad \Gamma \vdash \lceil n \rceil \,\widehat{\mathcal{R}}\, \lceil n \rceil : \mathsf{nat}$$

$$\frac{\Gamma \vdash M \,\mathcal{R}\, N : \mathsf{nat}}{\Gamma \vdash (\mathsf{succ}\, M) \,\widehat{\mathcal{R}}\, (\mathsf{succ}\, N) : \mathsf{nat}} \qquad \frac{\Gamma \vdash M \,\mathcal{R}\, N : \mathsf{nat}}{\Gamma \vdash (\mathsf{pred}\, M) \,\widehat{\mathcal{R}}\, (\mathsf{pred}\, N) : \mathsf{nat}}$$

$$\frac{\Gamma \vdash M_1 \,\mathcal{R}\, N_1 : \mathsf{nat} \qquad \Gamma \vdash M_2 \,\mathcal{R}\, N_2 : \mathsf{M}\ A \qquad \Gamma \vdash M_3 \,\mathcal{R}\, N_3 : \mathsf{M}\ A}{\Gamma \vdash (\mathsf{if0}\ M_1\ M_2\ M_3) \,\widehat{\mathcal{R}}\, (\mathsf{if0}\ N_1\ N_2\ N_3) : \mathsf{M}\ A}$$

$$\frac{\Gamma, x{:}A \vdash M \,\mathcal{R}\, N : B}{\Gamma \vdash (\lambda x{:}A_1.\, M) \,\widehat{\mathcal{R}}\, (\lambda x{:}A_2.\, N) : \Pi x{:}A.\, B}$$

$$\frac{\Gamma, x{:}\mathsf{M}\ A \vdash M \,\mathcal{R}\, N : \mathsf{M}\ A}{\Gamma \vdash (\mathsf{fix}\ x{:}A_1.\, M) \,\widehat{\mathcal{R}}\, (\mathsf{fix}\ x{:}A_2.\, N) : \mathsf{M}\ A}$$

$$\frac{\Gamma \vdash M_1 \,\mathcal{R}\, N_1 : \Pi x{:}A.\, B \qquad \Gamma \vdash M_2 \,\mathcal{R}\, N_2 : A}{\Gamma \vdash (M_1\, M_2) \,\widehat{\mathcal{R}}\, (N_1\, N_2) : B\{x := M_2\}} \quad \text{if } \Gamma \vdash A : *$$

$$\frac{\Gamma \vdash M_1 \,\mathcal{R}\, N_1 : \Pi x{:}A.\, B}{\Gamma \vdash (M_1\, A_1) \,\widehat{\mathcal{R}}\, (N_1\, A_2) : B\{x := A_1\}} \quad \text{if } \Gamma \vdash A : \Box$$

$$\frac{\Gamma \vdash M \,\mathcal{R}\, N : A}{\Gamma \vdash (\mathsf{unit}\ M) \,\widehat{\mathcal{R}}\, (\mathsf{unit}\ N) : \mathsf{M}\ A}$$

$$\frac{\begin{array}{c}\Gamma \vdash M_1 \,\mathcal{R}\, N_1 : \mathsf{M}\ A \\ \Gamma, x{:}A \vdash M_2 \,\mathcal{R}\, N_2 : \mathsf{M}\ B \\ \Gamma \vdash A_i : * \quad A_i =_{ml} \mathsf{M}\ A \ \text{ for } i = 1, 2\end{array}}{\Gamma \vdash (\mathsf{let}\ x{:}A_1 \Leftarrow M_1\ \mathsf{in}\ M_2) \,\widehat{\mathcal{R}}\, (\mathsf{let}\ x{:}A_2 \Leftarrow N_1\ \mathsf{in}\ N_2) : \mathsf{M}\ B}$$

$$\frac{\Gamma \vdash M \,\mathcal{R}\, N : A \qquad \Gamma \vdash A' : s \qquad A =_{ml} A'}{\Gamma \vdash M \,\widehat{\mathcal{R}}\, N : A'}$$

Fig. 19. Compatible closure of a confined relation

- *If* $\Gamma \vdash (\lambda x{:}A_1.\, M)\,\widehat{\mathcal{R}}\,(\lambda x{:}A_2.\, N) \,:A$,
  then $\Gamma \vdash (\lambda x{:}A_1.\, M)\,\widehat{\mathcal{R}}\,(\lambda x{:}A_2.\, N) \,: \Pi x{:}B.\, C$ *for some* B *and* C,
  and $A =_{ml} \Pi x{:}B.\, C$, $A_1 =_{ml} B$, *and* $A_2 =_{ml} B$,
  and $\Gamma, x\!:\!B \vdash M\,\mathcal{R}\,N \,: C$.

- *If* $\Gamma \vdash (\mathsf{fix}\ x{:}A_1.\, M)\,\widehat{\mathcal{R}}\,(\mathsf{fix}\ x{:}A_2.\, N) \,:A$,
  then $\Gamma \vdash (\mathsf{fix}\ x{:}A_1.\, M)\,\widehat{\mathcal{R}}\,(\mathsf{fix}\ x{:}A_2.\, N) \,: \mathsf{M}\ B$ *for some* B,
  and $A =_{ml} \mathsf{M}\ B$, $A_1 =_{ml} \mathsf{M}\ B$, *and* $A_2 =_{ml} \mathsf{M}\ B$,
  and $\Gamma, x\!:\!\mathsf{M}\ B \vdash M\,\mathcal{R}\,N \,: \mathsf{M}\ B$.

- *If* $\Gamma \vdash (M_1\, M_2)\,\widehat{\mathcal{R}}\,(N_1\, N_2) \,:A$,
  then *one of the following holds:*

  (i) $\Gamma \vdash (M_1\, M_2)\,\widehat{\mathcal{R}}\,(N_1\, N_2) \,: B\{x := M_2\}$ *for some* B,
  and $A =_{ml} B\{x := M_2\}$ *and* $A =_{ml} B\{x := N_2\}$,
  and $\Gamma \vdash M_1\,\mathcal{R}\,N_1 \,: \Pi x{:}C.\, B$ *and* $\Gamma \vdash M_2\,\mathcal{R}\,N_2 \,: C$,
  and $\Gamma \vdash C : *$.

  (ii) $\Gamma \vdash (M_1\, M_2)\,\widehat{\mathcal{R}}\,(N_1\, N_2) \,: B\{x := M_2\}$ *for some* B,
  and $A =_{ml} B\{x := M_2\}$ *and* $A =_{ml} B\{x := N_2\}$,
  and $\Gamma \vdash M_1\,\mathcal{R}\,N_1 \,: \Pi x{:}C.\, B$,
  and $\Gamma \vdash C : \Box$.

- *If* $\Gamma \vdash (\mathsf{unit}\ M)\,\widehat{\mathcal{R}}\,(\mathsf{unit}\ N) \,:A$,
  then $\Gamma \vdash (\mathsf{unit}\ M)\,\widehat{\mathcal{R}}\,(\mathsf{unit}\ N) \,: \mathsf{M}\ B$ *and* $A =_{ml} \mathsf{M}\ B$
  and $\Gamma \vdash M\,\mathcal{R}\,N \,: B$.

- *If* $\Gamma \vdash (\mathsf{let}\ x{:}A_1 \Leftarrow M_1\ \mathsf{in}\ M_2)\,\widehat{\mathcal{R}}\,(\mathsf{let}\ x{:}A_2 \Leftarrow N_1\ \mathsf{in}\ N_2) \,:A$
  then $\Gamma \vdash (\mathsf{let}\ x{:}A_1 \Leftarrow M_1\ \mathsf{in}\ M_2)\,\widehat{\mathcal{R}}\,(\mathsf{let}\ x{:}A_2 \Leftarrow N_1\ \mathsf{in}\ N_2) \,: \mathsf{M}\ C$
  and $A =_{ml} \mathsf{M}\ C$
  and $\Gamma \vdash M_1\,\mathcal{R}\,N_1 \,: \mathsf{M}\ B$ *and* $\Gamma, x\!:\!B \vdash M_2\,\mathcal{R}\,N_2 \,: \mathsf{M}\ C$
  and $\Gamma \vdash A_i : *$ *and* $A_i =_{ml} B$ *for* $i = 1, 2$.

Proof By induction of the derivation of $\Gamma \vdash M\,\widehat{\mathcal{R}}\,N \,: A$. ∎

MTS judgements $\Gamma \vdash M : A$ have various structural properties such as thinning, condensing, *etc.* that were presented earlier. The definition below gives analogous properties for confined relations, and we will need to verify that the confined relations we introduce do indeed satisfy one or more of these properties (we are not always need them to possess all of properties).

Definition 35 (Properties of confined relations)

(i) **Type Conversion:**

$$\begin{aligned} &\text{If } \Gamma \vdash M \,\mathcal{R}\, N : A \quad \text{and} \quad \Gamma \vdash A' : s \\ &\text{such that } A =_{ml} A' \\ &\text{then } \Gamma \vdash M \,\mathcal{R}\, N : A' \end{aligned}$$

(ii) **Context Conversion:**

$$\begin{aligned} &\text{If } \Gamma \vdash M \,\mathcal{R}\, N : A \quad \text{and} \quad \Gamma' \text{ is a valid context} \\ &\text{such that } \Gamma =_{ml} \Gamma' \\ &\text{then } \Gamma' \vdash M \,\mathcal{R}\, N : A \end{aligned}$$

(iii) **Thinning:** Let Γ and Δ be legal contexts such that $\Gamma \subseteq \Delta$. Then

$$\Gamma \vdash M \,\mathcal{R}\, N : A \quad \textit{implies} \quad \Delta \vdash M \,\mathcal{R}\, N : A.$$

(iv) **Condensing:** If $x \notin (\mathrm{FV}(\Delta) \cup \mathrm{FV}(M) \cup \mathrm{FV}(N) \cup \mathrm{FV}(B))$.

$$\Gamma, x\!:\!A, \Delta \vdash M \,\mathcal{R}\, N : B \quad \textit{implies} \quad \Gamma, \Delta \vdash M \,\mathcal{R}\, N : B$$

(v) **Spec:**

$$\Gamma, x\!:\!A, \Delta \vdash M_1 \,\mathcal{R}\, M_2 : B \quad \& \quad \Gamma \vdash N : A$$

$$\textit{implies}$$

$$\Gamma, \Delta\{x := N\} \vdash M_1\{x := N\} \,\mathcal{R}\, M_2\{x := N\} : B\{x := N\}.$$

(vi) **Precong:** If $\Gamma \vdash M, N : A$ and $\Gamma \vdash C[M], C[N] : B$ where M, N, $C[M]$ and $C[N]$ are objects, then

$$\Gamma \vdash M \,\mathcal{R}\, N : A \quad \textit{implies} \quad \Gamma \vdash C[M] \,\mathcal{R}\, C[N] : B.$$

(vii) **Comp:**

$$\Gamma \vdash M \,\widehat{\mathcal{R}}\, N : A \quad \textit{implies} \quad \Gamma \vdash M \,\mathcal{R}\, N : A.$$

(viii) **Sub-object:**

$$\begin{aligned} &\text{If } \Gamma, x\!:\!A, \Delta \vdash M_1 \,\mathcal{R}\, N_1 : B \quad \text{and} \quad \Gamma \vdash M_2 \,\mathcal{R}\, N_2 : A \quad \text{and} \quad \Gamma \vdash A* \\ &\text{and } \Gamma, \Delta\{x := M_2\} =_{ml} \Gamma, \Delta\{x := N_2\} \quad \text{and} \quad B\{x := M_2\} =_{ml} B\{x := N_2\} \\ &\text{then } \Gamma, \Delta\{x := M_2\} \vdash M_1\{x := M_2\} \,\mathcal{R}\, N_1\{x := N_2\} : B\{x := M_2\} \\ &\text{and } \Gamma, \Delta\{x := N_2\} \vdash M_1\{x := M_2\} \,\mathcal{R}\, N_1\{x := N_2\} : B\{x := N_2\}. \end{aligned}$$

(ix) **Sub-type:**

$$\begin{aligned}
&\text{If } \Gamma, x{:}K, \Delta \ \vdash\ M\,\mathcal{R}\,N\ : B \quad\text{and}\quad \Gamma \vdash A, A' : K \quad\text{and}\quad \Gamma \vdash K\Box\\
&\text{and } \Gamma, \Delta\{x := A\} =_{ml} \Gamma, \Delta\{x := A'\} \quad\text{and}\quad B\{x := A\} =_{ml} B\{x := A'\}\\
&\text{then } \Gamma, \Delta\{x := A\} \ \vdash\ M_1\{x := A\}\,\mathcal{R}\,N_1\{x := A'\}\ : B\{x := A\}\\
&\text{and } \Gamma, \Delta\{x := A'\} \ \vdash\ M_1\{x := A\}\,\mathcal{R}\,N_1\{x := A'\}\ : B\{x := A'\}.
\end{aligned}$$

Definition 36

(i) The *compatible closure* of a confined relation $\mathcal{R}$ is the confined relation $\widehat{\mathcal{R}}$ defined by the rules in Figure 19.

(ii) A confined relation is *natural* iff the properties **Type Conversion**, **Context Conversion**, **Thinning**, and **Spec** hold for it.

(iii) A confined relation is a *precongruence* iff the rule **Precong** is valid. A *congruence* is a confined relation that is both a precongruence and an equivalence relation.

The definitions below provide a way to induce a confined relation from a ground relation and vice versa.

Definition 37

(i) Let Γ be a legal context. Then a Γ-*closure* is a substitution σ defined inductively on the size of Γ as follows:
- if $\Gamma \equiv []$ then σ is the identity substitution, and
- if $\Gamma \equiv x_1 : A_1, \ldots, x_n : A_n$, then σ is a simultaneous substitution $\{x_1 := M_1, \ldots, , x_n := M_n\}$ such that for every $i \in \{1, \ldots, n\}$,

$$\vdash\ M_i : A_i\{x_1 := M_1, \ldots, x_{i-1} := M_{i-1}\}$$

(ii) The *confined extension* of a ground relation $\mathcal{R}_G$ is the confined relation $\mathcal{R}$ such that $(\Gamma\ \vdash\ M\,\mathcal{R}\,N\ : A)$ iff for all Γ-closures σ, $M\sigma\,\mathcal{R}_G\,N\sigma$.

(iii) If $\mathcal{R}$ is a confined relation, then its *ground restriction* is the ground relation

$$\{(M, N)\ \mid\ \cdot \vdash\ M\,\mathcal{R}\,N\ : A\}$$

(iv) If $\mathcal{R}$ is a confined relation, write $M\,\mathcal{R}\,N$ to mean a pair (M, N) is in the ground restriction of $\mathcal{R}$.

The definition of closure is slightly more complicated than other presentations because in the MTS setting one must take dependencies in the context into account when defining the notion of closure. The following proposition establishes that the notion of closure that we have defined above is the correct one in the sense that it generalizes substitution of a single closed term.

Proposition 38 *Let $\Gamma\ \vdash\ N : A$. For all Γ-closures σ, $\vdash\ N\sigma : A\sigma$.*

Given a well-formed context Γ, it is possible that there exist no Γ-closures due to the fact that some types may be uninhabited. The following proposition characterizes when closures may exist.

Proposition 39

- *If* $\Gamma \equiv [\cdot]$ *then a* Γ*-closure exists (namely, the identity substitution).*
- *If* $\Gamma \equiv \Delta, x : A$ *then a* Γ*-closure exists if there is a* Δ*-closure* σ *and a term* B *such that* $\vdash\ B : A\sigma$.

The following proposition states basic properties of confined relations.

Proposition 40

(i) The compatible closure of any reflexive confined relation is reflexive.

(ii) The confined extension of a ground relation is natural.

(iii) If $\mathcal{R}$ *is a ground relation, and if* Γ *is a context for which no closure exists, then for all* M,N *such that* $\Gamma \vdash\ M, N : A : *$, $\Gamma \vdash\ M\,\mathcal{R}\,N\ : A$.

Component (3) reflects the fact that uninhabited types may cause terms M,N to be trivially related by the confined extension of a ground relation. This in term, means that **Condensing** does not hold the confined extension of a ground relation. Referring to the definition of **Condensing** (see Definition 35), a $\Gamma, x : A, \Delta$-closure may not exist due to the type uninhabited A (and thus M and N are trivially related), but a Γ, Δ-closure may exist that causes M and N not to be related. This illustrates the importance of working with confined relations where the context is made explicit.

We can now define applicative similarity, the ground preorder from which we will define operational equality.

Definition 41 Given a ground relation R, let the ground relation $[R]$ between programs of the same type A be defined as follows:

$\cdot \vdash\ M\,[R]\,N\ : A$ iff whenever $M \Downarrow U$ there is a V with $N \Downarrow V$ and $\cdot \vdash\ U\,\widehat{R}\,V\ : A$.

An applicative simulation is a relation $\mathcal{S}$ such that $\mathcal{S} \subseteq [\mathcal{S}]$. Define *ground applicative similarity* $\leq_a$ to be union of all applicative simulations. Define *applicative similarity* $\leq_a$ to be the confined extension of ground applicative similarity.

Proposition 42

(i) Function $[\cdot]$ *is monotonic.*

(ii) The identity ground relation is an applicative simulation.

(iii) If $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *are applicative simulations, then so is* $\mathcal{S}_1\mathcal{S}_2$.

(iv) Ground similarity is the greatest fixed-point of $[\cdot]$ *and is the greatest applicative simulation.*

(v) $\cdot \vdash\ M \leq_a N\ : A$ *iff there is an applicative simulation* $\mathcal{S}$ *such that* $\cdot \vdash\ M\,\mathcal{S}\,N\ : A$.

(vi) Ground applicative similarity is a preorder.

Definition 43 An *applicative bisimulation* is a relation $\mathcal{S}$ such that both $\mathcal{S}$ and $\mathcal{S}^{-1}$ are applicative simulations. *Ground applicative bisimilarity* $\simeq_a$ is the ground relation on programs of type A such that

$$\cdot \vdash M \simeq_a N : A \quad \text{iff} \quad \cdot \vdash M \leq_a N : A \text{ and } \cdot \vdash N \leq_a M : A.$$

Applicative bisimilarity $\simeq_a$ is the confined extension of ground applicative bisimilarity. We take applicative similarity to be our notion of equality.

Since ground applicative similarity $\leq_a$ is a preorder (Proposition 42), its symmetric closure ground applicative bisimilarity a$\simeq_a$ is an equivalence relation. Furthermore, for any programs M and N of type A, $\cdot \vdash M \simeq_a N : A$ iff for some applicative bisimulation $\mathcal{S}$, $\cdot \vdash M \, \mathcal{S} \, N : A$.

Proposition 44 *Ground relations $\leq_a$ and $\simeq_a$ are both operationally adequate.*

We are now ready to prove that applicative similarity $\leq_a$ is a precongruence using a technique due to Howe [29]. The idea is to define another relation, compatible similarity $\leq_c$, that is obviously a precongruence and obviously contains $\leq_a$, and then show that $\leq_a$ contains $\leq_c$ (*i.e.*, the two relations are the same).

Definition 45 (compatible similarity) Define the confined relation *compatible similarity* $\leq_c$ to be the smallest relation closed under the following rule.

$$\frac{\Gamma \vdash L \,\widehat{\leq_c}\, M : A \qquad \Gamma \vdash M \leq_a N : A}{\Gamma \vdash L \leq_c N : A}$$

The following proposition gives a number of properties; the last is the desired property that $\leq_a$ is a precongruence.

Proposition 46

(i) *Applicative similarity is natural.*

(ii) *Confined relations $\widehat{\leq_c}$ and $\leq_c$ are reflexive.*

(iii) *Confined relations $\widehat{\leq_c}$ and $\leq_a$ both imply $\leq_c$.*

(iv) *If $\Gamma \vdash M_1 \leq_c M_2 : A$ and $\Gamma \vdash M_2 \leq_a M_3 : A$ then $\Gamma \vdash M_1 \leq_c M_3 : A$.*

(v) *The rules* **Sub-object** *and* **Sub-type** *(of Definition 35) hold for confined relation $\leq_c$.*

(vi) *If $\cdot \vdash U \leq_c M : A$ for value U then there is a value V such that $M \Downarrow V$ and $\cdot \vdash U \,\widehat{\leq_c}\, V : A$.*

(vii) *If $\cdot \vdash M \leq_c N : A$ and $M \Downarrow U$ then $\cdot \vdash U \leq_c N : A$.*

(viii) *If $\Gamma \vdash M \leq_c N : A$ then $\Gamma \vdash M \leq_a N : A$.*

(ix) *Applicative similarity is a precongruence.*

The fact that applicative similarity is a precongruence (and applicative bisimilarity is a congruence) allows us to reason compositionally about MTS programs. Given this, it is relatively straightforward to show that the equivalences of Figure 20 hold. As a

Congruence

If $(\Gamma \vdash M \mathrel{\widehat{\simeq_a}} N : A)$ then $(\Gamma \vdash M \simeq_a N : A)$.

Canonical Exclusivity

If $(\Gamma \vdash U \simeq_a V : A)$ then $(\Gamma \vdash U \mathrel{\widehat{\simeq_a}} V : A)$.

MTS Calculus

$$\Gamma \vdash (\lambda x{:}B.\,M)\,N \simeq_a M\{x := N\} : A$$

$$\Gamma \vdash \mathsf{fix}\ x{:}M.\,N \simeq_a N\{x := \mathsf{fix}\ x{:}M.\,N\} : A$$

$$\Gamma \vdash \mathsf{let}\ x{:}A \Leftarrow \mathsf{unit}\ N\ \mathsf{in}\ M \simeq_a M\{x := N\} : A$$

$$\Gamma \vdash \mathsf{let}\ x{:}A \Leftarrow M\ \mathsf{in}\ \mathsf{unit}\ x \simeq_a M : A$$

$$\begin{array}{ll}\Gamma \vdash & \mathsf{let}\ x_2{:}A_2 \Leftarrow (\mathsf{let}\ x_1{:}A_1 \Leftarrow M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3 \\ & \qquad\qquad\simeq_a \\ & \mathsf{let}\ x_1{:}A_1 \Leftarrow M_1\ \mathsf{in}\ (\mathsf{let}\ x_2{:}A_2 \Leftarrow M_2\ \mathsf{in}\ M_3) : A\end{array}$$

$$\Gamma \vdash \mathsf{succ}\,\lceil n \rceil \simeq_a \lceil n+1 \rceil : A$$

$$\Gamma \vdash \mathsf{pred}\,\lceil n+1 \rceil \simeq_a \lceil n \rceil : A$$

$$\Gamma \vdash \mathsf{pred}\,\lceil 0 \rceil \simeq_a \lceil 0 \rceil : A$$

$$\Gamma \vdash \mathsf{if0}\ \lceil 0 \rceil\ N_1\ N_2 \simeq_a N_1 : A$$

$$\Gamma \vdash \mathsf{if0}\ \lceil n+1 \rceil\ N_1\ N_2 \simeq_a N_2 : A$$

Strictness Law

$$\Gamma \vdash \mathsf{let}\ x{:}B \Leftarrow \perp\ \mathsf{in}\ M \simeq_a \perp : A$$

Fig. 20. Laws of equality in MTS

consequence, we have that MTS calculus is sound with respect to operational equivalence. This result provides the semantic foundation for the MTS framework.

Theorem 47 (soundness of MTS calculus) *For any objects M and N such that $\Gamma \vdash M : A$ and $\Gamma \vdash N : A$, $M =_{ml} N$ implies $M \simeq_a N$.*

6 Assessment and related work

Monadic type systems may prove useful in several applications, some of which are summarized below. In addition, monadic type systems may find applications in logic but we have not explored this possibility thus far.

Intermediate language for partial evaluation:

The computational metalanguage is well-suited as an evaluation-order independent intermediate language for partial evaluation [27]. The monadic structure guarantees the soundness of partial evaluation in the presence of computational effects [32, 48], and it facilitates the extension of type specialization to a language with first-class references [14, 30]. These works either take place in an untyped setting [32], or in a simply typed setting [14, 27, 30].

We expect the MTS-framework to contribute to the definition of sound partial evaluation for polymorphically typed languages (where specialization with respect to types coexists with specialization wrt. values in a single framework) and possibly also to an extension of type specialization to encompass polymorphism. It would also be interesting to consider type-directed partial evaluation [11] in this framework.

Intermediate language for compilation:

Various researchers have advocated the use of the metalanguage as an evaluation-order independent intermediate language for compiling [7, 28]. Many features of the metalanguage such as the reliance on monads to encapsulate effects, and the use of pointed types to distinguish call-by-name from call-by-value and certainly-converging terms from possibly-diverging terms have been incorporated into a recent proposal for a simply-typed evaluation-order independent intermediate language [38].

We believe that the MTS framework and the results of this paper can help provide a technical foundation for combining ideas from the PTS-based intermediate language of Meijer and Peyton Jones [31] and the language of Peyton Jones *et al.* [38].

Generic CPS translation framework:

Hatcliff and Danvy [28] have developed a generic framework for reasoning about CPS translations of simply-typed languages by factorizing the translations through a simply-typed version of the computational metalanguage. The results presented in the present work enable us to generalize this idea to treat in a single framework call-by-value, call-by-name, and mixed strategy CPS translations for PTSs. The general translation could be instantiated to the translations for F_2 [23], F_ω [22], non-dependent PTSs [10], as well as the recent CPS translations for PTSs with dependent types proposed by Barthe *et al.* [6].

7 Conclusion and directions for further research

The paper introduces monadic type systems, a powerful and flexible framework to capture computational effects in rich (polymorphic, dependent) type systems, and study operational semantics for monadic type systems with dependent types.

Much work remains to be done. For example, it seems important to extend the type structure of monadic type systems with further constructs as the type structure of MTSs, with only two type constructors (monads and dependent products), is too minimal for some applications.

Section 5 presents an operational semantics for MTSs, which smoothly generalizes certain aspects of Gordon's earlier work [21]. A more extensive investigation needs to be done to compare applicative similarity to other familiar notions of equivalence such as contextual similarity. In addition, the semantics is mainly concerned with the lifting monads. It would be interesting to scale up alternative semantics for notions of computation (see *e.g.*, [43]) to the framework of MTSs.

Finally, we are interested in exploiting our framework for CPS translation and partial evaluation. Based on the technical results of [6], we are currently extending the results of [28] to MTSs.

References

[1] P. Audebaud. Partial objects in the calculus of constructions. In *Proceedings of LICS'91*, pages 86–95. IEEE Computer Society Press, 1991.

[2] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[3] H. Barendregt. Introduction to Generalised Type Systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.

[4] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford Science Publications, 1992. Volume 2.

[5] G. Barthe. The relevance of proof-irrelevance. Manuscript, 1997.

[6] Gilles Barthe, John Hatcliff, and Morten Heine Sørensen. CPS translations and applications: the cube and beyond. In Olivier Danvy, editor, *Proceedings of the Second ACM SIGPLAN Workshop on Continuations*, number NS-96-13 in BRICS Notes, pages 4–1–4–31, Paris, France, January 1997. Dept. of Computer Science, Aarhus, Denmark.

[7] N. Benton. *Strictness Analysis of Lazy Functional Programs.* PhD thesis, University of Cambridge, 1992.

[8] L. Cardelli. Phase distinctions in type theory. Unpublished Mansucript, January 1988.

[9] T. Coquand. Typechecking dependent types. Privately circulated manuscript, 1998.

[10] T. Coquand and H. Herbelin. A-translation and looping combinators in pure type systems. *Journal of Functional Programming*, 4(1):77–88, January 1994.

[11] O. Danvy. Type-directed partial evaluation. In *Proceedings of POPL'96*, pages 242–257. ACM Press, 1996.

[12] Olivier Danvy and John Hatcliff. Cps transformation after strictness analysis. *Letters on Programming Languages and Systems*, 1(3):195–212, 1993.

[13] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proc. of the 9th Conference on Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 627–638, New Orleans, Louisiana, April 1993. Springer-Verlag.

[14] D. Dussart, J. Hughes, and P. Thiemann. Type specialisation for imperative languages. In *Proceedings of ICFP'97*, pages 204–216. ACM Press, 1997.

[15] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 10(2):235–271, 1992.

[16] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, Albuquerque, New Mexico, June 1993.

[17] J. Garrigue and D. Rémy. Extending ML with semi-explicit higher-order polymorphism. In *Proceedings of TACS'97*, volume 1281 of *Lecture Notes in Computer Science*, pages 20–46. Springer-Verlag, 1997.

[18] H. Geuvers. *Logics and type systems.* PhD thesis, University of Nijmegen, 1993.

[19] J-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* Thèse d'Etat, Université Paris 7, 1972.

[20] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Number 7 in Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[21] Andrew D. Gordon. *Functional Programming and Input/Output.* Cambridge University Press, 1994.

[22] R. Harper and M. Lillibridge. Explicit polymorphism and CPS conversion. In *Proceedings of POPL'93*, pages 206–219. ACM Press, 1993.

[23] Robert Harper and Mark Lillibridge. Polymorphic type assignment and cps conversion. *Lisp and Symbolic Computation*, 6(4):361–380, 1993.

[24] Robert Harper and Mark Lillibridge. Operational interpretations of an extension of f_ω with control operators. *Journal of Functional Programming*, 6(3):393–417, May 1996.

[25] Robert Harper and John C. Mitchell. On the type structure of standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[26] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995. ACM Press.

[27] J. Hatcliff and O. Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–541, October 1997.

[28] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pages 458–471, Portland, OG, January 1994. ACM Press.

[29] D.J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, February 1996.

[30] John Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 183–215, Dagstuhl, Germany, February 1996. Springer Verlag, Heidelberg.

[31] S. Peyton Jones and E. Meijer. Henk: a typed intermediate language. Proceedings of the ACM Workshop on Types in Compilation, 1997.

[32] J. Lawall and P. Thiemann. Sound specialization in the presence of computational effects. In *Proceedings of TACS'97*, volume 1281 of *Lecture Notes in Computer Science*, pages 165–190. Springer-Verlag, 1997.

[33] X. Leroy. Polymorphism by name for references and continuations. In *Proceedings of POPL'93*, pages 220–231. ACM Press, 1993.

[34] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[35] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Fla., January 1996. ACM Press.

[36] Eugenio Moggi. Notions of computations and monads. *Information and Computation*, 93:55–92, 1991.

[37] M. Odersky and K. Läufer. Putting type annotations to work. In *Proceedings of POPL'96*, pages 54–67. ACM Press, 1996.

[38] S. Peyton Jones, J. Launchbury, M. Shields, and A. Tolmach. The design of a typed intermediate language. In *Proceedings of POPL'98.* ACM Press, 1998.

[39] Simon L Peyton Jones, Cordelia Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, Keele, 1993.

[40] S.L. Peyton Jones and P.L. Wadler. Imperative functional programming. In *Proceedings of POPL'93*, pages 71–84. ACM Press, 1993.

[41] R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.

[42] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pages 717–740, July 1972.

[43] E. Ritter and A.M. Pitts. A fully abstract translation between a λ-calculus with reference types and Standard ML. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of TLCA'95*, volume 902 of *Lecture Notes in Computer Science*, pages 397–413. Springer-Verlag, 1995.

[44] Amr Sabry and John Field. Reasoning about explicit and implicit representations of state. In Paul Hudak, editor, *SIPL '93, ACM SIGPLAN Workshop on State in Programming Languages*, pages 17–30, Copenhagen, Denmark, June 1993. Yale University, Department of Computer Science, New Haven, CT. Technical Report YALEU/DCS/RR-968.

[45] Zhong Shao. Typed common intermediate format. In *Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997. USENIX.

[46] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proc. of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, USA, June 1995. ACM Press.

[47] Dave Tarditi, Greg Morrisett, P. Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, USA, May 1996. ACM Press.

[48] Peter Thiemann and Dirk Dussart. Partial evaluation for higher-order languages with state. Berichte des Wilhelm-Schickard-Instituts WSI-97-XX, Universitt Tbingen, April 1997.

[49] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[50] Philip L. Wadler. The essence of functional programming. In *Proc. 19th Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.

Proofs

A Example Compile-time Transformation

Consider the ML source term $(\lambda x.f@(!\ x))@(\mathsf{ref}\ 5)$ where $f : \mathsf{nat} \to \tau$. This term is transformed by $\mathcal{C}^M$ to

$$
\begin{array}{ll}
& \mathsf{let}\ x_1 : \mathsf{ref\ nat} \to \mathsf{M}\ \tau \Leftarrow \mathcal{C}^M[\![\lambda x.f@(!\ x)]\!]\ \mathsf{in} \\
& \mathsf{let}\ x_2 : \mathsf{ref\ nat} \Leftarrow \mathcal{C}^M[\![\mathsf{ref}\ 5]\!]\ \mathsf{in}\ x_1\ x_2 \\
\equiv & \mathsf{let}\ x_1 : \mathsf{ref\ nat} \to \mathsf{M}\ \tau \Leftarrow \mathsf{unit}\ \lambda x.\mathcal{C}^M[\![f@(!\ x)]\!]\ \mathsf{in} \\
& \mathsf{let}\ x_2 : \mathsf{ref\ nat} \Leftarrow (\mathsf{let}\ x_3 : \mathsf{nat} \Leftarrow \mathsf{unit}\ 5\ \mathsf{in}\ \mathsf{ref}_M\ \mathsf{nat}\ x_3)\ \mathsf{in}\ x_1\ x_2 \\
\equiv & \mathsf{let}\ x_1 : \mathsf{ref\ nat} \to \mathsf{M}\ \tau \Leftarrow \mathsf{unit}\ \lambda x. \\
& \quad \mathsf{let}\ x_4 : \mathsf{nat} \to \mathsf{M}\ \tau \Leftarrow \mathsf{unit}\ f\ \mathsf{in} \\
& \quad \mathsf{let}\ x_5 : \mathsf{nat} \Leftarrow (\mathsf{let}\ x_6 : \mathsf{ref\ nat} \Leftarrow \mathsf{unit}\ x\ \mathsf{in}\ !_M\ \mathsf{nat}\ x_6)\ \mathsf{in}\ x_4\ x_5\ \mathsf{in} \\
& \mathsf{let}\ x_2 : \mathsf{ref\ nat} \Leftarrow (\mathsf{let}\ x_3 : \mathsf{nat} \Leftarrow \mathsf{unit}\ 5\ \mathsf{in}\ \mathsf{ref}_M\ \mathsf{nat}\ x_3)\ \mathsf{in}\ x_1\ x_2 \\
\Rightarrow_{ml} & \mathsf{let}\ x_1 : \mathsf{ref\ nat} \to \mathsf{M}\ \tau \Leftarrow \mathsf{unit}\ \lambda x. \\
& \quad \mathsf{let}\ x_4 : \mathsf{nat} \to \mathsf{M}\ \tau \Leftarrow \mathsf{unit}\ f\ \mathsf{in} \\
& \quad \mathsf{let}\ x_6 : \mathsf{ref\ nat} \Leftarrow \mathsf{unit}\ x\ \mathsf{in} \\
& \quad \mathsf{let}\ x_5 : \mathsf{nat} \Leftarrow\ !_M\ \mathsf{nat}\ x_6\ \mathsf{in}\ x_4\ x_5\ \mathsf{in} \\
& \mathsf{let}\ x_3 : \mathsf{nat} \Leftarrow \mathsf{unit}\ 5\ \mathsf{in} \\
& \mathsf{let}\ x_2 : \mathsf{ref\ nat} \Leftarrow \mathsf{ref}_M\ \mathsf{nat}\ x_3\ \mathsf{in}\ x_1\ x_2 \\
\Rightarrow_{ml} & \mathsf{let}\ x_2 : \mathsf{ref\ nat} \Leftarrow \mathsf{ref}_M\ \mathsf{nat}\ 5\ \mathsf{in} \\
& (\lambda x.\mathsf{let}\ x_5 : \mathsf{nat} \Leftarrow\ !_M\ \mathsf{nat}\ x\ \mathsf{in}\ f\ x_5)\ x_2 \\
\Rightarrow_{ml} & \mathsf{let}\ x_2 : \mathsf{ref\ nat} \Leftarrow \mathsf{ref}_M\ \mathsf{nat}\ 5\ \mathsf{in} \\
& \mathsf{let}\ x_5 : \mathsf{nat} \Leftarrow\ !_M\ \mathsf{nat}\ x_2\ \mathsf{in}\ f\ x_5
\end{array}
$$

If the compiler added special rules for reasoning with state [44] it could derive an even more optimized program.

$$\mathsf{let}\ x_2 : \mathsf{ref\ nat} \Leftarrow \mathsf{ref}_M\ \mathsf{nat}\ 5\ \mathsf{in}\ f\ 5$$

The pervasive type annotations in the program can guide the compiler to choose good (unboxed) runtime representations for 5 and for $\mathsf{ref}\ 5$.

B Strong normalization of $\mu\kappa$-reduction

B.1 Finiteness of Developments

A development is a rewrite sequence in which the contracted redexes are descendants of the redexes of the original term. A standard result in λ-calculus, called Finiteness of Developments, shows that all such reduction sequences are finite. Among the many possible proofs of FD, one proceeds by defining underlined λ-terms. We take this path here.

Definition 48 The set $\mathcal{U}$ of *underlined λ-terms* is defined by the abstract syntax:

$$\mathcal{U} = V \mid \mathcal{U}\,\mathcal{U} \mid \lambda V.\,\mathcal{U} \mid (\underline{\lambda} V.\,\mathcal{U})\,\mathcal{U}$$

The notion $\underline{\beta}$ of *underlined β-reduction* is defined by the contraction rule

$$(\underline{\lambda} x.\, M)\; N \quad \to_{\underline{\beta}} \quad M\{x := N\}$$

Finiteness of Developments can be stated as:

Theorem 49 $\mathcal{U} = \mathrm{SN}(\underline{\beta})$.

B.2 Commuting conversions

In order to prove strong normalisation of μ-reduction, we need to consider a commuting conversion $\underline{\beta'}$.

Definition 50 The notion of reduction $\underline{\beta'}$ is defined by the contraction rule

$$P\;((\underline{\lambda} x.\, Q)\;R) \quad \to_{\underline{\beta'}} \quad (\underline{\lambda} x.\, P\,Q)\;R$$

provided $x \notin \mathrm{FV}(P)$.

$\underline{\beta'}$ preserves strong normalisation, in the sense that underlined λ-terms are $\underline{\beta\beta'}$-strongly normalising.

Theorem 51 $\mathcal{U} = \mathrm{SN}(\underline{\beta\beta'})$.

Proof Every term $M \in \mathcal{U}$ is $\underline{\beta}$-strongly normalising, so we can define $\mathsf{maxred}(M)$ to be the length of the longest $\underline{\beta}$-reduction sequence starting from M. Now assume that $M \notin \mathrm{SN}(\underline{\beta\beta'})$, so we have

$$M \to_{\underline{\beta\beta'}} M_1 \to_{\underline{\beta\beta'}} \dots$$

Without loss of generality, we can assume that:

(i) $N \in \mathrm{SN}(\underline{\beta\beta'})$ for every $N \in \mathcal{U}$ such that $\mathsf{maxred}(N) < \mathsf{maxred}(M)$;

(ii) $N \in \mathrm{SN}(\underline{\beta\beta'})$ for every strict subterm N of M.

We now proceed by case analysis on M. Necessarily M is of the form

$$(\underline{\lambda} x.\, P)\; Q\; R_1\; \dots\; R_n$$

Moreover, the first reduction step $M \to_{\underline{\beta}\beta'} M_1$ is a $\underline{\beta}'$-step, otherwise we would reach a contradiction. There are four possibilities, out of which we treat the last one in detail:

- $M \to_{\underline{\beta}'} (\underline{\lambda} x.\, P')\, Q\ R_1\ \dots\ R_n$ with $P \to_{\underline{\beta}'} P'$.
- $M \to_{\underline{\beta}'} (\underline{\lambda} x.\, P)\, Q'\ R_1\ \dots\ R_n$ with $Q \to_{\underline{\beta}'} Q'$.
- $M \to_{\underline{\beta}'} (\underline{\lambda} x.\, P)\, Q\ R_1\ \dots\ R'_i\ \dots R_n$ for some i with $R_i \to_{\underline{\beta}'} R'_i$.
- $Q \equiv (\underline{\lambda} y.\, S)\, T$ with $y \notin \mathrm{FV}(P)$ and $M \to_{\underline{\beta}'} (\underline{\lambda} y.\, (\underline{\lambda} x.\, P)\, S)\, T\ R_1\ \dots\ R_n$. By 2, each subterm of M is $\underline{\beta}\beta'$-strongly normalising, so necessarily the reduction sequence must contract a descendant of one of the redexes $(\underline{\lambda} x.\, P)\, S$ or $(\underline{\lambda} y.\, (\underline{\lambda} x.\, P)\, S)\, T$. In other words, the reduction sequence will be of the form

$$\begin{aligned}(\underline{\lambda} y.\, (\underline{\lambda} x.\, P)\, S)\, T\ R_1\ \dots\ R_n &\twoheadrightarrow_{\underline{\beta}\beta'} (\underline{\lambda} y.\, (\underline{\lambda} x.\, P')\, S')\, T'\ R'_1\ \dots\ R'_n\\ &\to_{\underline{\beta}}\ (\underline{\lambda} x.\, P')\, (S'\{y := T'\})\ R'_1\ \dots\ R'_n\\ &\to_{\underline{\beta}}\ \cdots\end{aligned}$$

  or

$$\begin{aligned}(\underline{\lambda} y.\, (\underline{\lambda} x.\, P)\, S)\, T\ R_1\ \dots\ R_n &\twoheadrightarrow_{\underline{\beta}\beta'} (\underline{\lambda} y.\, (\underline{\lambda} x.\, P')\, S')\, T'\ R'_1\ \dots\ R'_n\\ &\to_{\underline{\beta}}\ (\underline{\lambda} y.\, (P'\{x := S'\}))\, T'\ R'_1\ \dots\ R'_n\\ &\to_{\underline{\beta}}\ \cdots\end{aligned}$$

  with $X \twoheadrightarrow_{\underline{\beta}\beta'} X'$ for every term X.

  · The first case is impossible since $M \to_{\underline{\beta}} (\underline{\lambda} x.\, P)\, (S\{y := T\})\ R_1\ \dots\ R_n$ and hence by assumption, $(\underline{\lambda} x.\, P)\, (S\{y := T\})\ R_1\ \dots\ R_n \in \mathrm{SN}(\underline{\beta}\beta')$. On the other hand, $(\underline{\lambda} x.\, P')\, (S'\{y := T'\})\ R'_1\ \dots\ R'_n \notin \mathrm{SN}(\underline{\beta}\beta')$ and $(\underline{\lambda} x.\, P)\, (S\{y := T\})\ R_1\ \dots\ R_n \twoheadrightarrow_{\underline{\beta}\beta'} (\underline{\lambda} x.\, P')\, (S'\{y := T'\})\ R'_1\ \dots\ R'_n$, a contradiction.

  · The second case is impossible. Indeed, there are two possibilities:
  $P'\{x := S'\} \notin \mathrm{SN}(\underline{\beta}\beta')$. This possibility is to be ruled out since $P\{x := S\}\{y := T\} \in \mathrm{SN}(\underline{\beta}\beta')$ by 1 and thus $P\{x := S\} \in \mathrm{SN}(\underline{\beta}\beta')$, which together with $P\{x := S\} \twoheadrightarrow_{\underline{\beta}\beta'} P''\{x := S''\}$ contradicts $P''\{x := S''\} \notin \mathrm{SN}(\underline{\beta}\beta')$.
  the infinite reduction sequence proceeds further as

$$\begin{aligned}(\underline{\lambda} y.\, (P'\{x := S'\}))\, T'\ R'_1\ \dots\ R'_n &\twoheadrightarrow_{\underline{\beta}\beta'} (\underline{\lambda} y.\, (P''\{x := S''\}))\, T''\ R''_1\ \dots\ R''_n\\ &\to_{\underline{\beta}}\ (P''\{x := S''\}\{y := T''\})\ R''_1\ \dots\ R''_n\\ &\to_{\underline{\beta}}\ \cdots\end{aligned}$$

  This possibility is to be ruled out since $M \twoheadrightarrow_{\underline{\beta}} (P\{x := S\}\{y := T\})\ R_1\ \dots\ R_n$ so by 1, $(P\{x := S\}\{y := T\})\ R_1\ \dots\ R_n \in \mathrm{SN}(\underline{\beta}\beta')$, which together with $(P\{x := S\}\{y := T\})\ R_1\ \dots\ R_n \twoheadrightarrow_{\underline{\beta}\beta'} (P''\{x := S''\}\{y := T''\})\ R''_1\ \dots\ R''_n$ contradicts $(P''\{x := S''\}\{y := T''\})\ R''_1\ \dots\ R''_n \notin \mathrm{SN}(\underline{\beta}\beta')$.

■

C Proof of Lemma 2

Let $\Gamma = \{y_1 : \sigma_1, \dots, y_n : \sigma_n\}$. Let further $\Gamma' = \{y_1 : A'_1, \dots, y_n : A'_n\}$ where $A'_i = \mathcal{S}^M[\![A_i]\!]$ if y_i is not fix-bound and $A'_i = \mathcal{C}^M[\![A_i]\!]$ otherwise. Then

(C.1) $$\Gamma \vdash_{SML} e : \tau \quad \Rightarrow \quad G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e]\!] : \mathcal{C}^M[\![\tau]\!]$$

using $\mathcal{R}_{ML}$.

Proof The proof is by simultaneous induction on the type derivation of an expression using the auxiliary inductive hypothesis:

$$\Gamma \vdash_{SML} v : \tau \quad \Rightarrow \quad G_\Gamma, \Gamma' \vdash \mathcal{V}^M[\![v]\!] : \mathcal{V}^M[\![\tau]\!]. \tag{C.2}$$

First of all, G_Γ, Γ' is valid since for all $y_i : \mathcal{S}^M[\![\sigma_i]\!]$ in Γ' $G_\Gamma \vdash \mathcal{S}^M[\![\sigma_i]\!] : s$ where $s \in \{*^v, *^c, \sharp\}$.

- Hypothesis (C.2) is immediate for $\Gamma \vdash_{SML} i : \mathsf{nat}$, since $G_\Gamma, \Gamma' \vdash i : \mathcal{V}^M[\![\mathsf{nat}]\!]$ by the presence of constants.
- Hypothesis (C.2) for $\Gamma, x : \sigma \vdash_{SML} x : \tau \quad \tau \leq \sigma$. Suppose $x : \sigma$ in Γ with $\sigma = \forall\alpha_1 \ldots \alpha_n.\tau'$ and $\tau = \tau'\{\alpha_i \mapsto \tau_i\}$. Then
$$G_\Gamma, \Gamma' \vdash \mathsf{unit}\ (x\ \mathcal{V}^M[\![\tau_1]\!]\ \ldots\ \mathcal{V}^M[\![\tau_n]\!]) : \mathcal{C}^M[\![\tau]\!]$$
must be shown. By definition of Γ',
$$G_\Gamma, \Gamma' \vdash x : \mathcal{S}^M[\![\forall\alpha_1 \ldots \alpha_n.\tau']\!]$$
which is the same as
$$G_\Gamma, \Gamma' \vdash x : \Pi\alpha_1{:}*^v. \ldots \Pi\alpha_n{:}*^v.\ \mathcal{V}^M[\![\tau']\!].$$
Hence
$$G_\Gamma, \Gamma' \vdash x\ \mathcal{V}^M[\![\tau_1]\!]\ \ldots\ \mathcal{V}^M[\![\tau_n]\!] : \mathcal{V}^M[\![\tau'\{\alpha_i \mapsto \tau_i\}]\!].$$
- Hypothesis (C.2) for $\Gamma, f : \tau_2 \to \tau_1 \vdash_{SML} f : \tau_2 \to \tau_1$. Hence $f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!]$ in Γ' and we have to prove
$$G_\Gamma, \Gamma' \vdash \lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow f\ \mathsf{in}\ y_1\ y_2 : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].$$
By the (application) rule
$$G_\Gamma, \Gamma', y_2 : \mathcal{V}^M[\![\tau_2]\!], y_1 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!] \vdash y_1\ y_2 : \mathcal{C}^M[\![\tau_1]\!] \tag{C.3}$$
and by the assumption on f
$$G_\Gamma, \Gamma', y_2 : \mathcal{V}^M[\![\tau_2]\!] \vdash f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!]. \tag{C.4}$$
Applying (let) to (C.3) and (C.4) yields
$$G_\Gamma, \Gamma', y_2 : \mathcal{V}^M[\![\tau_2]\!] \vdash \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow f\ \mathsf{in}\ y_1\ y_2 : \mathcal{C}^M[\![\tau_1]\!]$$
And the claim follows by (abstraction).
$$G_\Gamma, \Gamma' \vdash \lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow f\ \mathsf{in}\ y_1\ y_2 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!].$$
- Hypothesis (C.2) for $\dfrac{\Gamma, x : \tau_2 \vdash_{SML} e : \tau_1}{\Gamma \vdash_{SML} \lambda x.e : \tau_2 \to \tau_1}$. By induction and $\mathcal{S}^M[\![\tau_2]\!] = \mathcal{V}^M[\![\tau_2]\!]$,
$$G_\Gamma, \Gamma', x : \mathcal{V}^M[\![\tau_2]\!] \vdash \mathcal{C}^M[\![e]\!] : \mathcal{C}^M[\![\tau_1]\!].$$
By the (abstraction) rule
$$G_\Gamma, \Gamma' \vdash \lambda x : \mathcal{V}^M[\![\tau_2]\!].\mathcal{C}^M[\![e]\!] : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!].$$
Since $\mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!] = \mathcal{V}^M[\![\tau_2 \to \tau_1]\!]$,
$$G_\Gamma, \Gamma' \vdash \lambda x : \mathcal{V}^M[\![\tau_2]\!].\mathcal{C}^M[\![e]\!] : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!].$$

- Hypothesis (C.2) for $\dfrac{\Gamma, f : \tau_2 \to \tau_1 \vdash_{SML} e : \tau_2 \to \tau_1}{\Gamma \vdash_{SML} \mathsf{fix}\ f.e : \tau_2 \to \tau_1}$. By induction and $\mathcal{C}^M[\![\tau]\!] = \mathsf{M}\ \mathcal{V}^M[\![\tau]\!]$,

  $$G_\Gamma, \Gamma', f : \mathsf{M}\ \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \vdash \mathcal{C}^M[\![e]\!] : \mathsf{M}\ \mathcal{V}^M[\![\tau_2 \to \tau_1]\!]$$

  By the (fixpoint) rule

  $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![\mathsf{fix}\ f.\mathcal{C}^M[\![e]\!]]\!] : \mathsf{M}\ \mathcal{V}^M[\![\tau_2 \to \tau_1]\!]$$

  and by (weakening)

  (C.5) $$G_\Gamma, \Gamma', y_2 : \mathcal{V}^M[\![\tau_2]\!] \vdash \mathsf{fix}\ f.\mathcal{C}^M[\![e]\!] : \mathsf{M}\ \mathcal{V}^M[\![\tau_2 \to \tau_1]\!].$$

  As seen before,

  (C.6) $$G_\Gamma, \Gamma', y_2 : \mathcal{V}^M[\![\tau_2]\!], y_1 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!] \vdash y_1\ y_2 : \mathcal{C}^M[\![\tau_1]\!]$$

  Apply (let) to (C.5) and (C.6) to obtain

  (C.7) $$G_\Gamma, \Gamma', y_2 : \mathcal{V}^M[\![\tau_2]\!] \vdash \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!] \Leftarrow \mathsf{fix}\ f.\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2 : \mathcal{C}^M[\![\tau_1]\!]$$

  Apply (abstraction) to (C.7)

  $$G_\Gamma, \Gamma', y_2 : \mathcal{V}^M[\![\tau_2]\!] \vdash$$
  $$\lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!] \Leftarrow \mathsf{fix}\ f.\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!]$$

- Hypothesis (C.1) for values $\Gamma \vdash_{SML} v : \tau$. By the auxiliary inductive hypothesis (C.1),

  $$G_\Gamma, \Gamma' \vdash \mathcal{V}^M[\![v]\!] : \mathcal{V}^M[\![\tau]\!]$$

  By the (unit) rule

  $$G_\Gamma, \Gamma' \vdash \mathsf{unit}\ (\mathcal{V}^M[\![v]\!]) : \mathsf{M}\ \mathcal{V}^M[\![\tau]\!]$$

  and by the definition of $\mathcal{C}^M$

  $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![v]\!] : \mathcal{C}^M[\![\tau]\!].$$

- $\dfrac{\Gamma \vdash_{SML} e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash_{SML} e_2 : \tau_2}{\Gamma \vdash_{SML} e_1 @ e_2 : \tau_1}$. By induction,

  $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e_1]\!] : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!] \quad \text{and} \quad G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e_2]\!] : \mathcal{C}^M[\![\tau_2]\!].$$

  By rule (application)

  (C.8) $$G_\Gamma, \Gamma', y_1 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!], y_2 : \mathcal{V}^M[\![\tau_2]\!] \vdash y_1\ y_2 : \mathcal{C}^M[\![\tau_2]\!].$$

  By rule (let) applied to the inductive hypothesis for e_2 and (C.8)

  (C.9) $$G_\Gamma, \Gamma', y_1 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!] \vdash \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathcal{C}^M[\![e_2]\!]\ \mathsf{in}\ y_1\ y_2 : \mathcal{C}^M[\![\tau_2]\!]$$

  and again by rule (let) applied to the inductive hypothesis for e_1 and (C.9)

  $$G_\Gamma, \Gamma' \vdash \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!] \Leftarrow \mathcal{C}^M[\![e_1]\!]\ \mathsf{in}$$
  $$\mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathcal{C}^M[\![e_2]\!]\ \mathsf{in}\ y_1\ y_2 : \mathcal{C}^M[\![\tau_2]\!]$$

  since $\mathcal{C}^M[\![\tau_2 \to \tau_1]\!] = \mathsf{M}\ (\mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!])$.

- $\dfrac{\Gamma \vdash_{SML} v : \tau_1 \quad \Gamma, x : \mathsf{gen}(\Gamma, \tau_1) \vdash_{SML} e : \tau_2}{\Gamma \vdash_{SML} \mathsf{let}\ x = v\ \mathsf{in}\ e : \tau_2}$.

  Call $\sigma = \mathsf{gen}(\Gamma, \tau_1) = \forall\alpha_1 \ldots \forall\alpha_n.\tau_1$. By induction (C.2) for v

  (C.10) $$G_\Gamma, \Gamma' \vdash \mathcal{V}^M[\![v]\!] : \mathcal{V}^M[\![\tau_1]\!]$$

  By (weakening)

  (C.11) $$G_\Gamma, \Gamma', \alpha_1 : *^v, \ldots, \alpha_n : *^v \vdash \mathcal{V}^M[\![v]\!] : \mathcal{V}^M[\![\tau_1]\!]$$

  By induction on n find

  (C.12) $$G_\Gamma, \Gamma' \vdash \lambda\alpha_1 : *^v. \ldots \lambda\alpha_n : *^v.\mathcal{V}^M[\![v]\!] : \mathcal{S}^M[\![\sigma]\!]$$

  By Lemma 1 there is an $s \in \{*^v, \sharp\}$ such that

  (C.13) $$G_\sigma \vdash \mathcal{S}^M[\![\sigma]\!] : s$$

  By definition of $\mathsf{gen}(\Gamma, \tau_1)$, G_σ is included in G_Γ, hence

  (C.14) $$G_\Gamma \vdash \mathcal{S}^M[\![\sigma]\!] : s$$

  By induction (C.1) on e and using (C.14) for the well-formedness of the assumptions

  (C.15) $$G_\Gamma, \Gamma', x : \mathcal{S}^M[\![\sigma]\!] \vdash \mathcal{C}^M[\![e]\!] : \mathcal{C}^M[\![\tau_2]\!]$$

  Applying (abstraction) to (C.15) yields

  (C.16) $$G_\Gamma, \Gamma' \vdash \lambda x : \mathcal{S}^M[\![\sigma]\!].\mathcal{C}^M[\![e]\!] : \mathcal{S}^M[\![\sigma]\!] \to \mathcal{C}^M[\![\tau_2]\!]$$

  Apply (application) to (C.12) and (C.16) to obtain

  (C.17) $$G_\Gamma, \Gamma' \vdash (\lambda x : \mathcal{S}^M[\![\sigma]\!].\mathcal{C}^M[\![e]\!])\ (\lambda\alpha_1 : *^v. \ldots \lambda\alpha_n : *^v.\mathcal{V}^M[\![v]\!]) : \mathcal{C}^M[\![\tau_2]\!]$$

- $\dfrac{\Gamma \vdash_{SML} e_1 : \mathsf{nat} \quad \Gamma \vdash_{SML} e_2 : \tau \quad \Gamma \vdash_{SML} e_3 : \tau}{\Gamma \vdash_{SML} \mathsf{if0}\ e_1\ e_2\ e_3 : \tau}$. By induction

  (C.18) $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e_1]\!] : \mathcal{C}^M[\![\mathsf{nat}]\!]$$

  and for $i = 2, 3$

  (C.19) $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e_i]\!] : \mathcal{C}^M[\![\tau]\!]$$

  and by (weakening) and the definition of $\mathcal{C}^M$

  (C.20) $$G_\Gamma, \Gamma', y : \mathsf{nat} \vdash \mathcal{C}^M[\![e_i]\!] : \mathsf{M}\ \mathcal{V}^M[\![\tau]\!].$$

  By rule (if)

  (C.21) $$G_\Gamma, \Gamma', y : \mathsf{nat} \vdash \mathsf{if0}\ y\ \mathcal{C}^M[\![e_2]\!]\ \mathcal{C}^M[\![e_3]\!] : \mathsf{M}\ \mathcal{V}^M[\![\tau]\!]$$

  Applying (let) to (C.18) and (C.21) yields

  (C.22) $$G_\Gamma, \Gamma' \vdash \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^M[\![e_1]\!]\ \mathsf{in}\ \mathsf{if0}\ y\ \mathcal{C}^M[\![e_2]\!]\ \mathcal{C}^M[\![e_3]\!] : \mathcal{C}^M[\![\tau]\!]$$

- $\dfrac{\Gamma \vdash_{SML} e : \mathsf{nat}}{\Gamma \vdash_{SML} \mathsf{succ}\ e : \mathsf{nat}}$. By induction,

  (C.23) $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e]\!] : \mathcal{C}^M[\![\mathsf{nat}]\!]$$

  Starting from $y : \mathsf{nat}$, apply rule (succ)

  (C.24) $$G_\Gamma, \Gamma', y : \mathsf{nat} \vdash \mathsf{succ}\ y : \mathsf{nat}$$

and then rule (unit)

(C.25) $$G_\Gamma, \Gamma', y : \mathsf{nat} \vdash \mathsf{unit}\ (\mathsf{succ}\ y) : \mathsf{M\ nat}$$

Now recall $\mathsf{nat} \equiv \mathcal{V}^M[\![\mathsf{nat}]\!]$ and hence $\mathsf{M\ nat} \equiv \mathcal{C}^M[\![\mathsf{nat}]\!]$ and apply (let) to (C.23) and (C.25) to obtain

(C.26) $$G_\Gamma, \Gamma' \vdash \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (\mathsf{succ}\ y) : \mathcal{C}^M[\![\mathsf{nat}]\!]$$

- The case $\mathsf{pred}\ e$ is analogous.

■

D Proof of Theorem 5

Suppose $e_1 \to e_2$ in ML. Then $\mathcal{C}^M[\![e_1]\!] =_{ml} \mathcal{C}^M[\![e_2]\!]$.
Proof

- case $e_1 \equiv (\lambda x.e)@v$ and $e_2 \equiv e\{x := v\}$:

$$\begin{array}{ll}
 & \mathcal{C}^M[\![e_1]\!] \\
\equiv & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathcal{C}^M[\![\lambda x.e]\!]\ \mathsf{in}\ \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathcal{C}^M[\![v]\!]\ \mathsf{in}\ y_1\ y_2 \\
\equiv & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{unit}\ \lambda x : \mathcal{V}^M[\![\tau_2]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in} \\
 & \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathsf{unit}\ \mathcal{V}^M[\![v]\!]\ \mathsf{in}\ y_1\ y_2 \\
\mapsto_{ml} & \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathsf{unit}\ \mathcal{V}^M[\![v]\!]\ \mathsf{in}\ \lambda x : \mathcal{V}^M[\![\tau_2]\!].\mathcal{C}^M[\![e]\!]\ y_2 \\
\mapsto_{ml} & \lambda x : \mathcal{V}^M[\![\tau_2]\!].\mathcal{C}^M[\![e]\!]\ \mathcal{V}^M[\![v]\!] \\
\mapsto_{ml} & \mathcal{C}^M[\![e]\!]\{x := \mathcal{V}^M[\![v]\!]\}
\end{array}$$

For e_2, $\mathcal{C}^M[\![e_2]\!] \equiv \mathcal{C}^M[\![e\{x := v\}]\!]$ which is $\equiv$ to $\mathcal{C}^M[\![e]\!]\{x := \mathcal{V}^M[\![v]\!]\}$ by Lemma 23 since $x : \mathcal{V}^M[\![\tau]\!]$.

- case $e_1 \equiv (\mathsf{fix}\ f.e)@v$ and $e_2 \equiv e\{f := \mathsf{fix}\ f.e\}@v$:

$$\begin{array}{ll}
 & \mathcal{C}^M[\![e_1]\!] \\
\equiv & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathcal{C}^M[\![\mathsf{fix}\ f.e]\!]\ \mathsf{in}\ \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathcal{C}^M[\![v]\!]\ \mathsf{in}\ y_1\ y_2 \\
\equiv & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{unit}\ (\lambda y_2 : \mathcal{V}^M[\![\tau_2]\!]. \\
 & \qquad \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2)\ \mathsf{in} \\
 & \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathsf{unit}\ \mathcal{V}^M[\![v]\!]\ \mathsf{in}\ y_1\ y_2 \\
\mapsto_{ml} & \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathsf{unit}\ \mathcal{V}^M[\![v]\!]\ \mathsf{in}\ (\lambda y_2 : \mathcal{V}^M[\![\tau_2]\!]. \\
 & \qquad \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2)\ y_2 \\
\mapsto_{ml} & (\lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2)\ (\mathcal{V}^M[\![v]\!]) \\
\mapsto_{ml} & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ (\mathcal{V}^M[\![v]\!]) \\
\mapsto_{ml} & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathcal{C}^M[\![e]\!]\{f := \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_1 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\}\ \mathsf{in}\ y_1\ (\mathcal{V}^M[\![v]\!])
\end{array}$$

$$\begin{array}{ll} & \mathcal{C}^M[\![e_2]\!] \\ \equiv & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathcal{C}^M[\![e\{f := \mathsf{fix}\ f.e\}]\!]\ \mathsf{in}\ \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathcal{C}^M[\![v]\!]\ \mathsf{in}\ y_1\ y_2 \\ =_{ml} & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathcal{C}^M[\![e\{f := \mathsf{fix}\ f.e\}]\!]\ \mathsf{in}\ y_1\ (\mathcal{V}^M[\![v]\!]) \end{array}$$

Here, we appeal to Lemma 22 to obtain

$$\mathcal{C}^M[\![e]\!]\{f := \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\} \equiv \mathcal{C}^M[\![e\{f := \mathsf{fix}\ f.e\}]\!]$$

- case $e_1 \equiv \mathsf{let}\ x = v\ \mathsf{in}\ e$ and $e_2 \equiv e\{x := v\}$:

$$\begin{array}{ll} & \mathcal{C}^M[\![e_1]\!] \\ \equiv & (\lambda x : \mathcal{S}^M[\![\sigma]\!].\mathcal{C}^M[\![e]\!])\ (\lambda \alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!]) \\ \mapsto_{ml} & \mathcal{C}^M[\![e]\!]\{x := (\lambda \alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!])\} \end{array}$$

  Convertibility $=_{ml}$ to $\mathcal{V}^M[\![e_2]\!]$ follows from Lemma 24.

- case $e_1 \equiv \mathsf{if0}\ 0\ e'_1\ e'_2$ and $e_2 \equiv e'_1$:

$$\begin{array}{ll} & \mathcal{C}^M[\![e_1]\!] \\ \equiv & \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathsf{unit}\ \lceil 0 \rceil\ \mathsf{in}\ \mathsf{if0}\ y\ \mathcal{C}^M[\![e'_1]\!]\ \mathcal{C}^M[\![e'_2]\!] \\ \mapsto_{ml} & \mathsf{if0}\ \lceil 0 \rceil\ \mathcal{C}^M[\![e'_1]\!]\ \mathcal{C}^M[\![e'_2]\!] \\ \mapsto_{ml} & \mathcal{C}^M[\![e'_1]\!] \end{array}$$

- case $e_1 \equiv \mathsf{if0}\ (i+1)\ e'_1\ e'_2$ and $e_2 \equiv e'_2$: analogous.

- case $e_1 \equiv \mathsf{succ}\ i$ and $e_2 \equiv i + 1$:

$$\begin{array}{ll} & \mathcal{C}^M[\![e_1]\!] \\ \equiv & \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathsf{unit}\ \lceil i \rceil\ \mathsf{in}\ \mathsf{unit}\ (\mathsf{succ}\ y) \\ \mapsto_{ml} & \mathsf{unit}\ (\mathsf{succ}\ \lceil i \rceil) \\ =_{ml} & \mathsf{unit}\ \lceil i + 1 \rceil \end{array}$$

- case $e_1 \equiv \mathsf{pred}\ i$ and $e_2 \equiv i - 1$: analogous.

∎

Lemma 22 *Suppose* $\Gamma, f : \tau_2 \to \tau_1 \vdash_{SML} e' : \tau'$ *and* $\Gamma \vdash_{SML} \mathsf{fix}\ f.e : \tau_2 \to \tau_1$ *then* $\mathcal{C}^M[\![e'\{f := \mathsf{fix}\ f.e\}]\!] \equiv \mathcal{C}^M[\![e']\!]\{f := \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\}$.

Proof By induction on e, the only interesting case is $e' \equiv f$:

$$\begin{array}{ll} & \mathcal{C}^M[\![f\{f := \mathsf{fix}\ f.e\}]\!] \\ \equiv & \mathcal{C}^M[\![\mathsf{fix}\ f.e]\!] \\ \equiv & \mathsf{unit}\ (\lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2) \end{array}$$

$$\mathcal{C}^M[\![f]\!]\{f := \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\}$$
$$\equiv \mathsf{unit}\ (\lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow f\ \mathsf{in}\ y_1\ y_2)\{f := \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\}$$
$$\equiv \mathsf{unit}\ (\lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2)$$

■

Lemma 23 *Suppose* $\Gamma, x : \tau \vdash_{SML} e : \tau'$ *and* $\Gamma \vdash_{SML} v : \tau$ *then* $\mathcal{C}^M[\![e\{x := v\}]\!] \equiv \mathcal{C}^M[\![e]\!]\{x := \mathcal{V}^M[\![v]\!]\}$.

Proof Induction on the proof of $\Gamma, x : \tau \vdash_{SML} e : \tau'$. ■

Lemma 24 *Suppose* $\Gamma, x : \forall\alpha_1 \ldots \forall\alpha_n.\tau \vdash_{SML} e : \tau'$, $\Gamma \vdash_{SML} v : \tau$, *and* $\forall\alpha_1 \ldots \forall\alpha_n.\tau = \mathsf{gen}(\Gamma, \tau)$ *then* $\mathcal{C}^M[\![e\{x := v\}]\!] =_{ml} \mathcal{C}^M[\![e]\!]\{x := \lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!]\}$.

Proof By induction on e, the only interesting case is $e \equiv x$, $\tau' = \tau\{\alpha_i := \tau_i\}$:
$\mathcal{C}^M[\![x\{x := v\}]\!] \equiv \mathcal{C}^M[\![v]\!]\{\alpha_i := \mathcal{V}^M[\![\tau_i]\!]\}$

$$\begin{array}{ll} & \mathcal{C}^M[\![x]\!]\{x := \lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!]\} \\ \equiv & \mathsf{unit}\ (x\ \mathcal{V}^M[\![\tau_1]\!]\ \ldots\ \mathcal{V}^M[\![\tau_n]\!])\{x := \lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!]\} \\ \equiv & \mathsf{unit}\ ((\lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!])\ \mathcal{V}^M[\![\tau_1]\!]\ \ldots\ \mathcal{V}^M[\![\tau_n]\!]) \\ =_{ml} & \mathsf{unit}\ (\mathcal{V}^M[\![v]\!]\{\alpha_i := \mathcal{V}^M[\![\tau_i]\!]\}) \end{array}$$

■

E Proof of Lemma 4

Proof

- $\dfrac{\Gamma \vdash_{SML} e : \tau}{\Gamma \vdash_{SML} \mathsf{ref}\ e : \mathsf{ref}\ \tau}$. By induction,

  (E.1) $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e]\!] : \mathcal{C}^M[\![\tau]\!]$$

  By Lemma 1 and rule (application),

  (E.2) $$G_\Gamma, \Gamma' \vdash \mathsf{ref}_M\ \mathcal{V}^M[\![\tau]\!] : \mathcal{V}^M[\![\tau]\!] \to \mathsf{M}\ (\mathsf{ref}\ \mathcal{V}^M[\![\tau]\!])$$

  By rule (weakening)

  (E.3) $$G_\Gamma, \Gamma', y : \mathcal{V}^M[\![\tau]\!] \vdash \mathsf{ref}_M\ \mathcal{V}^M[\![\tau]\!] : \mathcal{V}^M[\![\tau]\!] \to \mathsf{M}\ (\mathsf{ref}\ \mathcal{V}^M[\![\tau]\!])$$

  By rule (application) applied to (E.3) and $y : \mathcal{V}^M[\![\tau]\!]$,

  (E.4) $$G_\Gamma, \Gamma', y : \mathcal{V}^M[\![\tau]\!] \vdash (\mathsf{ref}_M\ \mathcal{V}^M[\![\tau]\!])\ y : \mathsf{M}\ (\mathsf{ref}\ \mathcal{V}^M[\![\tau]\!])$$

  Finally, apply rule (let) to (E.1) and (E.4)

  (E.5) $$G_\Gamma, \Gamma' \vdash \mathsf{let}\ y : \mathcal{V}^M[\![\tau]\!] \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ (\mathsf{ref}_M\ \mathcal{V}^M[\![\tau]\!])\ y : \mathsf{M}\ (\mathsf{ref}\ \mathcal{V}^M[\![\tau]\!])$$

  which yields the claim by $\mathsf{M}\ (\mathsf{ref}\ \mathcal{V}^M[\![\tau]\!]) \equiv \mathcal{C}^M[\![(\mathsf{ref}\ \tau)]\!]$.

- $\dfrac{\Gamma \vdash_{SML} e : \mathsf{ref}\ \tau}{\Gamma \vdash_{SML}\ !\ e : \tau}$. By induction,

(E.6) $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e]\!] : \mathcal{C}^M[\![\mathsf{ref}\ \tau]\!]$$

By Lemma 1, rule (application), and rule (weakening),

(E.7) $$G_\Gamma, \Gamma', y : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] \vdash\ !_M\ \mathcal{V}^M[\![\tau]\!] : \mathsf{ref}\ \mathcal{V}^M[\![\tau]\!] \to \mathsf{M}\ \mathcal{V}^M[\![\tau]\!]$$

By rule (application) applied to (E.7) and $y : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!]$,

(E.8) $$G_\Gamma, \Gamma', y : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] \vdash (!_M\ \mathcal{V}^M[\![\tau]\!])\ y : \mathsf{M}\ \mathcal{V}^M[\![\tau]\!]$$

Finally, apply rule (let) to (E.6) and (E.8)

(E.9) $$G_\Gamma, \Gamma' \vdash \mathsf{let}\ y : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ (!_M\ \mathcal{V}^M[\![\tau]\!])\ y : \mathsf{M}\ \mathcal{V}^M[\![\tau]\!]$$

which yields the claim by $\mathsf{M}\ \mathcal{V}^M[\![\tau]\!] \equiv \mathcal{C}^M[\![\tau]\!]$.

- $\dfrac{\Gamma \vdash_{SML} e_1 : \mathsf{ref}\ \tau \quad \Gamma \vdash_{SML} e_2 : \tau}{\Gamma \vdash_{SML} e_1 := e_2 : ()}$. By induction,

(E.10) $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e_1]\!] : \mathcal{C}^M[\![\mathsf{ref}\ \tau]\!]$$

and

(E.11) $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e_2]\!] : \mathcal{C}^M[\![\tau]\!]$$

By Lemma 1, rule (application), and rule (weakening),

(E.12) $$G_\Gamma, \Gamma', y_1 : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!], y_2 : \mathcal{V}^M[\![\tau]\!] \vdash\ :=_M\ \mathcal{V}^M[\![\tau]\!] : \mathsf{ref}\ \mathcal{V}^M[\![\tau]\!] \to \mathcal{V}^M[\![\tau]\!] \to \mathsf{M}\ ()$$

Use rule (application) twice to get

(E.13) $$G_\Gamma, \Gamma', y_1 : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!], y_2 : \mathcal{V}^M[\![\tau]\!] \vdash (\ :=_M\ \mathcal{V}^M[\![\tau]\!])\ y_1\ y_2 : \mathsf{M}\ ()$$

Apply rule (let) to (E.11) and (E.13),

(E.14) $$G_\Gamma, \Gamma', y_1 : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] \vdash \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau]\!] \Leftarrow \mathcal{C}^M[\![e_2]\!]\ \mathsf{in}\ (\ :=_M\ \mathcal{V}^M[\![\tau]\!])\ y_1\ y_2 : \mathsf{M}\ ()$$

Apply rule (let) to (E.10) and (E.14) to obtain the claim

$$\begin{aligned} G_\Gamma, \Gamma' \vdash\ & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] \Leftarrow \mathcal{C}^M[\![e_1]\!]\ \mathsf{in} \\ & \quad \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau]\!] \Leftarrow \mathcal{C}^M[\![e_2]\!]\ \mathsf{in}\ (\ :=_M\ \mathcal{V}^M[\![\tau]\!])\ y_1\ y_2 : \mathsf{M}\ () \end{aligned}$$

where $\mathsf{M}\ () \equiv \mathcal{C}^M[\![()]\!]$.

■

F Operational Semantics of F_ω

The small-step operational semantics in this section are taken from work by Harper and Lillibridge [22,24]. Following Felleisen and Hieb [15], each of them defines a set of values v, a set of evaluation contexts E, and a set of reduction rules to be applied in an evaluation context.

F.1 Standard Call-by-Name

$$\begin{array}{lrcl} \text{values} & v & ::= & \lambda x : \sigma.e \mid \Lambda\alpha : \kappa.e \\ \text{evaluation contexts} & E & ::= & [\,] \mid E\mathtt{@}e \mid E[\sigma] \end{array}$$

$$\begin{array}{lcl} E[(\lambda x : \sigma.e_1)\mathtt{@}e_2] & \to_{std-cbn} & E[e_1\{x := e_2\}] \\ E[(\Lambda\alpha : \kappa.e)[\sigma]] & \to_{std-cbn} & E[e\{\alpha := \sigma\}] \end{array}$$

F.2 Standard Call-by-Value

$$\begin{array}{lrcl} \text{values} & v & ::= & x \mid \lambda x : \sigma.e \mid \Lambda\alpha : \kappa.e \\ \text{evaluation contexts} & E & ::= & [\,] \mid E\mathtt{@}e \mid v\mathtt{@}E \mid E[\sigma] \end{array}$$

$$\begin{array}{lcl} E[(\lambda x : \sigma.e)\mathtt{@}v] & \to_{std-cbv} & E[e\{x := v\}] \\ E[(\Lambda\alpha : \kappa.e)[\sigma]] & \to_{std-cbv} & E[e\{\alpha := \sigma\}] \end{array}$$

F.3 ML-style Call-by-Value

$$\begin{array}{lrcl} \text{values} & v & ::= & x \mid \lambda x : \sigma.e \mid \Lambda\alpha : \kappa.v \\ \text{evaluation contexts} & E & ::= & [\,] \mid E\mathtt{@}e \mid v\mathtt{@}E \mid \Lambda\alpha : \kappa.E \mid E[\sigma] \end{array}$$

$$\begin{array}{lcl} E[(\lambda x : \sigma.e)\mathtt{@}v] & \to_{ml-cbv} & E[e\{x := v\}] \\ E[(\Lambda\alpha : \kappa.v)[\sigma]] & \to_{ml-cbv} & E[v\{\alpha := \sigma\}] \end{array}$$