BRICS

Basic Research in Computer Science

BRICS RS-02-34 M. Rhiger: A Foundation for Embedded Languages

A Foundation for Embedded Languages

Morten Rhiger

BRICS Report Series RS-02-34
ISSN 0909-0878 August 2002

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/02/34/`

A Foundation for Embedded Languages

Morten Rhiger

BRICS *
Department of Computer Science
University of Aarhus †

August, 2002

Abstract

Recent work on embedding object languages into Haskell use "phantom types" (i.e., parameterized types whose parameter does not occur on the right-hand side of the type definition) to ensure that the embedded object-language terms are simply typed. But is it a safe assumption that *only* simply-typed terms can be represented in Haskell using phantom types? And conversely, can *all* simply-typed terms be represented in Haskell under the restrictions imposed by phantom types? In this article we investigate the conditions under which these assumptions are true: We show that these questions can be answered affirmatively for an idealized Haskell-like language and discuss to which extent Haskell can be used as a meta-language.

*Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.

†Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
E-mail: `mrhiger@brics.dk`
Home page: `http://www.brics.dk/~mrhiger`

New affiliation as of August 1, 2002:
The IT University in Copenhagen,
Glentevej 67, DK-2400 Copenhagen NV, Denmark.
E-mail: `mir@it-c.dk`
Home page: `http://www.it-c.dk/~mir`

Contents

List of Figures

1 Introduction

A program is typically written in terms of library routines. Once stabilized, it may itself become a library routine and be used in other programs. This bottom-up style of programming makes program development and maintenance easier and more efficient since the programmer can rely on well-understood routines. One of the goals of module systems, macro systems, and separate compilation is precisely to ease the definition of new routines and the use of existing routines in new programs.

Similarly, new programming languages typically arise from extending (or, as Steele says [41], "growing") existing languages with new features. Programming languages built from existing pieces are easier to design, implement, and understand since, in principle, they comprise well-understood components. For example, many realistic programming languages embody the λ-calculus as a core component. Yet, it was not until Scott that the λ-calculus itself was given a meaning. He warned that the hitherto "formalistic play with symbols" was useless and that, eventually, symbols must be given an interpretation [40]. Scott developed domain theory to provide a foundation for the λ-calculus. The result is the denotational semantics as we know it today in which the λ-calculus is used as a meta-language to define the semantics of programming languages [42].

A programming language can also be embedded into another, instead of directly giving its semantics in terms of, e.g., domains. The result combines the domain-specific operations (such as domain-specific values, their types, and operations on them) of the object language with the domain-independent linguistic features (such as evaluation strategy and type system) of the meta-language. (There is an unfortunate clash of terminology between formal *domains* as complete, partially ordered sets and informal *domains* as specific areas of applications.) Already in the 1960's, this style was envisioned by Landin [26] who observed that the design of programming languages splits into "the choice of written appearances of programs" and "the choice of abstract entities that can be referred to in the language."

Statically typed higher-order languages provide powerful domain-independent linguistic features. In particular, the module system, type classes, and polymorphic types of Haskell [15] make it a natural candidate to host domain-specific components [23]. Examples of domain-specific languages embedded into Haskell include languages for geometric region analysis [3], interactive 3D animation [13], image synthesis and manipulation [14], interfacing with Microsoft's Component Object Model [17, 25], music description and composition [24], accessing SQL databases [27], robot control [33], and real-time vision processing [37].

The domain-specific operations may be provided by an external system such as a graphical display showing images, a sound device playing music, a database processing SQL queries, or a robot executing orders; or they may be provided by an interpreter implemented in the meta-language. The domain-specific language may also provide a notion of well-typed domain-specific objects and restrict the range of the domain-specific operations to these objects. It is then crucial that only well-typed domain-specific objects can be expressed in the meta-language. When the object-language type system is sufficiently simple it can be expressed directly by the meta-language types of the domain-specific operations.

"Phantom types" were introduced to embed stronger object-language type systems into Haskell [17, 25, 27]. (We call a formal type parameter of a parameterized type "phantom" if all instances of the parameterized type are independent of the actual type parameters. A more informal characterization says that a phantom type is one that occurs on the left-hand side of a definition but not on the right, but this characterization does not catch situations where the right-hand side does contain the type parameter but the type parameter is unused.) For example, phantom types are instrumental for embedding COM objects and for embedding SQL queries into Haskell [17, 25, 27]. They also provide a key for using Haskell's type inferencer as a theorem prover to show that normalization functions as embodied in type-directed partial evaluation preserve types and yield normal forms [6, 7]. These applications of phantom types show that embedded type systems provide a powerful tool for both designing embedded programming languages and reasoning about program correctness. All existing embeddings using phantom types, however, take the following key properties for granted.

Type soundness: If a domain-specific object is *ill-typed* then it *cannot* be represented by a well-typed meta-language term.

Completeness: If a domain-specific object is *well-typed* then it *can* be represented by a well-typed meta-language term.

Together, these properties simply states that the meta-language type system accepts *exactly* the well-typed object-language terms. In other words, the object-language type system is correctly simulated by the meta-language. In this work we formally prove that these properties hold when embedding a monomorphic, higher-order language into an idealized Haskell-like meta-language.

This paper is organized as follows. Section 2 gives an implementation of an embedding of the terms and types of the simply typed λ-calculus into Haskell. In Section 3 we present a semantics of an idealized Haskell-like meta-language and then prove the two properties mentioned above. In Section 4 we investigate the use of Haskell as a meta-language, in particular, which precautions to take when Haskell is used as a meta-language. This section also discusses extensions to the object and meta-languages. Section 5 gives an overview of related work and Section 6 concludes.

2 An embedded higher-order language

Let us consider a domain-specific object language for manipulating integers and higher-order functions, i.e., an extension of the simply typed λ-calculus. We include addition of integers but, depending on the domain, one can add other base types, finite products, lists, etc. and primitive operations on these types. In Haskell, the object language can be represented by the following data type.

```
type Ide = String
data Raw = INT Int | ADD Raw Raw
         | VAR Ide | LAM Ide Raw | APP Raw Raw

instance Show Raw where
  showsPrec _ t = ...
```

Here we have equipped the data type with a function showing terms as strings. An alternative to using a data type is to directly encode terms as strings (if, for example, terms are passed to an external system as concrete syntax).

2.1 Higher-order abstract syntax

The key to embedding a typed higher-order object language is to use higher-order abstract syntax [35] as the interface to constructing terms. In higher-order abstract syntax, variables and bindings of the object language are modelled by variables and bindings of the meta-language, in this case Haskell. It is precisely this connection that enables meta-language types to model object-language types.

A higher-order abstract syntax usually carries higher-order functions in the representation of lambdas. The following data type illustrates such a minimal higher-order abstract syntax.

```
data HoExp = LAM (HoExp → HoExp) | APP HoExp HoExp
```

However, short of higher-order matching in Haskell, elements of this data type cannot be decomposed. Therefore, we build (and reason about) first-order syntax. In order to maintain the connection between bindings in the object language and the meta-language we construct first-order syntax using a higher-order interface as illustrated in Figure 1. The higher-order implementation hides the (first-order) constructors for variables and lambdas; instead a new higher-order constructor groups together the construction of a symbolic variable and the lambda that binds it.

To facilitate automatic generation of variable names, terms are abstracted over a list of fresh variable names. (Generally, these names must be distinct to prevent variable clashes in the raw representation of the term.) The type of terms are `[Ide] → Raw`. The same list of variable names is passed to all subterms, except in the case of lambdas where the first name is used to construct a symbolic variable and the rest of the names are passed to the body of the lambda. Each variable in a term is therefore assigned the ith name in the given list where i is the de Bruijn level [9] of the variable in the term. For example, using the names `["a", "b"]` to construct the first-order term of `lam (λx → app (lam (λy → y)) (lam (λz → z)))` yields

```
LAM "a" (APP (LAM "b" (VAR "b")) (LAM "b" (VAR "b")))
```

2.2 An embedded type discipline

Unfortunately, the higher-order abstract syntax does not impose any type constraints on the embedded language. For example, Haskell accepts the expression

```
module Lambda(Exp, int, add, lam, app) where
  ...
  type Exp = [Ide] → Raw

  int :: Int → Exp
  add :: Exp → Exp → Exp
  lam :: (Exp → Exp) → Exp
  app :: Exp → Exp → Exp

  int i   = λns → INT i
  add a b = λns → ADD (a ns) (b ns)
  lam f   = λ(n:ns) → LAM n (f (λz→VAR n) ns)
  app a b = λns → APP (a ns) (b ns)
```

Figure 1: Higher-order abstract syntax.

```
app (int 1) (lam (λx → x))
```

even though its value represents the ill-typed object-language term (here presented in Haskell syntax) `1 (λx → x)`.

Therefore, we restrict the higher-order constructors to only yield representations of well-typed terms. To this end, we make the following observations about constructing representations of well-typed object-language terms:

- `int` produces an object-language term of type `Int`.
- When `add` is applied to two object-language terms of type `Int` it produces an object-language term of type `Int`.
- When `app` is applied to two object-language terms of types `a → b` and `a` it produces a term of object-language type `b`.
- When `lam` is applied to a function mapping an object-language term of type `a` into an object-language term of type `b` it produces an object-language term of type `a → b`.

These dependencies are just verbal formulations of the standard type rules for the simply-typed λ-calculus. They suggest that the (polymorphic) types of the constructors actually could reflect the object-language types. We thus parameterize the type `Exp` over the type of the represented object-language term and we restrict the types of the constructors according to the observations above. The type parameter of `Exp` is a phantom type: it is not used in the right-hand side of the definition of `Exp`.

The complete embedding of the simply-typed λ-calculus into Haskell is shown in Figure 2. Terms are given by an abstract data type: a `newtype` declaration hides their representations as functions of type `[Ide] → Raw` and the module only exports the typed constructors.

```
module Lambda(Exp, int, add, lam, app) where

  type Ide = String
  data Raw = INT Int | ADD Raw Raw
           | VAR Ide | LAM Ide Raw | APP Raw Raw

  newtype Exp t = E ([Ide] → Raw)
  make (E a) ns = a ns

  int :: Int → Exp Int
  add :: Exp Int → Exp Int → Exp Int
  lam :: (Exp a → Exp b) → Exp (a → b)
  app :: Exp (a → b) → Exp a → Exp b

  int i   = E (λns → INT i)
  add a b = E (λns → ADD (make a ns) (make b ns))
  lam f   = E (λ(n:ns) → LAM n (make (f (E (λz→VAR n))) ns))
  app a b = E (λns → APP (make a ns) (make b ns))

  instance Show (Exp t) where
    showsPrec i (E a) = showsPrec i r where
      r = a [c:i | i←("":map show [1..]), c←['a'..'z']]
```

Figure 2: A typed higher-order language embedded into Haskell.

As an example, Haskell rejects the expression `app (int 1) (lam (λx → x))`, since this expression is an attempt to represent the ill-typed object-language term `1 (λx → x)`. On the other hand, the object-language term `λf → 2 + (f 1)` has the type `(Int → Int) → Int`. It can thus be represented in Haskell by `lam (λf → add (int 2) (app f (int 1)))` of type `Exp ((Int → Int) → Int)`. It is, however, not clear just from the implementation in Figure 2 and the inferred types that all ill-typed terms are ruled out. Nor is it clear that all well-typed term are not ruled out. In the next section we present an idealized Haskell-like meta-language in which these properties do hold.

3 Soundness and completeness

We introduce the the syntax and type system of a simply typed λ-calculus (the object language) with constants of base type. We then consider an idealized meta-language without let-polymorphism but with a set of predefined polymorphic constant symbols. We give the syntax, type system, and denotational semantics of the meta-language. Soundness follows from a proof using a Kripke logical relation and completeness follows by structural induction. Section 4 discusses the differences between the idealized meta-language and Haskell.

3.1 Object language

To reason about object-language types and terms we use the following concrete representations.

$$\begin{array}{rcl} \sigma & ::= & \mathsf{Int} \mid \sigma_1 \to \sigma_2 \\ u & ::= & i \mid u_1 + u_2 \mid x \mid \lambda x.u \mid u_1\, u_2 \end{array}$$

We let Δ range over finite mappings from variables to object-language types. The following type rules assigns types to object-language terms.

$$\frac{}{\Delta \vdash i : \mathsf{Int}} \qquad \frac{\Delta \vdash u_1 : \mathsf{Int} \quad \Delta \vdash u_2 : \mathsf{Int}}{\Delta \vdash u_1 + u_2 : \mathsf{Int}}$$

$$\frac{\Delta(x) = \sigma}{\Delta \vdash x : \sigma} \qquad \frac{\Delta[x : \sigma_1] \vdash u : \sigma_2}{\Delta \vdash \lambda x.u : \sigma_1 \to \sigma_2} \qquad \frac{\Delta \vdash u_1 : \sigma_1 \to \sigma_2 \quad \Delta \vdash u_2 : \sigma_1}{\Delta \vdash u_1\, u_2 : \sigma_2}$$

These rules are standard and standard results apply to them. We shall only need the following weakening lemma.

Lemma 1 (Weakening). *If for all* $x \in \mathrm{dom}(\Delta)$, $\Delta'(x) = \Delta(x)$ *and* $\Delta \vdash u : \sigma$ *then* $\Delta' \vdash u : \sigma$

Proof. By induction on the derivation of $\Delta \vdash u : \sigma$. □

3.2 Meta-language

Let β range over a set $\mathbf{B}$ of base types. The types of the meta-language consist of base types, function types, and types of representations of terms.

$$\tau \quad ::= \quad \beta \mid \tau_1 \to \tau_2 \mid \mathsf{Exp}\,\sigma$$

In the type $\mathsf{Exp}\,\sigma$, the intention is that σ is the type of the represented object-language term (as also suggested by the examples at the end of Section 2). By construction, this σ cannot itself contain occurrences of Exp. This means that the object language cannot itself express encoded terms, such as used in a multi-stage framework.

Let x range over an infinite set $\mathbf{V}$ of variable names and $c_{t_1 \cdots t_n}$ over a set $\mathbf{C}$ of constant symbols. The type of a constant $c_{t_1 \cdots t_n}$ may depend on $t_1, \ldots, t_n$, where each t_i is either a meta-language type τ or an object-language type σ. (The type of the constant will always be a meta-language type.) Hence, a constant symbol $c_{t_1 \cdots t_n}$ corresponds to a particular monomorphic instance of a polymorphic constant symbol c. For example, a polymorphic identity function could be introduced by the set of constants $\{\mathsf{id}_\tau \mid \text{type } \tau\}$ where each id_τ would be given type $\tau \to \tau$. (The assignment of types to constant symbols is discussed below.) The syntax of our small language is now given as follows.

$$e \quad ::= \quad x \mid \lambda x.e \mid e_1\, e_2 \mid c_{t_1 \cdots t_n} \mid \mathsf{rec}\, x.e$$

A signature $\Sigma = (\mathbf{B}, \mathbf{C})$ lists base types and constant symbols and additionally assigns types $\Sigma(c_{t_1\cdots t_n})$ to constants $c_{t_1\cdots t_n}$. A type context Γ is a finite mapping from variables to types. Given a signature Σ, the type rules for the language are as follows.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_\Sigma x : \tau} \qquad \frac{\Gamma[x : \tau_1] \vdash_\Sigma e : \tau_2}{\Gamma \vdash_\Sigma \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_\Sigma e_1 : \tau_2 \to \tau \quad \Gamma \vdash_\Sigma e_2 : \tau_2}{\Gamma \vdash_\Sigma e_1\, e_2 : \tau}$$

$$\frac{\Sigma(c_{t_1\cdots t_n}) = \tau}{\Gamma \vdash_\Sigma c_{t_1\cdots t_n} : \tau} \qquad \frac{\Gamma[x : \tau] \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathsf{rec}\, x.e : \tau}$$

We define a base signature $\Sigma_0 = (\mathbf{B}_0, \mathbf{C}_0)$ containing an addition function and higher-order constructors,

$$\begin{array}{rcl} \mathbf{B}_0 & = & \{\mathsf{Int}\} \\ \mathbf{C}_0 & = & \mathbf{Z} \cup \{+, \mathsf{int}, \mathsf{add}\} \cup \bigcup_{\text{types } \sigma_1,\sigma_2} \{\mathsf{lam}_{\sigma_1,\sigma_2}, \mathsf{app}_{\sigma_1,\sigma_2}\} \end{array}$$

where $\mathbf{Z}$ is the set of integers. The associated assignment of types to constant symbols is given as follows. (This is where the types of the higher-order constructors are restricted.)

$$\begin{array}{rclrcl} \Sigma_0(i) & = & \mathsf{Int},\ \text{for each } i \in \mathbf{Z} & \Sigma_0(\mathsf{int}) & = & \mathsf{Int} \to \mathsf{Exp\,Int} \\ \Sigma_0(+) & = & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} & \Sigma_0(\mathsf{add}) & = & \mathsf{Exp\,Int} \to \mathsf{Exp\,Int} \to \mathsf{Exp\,Int} \end{array}$$

$$\begin{array}{rcl} \Sigma_0(\mathsf{lam}_{\sigma_1,\sigma_2}) & = & (\mathsf{Exp}\,\sigma_1 \to \mathsf{Exp}\,\sigma_2) \to \mathsf{Exp}\,(\sigma_1 \to \sigma_2) \\ \Sigma_0(\mathsf{app}_{\sigma_1,\sigma_2}) & = & \mathsf{Exp}\,(\sigma_1 \to \sigma_2) \to \mathsf{Exp}\,\sigma_1 \to \mathsf{Exp}\,\sigma_2 \end{array}$$

It is straightforward to extend the meta-language with other base types and constant symbols. The formal requirements that constant symbols must satisfy are discussed in Section 3.3.

3.2.1 Denotational semantics

To model the construction of object-language terms, we assume the existence of a discretely ordered [45, page 120] set $\mathbf{L}$ that is capable of representing terms and we assume the existence of the following injective functions with mutually disjoint ranges,

$$\begin{array}{rclrcl} \mathit{INT} & \in & \mathbf{Z} \to \mathbf{L} & \mathit{ADD} & \in & \mathbf{L} \times \mathbf{L} \to \mathbf{L} \\ \mathit{VAR} & \in & \mathbf{V} \to \mathbf{L} & \mathit{LAM} & \in & \mathbf{V} \times \mathbf{L} \to \mathbf{L} \\ \mathit{APP} & \in & \mathbf{L} \times \mathbf{L} \to \mathbf{L} & & & \end{array}$$

where $\mathbf{N}$ is the set of natural numbers. As an example one might take $\mathbf{L}$ to be, e.g., the set of finite strings of symbols. We need not require the constructor functions to be surjective. That is, we do not rule out "syntactically invalid" elements in $\mathbf{L}$.

We draw fresh object-language variables from the same source as meta-language variables, namely the infinite set $\mathbf{V}$. In particular, we assume that there is an injective function $\mathit{fresh} \in \mathbf{N} \to \mathbf{V}$.

In the semantical development that follows we shall use standard domain-theoretic notation: For CPOs A and B, we write $A_\perp$ for the lifted domain, $\perp_A$ for the bottom element of this domain, $\mathbf{up} \in A \to A_\perp$ for the lifting injection, $f^* \in A_\perp \to B_\perp$ for the strict extension of $f \in A \to B_\perp$, $f_\perp \in A_\perp \to B_\perp$ for the strict version of $f \in A \to B$. We shall write $(\cdot,\cdot)_\perp \in A_\perp \times B_\perp \to (A \times B)_\perp$ for the strict pairing function, e.g., the function for which $(\mathbf{up}(a), \mathbf{up}(b))_\perp = \mathbf{up}(a, b)$ and where $(a, b)_\perp = \perp_{A\times B}$ if either $a = \perp_A$ or $b = \perp_B$. We write $A \to_\mathrm{c} B$ for the continuous function space between A and B. The partial order on a domain A is $\sqsubset_A$.

An interpretation $\mathcal{I}$ of a signature Σ assigns predomains (i.e., bottomless CPOs) to base types and values to constant symbols. For a given interpretation $\mathcal{I}$, the meaning of types is defined as follows. (In the third case, σ is the semantic counterpart of a phantom type: it does not affect the predomain assigned to $\mathsf{Exp}\,\sigma$.)

$$\begin{array}{rcl} [\![\beta]\!]_\mathcal{I} &=& (\mathcal{I}(\beta))_\perp \\ [\![\tau_1 \to \tau_2]\!]_\mathcal{I} &=& [\![\tau_1]\!]_\mathcal{I} \to_\mathrm{c} [\![\tau_2]\!]_\mathcal{I} \\ [\![\mathsf{Exp}\,\sigma]\!]_\mathcal{I} &=& \mathbf{N}_\perp \to_\mathrm{c} \mathbf{L}_\perp \end{array}$$

The interpretation furthermore assigns values to constant symbols such that if $\Sigma(c_{t_1\cdots t_n}) = \tau$ then $\mathcal{I}(c_{t_1\cdots t_n}) \in [\![\tau]\!]_\mathcal{I}$.

The meaning of a type context Γ is the labelled product

$$[\![\Gamma]\!]_\mathcal{I} = \prod_{x\in\mathrm{dom}(\Gamma)} [\![\Gamma(x)]\!]_\mathcal{I}$$

Finally, to any well-typed meta-language expression $\Gamma \vdash_\Sigma e : \tau$ we assign a continuous function $[\![e]\!]_\mathcal{I} \in [\![\Gamma]\!]_\mathcal{I} \to_\mathrm{c} [\![\tau]\!]_\mathcal{I}$. The following is a standard call-by-name semantics for functional languages.

$$\begin{array}{rcl} [\![x]\!]_\mathcal{I}\,\rho &=& \rho(x) \\ [\![\lambda x.e]\!]_\mathcal{I}\,\rho &=& \lambda a.[\![e]\!]_\mathcal{I}\,\rho[x \mapsto a] \\ [\![e_1\,e_2]\!]_\mathcal{I}\,\rho &=& [\![e_1]\!]_\mathcal{I}\,\rho\,([\![e_2]\!]_\mathcal{I}\,\rho) \\ [\![c_{t_1\cdots t_n}]\!]_\mathcal{I}\,\rho &=& \mathcal{I}(c_{t_1\cdots t_n}) \\ [\![\mathsf{rec}\,x.e]\!]_\mathcal{I}\,\rho &=& \bigsqcup_{i\in\omega} \phi^i(\perp_{[\![\tau]\!]_\mathcal{I}}), \quad \text{where } \phi(a) = [\![e]\!]_\mathcal{I}\,\rho[x \mapsto a] \end{array}$$

The initial signature Σ_0 is given the interpretation $\mathcal{I}_0$: First, the type Int is interpreted by the integers, i.e., $\mathcal{I}_0(\mathsf{Int}) = \mathbf{Z}$. For the constant symbols, the interpretation is defined as follows.

$$\begin{array}{rcl} \mathcal{I}_0(i) &=& \mathbf{up}(i) \\ \mathcal{I}_0(+) &=& \lambda x.\lambda y.x +_\perp y \\ \mathcal{I}_0(\mathsf{int}) &=& \lambda i.\lambda n.\mathit{INT}_\perp(i) \\ \mathcal{I}_0(\mathsf{add}) &=& \lambda v_1.\lambda v_2.\lambda n.\mathit{ADD}_\perp(v_1(n), v_2(n))_\perp \\ \mathcal{I}_0(\mathsf{lam}_{\sigma_1,\sigma_2}) &=& \lambda f.\lambda n.\mathit{LAM}_\perp(\mathit{fresh}_\perp(n), \\ && \qquad f\,(\lambda z.\mathit{VAR}_\perp(\mathit{fresh}_\perp(n)))\,(n +_\perp \mathbf{up}(1)))_\perp \\ \mathcal{I}_0(\mathsf{app}_{\sigma_1,\sigma_2}) &=& \lambda v_1.\lambda v_2.\lambda n.\mathit{APP}_\perp(v_1(n), v_2(n))_\perp \end{array}$$

The two last equations hold for all object-language types σ_1 and σ_2.

It follows by construction that the functions involved in defining the semantics of terms and in defining the initial interpretation of the constant symbols are all continuous.

We define the obvious injective representation functions mapping object-language terms into $\mathbf{L}$ as follows.

$$\begin{array}{rcl@{\qquad}rcl}
\lceil i \rceil & = & \mathit{INT}(i) & \lceil u_1 + u_2 \rceil & = & \mathit{ADD}(\lceil u_1 \rceil, \lceil u_2 \rceil) \\
\lceil x \rceil & = & \mathit{VAR}(x) & \lceil \lambda x.u \rceil & = & \mathit{LAM}(x, \lceil u \rceil) \\
\lceil u_1\, u_2 \rceil & = & \mathit{APP}(\lceil u_1 \rceil, \lceil u_2 \rceil) & & &
\end{array}$$

This representation function need not be surjective: Some elements in $\mathbf{L}$ may not correspond to any object-language term. It is our goal to show, however, that any element of $\mathbf{L}$ that is denoted by an expression in the meta-language corresponds to an object-language term and, furthermore, that these terms are well typed.

3.3 Soundness

In this section we formally state and prove soundness. We end up with Corollary 1, a strong soundness result stating that if an expression has type $\mathsf{Exp}\,\sigma$ then it either diverges or yields the representation of an object-language term of type σ, and Corollary 2, a weak result that corresponds to the informal statement from Section 1.

In the rest of this section we restrict our attention to the initial signature Σ_0 and the initial interpretation $\mathcal{I}_0$.

Definition 1. *Given a type context Δ and a type σ we define a subset of $\mathbf{L}_\perp$ as follows.*

$$\mathcal{T}_\sigma^\Delta = \{\perp_\mathbf{L}\} \cup \{\mathbf{up}(l) \in \mathbf{L}_\perp \mid \exists u. \lceil u \rceil = l \wedge \Delta \vdash u : \sigma\}$$

The set $\mathcal{T}_\sigma^\Delta$ contains exactly the elements in $\mathbf{L}_\perp$ that are undefined or that correspond to object-language terms of type σ in type context Δ.

Lemma 2 (Admissibility of $\mathcal{T}$). *For any type σ and type context Δ the relation $\mathcal{T}_\sigma^\Delta$ is admissible. That is, it is pointed (i.e., $\perp_\mathbf{L} \in \mathcal{T}_\sigma^\Delta$) and it is chain complete (i.e., if $(l_i)_{i\in\omega}$ is a chain in $\mathbf{L}_\perp$ such that for all $i \in \omega$, $l_i \in \mathcal{T}_\sigma^\Delta$ then $\bigsqcup_{i\in\omega} l_i \in \mathcal{T}_\sigma^\Delta$).*

Proof. Pointedness follows trivially by the definition. Chain completeness follows from the fact that $\mathbf{L}_\perp$ is discretely ordered and that any chain in $\mathbf{L}_\perp$ therefore eventually becomes constant. $\square$

Definition 2. *A* world *is a type context Δ. Worlds are ordered as follows.*

$$\Delta' \succeq \Delta \iff \forall x \in \mathrm{dom}(\Delta). x \in \mathrm{dom}(\Delta') \wedge \Delta'(x) = \Delta(x)$$

We let $\#\Delta = \max(\{-1\} \cup \{i \mid \mathit{fresh}(i) \in \mathrm{dom}(\Delta)\})$ denote the largest fresh variable number already bound in the context Δ (or -1 if the context is empty).

It is easy to show that $\succeq$ is reflexive and transitive, that $\Delta' \succeq \Delta$ implies $\#\Delta' \geq \#\Delta$, and that $n > \#\Delta$ implies $n+1 > \#\Delta[g:\sigma]$ where $g = \mathit{fresh}(n)$.

Terms that are well typed in one world are also well typed in any larger world, a result due to weakening.

Lemma 3 (Monotonicity of $\mathcal{T}$). *For any type σ, if two type contexts satisfy $\Delta' \succeq \Delta$ then $\mathcal{T}_\sigma^{\Delta'} \supseteq \mathcal{T}_\sigma^{\Delta}$.*

Proof. A consequence of Lemma 1. □

A *logical relation* is a type-indexed relation over the denoted values defined in such a way that it is closed under abstraction and application [30]. We define such a unary logical relation containing values that *behave well*: Informally, all values of base type are well-behaved, a function is well-behaved if it maps well-behaved values to well-behaved result, and a representation of an object-language term is well-behaved if, given a fresh variable index, it has the correct type in a given type context.

Definition 3 (Logical relation $\mathcal{R}$). *Given a type τ and a type context Δ we define a subset of $[\![\tau]\!]_{\mathcal{I}_0}$ as follows.*

(1) $\mathcal{R}_\beta^\Delta = [\![\beta]\!]_{\mathcal{I}_0}$

(2) $\mathcal{R}_{\tau_1\to\tau_2}^\Delta = \{f \in [\![\tau_1]\!]_{\mathcal{I}_0} \to_{\mathrm{c}} [\![\tau_2]\!]_{\mathcal{I}_0} \mid \forall\Delta' \succeq \Delta.\forall a \in \mathcal{R}_{\tau_1}^{\Delta'}.f(a) \in \mathcal{R}_{\tau_2}^{\Delta'}\}$

(3) $\mathcal{R}_{\mathsf{Exp}\,\sigma}^\Delta = \{f \in \mathbf{N}_\perp \to_{\mathrm{c}} \mathbf{L}_\perp \mid \forall n > \#\Delta.f(\mathbf{up}(n)) \in \mathcal{T}_\sigma^\Delta\}$

In case (2), the restriction that $f \in \mathcal{R}_{\tau_1\to\tau_2}^\Delta$ must satisfy $f(\mathcal{R}_{\tau_1}^{\Delta'}) \subseteq \mathcal{R}_{\tau_2}^{\Delta'}$ for any $\Delta' \succeq \Delta$ (and not just for $\Delta' = \Delta$) accounts for situations where an argument to such a function carries free object-language variables not contained in Δ. This is utilized in showing soundness for $\mathsf{lam}_{\sigma_1,\sigma_2}$ (which introduces free object-language variables) in Lemma 8 below. The restriction also ensures that the logical relation is monotone.

The goal is now to show that any well-typed meta-language expression denotes a well-behaved value.

Lemma 4 (Admissibility of $\mathcal{R}_\tau$). *For any type τ and type context Δ the relation $\mathcal{R}_\tau^\Delta$ is admissible.*

Proof. Using admissibility of $\mathcal{T}_\sigma^\Delta$ (Lemma 2). Chain completeness follows by induction on τ. For the case $\tau = \beta$ we use the fact $\mathcal{R}_\beta^\Delta$ is the constantly true predicate. For the case $\tau = \mathsf{Exp}\,\sigma$ we use chain completeness of $\mathcal{T}_\sigma^\Delta$ and the fact that for any chain of continuous functions $(f_n)_{n\in\omega}$, $\left(\bigsqcup_{n\in\omega} f_n\right)(x) = \bigsqcup_{n\in\omega} f_n(x)$. Pointedness follows by induction on τ using pointedness of $\mathcal{T}_\sigma^\Delta$ and $[\![\beta]\!]_{\mathcal{I}_0}$. □

Together with Definition 3, the following lemma shows that $\mathcal{R}_\tau$ is a Kripke logical relation [30].

Lemma 5 (Monotonicity of $\mathcal{R}_\tau$). *For any type τ, if two type contexts satisfy $\Delta' \succeq \Delta$ then $\mathcal{R}_\tau^{\Delta'} \supseteq \mathcal{R}_\tau^{\Delta}$.*

Proof. By induction on τ.

Case $\tau = \beta$. Holds trivially since $\mathcal{R}_\beta^{\Delta'} = \mathcal{R}_\beta^{\Delta}$.

Case $\tau = \tau_1 \to \tau_2$. Follows from the transitivity of $\succeq$.

Case $\tau = \mathsf{Exp}\,\sigma$. Follows from the monotonicity of $\mathcal{T}$ (Lemma 3) and using the fact that $\#\Delta' \geq \#\Delta$. □

We extend the logical relation of values and types to a relation of environments and type contexts. This gives a notion of well-behaved environments.

Definition 4. *For any type contexts Δ and Γ we define a subset of $[\![\Gamma]\!]_{\mathcal{I}_0}$ as follows.*

$$\mathcal{R}_\Gamma^{\Delta} = \{\rho \in [\![\Gamma]\!]_{\mathcal{I}_0} \mid \forall x \in \mathrm{dom}(\Gamma).\rho(x) \in \mathcal{R}_{\Gamma(x)}^{\Delta}\}$$

This extension preserves monotonicity.

Lemma 6 (Monotonicity of $\mathcal{R}_\Gamma$). *For all type contexts Γ, Δ, and Δ', if $\Delta' \succeq \Delta$ then $\mathcal{R}_\Gamma^{\Delta'} \supseteq \mathcal{R}_\Gamma^{\Delta}$.*

Proof. Follows from Lemma 5. □

Adding a well-behaved value to an already well-behaved environment results in another well-behaved environment.

Lemma 7. *If $d \in \mathcal{R}_\tau^{\Delta}$ and $\rho \in \mathcal{R}_\Gamma^{\Delta}$ then also $\rho[x \mapsto d] \in \mathcal{R}_{\Gamma[x:\tau]}^{\Delta}$.*

Proof. Follows from Definition 4. □

Using the results established so far, soundness amounts to showing that the semantics of a well-typed expression is well-behaved. The following lemma shows that this result holds for the constant symbols defined by the initial interpretation.

Lemma 8. *For any constant symbol $c_{t_1\cdots t_n} \in \mathbf{C}_0$ with $\Sigma_0(c_{t_1\cdots t_n}) = \tau$ and for any type context Δ we have $\mathcal{I}_0(c_{t_1\cdots t_n}) \in \mathcal{R}_\tau^{\Delta}$.*

Proof. By analysis of the individual constant symbols.

Case $c_{t_1\cdots t_n} = i$. Holds trivially since all values of base type are well behaved.

Case $c_{t_1\cdots t_n} = +$. Holds trivially since $+$ produces a value of base type which is always well behaved.

Case $c_{t_1\cdots t_n} = \mathsf{int}$. Since $\Delta \vdash i : \mathsf{Int}$ we have $\mathbf{up}(INT(i)) \in \mathcal{T}_{\mathsf{Int}}^{\Delta}$ for any $i \in \mathbf{Z}$ and type context Δ. Therefore $\lambda i.\lambda n.INT_\perp(\mathbf{up}(i)) = \lambda i.\lambda n.\mathbf{up}(INT(i)) \in \mathcal{R}_{\mathsf{Int}\to\mathsf{Exp}\,\mathsf{Int}}^{\Delta}$ as required.

Case $c_{t_1 \cdots t_n} = \mathsf{add}$. Given $\Delta' \succeq \Delta$, $\Delta'' \succeq \Delta'$, $v_1 \in \mathcal{R}^{\Delta'}_{\mathsf{Exp\,Int}}$, $v_2 \in \mathcal{R}^{\Delta''}_{\mathsf{Exp\,Int}}$, and $n > \#\Delta''$ we must show that $ADD_\perp(v_1(\mathbf{up}(n)), v_2(\mathbf{up}(n)))_\perp$ is either $\perp_\mathbf{L}$ or equal to $\mathbf{up}\,\lceil u \rceil$ for some u with $\Delta'' \vdash u : \mathsf{Int}$.

Since $n > \#\Delta'' \geq \#\Delta' \geq \#\Delta$ we immediately have $v_2(\mathbf{up}(n)) \in \mathcal{T}^{\Delta''}_{\mathsf{Int}}$. Using Lemma 3 we also get $v_1(\mathbf{up}(n)) \in \mathcal{T}^{\Delta'}_{\mathsf{Int}} \subseteq \mathcal{T}^{\Delta''}_{\mathsf{Int}}$. Therefore, either $v_1(\mathbf{up}(n)) = \perp_\mathbf{L}$ or $v_2(\mathbf{up}(n)) = \perp_\mathbf{L}$, in which case we are done since $ADD_\perp(v_1(\mathbf{up}(n)), v_2(\mathbf{up}(n)))_\perp = \perp_\mathbf{L}$, or we have terms u_1 and u_2 with $v_1(\mathbf{up}(n)) = \mathbf{up}\,\lceil u_1 \rceil$, $v_2(\mathbf{up}(n)) = \mathbf{up}\,\lceil u_2 \rceil$, $\Delta'' \vdash u_1 : \mathsf{Int}$, and $\Delta'' \vdash u_2 : \mathsf{Int}$. We set $u = u_1 + u_2$ and have

$$ADD_\perp(v_1(\mathbf{up}(n)), v_2(\mathbf{up}(n)))_\perp = \mathbf{up}\,\lceil u_1 + u_2 \rceil$$

and

$$\frac{\Delta'' \vdash u_1 : \mathsf{Int} \quad \Delta'' \vdash u_2 : \mathsf{Int}}{\Delta'' \vdash u_1 + u_2 : \mathsf{Int}}$$

as required.

Case $c_{t_1 \cdots t_n} = \mathsf{lam}_{\sigma_1,\sigma_2}$. Given $\Delta' \succeq \Delta$, $n > \#\Delta'$, $g = \mathit{fresh}(n)$, and $f \in \mathcal{R}^{\Delta'}_{\mathsf{Exp}\,\sigma_1 \to \mathsf{Exp}\,\sigma_2}$ we must show that

$$LAM_\perp(\mathbf{up}(g), f\,(\lambda z.\,VAR_\perp(\mathbf{up}(g)))\,(\mathbf{up}(n+1)))_\perp$$

is either $\perp_\mathbf{L}$ or equal to $\mathbf{up}\,\lceil u \rceil$ for some u with $\Delta' \vdash u : \sigma_1 \to \sigma_2$

We have $\lambda z.\,VAR_\perp(\mathbf{up}(g)) = \lambda z.\mathbf{up}(VAR(g)) \in \mathcal{R}^{\Delta'[g:\sigma_1]}_{\mathsf{Exp}\,\sigma_1}$ since $\lceil g \rceil = VAR(g)$ and $\Delta'[g : \sigma_1] \vdash g : \sigma_1$. Since also $n + 1 > \#(\Delta'[g : \sigma_1])$ we have

$$f\,(\lambda z.\,VAR_\perp(\mathbf{up}(g)))\,(\mathbf{up}(n+1)) \in \mathcal{T}^{\Delta'[g:\sigma_1]}_{\sigma_2}$$

This value must therefore either be $\perp_\mathbf{L}$, in which case we are done, or be equal to $\mathbf{up}\,\lceil u' \rceil$ for some term u' with $\Delta'[g : \sigma_1] \vdash u' : \sigma_2$. We set $u = \lambda g.u'$ and have

$$LAM_\perp(\mathbf{up}(g), f\,(\lambda z.\,VAR_\perp(\mathbf{up}(g)))\,(\mathbf{up}(n+1)))_\perp = \mathbf{up}\,\lceil \lambda g.u' \rceil$$

and

$$\frac{\Delta'[g : \sigma_1] \vdash u' : \sigma_2}{\Delta' \vdash \lambda g.u' : \sigma_1 \to \sigma_2}$$

as required.

Case $c_{t_1 \cdots t_n} = \mathsf{app}_{\sigma_1,\sigma_2}$. (Follows the same structure as the case for add.) Given $\Delta' \succeq \Delta$, $\Delta'' \succeq \Delta'$, $v_1 \in \mathcal{R}^{\Delta'}_{\mathsf{Exp}\,(\sigma_1 \to \sigma_2)}$, $v_2 \in \mathcal{R}^{\Delta''}_{\mathsf{Exp}\,\sigma_1}$, and $n > \#\Delta''$ we must show that $APP_\perp(v_1(\mathbf{up}(n)), v_2(\mathbf{up}(n)))_\perp$ is either $\perp_\mathbf{L}$ or equal to $\mathbf{up}\,\lceil u \rceil$ for some u with $\Delta'' \vdash u : \sigma_2$.

Since $n > \#\Delta'' \geq \#\Delta' \geq \#\Delta$ we immediately have $v_2(\mathbf{up}(n)) \in \mathcal{T}^{\Delta''}_{\sigma_1}$. Using Lemma 3 we also get $v_1(\mathbf{up}(n)) \in \mathcal{T}^{\Delta'}_{\sigma_1 \to \sigma_2} \subseteq \mathcal{T}^{\Delta''}_{\sigma_1 \to \sigma_2}$. Therefore, either $v_1(\mathbf{up}(n)) = \perp_\mathbf{L}$ or $v_2(\mathbf{up}(n)) = \perp_\mathbf{L}$, in which case we are done since $APP_\bot(v_1(\mathbf{up}(n)), v_2(\mathbf{up}(n)))_\bot = \bot_\mathbf{L}$, or we have terms u_1 and u_2 with $v_1(\mathbf{up}(n)) = \mathbf{up}\,\lceil u_1 \rceil$, $v_2(\mathbf{up}(n)) = \mathbf{up}\,\lceil u_2 \rceil$, $\Delta'' \vdash u_1 : \sigma_1 \to \sigma_2$, and $\Delta'' \vdash u_2 : \sigma_1$. We set $u = u_1\, u_2$ and have

$$APP_\bot(v_1(\mathbf{up}(n)), v_2(\mathbf{up}(n)))_\bot = \mathbf{up}\,\lceil u_1\, u_2 \rceil$$

and

$$\frac{\Delta'' \vdash u_1 : \sigma_1 \to \sigma_2 \quad \Delta'' \vdash u_2 : \sigma_1}{\Delta'' \vdash u_1\, u_2 : \sigma_2}$$

as required. ☐

Other constant symbols can be added to the meta-language if they satisfy Lemma 8. The following main result states that evaluating a well-typed expression in a well-behaved environment yields a well-behaved value.

Theorem 1 (Soundness). *In the initial interpretation $\mathcal{I}_0$ of the initial signature Σ_0, if $\Gamma \vdash_{\Sigma_0} e : \tau$ and $\rho \in \mathcal{R}^\Delta_\Gamma$ then $[\![e]\!]_{\mathcal{I}_0}\, \rho \in \mathcal{R}^\Delta_\tau$.*

Proof. By structural induction on e.

Case $e = x$. Then $\Gamma(x) = \tau$ and $[\![x]\!]_{\mathcal{I}_0}\, \rho = \rho(x)$ so $[\![x]\!]_{\mathcal{I}_0}\, \rho \in \mathcal{R}^\Delta_\tau$ follows from Definition 4.

Case $e = \lambda x.e'$. Then $\tau = \tau_1 \to \tau_2$ where $\Gamma[x : \tau_1] \vdash_{\Sigma_0} e' : \tau_2$ and $[\![\lambda x.e']\!]_{\mathcal{I}_0}\, \rho = \lambda a.[\![e']\!]_{\mathcal{I}_0}\, \rho[x \mapsto a]$. We must show $[\![\lambda x.e']\!]_{\mathcal{I}_0}\, \rho \in \mathcal{R}^\Delta_{\tau_1 \to \tau_2}$. So given $\Delta' \succeq \Delta$ and $d \in \mathcal{R}^{\Delta'}_{\tau_1}$ we must show that $[\![e']\!]_{\mathcal{I}_0}\, \rho[x \mapsto d] \in \mathcal{R}^{\Delta'}_{\tau_2}$. From Lemma 6 it follows that $\rho \in \mathcal{R}^{\Delta'}_\Gamma$ and then from Lemma 7 we have that $\rho[x \mapsto d] \in \mathcal{R}^{\Delta'}_{\Gamma[x:\tau_1]}$. Thus, $[\![e']\!]_{\mathcal{I}_0}\, \rho[x \mapsto d] \in \mathcal{R}^{\Delta'}_{\tau_2}$ follows from the induction hypothesis.

Case $e = e_1\, e_2$. Then $\Gamma \vdash_{\Sigma_0} e_1 : \tau_2 \to \tau$, $\Gamma \vdash_{\Sigma_0} e_2 : \tau_2$, and $[\![e_1\, e_2]\!]_{\mathcal{I}_0}\, \rho = [\![e_1]\!]_{\mathcal{I}_0}\, \rho\, ([\![e_2]\!]_{\mathcal{I}_0}\, \rho)$. Using the induction hypothesis twice we get $[\![e_1]\!]_{\mathcal{I}_0}\, \rho \in \mathcal{R}^\Delta_{\tau_2 \to \tau}$ and $[\![e_2]\!]_{\mathcal{I}_0}\, \rho \in \mathcal{R}^\Delta_{\tau_2}$. Using reflexivity of $\succeq$, Definition 3(2) gives $[\![e_1\, e_2]\!]_{\mathcal{I}_0}\, \rho \in \mathcal{R}^\Delta_\tau$ as required.

Case $e = c_{t_1 \cdots t_n}$. Follows from Lemma 8.

Case $\mathsf{rec}\, x.e'$. Using pointedness for the base case and Lemma 7 for the induction step an induction on i shows that $\phi^i(\bot) \in \mathcal{R}^\Delta_\tau$ for all i, where $\phi(a) = [\![e']\!]_{\mathcal{I}_0}\, \rho[x \mapsto a]$. The result then follows from admissibility (Lemma 4). ☐

The following corollary states that an expression of type $\mathsf{Exp}\,\sigma$ evaluates to a function that, when passed an initial variable index, either diverges or yields a representation of an object-language term of type σ.

Corollary 1. *For any σ and e if $\emptyset \vdash_{\Sigma_0} e : \mathsf{Exp}\,\sigma$ then either*

(1) $[\![e]\!]_{\mathcal{I}_0}\, \emptyset\, (\mathbf{up}(0)) = \bot_\mathbf{L}$, *or*

(2) $[\![e]\!]_{\mathcal{I}_0}\, \emptyset\, (\mathbf{up}(0)) = \mathbf{up}\,\lceil u \rceil$ *for some term u with $\emptyset \vdash u : \sigma$.*

Proof. Since $\emptyset \in \mathcal{R}^{\emptyset}_{\emptyset}$, Theorem 1 gives

$$[\![e]\!]_{\mathcal{I}_0}\,\emptyset \in \mathcal{R}^{\emptyset}_{\mathsf{Exp}\,\sigma} = \{f \in \mathbf{N}_\perp \rightarrow_{\mathrm{c}} \mathbf{L}_\perp \mid \forall n > \#\emptyset . f(\mathbf{up}(n)) \in \mathcal{T}^{\emptyset}_{\sigma}\}$$

In particular, since $\#\emptyset = -1$

$$[\![e]\!]_{\mathcal{I}_0}\,\emptyset\,(\mathbf{up}(0)) \in \mathcal{T}^{\emptyset}_{\sigma} = \{\perp_{\mathbf{L}}\} \cup \{\mathbf{up}(l) \in \mathbf{L}_\perp \mid \exists u . \lceil u \rceil = l \wedge \emptyset \vdash u : \sigma\}$$

From which the result follows. ☐

The above corollary is a strong soundness result that relates types of meta-language expressions to types of object-language terms. In analogy with the informal statement in Section 1 we also have the following weaker result about the mere existence of object-language terms.

Corollary 2. *Given an object-language term u, if, for any σ, $\emptyset \not\vdash u : \sigma$ then there exist no e and σ with $\emptyset \vdash_{\Sigma_0} e : \mathsf{Exp}\,\sigma$ and $[\![e]\!]_{\mathcal{I}_0}\,\emptyset\,(\mathbf{up}(0)) = \mathbf{up}\,\lceil u \rceil$.*

Proof. Follows from Corollary 1 by a proof of contradiction. ☐

3.4 Completeness

We show that for any well-typed object-language term there exists a meta-language expression that evaluates to a representation of the term. It is straightforward to translate an object-language term into a Haskell expression that builds a representation of that term. Using the definitions in Figure 2, such a translation can be defined by induction over object-language terms as follows. (We rely on the requirement that the object-languages identifiers are a subset of the Haskell identifiers.)

$$\begin{array}{rcl@{\qquad}rcl}
\langle x \rangle & = & x & \langle \lambda x.u \rangle & = & \mathtt{lam}\ (\lambda x \rightarrow \langle u \rangle) \\
\langle i \rangle & = & \mathtt{int}\,i & \langle u_1\, u_2 \rangle & = & \mathtt{app}\,\langle u_1 \rangle\,\langle u_2 \rangle \\
\langle u_1 + u_2 \rangle & = & \mathtt{add}\,\langle u_1 \rangle\,\langle u_2 \rangle & & &
\end{array}$$

However, the idealized meta-language used in our formal development provides type-indexed constant symbols instead of polymorphic functions such as `lam` and `app`. We therefore present the translation as an extended type system where the constant symbols are indexed by the correct types. The translation uses int, add, $\mathsf{lam}_{\sigma_1,\sigma_2}$, and $\mathsf{app}_{\sigma_1,\sigma_2}$ to build integers, additions, lambdas, and applications. In addition, the type rules carry symbolic variable indices so that these can be related to the corresponding meta-language variables in the proofs below.

We let a translation context Ξ range over finite mappings from variables to pairs of types and integers. It is the intention that these integers denote the de Bruijn levels of the free variables in the term. The following rules then simply add a translation to the type system presented in Section 3.1. They define a predicate $n; \Xi \vdash e : \sigma \,/\, e$ expressing that at de Bruijn level n and in translation context Ξ the object-language term u has type σ and it translates to the meta-language expression e.

$$\frac{}{n;\Xi \vdash i : \mathsf{Int} \,/\, \mathsf{int}\, i} \qquad \frac{n;\Xi \vdash u_1 : \mathsf{Int} \,/\, e_1 \quad n;\Xi \vdash u_2 : \mathsf{Int} \,/\, e_2}{n;\Xi \vdash u_1 + u_2 : \mathsf{Int} \,/\, \mathsf{add}\, e_1\, e_2}$$

$$\frac{\Xi(x) = (\sigma, m)}{n;\Xi \vdash x : \sigma \,/\, x} \qquad \frac{n+1;\Xi[x : (\sigma_1, n)] \vdash u : \sigma_2 \,/\, e}{n;\Xi \vdash \lambda x.u : \sigma_1 \to \sigma_2 \,/\, \mathsf{lam}_{\sigma_1,\sigma_2}\, (\lambda x.e)}$$

$$\frac{n;\Xi \vdash u_1 : \sigma_1 \to \sigma_2 \,/\, e_1 \quad n;\Xi \vdash u_2 : \sigma_1 \,/\, e_2}{n;\Xi \vdash u_1\, u_2 : \sigma_2 \,/\, \mathsf{app}_{\sigma_1,\sigma_2}\, e_1\, e_2}$$

As shown by the following lemma, if an object-language term can be typed in the type system presented in Section 3.1 then it can also be translated in the system above. We say that a type context Δ and a translation context Ξ are related by $\Delta \approx \Xi$ when $\mathrm{dom}(\Delta) = \mathrm{dom}(\Xi)$ and when for all $x \in \mathrm{dom}(\Delta)$, $\Delta(x) = \sigma$ implies $\Xi(x) = (\sigma, m)$ for some m.

Lemma 9. *If $\Delta \vdash u : \sigma$, $\Delta \approx \Xi$, and n denotes the number of free variables in u then there exists an expression e such that $n;\Xi \vdash u : \sigma \,/\, e$.*

Proof. By induction on the derivation of $\Delta \vdash u : \sigma$. □

We first relate the type of the object-language term and the type of the meta-language expression it is translated to.

Lemma 10. *For any translation context Ξ, type context Γ, $n \in \mathbf{Z}$, and type σ, if $n;\Xi \vdash u : \sigma \,/\, e$ and if $\Gamma(x) = \mathsf{Exp}\,\sigma$ when $\Xi(x) = (\sigma, n)$ then $\Gamma \vdash_{\Sigma_0} e : \mathsf{Exp}\,\sigma$*

Proof. By induction on the derivation of $n;\Xi \vdash u : \sigma \,/\, e$. □

Not all well-typed object-language terms can be encoded *syntactically* in the meta-language since the fresh variable names drawn from the predetermined list might differ from the intended variable names. We will show, however, that a term and its encoding as given above are equal up to renaming of bound variables.

Definition 5 (Substitutions and α-convertibility). *A* substitution *s is a finite mapping of variables to variables. We let $(s\backslash x)$ be the restricted substitution satisfying $(s\backslash x)\,(x) = x$ and $(s\backslash x)\,(y) = s(y)$ for $x \neq y$. Substitutions extend to terms in such a way that $s(\lambda x.u) = \lambda x.u'$ where $u' = (s\backslash x)\,(u)$.*

*We say that a term u can be α-*converted *to a term u' under a substitution s, written $s \vdash u \longrightarrow u'$, if renaming bound variables in u according to s yields u'. The relation is defined by the following rules.*

$$\frac{}{s \vdash i \longrightarrow i} \qquad \frac{s(x) = y}{s \vdash x \longrightarrow y} \qquad \frac{y \notin (s\backslash x)\,(u) \quad s[x \mapsto y] \vdash u \longrightarrow u'}{s \vdash \lambda x.u \longrightarrow \lambda y.u'}$$

$$\frac{s \vdash u_1 \longrightarrow u_1' \quad s \vdash u_2 \longrightarrow u_2'}{s \vdash u_1 + u_2 \longrightarrow u_1' + u_2'} \qquad \frac{s \vdash u_1 \longrightarrow u_1' \quad s \vdash u_2 \longrightarrow u_2'}{s \vdash u_1\, u_2 \longrightarrow u_1'\, u_2'}$$

where $y \notin u$ indicates that the variable y does not occur (free or bound) in the expression u.

Two closed terms u_1 and u_2 are α-congruent in the traditional sense [1, page 26] if they are related by $\emptyset \vdash u_1 \longrightarrow u_2$.

Definition 6. *Given a translation context Ξ we define a substitution $\Xi^{\mathcal{S}}$ and an environment $\Xi^{\mathcal{E}}$ such that if $\mathit{fresh}(n) = g$ then*

(1) $(\Xi[x : (\sigma, n)])^{\mathcal{S}} = \Xi^{\mathcal{S}}[x \mapsto g]$

(2) $(\Xi[x : (\sigma, n)])^{\mathcal{E}} = \Xi^{\mathcal{E}}[x \mapsto \lambda z.\mathbf{up}(\mathit{VAR}(g))]$

If an object-language term u can be translated to a meta-language expression e then e will evaluate to a representation of u (modulo renaming).

Lemma 11. *For any integer n, term u, expression e, type σ, and translation context Ξ with $\mathrm{range}(\Xi) = \{(\sigma, i) \mid i < n\}$, if $n; \Xi \vdash u : \sigma \,/\, e$ then there exists a term u' such that $\Xi^{\mathcal{S}} \vdash u \longrightarrow u'$ and $[\![e]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) = \mathbf{up}\,\lceil u' \rceil$.*

Proof. By induction on the derivation of $n; \Xi \vdash u : \sigma \,/\, e$.

Case $n; \Xi \vdash i : \mathsf{Int} \,/\, \mathsf{int}\, i$. Then $[\![\mathsf{int}\, i]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) = \mathbf{up}(\mathit{INT}(i)) = \mathbf{up}\,\lceil i \rceil$ and indeed $\Xi^{\mathcal{S}} \vdash i \longrightarrow i$.

Case $n; \Xi \vdash u_1 + u_2 : \mathsf{Int} \,/\, \mathsf{add}\, e_1\, e_2$ where $n; \Xi \vdash u_1 : \mathsf{Int} \,/\, e_1$ and $n; \Xi \vdash u_2 : \mathsf{Int} \,/\, e_2$. Then by two applications of the induction hypothesis there exist u_1' and u_2' such that $\Xi^{\mathcal{S}} \vdash u_1 \longrightarrow u_1'$, $\Xi^{\mathcal{S}} \vdash u_2 \longrightarrow u_2'$, $[\![e_1]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) = \mathbf{up}\,\lceil u_1' \rceil$, and $[\![e_2]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) = \mathbf{up}\,\lceil u_2' \rceil$. We therefore have

$$\begin{aligned}
&[\![\mathsf{add}\, e_1\, e_2]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) \\
&\qquad = \mathit{ADD}_\perp\big([\![e_1]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)), [\![e_2]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n))\big)_\perp \\
&\qquad = \mathit{ADD}_\perp(\mathbf{up}\,\lceil u_1' \rceil, \mathbf{up}\,\lceil u_2' \rceil)_\perp \\
&\qquad = \mathbf{up}(\mathit{ADD}(\lceil u_1' \rceil, \lceil u_2' \rceil)) \\
&\qquad = \mathbf{up}\,\lceil u_1' + u_2' \rceil
\end{aligned}$$

and indeed $\Xi^{\mathcal{S}} \vdash u_1 + u_2 \longrightarrow u_1' + u_2'$.

Case $n; \Xi \vdash x : \sigma \,/\, x$ where $\Xi(x) = (\sigma, m)$. Then, with $g = \mathit{fresh}(m)$,

$$[\![x]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) = \Xi^{\mathcal{E}}(x)\, (\mathbf{up}(n)) = \mathbf{up}(\mathit{VAR}(g)) = \mathbf{up}\,\lceil g \rceil$$

and indeed $\Xi^{\mathcal{S}} \vdash x \longrightarrow g$.

Case $n; \Xi \vdash \lambda x.u : \sigma_1 \to \sigma_2 \,/\, \mathsf{lam}_{\sigma_1,\sigma_2}\, (\lambda x.e)$ where $n+1; \Xi[x : (\sigma_1, n)] \vdash u : \sigma_2 \,/\, e$. Then with $g = \mathit{fresh}(n)$, we have from the induction hypothesis that there exists u' such that $\Xi[x : (\sigma_1, n)]^{\mathcal{S}} \vdash u \longrightarrow u'$. Definition 6(1) then gives $\Xi^{\mathcal{S}}[x \mapsto g] \vdash u \longrightarrow u'$. Together with Definition 6(2) the induction hypothesis also gives

$$\begin{aligned}
[\![\lambda x.e]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) &= \mathit{LAM}_\perp\big(\mathbf{up}(g), [\![e]\!]_{\mathcal{I}_0}\, \Xi[x : (\sigma_1, n)]^{\mathcal{E}}\, (\mathbf{up}(n+1))\big)_\perp \\
&= \mathit{LAM}_\perp\big(\mathbf{up}(g), \mathbf{up}\,\lceil u' \rceil\big)_\perp \\
&= \mathbf{up}(\mathit{LAM}(g, \lceil u' \rceil)) \\
&= \mathbf{up}\,\lceil \lambda g.u' \rceil
\end{aligned}$$

Since $\text{range}(\Xi) \subseteq \{(\sigma, i) \mid i < n\}$ we have that for all z, $\Xi^{\mathcal{S}}(z) \neq g$. Hence indeed $\Xi^{\mathcal{S}} \vdash \lambda x.u \longrightarrow \lambda g.u'$.

Case $n; \Xi \vdash u_1\, u_2 : \sigma_2 \,/\, \mathsf{app}_{\sigma_1,\sigma_2}\, e_1\, e_2$ where $n; \Xi \vdash u_1 : \sigma_1 \,/\, e_1$ and $n; \Xi \vdash u_2 : \sigma_1 \to \sigma_2 \,/\, e_2$. (Follows the same structure as the case for addition.) Then by induction hypothesis there exist u_1' and u_2' such that $\Xi^{\mathcal{S}} \vdash u_1 \longrightarrow u_1'$, $\Xi^{\mathcal{S}} \vdash u_2 \longrightarrow u_2'$, $[\![e_1]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) = \mathbf{up}\, \lceil u_1' \rceil$, and $[\![e_2]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) = \mathbf{up}\, \lceil u_2' \rceil$. We therefore have

$$\begin{aligned}
&[\![\mathsf{app}_{\sigma_1,\sigma_2}\, e_1\, e_2]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) \\
&\quad = APP_{\perp}\big([\![e_1]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)), [\![e_2]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n))\big)_{\perp} \\
&\quad = APP_{\perp}(\mathbf{up}\, \lceil u_1' \rceil, \mathbf{up}\, \lceil u_2' \rceil)_{\perp} \\
&\quad = \mathbf{up}(APP(\lceil u_1' \rceil, \lceil u_2' \rceil)) \\
&\quad = \mathbf{up}\, \lceil u_1'\, u_2' \rceil
\end{aligned}$$

and indeed $\Xi^{\mathcal{S}} \vdash u_1\, u_2 \longrightarrow u_1'\, u_2'$. □

Theorem 2 (Completeness). *Given an object-language term u, if, for some σ, $\emptyset \vdash u : \sigma$ then there exists an expression e with $\emptyset \vdash_{\Sigma_0} e : \mathsf{Exp}\,\sigma$ and $[\![e]\!]_{\mathcal{I}_0}\, \emptyset\, (\mathbf{up}(0)) = \mathbf{up}\, \lceil u' \rceil$ for some u' with $\emptyset \vdash u \longrightarrow u'$.*

Proof. The term u must be closed. Hence, by Lemma 9, there exists an expression e with $0; \emptyset \vdash u : \sigma \,/\, e$. This e is the candidate we're seeking. Indeed, from Lemma 10, $\emptyset \vdash_{\Sigma_0} e : \mathsf{Exp}\,\sigma$ and, from Lemma 11, there exists a term u' with $[\![e]\!]_{\mathcal{I}_0}\, \emptyset\, (\mathbf{up}(n)) = \mathbf{up}\, \lceil u' \rceil$ and $\emptyset \vdash u \longrightarrow u'$. □

4 Haskell as a meta-language

There are a number of issues that must be addressed to use Haskell as a meta-language for embedded languages. Differences between Haskell and the idealized meta-language presented in the previous section influence the conditions under which Haskell can safely be used as a meta-language.

There are no built-in types $\mathsf{Exp}\,\sigma$ or constant symbols $\mathsf{lam}_{\sigma_1,\sigma_2}$ and $\mathsf{app}_{\sigma_1,\sigma_2}$ in Haskell. Instead, we must provide global definitions as in Figure 2 and argue that their semantics agree with the semantics of $\mathsf{lam}_{\sigma_1,\sigma_2}$ and $\mathsf{app}_{\sigma_1,\sigma_2}$. We have designed the semantics of these constant symbols to match the Haskell implementation. Furthermore, in Haskell, the types of these symbols are restricted outside the module of their implementation. In contrast, in the formal development they are given as constant symbols of the restricted types.

The formal treatment in this work uses strict data type constructors. In contrast, Haskell's constructors are non-strict. For the soundness result, the strict constructors guarantee that we only observe the types of "finite" object-language terms. In Haskell, it is possible to observe the shape of "infinite" terms using the top-level read-eval-print loop. For example, the following well-typed Haskell expression uses recursion to build an infinite object-language term.

```
let f = lam (λx → app f x) in f
```

Its meaning is the representation of the limit of the chain of incomplete terms

```
   ⊥
⊑  LAM "x1" (APP ⊥ (VAR "x1"))
⊑  LAM "x1" (APP (LAM "x2" (APP ⊥ (VAR "x2"))) (VAR "x1"))
⊑  ...
```

where $\perp$ denotes the bottom element of the CPO of (non-strict) data types. In a formal account for the soundness property using non-strict data types it may be desirable not to give types to such infinitely expanding terms. Instead, the soundness property might state that if a "finite" object-language term is denoted by an expression of type $\mathtt{Exp}\,\sigma$ then it has type σ.

Haskell's function space is lifted, e.g. $[\![\tau_1 \to \tau_2]\!]_{\mathcal{I}} = ([\![\tau_1]\!]_{\mathcal{I}} \to_{\mathrm{c}} [\![\tau_2]\!]_{\mathcal{I}})_{\perp}$, allowing observing termination of expressions of higher types using the top-level read-eval-print loop. In contrast, in the idealized meta-language an expression of higher type is always applied so its termination behavior is never observed independently. Changing the semantics to account for Haskell-like lifted function spaces does not appear to introduce any difficulties in the soundness proof.

Finally, the approach to embedding languages into higher-order host languages presented above inherits the known problems of higher-order abstract syntax [22]. Most importantly, a higher-order abstract syntax does not admit a notion of case analysis. In order to make any use of constructed object-language terms, the embedding we have presented directly represents terms using a first-order data type which can be printed as text. However, the standard function spaces of functional languages are generally too large for adequate higher-order abstract syntax. For example, the implementation in Figure 2 actually allows representations of "exotic terms", such as the following expression of type `Exp (Int → Int)`.

```
lam (λx → if show x = "a" then int 1 else int 2)
```

Although closed, this term does not correspond to any object-language term. Depending on the context, it may behave as either `λa → 1` or `λx → 2`. Here we have used the overloaded Haskell function `show` to obtain a representation of an object-language term as a string. Such `show`-like functions are not, however, tied to Haskell's overloading mechanism. They can be implemented in Haskell as well as in languages without overloading (such as ML [29]) as ordinary polymorphic functions of type `Exp a → String`.

Research on higher-order abstract syntax alleviates these problems by restricting the function spaces of the meta-language using, e.g., modal logic [10]. In Haskell and ML-like languages, however, such a requirement cannot be enforced by the type system. In the presence of `show`-like functions, there are no ways around the problems of higher-order abstract syntax other than to informally require the programmer to only observe the behavior of closed object-language terms. This requirement appears to match the typical use of embedded languages in existing applications.

Below, we briefly discuss the embedded type discipline in the context of extended object languages and in the context of extended meta-languages.

```
-- Finite products
pair :: Exp a → Exp b → Exp (a, b)
pfst :: Exp (a, b) → Exp a
psnd :: Exp (a, b) → Exp b

-- Lists
nil  :: Exp [a]
cons :: Exp a → Exp [a] → Exp [a]
hd   :: Exp [a] → Exp a
tl   :: Exp [a] → Exp [a]

-- Booleans
tt   :: Exp Bool
ff   :: Exp Bool
ift  :: Exp Bool → Exp a → Exp a → Exp a

iszero :: Exp Int → Exp Bool
isnull :: Exp [a] → Exp Bool
```

Figure 3: Extended object language.

4.1 Extended object language

As already mentioned, other domain-specific types and operations are easily added to the object language. Figure 3 presents the types of operations on finite products, lists, and booleans. (The actual definition of these primitives are similar to those presented in Figure 2 and are left out for conciseness.) It is straightforward to extend the proof of soundness and completeness to also handle the extensions outlined here.

It is also possible to extend the object language with types that do not exist in the meta-language. For example,

```
type Ref a = ()

ref :: Exp a → Exp (Ref a)
get :: Exp (Ref a) → Exp a
set :: Exp (Ref a) → Exp a → Exp ()
```

declares the types of pointer-manipulating object-language functions in Haskell. (Appropriate values can be defined as in Figure 2.) Values of type `Ref` will never be constructed and therefore we can choose a minimal representation, here as the unit type `()`. A similar technique is used to integrate COM objects into Haskell [17].

The domain-specific operations on base types considered here (i.e., `int`, `add`, and the operations of the extended language in Figure 3) are *meta-functions*: they do not represent first-class functions in the object language. For example, there is no object-language equivalent of a first-class zero predicate. Indeed, Haskell rejects the expression `cons iszero nil` (an attempt to represent a singleton list of a zero predicate). A first-class zero predicate can be embedded by

```
data Raw = ... | ISZERO'

iszero' :: Exp (Int → Bool)
iszero' =  E (λns → ISZERO')
```

and used as in `cons iszero' nil` of type `Exp [Int → Bool]`.

Since the object language is higher order we can also apply object-language β and η conversions at the meta-level. The two function compositions

```
lam . app :: Exp (a → b) → Exp (a → b)
app . lam :: (Exp a → Exp b) → Exp a → Exp b
```

are the extensional identity function on object-language functions, and the extensional identity function on meta-language functions. In fact, in the sense of binding-times and two-level coercions, the first composition coerces a dynamic function to static and back to dynamic again whereas the second composition coerces a static function to dynamic and back to static again, using two-level η-expansions [5]. Given, for example, a Haskell-like object language where the meta-language expression `iszero E` represents `(e ≡ 0)` (where `E` represents `e`) and the meta-language expression `iszero'` represents `(≡ 0)`. (In Haskell, `≡` denotes structural equality.) In the meta-language we then have the (extensional) identities

```
iszero  = app iszero'     -- of type Exp Int → Exp Bool
iszero' = lam iszero      -- of type Exp (Int → Bool)
```

since (for the first equality) for any meta-language expression `E` of type `Exp Int` representing `e`, `app iszero' E` represents `(≡ 0) e` and (for the second equality) `lam iszero'` represents `λx → x ≡ 0` and indeed `e ≡ 0` and `(≡ 0) e` are semantically identical, as are `(≡ 0)` and `λx → x ≡ 0`.

As opposed to Haskell, the object-language does not provide polymorphism and recursion. Polymorphic or recursive meta-language expressions that are accepted by Haskell's type system are unfolded in the object language. For example, it is possible to define polymorphic representations of object-language values in the meta-language as illustrated above for finite products and lists. Even though we cannot represent object-language term such as

```
let f = λx → x in (f 0, f True)
```

we can inline let-bound polymorphic values in the object-language term. For example, the following expression has type `Exp (Int, Bool)`.

```
let f = lam(λx → x) in pair (app f (int 0)) (app f tt)
```

It evaluates to the object-language term `((λa → a) 0, (λa → a) True)`.

4.2 Extended meta-language

It is possible to extend the meta-language with other base types, finite products, and lists à la Haskell, and also to extend the proofs of soundness and completeness.

It is also possible to change the evaluation strategy of the meta-language to call-by-value. Adding other effects than non-termination, however, might give an unsound embedding. This is the case for ML [29]. Using assignments, for example, some well-typed ML expressions do not represent well-typed object-language terms. In fact, as the following example shows, some expressions do not even yield representations of closed terms.

```
let val v = ref (int 0)
in  lam (fn x ⇒ (v := x; x));
    v
end
```

This expression returns whatever symbolic variable was generated at the time of constructing the object-language lambda. This problem is due to the use of higher-order abstract syntax, not to the typed embedding.

5 Related work

The soundness proof presented in this article uses a Kripke logical relation. The development is akin to Filinski's proof that type-directed partial evaluation implements a normalization function [16]. A corollary of Filinski's work is that type-directed partial evaluation is type preserving, a result we have also established using the typed embedded language described in this work [6, 7].

Completeness could also be stated by giving a semantics of the object language and then showing that a term and its encoding are semantically equivalent. Our proof is stronger in that it shows that a term and its encoding are syntactically equal modulo renaming of bound variables. It also avoids dealing with the semantics of the object language.

Yelland has used Haskell as a meta-language for hosting stack instructions for a core Java Virtual Machine [48]. In his work, the type of a stack cell is parameterized by its contents using a phantom type. Essentially, instructions are given types that reflect their behavior on the stack. Yelland proves a soundness result stating that instructions that are well-typed in Haskell will not err on execution. In Yelland's work, the object language is first order and has no notion of variables and bindings. Hence, he avoids higher-order abstract syntax. In contrast, we have considered a higher-order object language is this present work.

Phantom types provide a simple form of dependent types for *constructing* representations of simply-typed terms in Haskell. The embedding is limited, however, in that it does not allow a type-preserving *deconstruction* of simply-typed terms. This is partly due to the lack of higher-order matching in Haskell.

A dependently typed language, such as Martin-Löf's type theory [31], can directly express both type-safe construction and deconstruction of object-language terms. Previously, Yang has shown how to encode another class of dependently typed expressions in ML [46, 47].

There exists a number of logical frameworks that specifically, although not exclusively, address the problem of manipulating typed higher-order object-language terms [4, 12, 18, 19, 20, 21, 32, 34, 36]. Compared to Haskell, these languages typically provide a richer dependent-type system and a better support for higher-order abstract syntax via higher-order unification and matching. These logical frameworks are primarily aimed at *reasoning* about object-language terms and, in some cases, also entire logics. In contrast, we have explored the extent to which general-purpose languages such as Haskell are adequate for hosting typed object languages.

In the mid-1980's, Wand has defined the meaning of an object language (with primitive I/O and non-local jumps) in terms of the λ-calculus, viewing the latter as a semantic meta-language [44]. His translation is sound and complete in the sense that exactly the well-typed object-language terms have a well-typed meta-language counterpart. Thus, it can be seen as providing both a static and a dynamic semantics for the object language. (In fact, since the translated terms are in continuation-passing style, Wand also shows that the CPS transformation preserves well-typedness [28].) Wand's translation does not yield first-order data as result such as the embedding we consider here. Instead, it maps higher-order functions to higher-order functions. Furthermore, Wand's object-language type system is not presented in terms of his meta-language. In contrast, the embedded type discipline we consider here expresses the type system of the object language in terms of the type system of the meta-language (using phantom types).

Davies's $\lambda^{\bigcirc}$-calculus is an extension of the simply-typed λ-calculus for expressing staged evaluation [8]. In $\lambda^{\bigcirc}$, a term of type $\bigcirc\sigma$ evaluates in one stage to a value representing a term of type σ. Terms of type $\bigcirc\sigma$ are thus first-class representations of terms, much as the expressions of type $\mathsf{Exp}\,\sigma$ in our work are representations of object-language terms. There is a syntactic difference, however. In $\lambda^{\bigcirc}$ there is one uniform construct for building terms at the next stage. The meta-language in our work provides several constructors, one for each syntactic category of the object language. (Analogy: $\lambda^{\bigcirc}$ provides a type system for Lisp-like quasiquotation [2] whereas phantom types provide a type system for ML-like data types.) Another difference is that $\lambda^{\bigcirc}$-terms build other $\lambda^{\bigcirc}$-terms for a later stage. In contrast, the object-language terms in our work are always simply typed λ-terms. Consequently, we cannot embed object languages with an arbitrary number of stages. There is some hope, though, that type encodings as described by Yang [46] could achieve such a nested embedding, a staged version of which is provided in a language such as MetaML [43]. There is a growing interest in such statically typed, multi-stage languages because they provide a foundation for multi-stage evaluation and run-time code generation.

6 Conclusions

Phantom types make it possible to embed not only the (abstract) syntax but also the type system of a monomorphic object language into a statically typed meta-language such as Haskell. They have been used to design embedded languages [17, 25, 27] and to prove properties of programs [6, 7]. An experiment with run-time code generation for the rewrite engine of the logical framework MetaPRL [20] also confirms the utility of phantom types in a two-stage language [38, 39]. In this article, we have formally proved that phantom types yield sound and complete embeddings into an idealized meta-language and we have discussed the power and limitations of Haskell as such a meta-language.

Acknowledgements: I sincerely thank Olivier Danvy, Andrzej Filinski, Helmut Schwichtenberg, and Peter Sestoft for providing insightful comments on this work and Mikkel N. Hansen and Lasse R. Nielsen for kindly proof-reading earlier versions of this article. I am also grateful to the referees for their constructive comments and suggestions.

References

[1] Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics.* North-Holland, 1984.

[2] Alan Bawden. Quasiquotation in Lisp. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, number NS–99–1 in BRICS Note Series, pages 4–12, San Antonio, Texas, January 1999.

[3] William E. Carlson, Paul Hudak, and Mark P. Jones. An experiment using Haskell to prototype 'geometric region servers' for navy command and control. Technical Report 1031, Yale University, New Haven, Connecticut, November 1993.

[4] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, 1986.

[5] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 8(6):730–751, 1996.

[6] Olivier Danvy and Morten Rhiger. A simple take on typed abstract syntax in Haskell-like languages. In Herbert Kuchen and Kazunori Ueda, editors, *Proceedings of the Fifth International Symposium on Functional and Logic Programming*, number 2024 in Lecture Notes in Computer Science, pages

343–358, Tokyo, Japan, March 2001. Springer-Verlag. An extended version is available as the technical report BRICS RS-00-34.

[7] Olivier Danvy, Kristoffer Høgsbro Rose, and Morten Rhiger. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, 2001. An extended version is available as the technical report BRICS RS-01-16.

[8] Rowan Davies. A temporal-logic approach to binding-time analysis. In Edmund M. Clarke, editor, *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

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

[10] Jöelle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In P. de Groote and J. R. Hindley, editors, *Proceedings of the 3rd International Conference on Typed Lambda Calculi and Applications*, number 1210 in Lecture Notes in Computer Science, pages 147–163, Nancy, France, April 1997.

[11] Premkumar Devanbu and Jeff Poulin, editors. *Proceedings of the Fifth International Conference on Software Reuse*, Victoria, British Columbia, June 1998. IEEE Computer Society Press.

[12] Gilles Dowek, Amy Felty, Hugo Herbelin, Gerard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The COQ proof assistant user's guide. Technical Report 154, INRIA, Rocquencourt, France, 1993.

[13] Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In Chris Ramming, editor, *First Conference on Domain-Specific Languages*, pages 285–296, Santa Barbara, California, October 1997.

[14] Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. In Walid Taha, editor, *Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, number 1924 in Lecture Notes in Computer Science, pages 9–27, Montréal, Canada, September 2000.

[15] Joseph H. Fasel, Paul Hudak, Simon Peyton Jones, and Philip Wadler. Haskell special issue. *SIGPLAN Notices*, 27(5), May 1992.

[16] Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture

Notes in Computer Science, pages 378–395, Paris, France, September 1999. Springer-Verlag.

[17] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In Peter Lee, editor, *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 114–125, Paris, France, September 1999. ACM Press.

[18] Michael J. C. Gordon and Tom Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[19] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. A preliminary version appeared in the proceedings of the First IEEE Symposium on Logic in Computer Science, pages 194–204, June 1987.

[20] Jason Hickey. Metaprl homepage. `http://metaprl.org`.

[21] Jason Hickey. Nuprl-light: An implementation framework for higher-order logics. In William McCune, editor, *14th International Conference on Automated Deduction*, number 1249 in Lecture Notes in Artificial Intelligence, pages 395–399. Springer-Verlag, 1997.

[22] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In Giuseppe Longo, editor, *Proceedings of the Fourteenth Annual IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999. IEEE Computer Society Press.

[23] Paul Hudak. Modular domain specific languages and tools. In Devanbu and Poulin [11], pages 134–142.

[24] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3):465–483, 1996.

[25] Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components in Haskell. In Devanbu and Poulin [11], pages 224–233.

[26] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[27] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In Thomas Ball, editor, *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, pages 109–122, 1999.

[28] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, June 1985. Springer-Verlag.

[29] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[30] John C. Mitchell. Type systems for programming languages. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 8, pages 365–458. The MIT Press, 1990.

[31] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory*. International Series on Monographs on Computer Science No. 7. Oxford University Press, 1990.

[32] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[33] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In Gopal Gupta, editor, *Proceedings of the First International Symposium on Practical Aspects of Declarative Languages*, number 1551 in Lecture Notes in Computer Science, pages 91–105, San Antonia, Texas, January 1999. Springer-Verlag.

[34] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[35] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Mayer D. Schwartz, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 23, No 7, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[36] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206, Trento, Italy, 1999. Springer-Verlag.

[37] Alastair Reid, John Peterson, Greg Hager, and Paul Hudak. Prototyping real-time vision systems: An experiment in DSL design. In *Twenty-First International Conference on Software Engineering*, pages 484–493, Los Angeles, California, May 1999. ACM Press.

[38] Morten Rhiger. *Higher-Order Program Generation*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001.

[39] Morten Rhiger. Compiling embedded programs to byte code. In Shriram Krishnamurthi and C.R. Ramakrishnan, editors, *Proceedings of the Fourth International Symposium on Practical Aspects of Declarative Languages*, number 2257 in Lecture Notes in Computer Science, pages 120–136, Portland, Oregon, January 2002. Springer-Verlag.

[40] Dana Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993.

[41] Guy L. Steele Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, October 1999.

[42] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[43] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type safety. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming*, number 1443 in Lecture Notes in Computer Science, pages 918–929. Springer-Verlag, 1998.

[44] Mitchell Wand. Embedding type structure in semantics. In Mary S. Van Deusen and Zvi Galil, editors, *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 1–6. ACM Press, January 1985.

[45] Glynn Winskel. *The Formal Semantics of Programming Languages.* Foundation of Computing Series. The MIT Press, 1993.

[46] Zhe Yang. Encoding types in ML-like languages. In Paul Hudak and Christian Queinnec, editors, *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 289–300, Baltimore, Maryland, September 1998. ACM Press. Extended version available as the technical report BRICS RS-98-9.

[47] Zhe Yang. *Language Support for Program Generation: Reasoning, Implementation, and Applications.* PhD thesis, Computer Science Department, New York University, New York, New York, August 2001.

[48] Phillip M. Yelland. A compositional account of the Java Virtual Machine. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 57–69, San Antonio, Texas, January 1999. ACM Press.

Recent BRICS Report Series Publications

RS-02-34 **Morten Rhiger.** ***A Foundation for Embedded Languages*****. August 2002. 29 pp.**

RS-02-33 **Vincent Balat and Olivier Danvy.** ***Memoization in Type-Directed Partial Evaluation*****. July 2002. 18 pp. To appear in Batory and Consel, editors,** ***ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*****, GPCE '02 Proceedings, LNCS, 2002.**

RS-02-32 **Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde.** ***On Obtaining Knuth, Morris, and Pratt's String Matcher by Partial Evaluation*****. July 2002. 43 pp. To appear in Chin, editor,** ***ACM SIGPLAN ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*****, ASIA-PEPM '02 Proceedings, 2002.**

RS-02-31 **Ulrich Kohlenbach and Paulo B. Oliva.** ***Proof Mining: A Systematic Way of Analysing Proofs in Mathematics*****. June 2002. 47 pp.**

RS-02-30 **Olivier Danvy and Ulrik P. Schultz.** ***Lambda-Lifting in Quadratic Time*****. June 2002.**

RS-02-29 **Christian N. S. Pedersen and Tejs Scharling.** ***Comparative Methods for Gene Structure Prediction in Homologous Sequences*****. June 2002. 20 pp.**

RS-02-28 **Ulrich Kohlenbach and Laurenţiu Leuştean.** ***Mann Iterates of Directionally Nonexpansive Mappings in Hyperbolic Spaces*****. June 2002. 33 pp.**

RS-02-27 **Anna Östlin and Rasmus Pagh.** ***Simulating Uniform Hashing in Constant Time and Optimal Space*****. 2002. 11 pp.**

RS-02-26 **Margarita Korovina.** ***Fixed Points on Abstract Structures without the Equality Test*****. June 2002.**

RS-02-25 **Hans Hüttel.** ***Deciding Framed Bisimilarity*****. May 2002. 20 pp.**

RS-02-24 **Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach.** ***Static Analysis for Dynamic XML*****. May 2002. 13 pp.**

RS-02-23 **Antonio Di Nola and Laurenţiu Leuştean.** ***Compact Representations of BL-Algebras*****. May 2002. 25 pp.**