Typed Self-Representation

Tillmann Rendel Klaus Ostermann Christian Hofer
University of Aarhus, Denmark

Abstract

Self-representation – the ability to represent programs in their own language – has important applications in reflective languages and many other domains of programming language design. Although approaches to designing typed program representations for sublanguages of some base language have become quite popular recently, the question whether a fully metacircular typed self-representation is possible is still open. This paper makes a big step towards this aim by defining the F_ω^* calculus, an extension of the higher-order polymorphic lambda calculus F_ω that allows typed self-representations. While the usability of these representations for metaprogramming is still limited, we believe that our approach makes a significant step towards a new generation of reflective languages that are both safe and efficient.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Theory

Keywords Lambda Calculus, Language Design, Reflection, Self Interpretation, Types

1. Introduction

It is a basic result of computability theory that every Turing-complete programming language allows representing programs in that language as data, and writing a metacircular interpreter in terms of this self-representation. There is a long history in programming languages on designing self-representations and metacircular interpreters which dates back at least to the early days of LISP [18].

Self-representation plays a crucial role in the design of reflective languages. For example, the classical reflective tower [26] consists of a stack of interpreters such that each interpreter is program data on the next level. In such reflective languages, self-representation is usually formalized in the form of a *quote* function, which converts terms to representations of terms, and an unquote or *eval* function which converts term representations back to terms.

There are many good reasons to consider the possibility of statically typed self-representation. The expressive power of dynamically typed reflective languages is well-known, but the lack of static typing makes this power hard to control. Furthermore, statically typed program representations allow *tagless* interpreters, which are an order of magnitude faster than ordinary interpreters which use tagged unions to represent values. Another motivation for this work stems from Steele's idea of growable languages [27], which – according to Steele – necessitates that features provided by a library look like built-in language features. The ability to represent a whole statically typed language within the language itself can be considered as a realization of Steele's idea in that the language itself can be made to look as if it were provided by a library.

Despite these exciting possibilities, the question of whether it is possible to design languages that allow typed self-representations is a long-standing open question. Reynolds has hinted at the question of typed self-representation in his classic paper on definitional interpreters [24]. There have been attempts, notably those by Pfenning and Lee [21] and more recently by Carette et al. [7], but none of these attempts have resulted in a fully metacircular typed self-representation (more about that later).

In this paper, we present our attempt to make typed self-representation possible and useful. More specifically, the contributions of this work are as follows:

- We give a definition what typed self-representation is, by specifying a set of properties it should minimally fulfill.
- We identify problems of current approaches of typed self-representation.
- We present a language, which allows typed self-representation. We show how a metacircular interpreter can be written in the language, and we discuss the scope of alternative interpretations that can be defined on this self-representation. The language, which we call F_ω^*, is an extension of the higher-order polymorphic lambda calculus F_ω. The name is because the crucial addition to F_ω, which enables typed self-representation, is a variant of the *type:type* rule from the λ^* calculus [8].
- We have formalized F_ω^* in the Coq theorem prover [5] and present a machine-checked proof that the proposed typed self-representation is indeed metacircular. Furthermore, we have implemented both a type checker and interpreter for F_ω^*.
- We present an embedding of F_ω^* into the pure type systems framework [4, 29, 3]. By this embedding we can "reuse" the meta-theory of pure type systems and give an accurate comparison with other PTS instances.
- We conclude with a detailed discussion of the remaining open issues in our representation, and future work, in particular the issue of quoting terms with free variables and the use of type-indexed datatypes to get more flexible self-representations.

The remainder of this paper is structured as follows. The next section discusses the notion of typed self-representation. F_ω^* and its self-embedding are introduced in Sec. 3. Section 4 presents our technical results and open issues. Section 5 discusses the issue of integrating typed self-representation into programming languages. Section 6 discusses the design decisions, related approaches, open issues and future work, and the final Section concludes.

PLDI'09, June 15–20, 2009, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-392-1/09/06... $5.00

2. Typed Self-Representation

Self-representation refers to the possibility of representing terms of a programming language in the language itself. In the untyped setting, this is typically implemented by a *quote* mechanism which transforms a term into its representation, and a corresponding unquote or *eval* function that transforms a representation back to the represented term. Correspondingly, a *typed* self-representation is a self-representation of terms within a typed language, mapping typed terms to typed representations of terms.

Like with untyped self-representation, a code transformation quote is used to transform terms to representations of terms, which can be processed by programs written in the language itself. In the context of static typing, however, we want to describe both the quote transformation and the individual operations on quoted terms on the type level to guarantee the well-formedness of these operations. These additional guarantees are an important reason to search for a typed self-representation in the first place. It should be noted, however, that this does not necessitate a corresponding quote transformation on types, transforming them to terms or types of the language. Instead, the type of the representation of a term may simply reflect its character as being such a representation.

In a setting where quoted terms are represented by encoding their abstract syntax trees as a data structure, the notion of typed self-representation may seem to be intuitive. However, the notion becomes less clear if one considers sophisticated encoding techniques such as higher-order abstract syntax [20] or "final" representations of programs [7]. Therefore, we first have to clarify the notion of typed self-representation. We consider typed self-representation to be constituted by the following five properties. After the presentation of our solution, we will revisit these properties and discuss to what degree our solution has the properties proposed here.

1. **Representation.** There is a family of types $\mathsf{Expr_T}$ such that $\mathsf{quote(t)}$ has type $\mathsf{Expr_T}$ if and only if t has type T.
2. **Adequacy.** Every term t of type $\mathsf{Expr_T}$ corresponds to a term t' of type T, which means that for every t as above there exists a t' such that $t = \mathsf{quote}(t')$.
3. **First Class Interpretations.** It is possible to express operations on quoted terms so that they are well-typed for all terms of type $\mathsf{Expr_T}$, without the need to refer to any specific such terms.
4. **Self Interpretation.** There is a family of contexts $\mathsf{eval_T}\langle\rangle$ such that $\mathsf{eval_T}\langle\mathsf{quote(t)}\rangle$ is observational equivalent to t if t has type T.
5. **Reflection.** $\mathsf{quote(t)}$ exhibits the intensional structure of t in a useful way.

The properties are formulated in terms of families of types and contexts to make them applicable to different languages and type systems, even if they do not provide functional abstraction. For a functional language, one can read "family of types" as "type constructor", and "context" as "function".

The first property ensures the full circularity of the representation: The quote transformation can transform every well-typed term into a well-typed term. Furthermore, $\mathsf{quote(t)}$ can be seen as containing a proof of the well-typedness of t. But to be more precise, we also have to say something about the typing context. While one may like to allow free variables in t, and consequently allow arbitrary typing contexts in Prop. 1, this leads to some complications as we will see below. Hence for now we only demand that quoting works correctly for closed terms, i.e., $\emptyset \vdash t : T$ if and only if $\emptyset \vdash \mathsf{quote(t)} : \mathsf{Expr_T}$.

Property 1 ensures that the type family $\mathsf{Expr_T}$ is expressive enough to encompass all representations of well-typed terms. On the other hand, Prop. 2 states that $\mathsf{Expr_T}$ is precise enough to rule out exotic terms of type $\mathsf{Expr_T}$ which do not correspond to any term of type T. Together, these properties specify that the extension of $\mathsf{Expr_T}$ is precisely the set of quoted terms. Obviously, this property cannot hold for languages which allow diverging terms of type $\mathsf{Expr_T}$. However, a weaker variant is applicable, which requires adequacy only for normalizing terms of type $\mathsf{Expr_t}$.

While Prop. 1 already implies that quoted terms are first-class values at least in the sense that they can be separately typed, Prop. 3 requires the same for operations on quoted terms.

Property 4 guarantees that quoted terms contain enough information to write a standard evaluator. Although there may be useful weaker self-representations, we believe that this property is essential for many applications in reflective languages.

Property 5 aims at the capability to apply interesting non-standard interpretations on quoted terms. Recurring examples are pretty printing, size measuring, CPS transformation or converting into an untyped representation. The definition of the property is intentionally vague, as it is not clear which kind of interpretations could or should be possible in a typed setting. Hence we do not want to prematurely preclude other possible approaches to typed-self representation by a too narrow definition.

However, without this property, the trivial encoding, using the identity function as quote, or simple wrappers around terms such as boxing [9], or staging annotations [28] would count as forms of typed self-representation, as they fulfill some or all of the other properties above. These representations, however, do not give access to the intensional structure of terms. This precludes exploring the syntactic structure of a term, e.g., whether it is a function application or a λ-abstraction, or changing the meaning of the language primitives in a non-standard interpretation.

3. $\mathbf{F}^*_\omega$ and its Self-Embedding

A language with the properties introduced in the last section has to feature both a term language simple enough to allow quote to encode all features of the language and a type system rich enough to encode the family of types $\mathsf{Expr_T}$. Experiments by Pfenning et al. [21] suggest that to support a level of abstraction in the encoded language, a higher level of abstraction has to be supported in the encoding language, e.g., second-order types are needed to represent first-order types. With self-representation, encoded and encoding language are identical. While this seems to preclude typed self-representation at first, we will show how to escape from this circle after we have introduced the style of quoting and evaluating we aim at.

3.1 Encoding of terms and types

Instead of a representation of terms as an abstract syntax tree, we choose a Church encoding, which has a number of advantages over a datatype based representation, most notably that it keeps the language simple since it is not necessary to add, and subsequently represent, datatypes. Furthermore, we do not represent the types of the language explicitly, but each term representation is associated with a type in the meta-language that ensures type-safety. In order to guarantee the type-safe representation of variables, we make use of higher-order abstract syntax (HOAS) [20]. Our encoding has been mainly inspired by related approaches of Pfenning / Lee [21] and Carette et al. [7] which we will discuss in Sec. 6. In the following, we will first describe our encoding technique in an untyped setting, and subsequently develop our solution to the typing problem.

A lambda term such as the church numeral two in the untyped lambda calculus

$$\lambda \mathsf{f} \,.\, \lambda \mathsf{x} \,.\, \mathsf{f(fx)}$$

can be encoded by abstracting over the meaning of the nodes in the syntax tree. Since lambda terms contain only abstractions and applications, we can encode the Church numeral two as

$$\lambda \mathsf{lam} \,.\, \lambda \mathsf{app} \,.\, \mathsf{lam}(\lambda \mathsf{f} \,.\, \mathsf{lam}(\lambda \mathsf{x} \,.\, \mathsf{app}\,\mathsf{f}\,(\mathsf{app}\,\mathsf{f}\,\mathsf{x}))).$$

The lam and app binders fulfill the analogous function in our encoding as the f and x binders in the Church encoding of numerals. Due to the use of HOAS, the bound variables f and x are not represented by some first-order representation like de Bruijn indices [10], but by employing the lambda binding mechanism of the meta-language.

A context $\mathsf{eval}\langle\rangle$, as described in the previous section, for evaluating such quoted terms can be defined as

$$\mathsf{eval}\langle \mathsf{t}\rangle = \mathsf{t}\,(\lambda \mathsf{f} \,.\, \mathsf{f})\,(\lambda \mathsf{f} \,.\, \lambda \mathsf{x} \,.\, \mathsf{f}\,\mathsf{x}).$$

To lift this encoding to a typed lambda calculus, one has to give types for lam and app. A first attempt might be the following System F types:

$$\mathsf{App}_{attempt} = \forall \mathsf{A} :: \ast \,.\, \forall \mathsf{B} :: \ast \,.\, (\mathsf{A} \to \mathsf{B}) \to \mathsf{A} \to \mathsf{B}$$

$$\mathsf{Lam}_{attempt} = \forall \mathsf{A} :: \ast \,.\, \forall \mathsf{B} :: \ast \,.\, (\mathsf{A} \to \mathsf{B}) \to (\mathsf{A} \to \mathsf{B}).$$

Note how these versions of App and Lam are polymorphic in the domain and codomain of the processed function. We call the set of types (like App and Lam) that describe a language a *language interface*.

Unfortunately, the types for App and Lam above do not allow interesting interpreters beyond self-evaluation, since all reasonable functions with these types are identity functions. In type systems with abstraction over type functions, such as F_ω, we can add another parameter $\mathsf{R} :: \ast \Rightarrow \ast$ which represents the change in the type we want to perform with app and lam. While a self-interpreter can choose $\mathsf{R} = \mathsf{ID} = \lambda \mathsf{T} : \ast \,.\, \mathsf{T}$, other interpreters are free to choose more interesting type functions. Using R, the types of app and lam are:

$$\mathsf{App}\,\mathsf{R} = \forall \mathsf{A} :: \ast \,.\, \forall \mathsf{B} :: \ast . \mathsf{R}\,(\mathsf{A} \to \mathsf{B}) \to \mathsf{R}\,\mathsf{A} \to \mathsf{R}\,\mathsf{B}$$

$$\mathsf{Lam}\,\mathsf{R} = \forall \mathsf{A} :: \ast \,.\, \forall \mathsf{B} :: \ast . (\mathsf{R}\,\mathsf{A} \to \mathsf{R}\,\mathsf{B}) \to \mathsf{R}\,(\mathsf{A} \to \mathsf{B}).$$

Note how the types now depend on R as free variable.

However, for a metacircular self-representation of all terms in F_ω, we have to account for the type abstraction and application operations in these types, too. We call the types that describe type abstraction and application TLam and TApp, respectively. For example, we want to represent the polymorphic identity function $\Lambda \mathsf{T} :: \ast \,.\, \lambda \mathsf{x} : \mathsf{T} \,.\, \mathsf{x}$ as

$$\begin{aligned}
&\Lambda \mathsf{R} :: \ast \Rightarrow \ast \,.\, \lambda \mathsf{lam} : \mathsf{Lam}\,\mathsf{R} \,.\, \lambda \mathsf{app} : \mathsf{App}\,\mathsf{R} \,.\\
&\quad \lambda \mathsf{tlam} : \mathsf{TLam}\,\mathsf{R} \,.\, \lambda \mathsf{tapp} : \mathsf{TApp}\,\mathsf{R} \,.\\
&\quad \mathsf{tlam}\,[\ast]\,[\forall \mathsf{T} :: \ast \,.\, \mathsf{T} \to \mathsf{T}]\,(\Lambda \mathsf{T} :: \ast \,.\, \mathsf{lam}\,[\mathsf{T}][\mathsf{T}](\lambda \mathsf{x} : \mathsf{R}\,\mathsf{T} \,.\, \mathsf{x}))
\end{aligned}$$

with suitable types TLam and TApp in addition to the types Lam and App introduced above.

We model tlam and tapp analogous to lam and app. The function that is processed is a function mapping types to terms, i. e., a polymorphic term. The first parameter of tlam and tapp states the domain (on types, i. e, it is a kind, in the example: $\ast$), while the second parameter states the codomain. However, the codomain can depend on the respective inhabitant of the domain, i. e., it is a family of types indexed by types. Thus, it has to be parameterized by the respective inhabitant of the domain ($\forall \mathsf{T} :: \ast \,.\, \mathsf{T} \to \mathsf{T}$ in the example). The third parameter is the actual polymorphic term, represented by a HOAS encoding on the type level.

Unfortunately, as the example illustrates for TLam, the types TLam and TApp cannot be expressed in F_ω, since they have to

$$\begin{aligned}
\mathsf{App} &\equiv \lambda \mathsf{R} :: \ast \Rightarrow \ast \,.\, \forall \mathsf{A} :: \ast \,.\, \forall \mathsf{B} :: \ast.\\
&\qquad \mathsf{R}\,(\mathsf{A} \to \mathsf{B}) \to \mathsf{R}\,\mathsf{A} \to \mathsf{R}\,\mathsf{B}\\
\mathsf{Lam} &\equiv \lambda \mathsf{R} :: \ast \Rightarrow \ast \,.\, \forall \mathsf{A} :: \ast \,.\, \forall \mathsf{B} :: \ast.\\
&\qquad (\mathsf{R}\,\mathsf{A} \to \mathsf{R}\,\mathsf{B}) \to \mathsf{R}\,(\mathsf{A} \to \mathsf{B}).\\
\mathsf{TApp} &\equiv \lambda \mathsf{R} :: \ast \Rightarrow \ast \,.\, \forall \mathsf{S} :: \Box \,.\, \forall \mathsf{T} :: \mathsf{S} \Rightarrow \ast.\\
&\qquad \mathsf{R}\,(\forall \mathsf{X} :: \mathsf{S} \,.\, \mathsf{T}\,\mathsf{X}) \to (\forall \mathsf{X} :: \mathsf{S} \,.\, \mathsf{R}\,(\mathsf{T}\,\mathsf{X}))\\
\mathsf{TLam} &\equiv \lambda \mathsf{R} :: \ast \Rightarrow \ast \,.\, \forall \mathsf{S} :: \Box \,.\, \forall \mathsf{T} :: \mathsf{S} \Rightarrow \ast.\\
&\qquad (\forall \mathsf{X} :: \mathsf{S} \,.\, \mathsf{R}\,(\mathsf{T}\,\mathsf{X})) \to \mathsf{R}\,(\forall \mathsf{X} :: \mathsf{S} \,.\, \mathsf{T}\,\mathsf{X})\\
\mathsf{Expr} &\equiv \lambda \mathsf{A} :: \ast \,.\, \forall \mathsf{R} :: \ast \Rightarrow \ast.\\
&\qquad \mathsf{App}\,\mathsf{R} \to \mathsf{Lam}\,\mathsf{R} \to \mathsf{TApp}\,\mathsf{R} \to \mathsf{TLam}\,\mathsf{R} \to \mathsf{R}\,\mathsf{A}
\end{aligned}$$

Figure 1. The F^*_ω language interface

allow abstraction over polymorphic terms of arbitrary kinds, but F_ω does not allow kind-polymorphism. Indeed, even adding kind-polymorphism is not enough, because the representation of kind polymorphism would again have to abstract over the form of the kind being abstracted over, employing another, higher level of polymorphism.

The key idea to break this circle is to add a constant $\Box$ which stands for the kind of all kinds. Furthermore, we unify the syntactic categories of types and kinds, such that the regular type-level λ can be used to express kind functions, and the regular expression-level Λ to express kind-polymorphic terms. This extended version of F_ω allows giving sensible types for tapp and tlam, as shown in Fig. 1, which also shows how the type family Expr_T from the previous section can be encoded directly as a type function. Hence the type of the representation of our polymorphic identity function is $\mathsf{Expr}\,(\forall \mathsf{T} :: \ast \,.\, \mathsf{T} \to \mathsf{T})$, as desired.

3.2 The language F^*_ω

Before we discuss our language w. r. t. typed self-representation, we first have to formalize the extensions to F_ω we had to introduce to define the language interface. We call this language F^*_ω, because it combines features of F_ω and λ^* [12]. Its formal definition can be found in Fig. 2.

F^*_ω features the same term language as F_ω, but collapses the *syntactic* categories of types and kinds, similar to how in λ^* all syntactic categories are collapsed. Therefore every (well-typed) F_ω term is a (well-typed) F^*_ω term as well, and similarly (well-kinded) F_ω types are (well-kinded) F^*_ω types. Since we collapse types and kinds, F_ω kinds are again well-kinded F^*_ω types. On the other hand, every typing statement in λ^* can be transformed into a kinding statement in F^*_ω, if one substitutes $\ast$ by $\Box$, and adapts the syntax to account for the differences in the syntax definitions. In this sense, every well-typed λ^* term corresponds to a well-kinded F^*_ω type.

Note that we still distinguish typing and kinding, as the former is about well-formedness of terms, while the latter is about well-formedness of types. Furthermore, the typing and kinding rules ensure that in all bindings of form $\mathsf{x} : \mathsf{T}$, T must be a proper type, while in bindings of the form $\mathsf{X} :: \mathsf{T}$, T must be a proper kind. In the same way, we distinguish three function types: $\mathsf{T} \to \mathsf{T}$, where domain and codomain are proper types, $\forall \mathsf{X} :: \mathsf{T} \,.\, \mathsf{T}$, where the domain is a proper kind, while the codomain is a type-indexed family of proper types, and $\Pi \mathsf{X} :: \mathsf{T} \,.\, \mathsf{T}$, where the domain is a proper kind, while the codomain is a type-indexed family of proper kinds. These different syntactic forms follow the standard F_ω syntax definition and emphasize the difference between statements about terms and statements about types. As we will argue later, they are not strictly necessary.

Abstraction over terms is written with a lower-case lambda $\lambda x : T \,.\, b$ while abstraction over types and kinds uses a upper-

Syntax

x	$:=$		*term variables*
X	$:=$		*type/kind variables*
t	$:=$		*terms*:
		x	*variable*
	$\mid$	$t\,t$	*application*
	$\mid$	$t\,[T]$	*type application*
	$\mid$	$\lambda x : T\,.\,t$	*abstraction*
	$\mid$	$\Lambda X :: T\,.\,t$	*type abstraction*
T	$:=$		*types/kinds*:
		$*$	*kind of proper types*
	$\mid$	$\Box$	*kind of all kinds*
	$\mid$	X	*type/kind variable*
	$\mid$	$T\,T$	*type/kind application*
	$\mid$	$\lambda X :: T\,.\,T$	*type/kind abstraction*
	$\mid$	$T \rightarrow T$	*function type*
	$\mid$	$\forall X :: T\,.\,T$	*polymorphic type*
	$\mid$	$\Pi X :: T\,.\,T$	*dependent kind*

$T_1 \Rightarrow T_2$ is syntactic sugar for $\Pi X :: T_1\,.\,T_2$ if $X \notin FV(T_2)$

Γ	$:=$		*contexts*
		$\emptyset$	*empty context*
	$\mid$	$\Gamma, x : T$	*term variable binding*
	$\mid$	$\Gamma, X :: T$	*type/kind variable binding*

Typing

$$\frac{\Gamma \vdash T :: * \qquad x \notin dom(\Gamma)}{\Gamma, x : T \vdash x : T} \quad (\text{T-VAR})$$

$$\frac{\Gamma \vdash t : T_1 \qquad \Gamma \vdash T_2 :: * \qquad x \notin dom(\Gamma)}{\Gamma, x : T_2 \vdash t : T_1} \quad (\text{T-WEAK1})$$

$$\frac{\Gamma \vdash t : T_1 \qquad \Gamma \vdash T_2 :: \Box \qquad X \notin dom(\Gamma)}{\Gamma, X :: T_2 \vdash t : T_1} \quad (\text{T-WEAK2})$$

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\,t_2 : T_2} \quad (\text{T-APP})$$

$$\frac{\Gamma \vdash t : (\forall X :: T_1\,.\,T_2) \qquad \Gamma \vdash T_3 :: T_1}{\Gamma \vdash t\,[T_3] : T_2[X \mapsto T_3]} \quad (\text{T-TAPP})$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (\lambda x : T_1\,.\,t) : (T_1 \rightarrow T_2)} \quad (\text{T-ABS})$$

$$\frac{\Gamma \vdash T_1 :: \Box \qquad \Gamma, X :: T_1 \vdash t : T_2}{\Gamma \vdash (\Lambda X :: T_1\,.\,t) : (\forall X :: T_1\,.\,T_2)} \quad (\text{T-TABS})$$

$$\frac{\Gamma \vdash t : T_1 \qquad \Gamma \vdash T_1 :: * \qquad T_1 =_\beta T_2}{\Gamma \vdash t : T_2} \quad (\text{T-CONV})$$

Kinding

$$\frac{}{\emptyset \vdash * :: \Box} \quad (\text{K-TYPE})$$

$$\frac{}{\emptyset \vdash \Box :: \Box} \quad (\text{K-KIND})$$

$$\frac{\Gamma \vdash T :: \Box \qquad X \notin dom(\Gamma)}{\Gamma, X :: T \vdash X :: T} \quad (\text{K-VAR})$$

$$\frac{\begin{array}{c}\Gamma \vdash T :: T_1 \\ \Gamma \vdash T_1 :: \Box \\ T_1 =_\beta T_2\end{array}}{\Gamma \vdash T :: T_2} \quad (\text{K-CONV})$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 :: T_2 \\ \Gamma \vdash T_3 :: * \qquad x \notin \Gamma\end{array}}{\Gamma, x : T_3 \vdash T_1 :: T_2} \quad (\text{K-WEAK1})$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 :: T_2 \\ \Gamma \vdash T_3 :: \Box \qquad X \notin \Gamma\end{array}}{\Gamma, X :: T_3 \vdash T_1 :: T_2} \quad (\text{K-WEAK2})$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 :: (\Pi X :: T_3\,.\,T_4) \\ \Gamma \vdash T_2 :: T_3\end{array}}{\Gamma \vdash T_1\,T_2 :: T_4[X \mapsto T_2]} \quad (\text{K-APP})$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 :: \Box \\ \Gamma, X :: T_1 \vdash T_2 :: T\end{array}}{\Gamma \vdash (\lambda X :: T_1\,.\,T_2) : (\Pi X :: T_1\,.\,T)} \quad (\text{K-ABS})$$

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash (T_1 \rightarrow T_2) :: *} \quad (\text{K-FUN})$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 :: \Box \\ \Gamma, X :: T_1 \vdash T_2 :: *\end{array}}{\Gamma \vdash \forall X :: T_1\,.\,T_2) :: *} \quad (\text{K-TFUN})$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 :: \Box \\ \Gamma, X :: T_1 \vdash T_2 :: \Box\end{array}}{\Gamma \vdash (\Pi X :: T_1\,.\,T_2) :: \Box} \quad (\text{K-KFUN})$$

Figure 2. Definition of F_ω^*

case lambda $\Lambda X :: T\,.\,B$. Type application is written with brackets $f\,[T]$.

The type and kind language consists of two constants, the kind of all types $*$ and the kind of all kinds $\Box$. Abstraction over types or kinds is written with a lower-case lambda $\lambda X :: T\,.\,B$. There are different syntactic forms for the three different kinds of functions: A function $\lambda x : T\,.\,b$ has a type of the form $T \rightarrow B$, a polymorphic function $\Lambda X :: T\,.\,b$ has a type of the form $\forall X :: T\,.\,B$ and a type or kind function $\lambda X :: T\,.\,B$ has a kind of the form $\Pi X :: T\,.\,B$, which can be abbreviated as $T \Rightarrow B$ if X is not free in B.

The definitions of β reduction and β equivalence on terms and types are not shown, but they are defined as usual and can be found in our Coq formalization.

The typing rules mostly follow the corresponding rules of F_ω, but (T-WEAK2) and (T-TABS) have to explicitly check that the mentioned kinds are themselves well-kinded using a judgment of the

Quoting

$$\frac{\vdash t : T \rhd t' \qquad \mathsf{R}, \mathsf{lam}, \mathsf{app}, \mathsf{tlam}, \mathsf{tapp} \notin t, T}{\langle t \rangle = \Lambda \mathsf{R} :: * \Rightarrow * \,.\, \lambda \mathsf{lam} : \mathsf{Lam}\,\mathsf{R} \,.\, \lambda \mathsf{app} : \mathsf{App}\,\mathsf{R} \,.\, \lambda \mathsf{tlam} : \mathsf{TLam}\,\mathsf{R} \,.\, \lambda \mathsf{tapp} : \mathsf{TApp}\,\mathsf{R} \,.\, t'} \quad (\text{QUOTE})$$

Pre-Quoting

$$\frac{\begin{array}{c}\Gamma \vdash t : T \boxed{\rhd t_2} \\ \Gamma \vdash T_2 :: * \qquad x \notin dom(\Gamma)\end{array}}{\Gamma, x : T_2 \vdash t : T_1 \boxed{\rhd t_2}} \quad (\text{Q-WEAK1})$$

$$\frac{\begin{array}{c}\Gamma \vdash t : T_1 \boxed{\rhd t_2} \\ \Gamma \vdash T_2 :: \Box \qquad X \notin dom(\Gamma)\end{array}}{\Gamma, X :: T_2 \vdash t : T_1 \boxed{\rhd t_2}} \quad (\text{Q-WEAK2})$$

$$\frac{\begin{array}{c}\Gamma \vdash t_2 : T_1 \boxed{\rhd t_4} \\ \Gamma \vdash t_1 : T_1 \to T_2 \boxed{\rhd t_3}\end{array}}{\begin{array}{c}\Gamma \vdash t_1\, t_2 : T_2 \\ \boxed{\rhd \mathsf{app}\, [T_1]\, [T_2]\, t_3\, t_4}\end{array}} \quad (\text{Q-APP})$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 :: * \\ \Gamma, x : T_1 \vdash t : T_2 \boxed{\rhd t_2}\end{array}}{\begin{array}{c}\Gamma \vdash (\lambda x : T_1 \,.\, t) : (T_1 \to T_2) \\ \boxed{\rhd \mathsf{lam}\, [T_1]\, [T_2]\, (\lambda x : R\, T_1 \,.\, t_2)}\end{array}} \quad (\text{Q-ABS})$$

$$\frac{\begin{array}{c}\Gamma \vdash T_3 :: T_1 \\ \Gamma \vdash t_1 : (\forall X :: T_1 \,.\, T_2) \boxed{\rhd t_2}\end{array}}{\begin{array}{c}\Gamma \vdash t_1\, [T_3] : T_2[X \mapsto T_3] \\ \boxed{\rhd \mathsf{tapp}\, [T_1]\, [\lambda X :: T_1 \,.\, T_2]\, t_2\, [T_3]}\end{array}} \quad (\text{Q-TAPP})$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 :: \Box \\ \Gamma, X :: T_1 \vdash t : T_2 \boxed{\rhd t_2}\end{array}}{\begin{array}{c}\Gamma \vdash (\Lambda X :: T_1 \,.\, t) : (\forall X :: T_1 \,.\, T_2) \\ \boxed{\rhd \mathsf{tlam}\, [T_1]\, [\lambda X :: T_1 \,.\, T_2]\, (\Lambda X :: T_1 \,.\, t_2)}\end{array}} \quad (\text{Q-TABS})$$

$$\frac{\Gamma \vdash T :: * \qquad x \notin dom(\Gamma)}{\Gamma, x : T \vdash x : T \boxed{\rhd x}} \quad (\text{Q-VAR})$$

$$\frac{\begin{array}{c}\Gamma \vdash t : T_1 \boxed{\rhd t_2} \\ \Gamma \vdash T_2 :: * \qquad T_1 =_\beta T_2\end{array}}{\Gamma \vdash t : T_2 \boxed{\rhd t_2}} \quad (\text{Q-CONV})$$

Figure 3. Quoting of F^*_ω. Additions to the typing rules in Fig. 2 are marked by boxes.

form $\Gamma \vdash T :: \Box$. This is not needed in F_ω because all syntactically possible F_ω kinds are already well-formed.

The kinding rules of F^*_ω are substantially more complex then the kinding rules of F_ω, since kinds may now contain variables and all contained kinds have to be checked for well-formedness using judgments of the form $\Gamma \vdash T :: \Box$.

3.3 Self Representation in F^*_ω

Following the examples in Sec. 3.1, quoting is formalized as a type-directed transformation, see Fig. 3. Quoting is defined in two steps: A *pre-quote* computation that is embedded into the typing rules, and a *quote* operation which turns a pre-quote into a quote by closing over its free variables from the language interface: The judgment $\Gamma \vdash t : T \rhd t'$ handles the introduction of free variables R, app, lam, tapp and tlam, while the rule (QUOTE) defines a term $\langle t \rangle$ of type $\Gamma \vdash \langle t \rangle$:$\mathsf{Expr}\, T$ which binds these free variables. The rules for $\Gamma \vdash t : T \rhd t'$ are a syntactic superset of the rules for $\Gamma \vdash t : T$ in Fig. 2 that do not add any new constraints, hence every well-typed term can be quoted. The newly added parts are framed.

Note that according to that rule, a term t that contains the variables R, app, lam, tapp and tlam cannot be quoted, but a simple α-renaming is sufficient to remove potential name clashes: As the pre-quoting relation is closed under α-renaming, there is an α-equivalent term which can be pre-quoted, if a pre-quoting judgment can be derived for the original term. Similarly, if a pre-quoting judgment $\Gamma \vdash t : T \rhd t'$ is derived for a term t such that T contains these variables, there is an α-equivalent type, such that t can be quoted, by the same argument.

Having shown the quoting mechanism, we can now demonstrate, how to implement a meta-circular interpreter for quoted F^*_ω terms in F^*_ω. In order to define the interpreter, we have to specify

```
       ID ≡ λT :: ∗ . T
  selfapp  : App ID
  selfapp ≡ ΛT1 :: ∗ . ΛT2 :: ∗ . λf : (T1 → T2) . λx : T1 . f x
  selflam  : Lam ID
  selflam ≡ ΛT1 :: ∗ . ΛT2 :: ∗ . λf : (T1 → T2) . f
 selftapp  : TApp ID
 selftapp ≡ ΛT1 :: □ . ΛT2 :: (T1 ⇒ ∗).
              λf : (∀X :: T1 . T2 X) . ΛX : T1 . f [X]
 selftlam  : TLam ID
 selftlam ≡ ΛT1 :: □ . ΛT2 :: (T1 ⇒ ∗) . λf : (∀X :: T1 . T2 X) . f
     eval  : ∀A :: ∗ . Expr A → A
     eval ≡ ΛA :: ∗ . λe : Expr A.
              e ID selfapp selflam selftapp selftlam
```

Figure 4. Self Interpreter. The types of the terms are given for better readability.

the meaning of (type) application and (type) abstraction in the interpretation, and give an according definition of R that makes the interpretation well-typed. For the meta-circular interpreter, this is shown in Fig. 4. All the operations are basically identity functions, and R is accordingly the identity function on type level.

4. Metatheory of F^*_ω

We establish basic properties of F^*_ω by encoding it as a pure type system (PTS) [4, 29, 3]. PTS are a family of lambda calculi, parameterized by the sorts of terms they contain (e.g. expressions, types, kinds, ...), their relations (e.g. expressions are qualified by types) and the allowed abstractions (e.g. terms can abstract over types).

F^*_ω is the following PTS instance.

$$\mathcal{S} = \{*, \square\}$$

$$\mathcal{A} = \{* : \square, \square : \square\}$$

$$\mathcal{R} = \{(*, *, *), (\square, *, *), (\square, \square, \square)\}$$

The set of sorts $\mathcal{S}$ specifies that we have two levels, terms whose types are classified by $*$ and types whose types are classified by $\square$. The axioms $\mathcal{A}$ correspond to the rules (K-TYPE) and (K-KIND), and the relations $\mathcal{R}$ allow the three types of abstraction available in our language: terms can abstract over terms, terms can abstract over types, and types can abstract over types.

The PTS instances corresponding to F_ω, λ^* and F^*_ω are compared in Fig. 5, which suggests that F^*_ω contains F_ω on the term level, and λ^* on the type level (renaming $*$ to $\square$).

While the representation of F^*_ω as a pure type system may be less accessible for readers not familiar with PTS, it allows us to easily derive a number of properties which hold for all PTS, including subject reduction. To do so, we have to show that our typing and kinding rules indeed correspond to the PTS rules specialized for the instance defined above. A F^*_ω term is converted to PTS syntax by dropping all brackets and replacing :: with :, Λ with λ, and $\forall$ with Π. While the syntactic categories collapse in this process, the distinction between terms and types is still upheld by the type system, which distinguishes between terms whose types are of sort $*$, and types whose types are of sort $\square$. Therefore, each well-typed PTS term can be translated back to F^*_ω in a type-directed manner, replacing : by :: if the qualifier has type $\square$, Π by $\forall$ if the corresponding term has type $*$, λ to Λ if the type of the whole term has type $*$, but the qualifier in the λ-Expression has type $\square$, and $f\ a$ to $f\ [a]$ if the type of the type of a is $\square$, but the type of the type of $f\ a$ is $*$.

The treatment of the term-type-dichotomy on the syntactic level allows us to omit some checks of the form $\Gamma \vdash t : *$ resp. $\Gamma \vdash t : \square$ in the typing and kinding rules in Fig. 2, which have to be done in the typing rules for general PTS as presented in [3]. However, it is easy to see that the omitted checks are syntactically ensured in F^*_ω. By this embedding into the PTS framework we can hence reuse the meta-theory of PTS, and in particular we have:

THEOREM 1 (Subject Reduction).

$$\Gamma \vdash t : T \text{ and } t \rightarrow_\beta t' \implies \Gamma \vdash t' : T$$

Proof. Follows by encoding of F^*_ω as PTS and Thm. 5.2.15 in [3]. ∎

Furthermore, since F^*_ω contains λ^* in the sense described above, we get the following results:

THEOREM 2. *The kind system of F^*_ω is undecidable. As a logic, it is inconsistent.*

Proof. Follows from the undecidability [23] and inconsistency [8, 12] of the λ^* type system. ∎

While the inconsistency result is not relevant for our purpose of using F^*_ω as a programming language[1], the undecidability of the kind (and hence type) system is potentially a more serious issue. We believe that the undecidability is, as in many other undecidable type systems, not very significant in practical programming, since the programs that lead to non-termination of the type checking algorithm are quite esoteric [8]. This is supported by anecdotal evidence in the form of the experiments we made with our F^*_ω typechecker implementation, where we never encountered the problem that the typechecker loops.

Let us now discuss the properties of quoting and self evaluation, in particular whether they comply to the criteria which we set up in Sec. 2. In the paper, we only provide short proof sketches. A full formalization in Coq using a locally nameless representation and corresponding machine-checked proofs are available for download[2].

4.1 Representation Property

We first have to show that our quote transformation is applicable to all well-typed terms.

LEMMA 1. *For all t, T and Γ which do not contain* R, lam, app, tlam *or* tapp,

$$\Gamma \vdash t : T \iff \exists t' . \Gamma \vdash t : T \rhd t'.$$

Proof. Assume t' exists so that $\Gamma \vdash t : T \rhd t'$, then the inference tree for $\Gamma \vdash t : T \rhd t'$ can be transformed into an inference tree for $\Gamma \vdash t : T$ by removing all parts not framed in Fig. 3.

Now assume that $\Gamma \vdash t : T$. Then the inference tree of $\Gamma \vdash t : T$ can be extended to an inference tree for $\Gamma \vdash t : T \rhd t'$ for some t' by filling in the parts framed in Fig. 3, proceeding from top to bottom, since no new premises have to be added. ∎

THEOREM 3 (Existence of Quoting). *For all t and T, where w. l. o. g.* R, lam, app, tlam, tapp $\notin$ t, T,

$$\emptyset \vdash t : T \implies \exists t' . \langle t \rangle = t'$$

Proof. Follows by inspection of (QUOTE) and Lemma 1 ∎

F^*_ω allows encoding the family of types Expr_T directly as type function Expr as shown in Fig. 1. We can prove that Expr adheres to Prop. 1 in Sec. 2.

LEMMA 2. *For all t, T and Γ which do not contain* R, lam, app, tlam *or* tapp, *with Γ' defined as Γ prefixed by* R :: $* \Rightarrow *$, lam : TLam R, app : TApp R, tlam : TLam R, tapp : TApp R *and with each occurrence of $t : T$ replaced by $t :$* R T,

$$\Gamma \vdash t : T \rhd t' \implies \Gamma' \vdash t' : \mathsf{R}\,T.$$

Proof. Follows by induction on the structure of the inference tree of $\Gamma \vdash t : T \rhd t'$. See the Coq formalization for details. ∎

THEOREM 4 (Welltyped Quote). *For all t and T,*

$$\emptyset \vdash t : T \quad \wedge \quad \langle t \rangle = t' \implies \emptyset \vdash t' : \mathsf{Expr}\,T$$

Proof. Follows by inspection of (QUOTE) and Lemma 2 ∎

	F_ω	F^*_ω	λ^*
$\mathcal{S}$	$*$	$*$	
	$\square$	$\square$	$*$
$\mathcal{A}$	$* : \square$	$* : \square$	
		$\square : \square$	$* : *$
$\mathcal{R}$	$(*, *, *)$	$(*, *, *)$	
	$(\square, *, *)$	$(\square, *, *)$	
	$(\square, \square, \square)$	$(\square, \square, \square)$	$(*, *, *)$

Figure 5. Comparison of PTS instances

[1] It would be interesting, though, to analyze whether the inconsistency is confined to the kind system or whether it leaks into the type system

[2] `http://www.daimi.au.dk/~rendel/metacircular`

4.2 Adequacy Property

While we do not have a formal proof of adequacy yet, we believe that the self-representation presented in this paper is weakly adequate in the sense of Prop. 2 on the ground of the following argument.

Inspection of the typing rules in Fig. 2 shows that every normalizing, closed term of type Expr_T for some T must have been ultimately built by λ and Λ abstractions of the form

$$\Lambda \mathsf{R} :: \ast \Rightarrow \ast \,.\, \lambda \mathsf{lam} : \mathsf{Lam\,R}. \\ \lambda \mathsf{app} : \mathsf{App\,R} \,.\, \lambda \mathsf{tlam} : \mathsf{TLam\,R} \,.\, \lambda \mathsf{tapp} : \mathsf{TApp\,R} \,.\, \mathsf{body}$$

for some term body with

$$\mathsf{R} :: \ast \Rightarrow \ast, \mathsf{lam} : \mathsf{TLam\,R}, \mathsf{app} : \mathsf{TApp\,R}, \\ \mathsf{tlam} : \mathsf{TLam\,R}, \mathsf{tapp} : \mathsf{TApp\,R} \;\vdash\; \mathsf{body} : \mathsf{R\,A}.$$

Since R is entirely abstract in body, the result of type R A must have been produced by applications of the functions lam, app, tlam and tapp, due to relational parametricity.

4.3 First-Class Interpretation Property

F_ω^* allows encoding an interpretation with respect to some representation type function R directly as a term of type

$$\forall \mathsf{A} : \Box \,.\, \mathsf{Expr\,A} \to \mathsf{R\,A}.$$

See eval in Fig. 4 for an example of such a term.

4.4 Self Interpretation Property

We can prove that eval T t′ is observational equivalent to t if t has type T and $\langle \mathsf{t} \rangle = \mathsf{t}'$. In fact, eval T t′ is even β-equivalent to t, which means that there is a potential for partial evaluation that might allow self-interpretation to be done without the order-of-magnitude slowdown in performance that is usually associated with it.

LEMMA 3. *For all t, T and Γ which do not contain* R, lam, app, tlam *or* tapp,

$$\Gamma \vdash t : T \;\triangleright\; t' \quad\Longrightarrow\quad t =_\beta \mathsf{eval}\, T\, t'$$

Proof. Follows by induction on the structure of the inference tree of $\Gamma \vdash t : T \triangleright t'$. See the Coq formalization for details. ■

THEOREM 5. *For all t and T,*

$$\emptyset \vdash t : T / \langle t \rangle = t' \quad\Longrightarrow\quad t =_\beta \mathsf{eval}\, T\, t'$$

Proof. Follows by inspection of (QUOTE) and Lemma 3 ■

4.5 Reflection Property

This property demands that other, non-trivial interpretation besides standard evaluation must be possible. To this end, Figure 6 illustrates how an interpretation that measures the size of a term can be expressed. Our language does not support natural numbers directly, but we assume that a standard Church encoding of natural numbers is used.

Unfortunately, a closer look at the definition of ctlam reveals a significant problem: It is not quite clear which type argument to supply to f. It is obvious that the type argument has no significance, since the type function $\forall X :: T_1 \,.\, Nat$ is constant function. But still, we have to supply one. In the figure, we have used a constant $\perp :: (\Pi S :: \Box \,.\, S)$ which can easily be added to the language. Another solution is to add syntax for coercing a term of type $\forall X :: T_1 \,.\, T_2$ to T_2 if T_1 is not free in T_2. Both of these solutions work fine for any constant representation function. This means, that we can at least define reflective interpretations into an untyped domain. However, the solutions are still somewhat unsatisfactory. We will revisit this problem in Sec. 6.

$$\begin{aligned}
\mathsf{INTR} &\equiv \lambda \mathsf{T} :: \ast \,.\, \mathsf{Nat}\\
\mathsf{capp} &\equiv \Lambda \mathsf{T}_1 :: \ast \,.\, \Lambda \mathsf{T}_2 :: \ast \,.\, \lambda \mathsf{f} : \mathsf{Nat} \,.\, \lambda \mathsf{x} : \mathsf{Nat} \,.\, \mathsf{f} + \mathsf{x} + 1\\
\mathsf{clam} &\equiv \Lambda \mathsf{T}_1 :: \ast \,.\, \Lambda \mathsf{T}_2 :: \ast \,.\, \lambda \mathsf{f} : \mathsf{Nat} \to \mathsf{Nat} \,.\, (\mathsf{f}\,0) + 1\\
\mathsf{ctapp} &\equiv \Lambda \mathsf{T}_1 :: \Box \,.\, \Lambda \mathsf{T}_2 :: (\mathsf{T}_1 \Rightarrow \ast) \,.\, \lambda \mathsf{f} : \mathsf{Nat} \,.\, \Lambda \mathsf{X} :: \mathsf{T}_1 \,.\, \mathsf{f} + 1\\
\mathsf{ctlam} &\equiv \Lambda \mathsf{T}_1 :: \Box \,.\, \Lambda \mathsf{T}_2 :: (\mathsf{T}_1 \Rightarrow \ast).\\
&\qquad \lambda \mathsf{f} : (\forall \mathsf{X} :: \mathsf{T}_1 \,.\, \mathsf{Nat}) \,.\, \mathsf{f}[\perp \mathsf{T}_1] + 1\\
\mathsf{ceval} &: \forall \mathsf{A} :: \ast \,.\, \mathsf{Expr\,A} \to \mathsf{Nat}\\
\mathsf{ceval} &\equiv \Lambda \mathsf{A} :: \ast \,.\, \lambda \mathsf{e} : \mathsf{Expr\,A}.\\
&\qquad \mathsf{e\ INTR\ capp\ clam\ ctapp\ ctlam}
\end{aligned}$$

Figure 6. Interpreter which measures the size of a term

4.6 Coq Formalization

We have formalized F_ω^* in the Coq theorem prover [5] using a locally nameless representation as proposed by Aydemir et al. [2]. The representation uses de Bruijn indices to represent bound variables and atoms with decidable equality to represent free variables. This allows easy reasoning up to α equivalence, in fact, term equality *is* α equivalence. However, a number of technical lemmas have to be formulated and proved which connect the various forms of substitution for free and for bound variables. Unfortunately, the approach described by Aydemir et al. seems not to scale well in the number of syntactic categories. Even with only two syntactic categories as for F_ω^*, we need up to five versions for some of the lemmas opposed to only one version in the case of a single syntactic category. This explosion of technical proof obligations somewhat overshadows our experience with Coq and the locally nameless approach.

We have formalized the typing, kinding and quoting relation, and proved Lemmas 2 and 3 and Theorems 4 and 5 as stated in Sec. 4. In our ongoing work about this topic, we plan to formalize our proof sketch from Sec. 4.5. It would be also interesting to formalize the embedding into the PTS framework.

5. Language Integration

The obvious application of our quoting mechanism is to integrate it into the programming language itself, which in turn enables a lot of applications well-known from untyped reflective languages. The application we are most interested in is a *polymorphic embedding* [16] of the host language, meaning that programs written in the language can be given a non-standard meaning by enabling a parameterization with the desired denotation of the language constructs.

The main conceptual challenge in integrating the quote mechanism into the language is the change of environment as described in Thm. 4, or, equivalently, the question of how to deal with free variables in quoted terms.

While the exact way how a quote function is integrated into the language is not in the scope (and page limit) of this paper, we want to sketch different ways how typed self-representation can be used to this end. Hence we assume that the syntax of our language is extended with terms of the form quote t.

5.1 Static Quoting

One way of interpreting quoting is by a transformation during type checking. The easiest (but most restricted) way of dealing with free variables is to allow only quoting of closed terms. Hence the typing rule for quoted terms is:

$$\frac{\emptyset \vdash t : T}{\emptyset \vdash \mathsf{quote}\, t : \mathsf{Expr}\, T}$$

The quotes can simply be “compiled away” by a syntax-directed transformation during type checking, i.e., $\mathsf{quote}\, t$ is transformed to $\langle t \rangle$.

If we want to allow free variables within quoted terms, they have to be lifted from a type A to a type $\mathsf{R\,A}$. One way to do this is letting the programmer specify their *lift* function by making it part of the language interface. For example, for a self-interpreter, where R is the identity function, the lift function could be the identity function. For the interpreter in Fig. 6 the lift function could be the function returning zero for all inputs.

In order to allow this, one could extend the language interface by a *lift* function of type

$$\mathsf{Lift} \equiv \Lambda \mathsf{R} :: * \,.\, \forall \mathsf{A} :: * \,.\, \mathsf{A} \to \mathsf{R\,A},$$

which lets the programmer control how external terms are lifted into the respective representation. The language interface component Expr hence becomes

$$\mathsf{Expr} \equiv \ldots \to \mathsf{TLam\,R} \to \mathsf{Lift\,R} \to \mathsf{R\,A}.$$

To this end, the definition of quoting in Fig. 3 must be changed such that the quote of every occurrence of a variable x that is bound outside the quote is $\mathsf{lift\,x}$. Occurrences of variables bound inside the quote can easily be distinguished from those bound outside by splitting the environment into two parts – one for the externally bound variables, and one for the internal ones.

5.2 Dynamic Quoting

Another interesting, but more speculative possibility would be to perform the quote transformation during reduction. This means that free variables in the term to be quoted may have been substituted with other terms during reduction before the actual quote transformation takes place. This implies that the result of the transformation depends on the reduction strategy (call-by-value, call-by-name etc.). This will make the result of quoting harder to predict. On the other hand, dynamic quoting enables exciting new possibilities, such as a *propagating quote*: Assuming that we have a fixed-point operator μ, and are given a representation R and interpretation functions lam, app, tlam, and tapp, we could then write a lift function such as

$$\begin{aligned}&\mu\, \mathsf{lift} : \mathsf{Lift\,R} \,.\, \Lambda \mathsf{A} :: * \,.\, \lambda \mathsf{x} : \mathsf{A} \,.\\ &\quad (\mathsf{quote\,x})\, \mathsf{R\,lam\,app\,tlam\,tapp\,lift}\end{aligned}$$

which would transitively quote every term that is bound outside the current quote and interpret it with the given interpretation. The exact design of dynamic quoting and an evaluation of its practical utility is part of our future work.

6. Discussion

We can now reflect on the design decisions that we have made on the way. This will lead to the identification of a number of areas of future work.

First, we will discuss the most important design choices we have made in our implementation: the choice of language, the representation of terms, and of variables. Then, we will discuss the problems arising from the integration of externally defined terms.

6.1 The choice of F_ω^*

Pfenning / Lee [21] have been the first to explore metacircularity in the context of statically typed functional languages, in particular, in extensions of System F.

They are able to represent terms of F_2 in F_3, and terms of F_ω in F_ω^+. The latter is their extension of F_ω with kind variables and kind polymorphism on terms. They claim that it is strongly normalizing and type-checking is decidable.

However, they do not achieve metacircularity, missing Prop. 1. As we will discuss below, they furthermore do not achieve Prop. 5. In order to embed F_ω^+, they would need a language that allows for the representation of kind abstraction and kind application, which would require a notion of sorts. By fusing the categories of types and kinds, we are able to evade this problem and achieve full metacircularity, for the price of losing strong normalization and decidable type-checking.

Carette et al. [7] (in the journal version) attempt to write a self-interpreter without drawing on higher-rank and higher-kind polymorphism. Instead they use a simply-typed lambda calculus extended with let-bound polymorphism. However, they cannot define a transformation *quote* that returns a term in their language. They can only specify a pre-encoding that contains free variables and has to be put in the context of an evaluator in order to build a closed term. Therefore, it is impossible to type-check a term separately (Prop. 1). Accordingly, it is impossible to talk of adequacy (Prop. 2). Furthermore, interpretations can therefore not be regarded as first-class citizens (Prop. 3). In the sense of Prop. 4, however, they define a self interpretation, and their representation allows for non-standard interpretations in the sense of the reflection property (Prop. 5).

Another conceivable choice of language would be to use λ^* rather than F_ω^*. However, it is not obvious whether and how typed self-representation in λ^* is possible. The main problem in λ^* is that terms have exactly the same structure as types, and hence terms like $*$ or $\Pi \mathsf{x} : \mathsf{T} \,.\, \mathsf{T}'$ must also be represented. Another problem is the fact that types may depend on terms. Intuitively, it is not clear how to abstract over the interpretation of a term, if the type of the term depend on a fixed interpretation of the term. For example, to consider types that depend on terms, the App type, which (omitting the boilerplate declarations) is $R\,(A \to B) \to R\,A \to R\,B$ in F_ω^* (see Fig. 1) would have to become something like $R\,(\Pi t : A \,.\, B\,t) \to \Pi t' : R\,A \,.\, X$ in λ^* – but then it is not clear which reasonable type to fill in for the X placeholder.

6.2 Representation of Terms

Our representation of terms contains several design decisions. Our basic decision is to follow Pfenning / Lee [21] in not representing types explicitly, but mapping them to types in the meta-language. This automatically precludes the construction of non-welltyped terms. However, as a consequence, the representation of variables also has to be delegated to the meta-language. Therefore, we likewise use higher-order abstract syntax (HOAS).

The central difference to Pfenning / Lee [21] is in how we embed object language terms into the meta-language without an explicit notion of data types. The concept of representing terms within lambda calculi goes back to Church’s numerals and booleans. Böhm / Berarducci [6] first defined a representation in the typed setting of System F.

Pfenning / Paulin-Mohring [22] have extended this approach to allow Church encodings for inductive definitions of generalized abstract data types (GADTs), i. e. data types whose (polymorphic) constructors are non-uniform in their type variables, within F_ω. Pfenning / Lee [21] apply this approach to their embeddings of System F terms in F_3 and of F_ω terms in F_ω^+.

However, this representation only works for inductively defined data types. In particular, the data type must not appear in a negative position of a constructor. In our representation, lam has type

$$\lambda \mathsf{R} :: * \Rightarrow * \,.\, \forall \mathsf{A} :: * \,.\, \forall \mathsf{B} :: * \,.\, (\mathsf{R\,A} \to \mathsf{R\,B}) \to \mathsf{R}\,(\mathsf{A} \to \mathsf{B}),$$

and $\mathsf{R\,A}$ appears in negative position. Pfenning / Lee [21] instead propose a solution, where lam has type[3]

$$\forall \mathsf{A} :: \ast \,.\, \forall \mathsf{B} :: \ast \,.\, (\mathsf{A} \to \pi\, \mathsf{B}) \to \pi(\mathsf{A} \to \mathsf{B}),$$

which strongly restricts the range of definable interpretations to basically only the standard interpretation [21, p. 150]. Besides not being metacircular, their approach therefore cannot satisfy property 5 of a typed self-representation (see Sec. 2).

The representation of terms that we have chosen has been inspired by Carette et al. [7], who developed it in the context of a simply typed lambda calculus. In this setting, the representation allows for implementing simple alternative semantics, like counting the number of terms. We have defined a straightforward extension of their representation to type abstraction and application. In Sec. 3.3 we discussed the limitation of this extension: If the representation is a constant function, then we have the problem that no appropriate type parameter is available in the TLam case. We have discussed two different solutions to this problem and can conclude that in the case of non-polymorphic representations, these solutions are sufficient.

However, one could argue that our solutions are somewhat ad hoc since they do not address the real cause of the problem, which lies in the design of the language interface itself: Each tlam gets domain S and codomain T and a parametric function f as a parameter. Depending on the type $\mathsf{R}\ (\forall \mathsf{X} :: \mathsf{S}\,.\,\mathsf{T\,X})$, it has potentially three options. Firstly, if there is a closed term that inhabits the type, it can return this term (in our case, it could return the Church encoding of a constant number). Secondly, if the type is β-equivalent to a type $\forall \mathsf{X} :: \mathsf{S}\,.\,\mathsf{R}\,(\mathsf{T\,X})$, it can simply return f. The only alternative option to make use of f, which is the only available term, was to supply a type $\mathsf{X} :: \mathsf{S}$. However, as S can be any kind, this can only be a type $\perp \mathsf{S}$. Hence we conclude that our current representation is not very good at representing type abstraction.

6.3 Accessing the intensional structure of programs

However, the representation of type abstraction is just one issue in a wider context of questions related to type-safe program representations. A general challenge for the expressivity of each kind of representation is the "degree of access to the intensional structure of programs" [21, p. 156]. Carette et al. [7] recognize that their representation is too limited in this regard, as they cannot define type functions which destruct or inspect their argument. In this representation, they cannot express a partial evaluation or a transformation to continuation-passing style (CPS). To overcome this, they propose an ad-hoc solution involving adding an additional type argument to R for every different type function which is to be implemented. However, this means that adding a new interpretation to the language potentially necessitates a redefinition of the language interface. This non-modularity strongly suggests the search for a better solution.

However, in F_ω^* we have a very expressive type system at our disposal. We can therefore overcome this limitation by using type-indexed types [15]. To demonstrate this capability, we have so far developed an embedding of the simply typed lambda calculus with natural numbers into F_ω^*, and defined an evaluator and a call-by-name continuation passing style interpreter (the latter is modeled after the corresponding interpreter in [7]). To this aim, we constructed a simply typed universe as in [1], with a constructor Num for representing the base type Nat, and a constructor Arrow for representing function types. However, we had to find a way to represent typecasing [14]. We used a simple Church encoding of the types in the universe, defining the universe U and the constructors Num and Arrow as in Fig. 7.

[3] Where π corresponds roughly to our Expr.

$$
\begin{array}{lcl}
\mathsf{U} & = & \Pi\, \mathsf{X} :: \Box \,.\, \mathsf{X} \Rightarrow (\mathsf{X} \Rightarrow \mathsf{X} \Rightarrow \mathsf{X}) \Rightarrow \mathsf{X} \\
\mathsf{Num} & = & \lambda \mathsf{X} :: \Box \,.\, \lambda \mathsf{N} :: \mathsf{X} \,.\, \lambda \mathsf{Ar} :: \mathsf{X} \Rightarrow \mathsf{X} \Rightarrow \mathsf{X} \,.\, \mathsf{N} \\
\mathsf{Arrow} & = & \lambda \mathsf{A} :: \mathsf{U} \,.\, \lambda \mathsf{B} :: \mathsf{U} \,.\, \lambda \mathsf{X} :: \Box \,.\, \lambda \mathsf{N} :: \mathsf{X} \,.\, \lambda \mathsf{Ar} : \mathsf{X} \to \mathsf{X} \to \mathsf{X}. \\
 & & \qquad \mathsf{Ar}(\mathsf{A\,N\,Ar})(\mathsf{B\,N\,Ar}) \\
 & & \\
\mathsf{NumR} & = & \lambda \mathsf{R} :: (\mathsf{U} \Rightarrow \ast) \,.\, \mathsf{Nat} \to \mathsf{R\,Num} \\
\mathsf{App} & = & \lambda \mathsf{R} :: (\mathsf{U} \Rightarrow \ast) \,.\, \forall \mathsf{S} :: \mathsf{U} \,.\, \forall \mathsf{T} :: \mathsf{U}. \\
 & & \qquad \mathsf{R}(\mathsf{Arrow\,S\,T}) \to \mathsf{R\,S} \to \mathsf{R\,T} \\
\mathsf{Lam} & = & \lambda \mathsf{R} :: (\mathsf{U} \Rightarrow \ast) \,.\, \forall \mathsf{S} :: \mathsf{U} \,.\, \forall \mathsf{T} :: \mathsf{U}. \\
 & & \qquad (\mathsf{R\,S} \to \mathsf{R\,T}) \to \mathsf{R}(\mathsf{Arrow\,S\,T}) \\
\mathsf{Expr} & = & \lambda \mathsf{A} :: \ast \,.\, \forall \mathsf{R} :: (\mathsf{U} \Rightarrow \ast). \\
 & & \qquad \mathsf{NumR\,R} \to \mathsf{App\,R} \to \mathsf{Lam\,R} \to \mathsf{R\,A}
\end{array}
$$

Figure 7. Embedding of STLC using type-indexed types

$$
\begin{array}{lcl}
\mathsf{R_{eval}} & = & \lambda \mathsf{X} :: \mathsf{U} \,.\, \mathsf{X} \ast \mathsf{Nat}(\lambda \mathsf{A} :: \ast \,.\, \lambda \mathsf{B} :: \ast \,.\, \mathsf{A} \to \mathsf{B}) \\
\mathsf{int_{eval}} & = & \lambda \mathsf{x} : \mathsf{Nat} \,.\, \mathsf{x} \\
\mathsf{app_{eval}} & = & \Lambda \mathsf{A} :: \mathsf{U} \,.\, \mathsf{lambdaB} : \mathsf{U} \,.\, \lambda \mathsf{f} : (\mathsf{R_{eval}A}) \to (\mathsf{R_{eval}B}). \\
 & & \qquad \lambda \mathsf{x} : (\mathsf{R_{eval}A}) \,.\, \mathsf{f\,x} \\
\mathsf{lam_{eval}} & = & \Lambda \mathsf{A} :: \mathsf{U} \,.\, \lambda \mathsf{B} : \mathsf{U} \,.\, \lambda \mathsf{f} : (\mathsf{R_{eval}A}) \to (\mathsf{R_{eval}B}) \,.\, \mathsf{f} \\
\mathsf{eval} & = & \Lambda \mathsf{A} :: \mathsf{U} \,.\, \lambda \mathsf{e} : \mathsf{Expr\,A} \,.\, \mathsf{e}\,[\mathsf{R_{eval}}]\, \mathsf{int_{eval}}\, \mathsf{app_{eval}}\, \mathsf{lam_{eval}}
\end{array}
$$

Figure 8. An evaluator using type-indexed types

Every interpretation has to define a decoding function $\mathsf{R} :: \mathsf{U} \Rightarrow \ast$ on types that reflects the types of the produced values. We call it R, as its role is similar to the type constructor in our representation. Accordingly, the language interface NumR, App, Lam, and Expr is similar to the one given in Fig. 1.

The code for the evaluator is shown in Fig. 8. The decoding function for the evaluator $\mathsf{R_{eval}}$ represents numbers by the type Nat and arrows by $\to$. The code for the call-by-name CPS interpreter is shown in Fig. 9. Its decoding function $\mathsf{R_{cps}}$ is more complex. $\mathsf{R'_{cps}}$ reflects that numbers and arrows have to be represented differently.

These examples demonstrate that very expressive interpretations can be encoded using the same language interface in this style. In the future, we want to analyze how this approach can be generalized to the full self-embedding of F_ω^* by finding an encoding that encompasses the representation of universal quantification. It should be noted, however, that this is still concerned with the representation of terms and finding an appropriate mapping for the types that reflects this representation. Therefore, we do not require a representation of dependent kinds, as they are not contained in the language interface. Furthermore, the interpretations themselves still are parametric. For example, the interpretation of lam cannot depend on the type of the function's parameter. This is, because interpretations are ordinary F_ω^* terms, and F_ω^* does not support type-indexed functions. As part of our future work, we also want to consider defining an explicit representation of types by terms. This would imply the possibility of defining type-indexed functions.

6.4 Representation of variables

Representation of variables, both bound and free, can be seen as the main challenge in typed self representation. We already discussed the issue of free variables in Sec. 5. There, the problem has been described as one of lifting external terms into the respective interpretation using a static or dynamic quoting approach. While bound and free variables may be represented differently in some representations, they are always closely related, because an open term can be closed by adding lambda binders around it. That means that even a typed self-representation which restricts itself to closed terms should account for free variables, since they naturally occur during construction of closed terms.

$$
\begin{array}{lcl}
\mathsf{R_{cps}} & = & \lambda \mathsf{X} :: \mathsf{U} \,.\, \mathsf{Ct}(\mathsf{R'_{cps}}\,\mathsf{X}) \\
\mathsf{R'_{cps}} & = & \lambda \mathsf{X} :: \mathsf{U} \,.\, \mathsf{X} * \mathsf{Nat}(\lambda \mathsf{A} :: * \,.\, \lambda \mathsf{B} :: * \,.\, \mathsf{Ct\,A} \rightarrow \mathsf{Ct\,B}) \\
\mathsf{Ct} & = & \lambda \mathsf{A} :: * \,.\, \forall \mathsf{W} :: * \,.\, (\mathsf{A} \rightarrow \mathsf{W}) \rightarrow \mathsf{W} \\
\\
\mathsf{int_{cps}} & = & \lambda \mathsf{x} : \mathsf{Nat} \,.\, \lambda \mathsf{W} : * \,.\, \lambda \mathsf{k} : (\mathsf{Nat} \rightarrow \mathsf{W}) \,.\, \mathsf{k\,x} \\
\mathsf{app_{cps}} & = & \Lambda \mathsf{A} :: \mathsf{U} \,.\, \Lambda \mathsf{B} : \mathsf{U} \,.\, \lambda \mathsf{f} : \mathsf{R_{cps}}(\mathsf{Arrow\,A\,B}) \,.\, \lambda \mathsf{x} : \mathsf{R_{cps}A}. \\
 & & \quad \Lambda \mathsf{W} :: * \,.\, \lambda \mathsf{k} : ((\mathsf{R'_{cps}B}) \rightarrow \mathsf{W}). \\
 & & \qquad \mathsf{f\,W}(\lambda \mathsf{g} : ((\mathsf{R_{cps}A}) \rightarrow (\mathsf{R_{cps}B}))\,.\, \mathsf{g\,x}\,[\mathsf{W}]\,\mathsf{k}) \\
\mathsf{lam_{cps}} & = & \Lambda \mathsf{A} :: \mathsf{U} \,.\, \Lambda \mathsf{B} :: \mathsf{U} \,.\, \lambda \mathsf{f} : (\mathsf{R_{cps}A} \rightarrow \mathsf{R_{cps}B}). \\
 & & \quad \Lambda \mathsf{W} :: * \,.\, \lambda \mathsf{k} : ((\mathsf{R_{cps}A} \rightarrow \mathsf{R_{cps}B}) \rightarrow \mathsf{W}) \,.\, \mathsf{k\,f} \\
\mathsf{cps} & = & \Lambda \mathsf{A} :: \mathsf{U} \,.\, \lambda \mathsf{e} : \mathsf{Expr\,A} \,.\, \mathsf{e}\,[\mathsf{R_{cps}}]\,\mathsf{int_{cps}}\,\mathsf{app_{cps}}\,\mathsf{lam_{cps}}
\end{array}
$$

Figure 9. A call-by-name CPS interpreter using type-indexed types

In the tradition of Pfenning/Lee [21] and Carette et al. [7], we have chosen to use HOAS to represent variables of the embedded language as variables of the host language. But there is an important difference between their approaches regarding the type of the variables. Pfenning/Lee [21] represent embedded variables of type A by host variables of type A, while Carette et al. [7] use host variables of type $\mathsf{R\,A}$. Pfenning and Lee's approach allow them to represent terms with free variables similar to how they represent terms with bound variables. A term of type A with a free variable of type B can be represented as $\mathsf{A} \rightarrow \pi\,\mathsf{B}$. Indeed, their constructor lam for representing lambda abstraction can also be used to wrap a term with a free variable in a lambda binder to bind it, independently of the semantics.

However, as we have already discussed in the previous section, this representation of variables does not allow for interesting semantics, since the values of all variables have to be given in the host language semantics, not some embedded semantics.

To have more flexible variables, which values can be expressed according to the embedded semantics, Carette et al. [7] use terms of type $(\mathsf{R\,A} \rightarrow \mathsf{R\,B})$ to represent bound variables. While this allows interesting semantics, the authors do not present an analogue to the lam function above and do not discuss the representation of free variables.

Naively, one would like to represent terms of type B with a free variable of type A as $\mathsf{ExprA} \rightarrow \mathsf{ExprB}$. Consider, we had a datatype of terms containing a free variable that is constructed from this function type. In order to work with such terms, we had to find a way to decompose them. It is well known that one can not generally decompose datatypes which contain function types (like the type of lam). Fortunately, we can further restrict the type of functions that are allowed to be used to represent terms with free variables: A term should not be allowed to have access to the internals of the representation of the variable. This corresponds to the requirement that the function has to be parametric. Fegaras / Sheard [11] have shown that terms constructed from those functions can be decomposed. Recently, Washburn / Weirich [30] proposed a representation of the *untyped* lambda calculus into the F_ω fragment of Haskell that allows for a binding operation of type $(\mathsf{Expr} \rightarrow \mathsf{Expr}) \rightarrow \mathsf{Expr}$, and still is able to apply the technique of [11]. The authors believe that this approach can be adapted to the typed self-representation described in this work.

6.5 Further Approaches

20 years ago Hagiya [13] presented a very innovative approach to writing a meta-circular interpreter in the setting of typed languages. It is based on a variant of the lambda calculus with Boolean and integer constants and symbols (variable constants), let bindings, if expressions, errors and casts. The language has type annotations, in particular for functions and fixed-point operations. It is a dependently-typed system, and types are considered just normal values. The base evaluator is untyped. A quote operation is defined in the style of quoting in Lisp, giving a term of type exp.

A meta-circular interpreter evaluates each quoted term to a dependent pair of its type and its value. It therefore works both as an evaluator and a type-checker. In this way, well-typedness for the meta-circular interpreter is shown, by running it on top of itself. The language can be redefined or extended by modifying the meta-circular interpreter. This new version can be run on top of the old version, as in the original ideas about reflection (e. g., [26]). This is certainly a great result, however it comes with its own problems. The phase distinction between evaluation and type-checking is abolished. Some normalization takes place under binders to ensure well-typedness. In this way, type-checking is not only undecidable from a theoretical point of view, but it is also clear that it has only limited applicability in practice. Still, there is no indication that type-checking is sound. It is not clear, how well-typedness of terms can be established in this dependently-typed system without a complete evaluation. We would therefore consider it more related to systems of *untyped* self-representation. Seen from the perspective of our properties of a typed self-representation, Prop. 1 is not fulfilled: each representation of a term is of type exp. Accordingly, Prop. 2 and Prop. 3 are not applicable. Prop. 4 is fulfilled, as a meta-circular interpreter is given. The kind of representation allows full access to the structure of the code, guaranteeing Prop. 5.

Another early approach of implementing reflection in the context of typed languages is from Läufer / Odersky [17]. They write a meta-interpreter for typed terms of the SK calculus in Haskell, together with a reification mechanism that lifts natural numbers to their representations, and a reflection mechanism that evaluates represented expressions to values. Although an interesting result in itself, it does not create a self-representation of the SK calculus, but a representation of SK terms in Haskell.

Our idea of propagating dynamic quoting is similar to the idea of MapClosure [25]. Apart from the fact that MapClosure is not statically safe, an operational difference is that MapClosure allows changing the meaning of bindings in the environment, whereas we allow changing the meaning of the language primitives.

Nanevski [19] proposes a language with explicit distinction of free and bound variables, and corresponding distinct abstraction mechanisms. In contrast to this work, Nanevski's encoding is not metacircular, but it would be interesting to check whether Nanevski's ideas could be transferred to our setting to address the representation of free variables. Nanevski provides pattern matching for intensional code analysis. However, the lack of polymorphic patterns precludes operations which work for terms of arbitrary types. It seems therefore not to be possible to implement interesting non-standard interpretations like size measurement in Nanevski's system.

7. Conclusion

Achieving a typed self-representation of programs is a longstanding goal in programming language development. We have analyzed the difficulties associated with its accomplishment and have seen that F_ω and related languages are probably not sufficient for achieving this goal. To remedy the problem, we have proposed a new language, F^*_ω, which is the first language that allows typed self-representation. We have analyzed the metatheoretical properties of F^*_ω and have established the main technical results in a mechanically checked proof. However, we have also seen that our approach to representing programs is not satisfactory with respect to representing type abstraction and application, hence our future work will concentrate on better representations of the type structure of terms.

Acknowledgments

The authors would like to thank Thomas Streicher and the anonymous reviewers for their insightful suggestions that helped to improve the paper. Tillmann Rendel and Christian Hofer are supported by the ScalPL project of the European Research Commission.

References

[1] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 1–20. Kluwer, B.V., 2003.

[2] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *POPL '08*, pages 3–15, New York, NY, USA, 2008. ACM.

[3] H. P. Barendregt. Lambda calculi with types. In *Handbook of logic in computer science (vol. 2): background: computational structures*, pages 117–309. Oxford University Press, 1992.

[4] S. Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Technical report, Department of Computer Science, Carnegie-Mellon University and Dipartimento Matematica, Universita di Torino, 1988.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:The Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[6] C. Böhm and A. Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.*, 39:135–154, 1985.

[7] J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated. In *APLAS'07, extended version to appear in Journal of Functional Programming*, pages 222–238. Springer LNCS 4807, 2007.

[8] T. Coquand. An analysis of Girard's paradox. In *In Symposium on Logic in Computer Science*, pages 227–236. IEEE Computer Society Press, 1986.

[9] R. Davies and F. Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001.

[10] 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.

[11] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *POPL '96*, pages 284–294. ACM, 1996.

[12] J. Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thése de doctorat detat, Université de Paris VII, 1972.

[13] M. Hagiya. Meta-circular interpreter for a strongly typed language. *J. Symb. Comput.*, 8(6):651–680, 1989.

[14] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95*, pages 130–141. ACM, 1995.

[15] R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. *Sci. Comput. Program.*, 51(1-2):117–151, 2004.

[16] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In *GPCE'08*. ACM, 2008.

[17] K. Läufer and M. Odersky. Self-interpretation and reflection in a statically typed language. In *Proc. OOPSLA Workshop on Reflection and Metalevel Architectures*. ACM, Oct. 1993.

[18] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[19] A. Nanevski. Meta-programming with names and necessity. In *ICFP*, pages 206–217, 2002.

[20] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI '88*, pages 199–208. ACM, 1988.

[21] F. Pfenning and P. Lee. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science*, 89(1):137–159, 1991.

[22] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the calculus of constructions. In *Proceedings of the 5th International Conference on Mathematical Foundations of Programming Semantics*, pages 209–228. Springer-Verlag, 1990.

[23] M. B. Reinhold. Typechecking is undecidable when "type" is a type. Technical report, Massachusetts Institute of Technology, 1989.

[24] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740. ACM, 1972.

[25] J. M. Siskind and B. A. Pearlmutter. First-class nonstandard interpretations by opening closures. In *POPL '07*, pages 71–76. ACM, 2007.

[26] B. C. Smith. Reflection and semantics in LISP. In *POPL '84*, pages 23–35. ACM, 1984.

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

[28] W. Taha, Z. Benaissa, and T. Sheard. Multi-stage programming: Axiomatization and type-safety. In *In 25th International Colloquium on Automata, Languages, and Programming*, pages 918–929. Springer-Verlag, 1998.

[29] J. Terlouw. Een nadere bewijstheoretische analyse van GSTT's. Technical report, Department of Computer Science, Catholic University, Nijmegen, The Netherlands, 1989.

[30] G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism*. *J. Funct. Program.*, 18(1):87–140, 2008.