Structural Types for the Factorisation Calculus

Author:
A. Bates

Supervisor:
R. N. S. Rowe

June 2014

Abstract

The factorisation calculus of Jay and Given-Wilson, a fundamental model of pattern matching, introduces the factorisation combinator, whose behaviour is conditional on the structure of its arguments. This combinator is typeable with polymorphic types such as System **F** types; this system does not have the property of type assignment decidability. We develop a new type system for the factorisation calculus, the structural type system, which augments Curry types with structural modifiers. These structural modifiers carry information sufficient to type the factors of an application solely from the type of the application itself. With this, we are able to develop a principal types algorithm, and show soundness and completeness properties.

We apply our type system to the theory of typed self-interpretation. Jay and Palsberg introduce an extension to the factorisation calculus which supports typed self-interpretation, again without decidable type assignment. We modify their calculus in order to preserve the properties of the principal types algorithm while retaining the self-interpreting terms. In this way we show the existence of a self-recogniser and self-enactor for a statically-typed calculus with decidable type inference.

Thanks to Reuben, my supervisor, for making a tremendously enjoyable project possible!

Contents

Introduction

The factorisation calculus of Barry Jay and Thomas Given-Wilson, first presented in 2011, attempts to be the fundamental model of pattern matching. The calculus, a combinatory calculus similar to combinatory logic, introduces the factorisation combinator F, whose reduction behaviour is different depending on whether its first argument is an operator or an application. This enables F to identify and decompose the internal structure of a term, and test for intensional equality, something not possible in combinatory logic.

Naturally, one may consider typing the factorisation calculus, extending it with a type system to characterise the behaviour of terms. The Curry type system of combinatory logic, simple but inexpressive, is not powerful enough to capture the two behaviours of the factorisation combinator. Jay and Given-Wilson show that the polymorphic types of System **F** are sufficient to type F. However, there is no known polymorphic type system that can type F for which type assignment is decidable. This is a very desirable property of a type system, both practically and theoretically; the ability to compute the type of a term algorithmically.

We attempt to develop a type system for the factorisation calculus for which type assignment is decidable. Our system, called the structural type system, is built on the idea of extending simple Curry types with structural modifiers, information which allows recovering the type of factors of an application solely from the type of the application itself (section 2.1). This idea is sufficient to characterise the behaviour of F when applied to a compound; we require a second type to capture the behaviour in the atomic case. With this construction we show a subject reduction result essential to any useful type system (section 2.2). Then, we are able to devise an algorithm to compute the principal types for a term, based on type unification. Crucially, our type system is a restrictive enough extension to the Curry system for us to show soundness and completeness properties for this algorithm, hence proving type assignment decidability (section 2.3).

Barry Jay, together with Jens Palsberg, went on to employ the expressive power of the factorisation calculus to the field of typed self-interpretation. Self-interpreters for a language or calculus are terms constructed in that calculus that are able to recognise or execute descriptions of other terms. Jay and Palsberg extend the factorisation calculus with additional combinators necessary to develop a self-recogniser and a self-enactor. A self-recogniser recovers the original term from its description, and a self-enactor executes a description to produce a description of the result. Their notion of describing terms behaves well with respect to their type system, again based on System **F** types, which of course does not have property of decidable type assignment. The authors leave open the problem of showing the possibility of self-interpretation for a statically-typed language with decidable type assignment.

We extend our structural type system to a reduced version of this calculus in which typed self-interpretation is still possible (chapter 3). Many of the properties of the type system for the factorisation calculus carry through here, including decidable type assignment. Our main contribution is in showing this result, and hence answering the problem left open by Jay and Palsberg. We find our calculus unwieldy, however, and have not yet devised an

implementation of sufficient efficiency to compute the principal types of the self-interpreters.

Chapter 1

Background

1.1 Combinatory Logic

A term rewriting system [9] is a collection of function symbols that combine to yield terms, together with reduction rules for these function symbols which model function application. Combinatory logic (CL) is one such term rewriting system, originally introduced by H.B. Curry as a variable-free model of computation [2]. The function symbols are S and K, and terms are defined thusly.

Definition 1.1.1. Terms t_1, t_2 are given by

$$t_1, t_2 ::= S \mid K \mid (t_1 t_2)$$

The term $(t_1 t_2)$ is read as *the application of* t_1 *to* t_2. Application is by convention left-associative, so for convenience leftmost, outermost parentheses are omitted.

Definition 1.1.2. Let t_1, t_2, t_3 be arbitrary terms. Then rewrite rules for the calculus are given by the reduction rules for the combinators

$$\begin{aligned} K\ t_1\ t_2 &\to t_1 \\ S\ t_1\ t_2\ t_3 &\to t_1 t_3 (t_2 t_3) \end{aligned}$$

and by the inductive definition

$$M \to N \implies \begin{cases} MP \to NP \\ PM \to PN \end{cases}$$

A term is called reducible if it (or any subterm) matches the pattern on the left-hand side of some rewrite rule, and it reduces to (the corresponding substitution of) the right-hand side. For example, if t_1 is an arbitrary term, $SKKt_1 \to Kt_1(Kt_1) \to t_1$. The reflexive, transitive closure of reduction defines an equivalence relation on terms, called *extensional equivalence* or *behavioural equivalence*. So the term SKK is behaviourly equivalent to the identity function, which we denote with I.

A term is in *normal form* if it is not reducible. A term is *normalising* if some reduction sequence yields a normal form, and *strongly normalising* if all do. In this way a strongly normalising term corresponds to a terminating program.

Fundamentally, combinatory logic is equivalent in expressiveness to lambda calculus (LC) [3]. Therefore, combinatory logic is Turing complete: it is able to represent any Turing-computable function on the natural numbers.

1.2 The Curry Type Assignment System

Type systems define a notion of 'typeability' such that typeable terms are in some sense well-defined. The Curry type assignment system [2, 3] was introduced by H.B. Curry for this purpose, in the context of lambda calculus. LC terms are built from *abstraction* and *application*, where intuitively abstraction builds functions and application applies abstractions to arguments. Therefore a well-defined application $t_1 t_2$ should require t_1 to be an abstraction. Type systems for LC, in particular Curry types, enforce this by distinguishing well-defined terms in the type system.

The Curry type system applies equivalently to combinatory logic, and it is this context that we will be concerned with.

Definition 1.2.1 (Curry Types)**.** Types A, B are constructed from type variables ϕ and the arrow constructor $\to$ by the following grammar

$$A, B ::= \phi \mid A \to B$$

The arrow constructor is right-associative, so we omit rightmost, outermost brackets.

Definition 1.2.2. Let A, B, C be arbitrary types. Assignable types to a term are defined by the following inference rules

$$(K): \quad \frac{}{\vdash K : A \to B \to A}$$

$$(S): \quad \frac{}{\vdash S : (A \to B \to C) \to (A \to B) \to A \to C}$$

$$(\to E): \quad \frac{\vdash M : A \to B \quad \vdash N : A}{\vdash MN : B}$$

Certainly the set of (Curry-) typeable terms is a strict subset of the set of all CL terms; the term $SII(SII)$, for example, cannot be typed. In fact a (Curry-) typeable term is strongly normalising [9], so we say the type system itself is strongly normalising. Hence, the Curry type system is restrictive enough to only type 'terminating programs'. More expressive type systems, such as the polymorphic System **F** (section 1.4), type a larger class of terms while retaining this property.

An interesting property of this system is decidable type inference [8]; there exists a terminating algorithm to decide whether a term is typeable and, if so, to return its most general type. Such an algorithm is based on *unification*, that is, deciding if two types can be made equal by means of *type substitution*. Robinson [8] showed that type unification returns the most general unifier, upon which the principal type inference algorithm hinges.

1.3 Factorisation Calculi

Factorisation calculi, in particular SF-calculus, recently introduced by Jay and Given-Wilson [5], are a family of combinatory calculi proposed as the fundamental model of pattern matching. Combinatory logic, and equivalently lambda calculus, are able to identify extensional

(behavioural) equivalence, but not intensional (structural) equivalence [5]. For example, there is no term in CL to differentiate SKK and SKS, as both behave as the identity. Perhaps surprisingly, such an equivalence function is Turing-computable.

Structure is built in combinatory logic through application, and the factorisation calculi introduce a combinator F whose behaviour exposes this structure. Put another way, reduction of F is conditional on the structure of its first argument, differentiating between atomic terms and applications. In such a way, Jay and Given-Wilson are able to develop a term in SF-calculus to decide equality of arbitrary normal forms.

SF-calculus is the fundamental factorisation calculus, and of primary interest to us. We present its construction here.

Definition 1.3.1. Terms t_1, t_2 in SF-calculus are given by

$$t_1, t_2 ::= S \mid F \mid (t_1 t_2)$$

As usual, leftmost, outermost parentheses are omitted.

Definition 1.3.2. A factorable or matchable form is a term that is a partial application. Therefore, if M and N are arbitrary terms, the factorable forms are S, SM, SMN, F, FM, FMN.

Definition 1.3.3. The terms S, F are variously called atomic terms or operators. A compound is a factorable application.

It is necessary to restrict the behaviour of F to factorable forms in order to preserve confluence [5]. Reduction is then given by

Definition 1.3.4. Let O, M, N, X, Y, Z, P be arbitrary terms. Single-step reduction ($\to$) of terms is given by the following rewrite rules

$$\begin{aligned} SXYZ &\to XZ(YZ) && \\ FOMN &\to M && \text{if } O \text{ atomic} \\ F(PQ)MN &\to NPQ && \text{if } PQ \text{ compound} \end{aligned}$$

and by the inductive definition

$$M \to N \implies \begin{cases} MP \to NP \\ PM \to PN \end{cases}$$

Reduction is defined as the reflexive, transitive closure of single-step reduction, denoted $\to^*$, or $\to$ when no distinction is necessary.

SF-calculus subsumes the expressiveness of combinatory logic, as K is behaviourly equivalent to FF (and FS). The main result is the development of a term to decide equality of normal forms (recall such a term does not exist in CL). There exists a term which compares two atoms of the calculus for equality (*eqatom* [5]), and F is able to distinguish atoms from compounds, so together these allow the recursive comparison of the structure of two terms. In fact, SF-calculus can be extended with additional combinators (such as the constructor C of SFC-calculus [5]) while preserving this property.

1.4 System F Types for *SF*-Calculus

Consider extending the Curry type system to SF-calculus. This requires a principal type for the factorisation combinator. In fact, no such principal type exists with Curry types. Fundamentally, this is because the type of an application does not determine the type of its factors; there is not enough information to construct a type for F. It is possible, however, to type the factorisation combinator with a more expressive type system. Jay and Given-Wilson present such a type using System **F** types.

System **F** [9] is a reformulation of Curry types to introduce *universal quantification* over type variables. This yields a tremendous increase in expressibility. Types are extended to allow universal quantification over type variables, and the inference rules are extended to allow introduction and elimination of these quantifiers. This is sufficient to type F with type scheme [3]

$$F : T \to U \to (\forall Z.(Z \to T) \to Z \to U) \to U$$

The quantifier is necessary to represent the unknown type of the right-hand factor in the compound.

Unlike the Curry type system, System **F** does not have the property of decidable type inference [11]. In some sense, it is too expressive for this property to be retained. While it is true that some fragments of System **F** preserve decidability (for example, the type system of ML), the nested quantifier of the above type means none of these restrictions are sufficient. It is this property, decidability of type inference, that we wish to preserve in developing a type system for SF-calculus.

1.5 Self-Interpretation

A self-interpreter for a programming language is an interpreter constructed in the same language that it is interpreting. Interpreters accept, as input, descriptions of a program, which are constructed using a *quote* function [7], mapping programs to their description. Interpreter behaviour, conditional on this input, fits broadly into one of two categories. A *recogniser* recovers the original program from the quotation, and an *enactor* executes the program description.

Self-interpreters are available for many programming languages [4] and for more fundamental models of computation, such as the lambda calculus. In the lambda calculus, self-recognisers and self-enactors are themselves lambda terms. Self-interpretation in this context has been extensively studied; Mogensen, for example, presents a self-enactor which executes its argument in linear time [6]. Likewise, Berarducci and Böhm construct self-interpreters as solutions to systems of equations in LC [1], solutions which surprisingly do not involve a fixed-point combinator and have a normal form.

1.6 Typed Self-Interpretation

Most of the literature on self-interpretation, [1, 6] included, concerns untyped systems. Typed self-interpretation is self-interpretation for statically-typed languages. This concept was first discussed by Rendel et al. [7], who present a self-recogniser for a typed lambda calculus. Additionally, they identify several desirable properties of a system exhibiting typed self-interpretation. For example, *representation* ensures the process of quotation respects typing

and furthermore that the quotation of distinct types have distinct types. *Adequacy* specifies that every term of a quotation type is, in fact, a quotation.

Rendel et al. leave open the problem of specifying a self-enactor for a statically-typed language.

1.7 Factorisation Calculi and Self-Interpretation

The factorisation combinator of SF-calculus (section 1.3) models the fundamental behaviour of pattern matching. This behaviour is exploited by Jay and Palsberg with application to typed self-interpretation [4]. They present an extension to SF-calculus, called the *blocking factorisation calculus*, which admits a self-recogniser `unquote` and self-enactor `enact`. These terms are typeable with a System **F**-like type system, an extension to that considered for the factorisation calculi (section 1.4). Of course, such a type system does not yield decidable type assignment.

We are concerned with modifying the blocking factorisation calculus to support a type system with decidable type assignment. As such, we present an overview of the construction of the self-recogniser and self-enactor of Jay and Palsberg, which we will appeal to many times in the discussion of our work.

Definition 1.7.1. Terms t_1, t_2 in the blocking factorisation calculus are defined by

$$t_1, t_2 ::= S \mid F \mid K \mid B \mid Y \mid E \mid x \mid t_1 t_2$$

The S and F combinators of SF-calculus are joined by the fixed-point combinator Y, a quotation constructor B, an operator equality tester E, the usual K combinator (for typing simplicity) and variables, denoted with lowercase characters.

Definition 1.7.2. The reduction rule for E is

$$EMNs\,t \to \begin{cases} s & \text{if } M = N \text{ are equal operators} \\ t & \text{otherwise, if } M, N \text{ factorable} \end{cases}$$

The fixed-point combinator Y behaves as usual

$$Yt \to t\,(Yt)$$

Similarly, S, F and K follow their usual reduction rules; B is a constructor, and therefore has no reduction behaviour.

Before continuing, we recap the syntactic sugar used in the construction of the self-interpreters.

- the identity operator, I
- λ abstraction, written $\lambda x.t$ for some variable x and term t
- 'let' and 'let rec' bindings, written `let` $x = s$ `in` t and `let rec` $f = t$
- extensions, a pattern matching construct, written $p \to s \mid t$, where p, s are arbitrary terms and t is an abstraction

These constructs are de-sugared in the following way into the blocking factorisation calculus. I abbreviates SKK. `let` $x = s$ `in` t de-sugars to $(\lambda x.t)\, s$, and `let rec` $f = t$ to $Y\,(\lambda f.t)$. Abstractions are rewritten as combinators in the calculus by the standard conversion of lambda terms into combinatory logic:

$$\begin{aligned}
\lambda x.x &= I \\
\lambda x.t &= Kt && \text{if } x \text{ not free in } t \\
\lambda x.tx &= t && \text{if } x \text{ not free in } t \\
\lambda x.\,(ru) &= S\,(\lambda x.r)\,(\lambda x.u) && \text{otherwise}
\end{aligned}$$

Extensions are de-sugared as follows:

$$\begin{aligned}
x \to s \mid r &= \lambda x.s \\
O \to s \mid r &= \lambda x.E\,O\,x\,s\,(r\,x) \\
p\,q \to s \mid r &= \lambda x.F\,x\,(r\,x)\,(\lambda y.\,(p \to (q \to s \mid r'\,y) \mid r')\,y) \\
&\qquad \text{where } r' = \lambda y.\lambda z.r\,(y\,z)
\end{aligned}$$

Extensions model pattern matching. They are applied to terms; if this argument matches the pattern on the left-hand side of the arrow then the result is the (appropriate substitution of) the pattern on the right-hand side. If no match is possible, the term is applied to the right-hand side of the extension constructor ($|$). In the first case above, x is a variable, and will match any term, resulting in s (note, x may be free in s, and so will be substituted accordingly). Operator matching is straightforward using the operator equality combinator E. When matching applications, first the left-hand side, then the right-hand side is checked for match equality, and the appropriate action is taken depending on the outcome.

Example 1.7.3. Let t be a matchable form. Then

$$(Bx \to x \mid r)\,t \to \begin{cases} t' & \text{if } t \text{ is of the form } Bt' \text{ for some } t' \\ rt & \text{otherwise} \end{cases}$$

This pattern matching construct is very powerful, and results directly from the expressiveness of F. Using this, the self-recogniser and self-enactor are constructed with relative simplicity.

Quotation in the language is achieved by the following function:

$$\begin{aligned}
\mathit{quote}\,(x) &= x \\
\mathit{quote}\,(O) &= BO \\
\mathit{quote}\,(MN) &= \mathit{quote}\,(M)\;\mathit{quote}\,(N)
\end{aligned}$$

This method cannot be expressed as a term in the language using extensions, as it maps an arbitrary term (not just a matchable form) to a quoted term. Note, however, that quoted terms are always in normal form, and therefore are matchable [4]. This justifies the following use of extensions in specifying a self-recogniser.

$$
\begin{aligned}
&\texttt{let rec unquote} = \\
&\ B\,x \to x \\
&\mid x\,y \to (\texttt{unquote}\,x)\,(\texttt{unquote}\,y) \\
&\mid x \to x
\end{aligned}
$$

We omit the specification of the self-enactor here; our enactor (section 3.4) is a small modification to that of Jay and Palsberg.

The blocking factorisation calculus uses System **F** types, extended from their use in typing the original presentation of the factorisation calculus (section 1.4). Y, K are typed as usual; B has principal type scheme $A \to A$, the identity type. In particular, the type of a term is the same as its quotation. This means that this system does not satisfy the adequacy property of Rendel et al. (section 1.6), because any unquoted term has a quotation type. One suggested refinement would be to introduce an 'Expr' type constructor in order to distinguish the quotation types; the type scheme for B would then be $A \to \mathrm{Expr}\,A$ [4].

The combinator E, the operator equality tester, does not have a principal System **F** type. Jay and Palsberg present a family of principal types for E, one for each operator $O \neq E$. Certainly, EE is not typeable. This limitation is not too restrictive; the self-enactor is constructed so as not to explicitly mention testing for E, and is therefore typeable.

Finally, `unquote` and `enact` are themselves typeable, and, as would be expected, have identity types [4].

Chapter 2

Structural Types for the Factorisation Calculus

2.1 The Structural Type System

Our aim is to develop a type system for *SF*-calculus with decidable type inference. The System **F**-based type system originally presented (section 1.4) does not have this property. This is a consequence of the undecidability of inference in unrestricted System **F**, and the need for the principal type for F to have a nested quantifier, making any known restriction to recover decidability inapplicable. Intuitively, our approach is to extend the Curry type system (section 1.2) to be able to capture the behaviour of the factorisation combinator. Such an extension must be sufficiently minimal to preserve type inference decidability.

The requirement for the System **F** types for F to quantify the second argument is fundamentally because the type of an application does not determine the type of its factors. The quantification then serves to cover all possible types for the argument (right-hand term) of the compound. We eliminate this requirement by developing a type system which retains the types of the factors explicitly in the type for an application. We do this by extending Curry types with structural modifiers.

A structural modifier to a type indicates that a term of this type reduces to an application, the argument of which has type specified by the structural modifier. This is denoted in the following way. If A, B are types, then $[A]B$ is a type, where $[A]$ is the structural modifier. If a term t has type $[A]B$, then t is an application $t_1 t_2$, t_2 has type A, and t_1 has type $A \to B$ (following the $(\to E)$ inference rule of Curry types). t is treated as though it has type B, but with additional information. Note the type specified by the structural modifier is sufficient to infer the types for both factors of the application.

Of course, structural modifiers may be compounded to form structural sequences, representing a term constructed using multiple applications. Arguments to arrow types may therefore have types prepended with arbitrary structural sequences. The arrow type does not care how its argument is constructed, only that it has the correct functional behaviour, so we need to be able to infer the type of an application independent of the structural modifiers of the argument. This is achieved using structural variables, denoted Ψ, which are able, through substitution, to represent any structural sequence, potentially including other structural variables.

Definition 2.1.1. Types, ranged over by A, B, and structural sequences, ranged over by Θ,

are defined by the following grammar

$$\begin{aligned} A, B &::= \Theta\tau \\ \tau &::= \phi \mid (A \to B) \\ \Theta &::= \epsilon \mid [A]\Theta \mid \Psi \end{aligned}$$

Let $\mathcal{T}$ denote the set of all types, and $\mathcal{T}_\theta$ the set of all structural sequences. Let $\mathcal{T}_\tau$ be the set of terms in the grammar ranged over by τ above.

ϵ denotes the empty structural sequence. As usual, the arrow constructor is right-associative, so we omit rightmost, outermost parentheses, where possible. By definition, modifiers are read left-to-right, so informally a term t of type $[B][A]C$ behaves as an application $t_1 t_2 t_3$, where t_1, t_2, t_3 have types $A \to B \to C, A, B$ respectively.

We have restricted the form of structural sequences such that they may contain at most one structural variable, at the end of the sequence. In brief, this restriction is necessary to remove ambiguity in type inference, and is explained in more detail in section 2.3.

We require a notion of type substitution, both in order to define the inference rules for our type system, as well as to reason about unification in the context of type inference. A type variable substitution is the natural extension of a type variable substitution in the Curry type system to our grammar, ensuring that the substitution acts recursively on any structural modifiers. A structural variable substitution is the mechanism through which structural variables can represent arbitrary structural sequences.

Definition 2.1.2. Let ϕ be a type variable, $A \in \mathcal{T}_\tau$. A type variable substitution $\mathcal{S} = (\phi \mapsto A) : (\mathcal{T} \to \mathcal{T}) \cup (\mathcal{T}_\theta \to \mathcal{T}_\theta)$ is a map satisfying

$$\begin{aligned} \mathcal{S}(\Theta\tau) &= (\mathcal{S}\Theta)(\mathcal{S}\tau) \\ \mathcal{S}\phi' &= \begin{cases} A & \text{if } \phi' = \phi \\ \phi' & \text{otherwise} \end{cases} \\ \mathcal{S}(B \to C) &= (\mathcal{S}B) \to (\mathcal{S}C) \\ \mathcal{S}\epsilon &= \epsilon \\ \mathcal{S}([B]\Theta) &= [\mathcal{S}B](\mathcal{S}\Theta) \\ \mathcal{S}\Psi &= \Psi \end{aligned}$$

Let Ψ be a structural variable, $\Theta \in \mathcal{T}_\theta$. A structural variable substitution $\mathcal{S} = (\Psi \mapsto \Theta) : (\mathcal{T} \to \mathcal{T}) \cup (\mathcal{T}_\theta \to \mathcal{T}_\theta)$ is a map satisfying

$$\begin{aligned} \mathcal{S}(\Theta\tau) &= (\mathcal{S}\Theta)(\mathcal{S}\tau) \\ \mathcal{S}\phi &= \phi \\ \mathcal{S}(B \to C) &= (\mathcal{S}B) \to (\mathcal{S}C) \\ \mathcal{S}\epsilon &= \epsilon \\ \mathcal{S}([B]\Theta') &= [\mathcal{S}B](\mathcal{S}\Theta') \\ \mathcal{S}\Psi' &= \begin{cases} \Theta & \text{if } \Psi' = \Psi \\ \Psi' & \text{otherwise} \end{cases} \end{aligned}$$

A substitution is the composition of type and structural variable substitutions. $B \in \mathcal{T}$ is a substitution instance of $A \in \mathcal{T}$ if there exists a substitution $\mathcal{S}$ such that $B = \mathcal{S}A$.

Note that substitutions cannot alter the form of a type, beyond structural variable substitution. For example, any substitution instance of a type with a leading structural modifier will have a leading structural modifier. More generally, substitutions respect the grammar of types and of structural sequences; they are homomorphisms.

We will always omit the empty structural sequence (ϵ) when presenting types. It is important for the reader not to become confused by this. For example, the types ϕ_1 and $[\phi_2]\phi_3$ are not unifiable as there is no substitution mapping the empty structural sequence (prepending ϕ_1) to a structural modifier. In this type system, the most general type is $\Psi\phi$, which though substitution can be mapped to any type. We will often denote this most general type with Γ. The most general function type is $\Psi_3(\Psi_1\phi_1 \to \Psi_2\phi_2) = \Psi_3(\Gamma_1 \to \Gamma_2)$.

We now introduce the inference rules for the type system. As usual, the inference rules typing the combinators S and F will assign to these terms some substitution instance of a principal type. The rule is less clear in the case of typing an application, however. It seems natural for this rule to be the way in which structural modifiers are introduced into application types. One such possibility is

$$(\to E)_{\text{attempt}}: \quad \frac{\vdash M : \Theta(A \to B) \quad \vdash N : A}{\vdash MN : [A]\Theta B}$$

In this way structural sequences are constructed through repeated application of $(\to E)_{\text{attempt}}$. This inference rule does not yield a type system with certain 'nice' properties. Specifically, it does not satisfy the subject reduction property (section 2.2). This is illustrated in the following example.

Example 2.1.3. Consider the combinator K (an abbreviation of FF). In this scheme we would expect $\vdash K : \phi_1 \to \phi_2 \to \phi_1$. Consider terms t_1, t_2 with $\vdash t_1 : \phi_1$, $\vdash t_2 : \phi_2$. Then by (repeated) application of $(\to E)_{\text{attempt}}$, $\vdash Kt_1t_2 : [\phi_2][\phi_1]\phi_1$. However, $Kt_1t_2 \to t_1$, so, typing cannot respect reduction, since to do so would require $\vdash t_1 : [\phi_2][\phi_1]\phi_1$. This does not necessarily hold.

Our approach, therefore, is to embed all structural information in the types of the combinators themselves. The inference rule for application is then very similar to that of Curry's system. It is illustrative to consider the principal type for a less complicated combinator, before presenting the cases for S and F.

Example 2.1.4. Consider K again. In the Curry type system, $\vdash K : \phi_1 \to \phi_2 \to \phi_1$. Let $\vdash x : \Psi_1\phi_1$. Using structural types, we require the type for Kx to encode the type for x. Therefore $\vdash K : [\Psi_1\phi_1](\Psi_2\phi_2 \to \Psi_1\phi_1)$. Notice, if $\vdash y : \Psi_2\phi_2$, $\vdash Kxy : \Psi_1\phi_1$ (assuming an inference rule for application similar to that of Curry's system). So K would have principal type $\Gamma_1 \to [\Gamma_1](\Gamma_2 \to \Gamma_1)$ ($\Gamma_i = \Psi_i\phi_i$), which respects the combinator's reduction behaviour.

Using this approach we can straightforwardly obtain the principal type for S. However, given the two different behaviours of F, conditional on the structure of the first argument, the implication is that there is no single principal type. This is because the additional structural information required by the type when the first argument is a compound leads to an overspecification of the type when the first argument is atomic. We therefore present two principal types for F, representing the two reduction rules.

Definition 2.1.5. Define the principle types of the combinators to be

$$\begin{aligned} S : \mathbb{T}_S &= \mathbb{T}_X \to [\mathbb{T}_X](\mathbb{T}_Y \to [\mathbb{T}_Y][\mathbb{T}_X](\Gamma_1 \to \Gamma_3)), \text{ where} \\ \mathbb{T}_X &= \Psi_4(\Gamma_1 \to \Psi_5(\Gamma_2 \to \Gamma_3)) \\ \mathbb{T}_Y &= \Psi_6(\Gamma_1 \to \Gamma_2) \\ \Gamma_i &= \Psi_i \phi_i \end{aligned}$$

$$\begin{aligned} F : \mathbb{T}_{F_{\text{atom}}} &= (\Gamma_1 \to \Gamma_2) \to [\Gamma_1 \to \Gamma_2](\Gamma_3 \to [\Gamma_3][\Gamma_1 \to \Gamma_2](\Gamma_4 \to \Gamma_3)), \text{ where} \\ \Gamma_i &= \Psi_i \phi_i \end{aligned}$$

$$\begin{aligned} F : \mathbb{T}_{F_{\text{comp}}} &= \mathbb{T}_{PQ} \to [\mathbb{T}_{PQ}](\Gamma_4 \to [\Gamma_4][\mathbb{T}_{PQ}](\mathbb{T}_N \to \Gamma_3)), \text{ where} \\ \mathbb{T}_N &= \Psi_5(\mathbb{T}_P \to [\mathbb{T}_P]\Psi_5(\Gamma_1 \to \Gamma_3)) \\ \mathbb{T}_P &= \Psi_2(\Gamma_1 \to \mathbb{T}_{PQ}) \\ \mathbb{T}_{PQ} &= [\Gamma_1]\Gamma_2 \\ \Gamma_i &= \Psi_i \phi_i \end{aligned}$$

Before further discussion, we present the inference rules for the type system. Because they are not responsible for building structural information into types, their behaviours are largely unchanged from the equivalent Curry type inference rules.

Definition 2.1.6. Let $\mathcal{S}$ denote any substitution. The inference rules for the type system are

$$(S): \quad \frac{}{\vdash S : \mathcal{S}\mathbb{T}_S}$$

$$(F_{\text{atom}}): \quad \frac{}{\vdash F : \mathcal{S}\mathbb{T}_{F_{atom}}}$$

$$(F_{\text{comp}}): \quad \frac{}{\vdash F : \mathcal{S}\mathbb{T}_{F_{comp}}}$$

$$(\to E): \quad \frac{\vdash M : \Theta(A \to B) \quad \vdash N : A}{\vdash MN : B}$$

The two types for F hint at an intersection type system [10], but we do not require the additional expressiveness of such a system in general, so we do not explore the connection further here. The design of the principal types reflects the need to distinguish atomic terms and factorable forms solely by their types. We formalise this notion in several lemmas.

Lemma 2.1.7 (Soundness of Substitution)**.** *Let $\mathcal{S}$ be a substitution. If $\vdash M : A$ then $\vdash M : \mathcal{S}A$.*

Proof. By induction on the structure of a derivation.

1. (S), (F_{atom}), (F_{comp}): Assume A has been derived from one of the inference rules for either S or F. This rule is the only inference rule in the derivation structure. Therefore $A = \mathcal{S}'\mathbb{T}_*$, a substitution instance of some $\mathbb{T}_*$. For any substitution $\mathcal{S}$, $\mathcal{S}A = \mathcal{S}(\mathcal{S}'\mathbb{T}_*) = (\mathcal{S} \circ \mathcal{S}')\mathbb{T}_*$. As the composition of two substitutions is itself a substitution, $M : \mathcal{S}A$ is derivable by the inference rule concerning $\mathbb{T}_*$.

2. $(\to E)$: Assume A has been derived through the $(\to E)$ inference rule, so $M = PQ$, an application. There exists sub-derivations such that $\vdash P : \Theta(B \to A)$, $\vdash Q : B$, for some type B and structural sequence Θ. Consider a substitution $\mathcal{S}$. The claim holds inductively for the sub-derivations, so $\vdash P : \mathcal{S}(\Theta(B \to A))$, $\vdash Q : \mathcal{S}B$. By the definition of substitution, $\vdash P : (\mathcal{S}\Theta)((\mathcal{S}B) \to (\mathcal{S}A))$. Therefore, by $(\to E)$, $\vdash PQ : \mathcal{S}A$.

□

Lemma 2.1.8. *If a compound is typeable then its type has a leading structural modifier. Symbolically, if PQ is a compound and $\vdash PQ : A$ then $A = [B]\Theta\tau$ for some B, Θ, τ.*

Proof. By definition 1.3.3, a compound is the application of a combinator F or S to either a term, or the composition of two terms. By definition 2.1.6, this leading combinator is typed with a substitution instance of $\mathbb{T}_S, \mathbb{T}_{F_{\text{atom}}}$ or $\mathbb{T}_{F_{\text{comp}}}$. The application of $(\to E)$ one or two times to these principle types yields a type with a leading structural modifier, a form which is preserved by substitution (lemma 2.1.7). □

Lemma 2.1.9. *Let t be a matchable form, and suppose Ft is typeable. Then the leading F is typed with a substitution instance of exactly one of the principal types for the factorisation combinator. So $\mathbb{T}_{F_{atom}}$, $\mathbb{T}_{F_{comp}}$ are in some sense mutually exclusive.*

Proof. As Ft is typeable, F must be typed with at least one of the inference rules $(\mathbb{T}_{F_{\text{atom}}})$, $(\mathbb{T}_{F_{\text{comp}}})$. We need to show that it cannot be both. The argument to F is either an operator or a compound. In the latter case, by lemma 2.1.8, this first argument has a leading structural modifier. In the former case, through inspecting the principal types for S and F, and knowing that substitution instances of these types cannot introduce leading structural modifiers, the first argument does not have a leading structural modifier. Each principal type for F matches exactly one of these cases, so the corresponding inference rules apply with mutual exclusivity. □

We end this section by presenting a straightforward result on the expressiveness of the structural type system compared with the Curry type system.

Definition 2.1.10. The *functional form* of a structural type is the underlying Curry type obtained by removing all structural sequences that occur in the type.

Example 2.1.11. Recall the combinator K is an abbreviation for FS. By application of $(\to E)$ to the principal types for F and S, clearly

$$\vdash FS : [\mathbb{T}_S]\,(\Gamma_3 \to [\Gamma_3]\,[\mathbb{T}_S]\,(\Gamma_4 \to \Gamma_3))$$

This has functional form equivalent to the principal Curry type for K in SK-calculus, $\phi_1 \to \phi_2 \to \phi_1$.

Remark 2.1.12. Our notion of type variable substitution (definition 2.1.2) is consistent with the notion of substitution of Curry types. Technically, the processes of converting between SF- and SK-terms, and applying type variable substitutions in either of these type systems, commute.

In general, we may consider a typeable SK-term and ask whether it is typeable with structural types (after K is de-sugared to FS). The following lemma says such a term is always typeable with the same functional form as the Curry type.

Lemma 2.1.13. *Let t be an SK-term, and t' the corresponding SF-term. Suppose $\vdash t : A$ in the Curry type system, so A is a Curry type. Then there exists a structural type A' with functional form A such that $\vdash t' : A'$ in the structural type system.*

Proof. By induction on the structure of an SK-term.

1. S: Let $\vdash S : A$, so A is some substitution instance of the principal (Curry) type for S. This substitution induces a type variable substitution of structural types, say, $\mathcal{S}$. Now, the corresponding SF-term is simply S, and $\vdash S : \mathbb{T}_S$. Note $\mathbb{T}_S$ has functional form equal to the principal Curry type for S. By the inference rule for S, $\vdash S : \mathcal{S}\mathbb{T}_S$, and, as substitution preserves functional form, $\mathcal{S}\mathbb{T}_S$ has the same functional form as A.

2. K: By example 2.1.11, the translation of K into SF-calculus has the same functional form as the principal Curry type for K. This is preserved through substitution, as argued in the case for S.

3. t_1t_2: Assume inductively the result holds for SK-terms t_1 and t_2. Suppose $\vdash t_1t_2 : A$. Then, by the (Curry) inference rule $(\to E)$, there exists a type B such that $\vdash t_1 : B \to A$, $\vdash t_2 : B$. By induction, there exists structural types such that $\vdash t_1' : \Theta(B' \to A')$, $\vdash t_2' : B'$ in the structural type system. Furthermore $\Theta(B' \to A')$ and B' have functional form equal to their Curry equivalents. By the $(\to E)$ inference rule, $\vdash t_1't_2' : A'$, which must have functional form equivalent to A.

□

2.2 Subject Reduction

Our first result for the structural type system is that of subject reduction. A subject reduction property is a crucial property of a type system. Informally, the property states that reduction respects typing; equivalently, the result of running a program has the same type as the program itself. Our proof is similar in structure to the corresponding result for the Curry type system.

Theorem 2.2.1. *Suppose $t_1 \to t_2$ and $\vdash t_1 : A$. Then $\vdash t_2 : A$.*

Proof. By induction on the definition of reduction (definition 1.3.4).

There are three base cases, corresponding to the three rewrite rules. Let O, M, N, X, Y, Z be arbitrary terms.

1. $(SXYZ \to XZ(YZ))$: Suppose $\vdash SXYZ : A$. Because we have typed an application we must have deduced the type A using $(\to E)$, so $\vdash SXY : \Theta_1(B \to A)$ and $\vdash Z : B$ for some type B and structural sequence Θ_1. Similarly $\vdash SXY$ is an application, so again by $(\to E)$ we have $\vdash SX : \Theta_2(C \to \Theta_1(B \to A))$ and $\vdash Y : C$. By the same reasoning $\vdash S : \Theta_3(D \to \Theta_2(C \to \Theta_1(B \to A)))$ and $\vdash X : D$. Now the inference rule (S) dictates that the type for S must be a substitution instance of the principle type for S, $\mathbb{T}_S$, so there exists a type E and structural sequences $\Theta_4, \Theta_5, \Theta_6$ such that

$$D = \Theta_4(B \to \Theta_5(E \to A))$$
$$C = \Theta_5(B \to E)$$

Necessarily $\Theta_1, \Theta_2, \Theta_3$ are fixed; $\Theta_1 = [C][D], \Theta_2 = [D], \Theta_3 = \epsilon$.

Using the types for X, Y and Z derived above we can type the contractum with A as follows

$$(\to E)\ \frac{(\to E)\ \dfrac{\vdash X : \Theta_4(B \to \Theta_5(E \to A)) \qquad \vdash Z : B}{\vdash XZ : \Theta_5(E \to A)} \qquad \dfrac{\vdash Y : \Theta_5(B \to E) \qquad \vdash Z : B}{\vdash YZ : E}\ (\to E)}{\vdash XZ(YZ) : A}$$

2. $(FOMN \to M)$: Suppose $\vdash FOMN : A$, with O atomic. Through repeated, backward reasoning of $(\to E)$ we obtain $\vdash N : B, \vdash M : C, \vdash O : D$ and $\vdash F : \Theta_3(D \to \Theta_2(C \to \Theta_1(B \to A)))$ for some types B, C, D and structural sequences $\Theta_1, \Theta_2, \Theta_3$. O is atomic, so is exactly either S or F. Therefore, by definition 2.1.6, D must be a substitution instance of one of $\mathbb{T}_S, \mathbb{T}_{F_{\text{atom}}}$ or $\mathbb{T}_{F_{\text{comp}}}$. Any substitution instance of these principle types must be of the form $T_1 \to T_2$, without any leading structural modifier. Therefore we deduce that the leading combinator F must have a type derived from $\mathbb{T}_{F_{\text{atom}}}$, because the type of the first argument in $\mathbb{T}_{F_{\text{atom}}}$ matches the form $T_1 \to T_2$. By lemma 2.1.9, this F cannot have a type derived from $\mathbb{T}_{F_{\text{comp}}}$. Crucially, $\mathbb{T}_{F_{\text{atom}}}$ forces C to be equal to the resultant type A.

   $FOMN$ reduces to M, therefore the contractum is typeable with A.

3. $(F(PQ)MN \to NPQ)$: Suppose $\vdash F(PQ)MN : A$, where PQ is a compound. As in the atomic case we deduce $\vdash N : B, \vdash M : C, \vdash PQ : D$ and $\vdash F : \Theta_3(D \to \Theta_2(C \to \Theta_1(B \to A)))$ for some types B, C, D and structural sequences $\Theta_1, \Theta_2, \Theta_3$. The term PQ is a compound, so, by lemma 2.1.8, D has a leading structural modifier.

   The leading F can therefore be typed with $\mathbb{T}_{F_{\text{comp}}}$, which requires its first argument to have a leading structural modifier. Hence this combinator cannot be typed with $\mathbb{T}_{F_{\text{atom}}}$ by lemma 2.1.9. Consequently, by the inference rule $(\mathbb{T}_{F_{\text{comp}}})$,

$$\begin{aligned} D &= [E]\Theta_4 G \\ H &= \Theta_4(E \to D) \\ B &= \Theta_5(H \to [H]\Theta_5(E \to A)) \end{aligned}$$

   Also, $\vdash Q : E$, as the application PQ is typeable.

   The contractum NPQ is typeable with A as follows

$$(\to E)\ \frac{(\to E)\ \dfrac{\vdash N : \Theta_5(H \to [H]\Theta_5(E \to A)) \qquad \vdash P : H}{\vdash NP : [H]\Theta_5(E \to A)} \qquad \vdash Q : E}{\vdash NPQ : A}\ (\to E)$$

4. Inductively assume the claim holds for terms M and N, such that if $M \to N$ and $\vdash M : A$, then $\vdash N : A$. Let P be an arbitrary term and consider the two cases of the inductive definition 1.3.4

   (a) $(M \to N \implies PM \to PN)$: Assume $\vdash PM : B$. Then, by $(\to E)$, $\vdash P : \Theta(A \to B)$ for some A, Θ. By the induction hypothesis $\vdash N : A$, so, by $(\to E)$, $\vdash PN : B$.

(b) $(M \to N \implies MP \to NP)$: Assume $\vdash MP : B$. Then, by $(\to E)$, A is really $\Theta(C \to B)$ for some C, Θ, and $\vdash P : C$. MP reduces to NP, and by the induction hypothesis $\vdash N : \Theta(C \to B)$, so, by $(\to E)$, $\vdash NP : B$.

□

Corollary 2.2.2. *Reduction* $(\to^*)$ *preserves typing.*

With this fundamental result we are ready to introduce the type inference algorithm for the type system.

2.3 Type Inference

In this section we present the type inference algorithm for the structural type system, and prove soundness and completeness properties for it. This exactly shows that type assignment is decidable. This is perhaps unsurprising considering that our type system is based on Curry types, a very restrictive type system.

Type inference for Curry types yields the principal type for a term; the principal type is the most general type with which the term can be typed, and all other typing possibilities can be obtained from this through substitution. By virtue of there being two principal types for F, type inference in our type system will yield a finite set of principal types (bounded in size by the number of occurrences of the factorisation combinator in the term), the elements of which are non-unifiable principal types. The inference algorithm hinges on an algorithm to unify types, which we present also.

Type unification is a process which attempts to make equal two types through substitution. If unification is successful, this substitution is returned. Unification for structural types is the natural extension of unification for Curry types (originating from Robinson's work [8]) to structural types. In this instance, we require two algorithms, one to unify structural sequences and one to unify types themselves. Reflecting the grammar of types, these algorithms are mutually-recursive.

Before stating the algorithms it is useful to explain how their development dictated the grammar of structural sequences (definition 2.1.1). We require that structural variables occur at most once in a sequence, at the end. This limitation simplifies unification. Consider unifying an unrestricted structural sequence of two structural variables $\Psi_1\Psi_2$, with the structural sequence $[A][B]$ (A, B arbitrary types). Unification is possible; there are three unification possibilities:

$$\Psi_1 \mapsto \epsilon, \Psi_2 \mapsto [A][B]$$
$$\Psi_1 \mapsto [A], \Psi_2 \mapsto [B]$$
$$\Psi_1 \mapsto [A][B], \Psi_2 \mapsto \epsilon$$

There is no 'correct' choice here; any of the possibilities may cause unification failure at a later stage of the algorithm. Of course, all possibilities could be tested, but this is computationally expensive. It is sufficient to restrict the form of structural sequences as we have done, which eliminates this problem, while retaining typeability of S and F.

Definition 2.3.1 (Structural Type Unification)**.** Unification of types is achieved through the following algorithms. These algorithms yield substitutions. Here, a type variable ϕ (equivalently, a structural variable Ψ) is 'contained in' a type A, denoted $\phi \in A$, if it occurs anywhere in the structure of A.

$$
\begin{array}{l}
\text{unify } (\Theta_1 A')\ (\Theta_2 B') \\
\quad = (\text{unify } SA'\ SB') \circ S \\
\quad \text{where} \\
\qquad S = \text{unifySeq } \Theta_1\ \Theta_2 \\
\text{unify } \phi\ \phi \\
\quad = \text{Id} \\
\text{unify } \phi\ B' \\
\quad = (\phi \mapsto B') \qquad \text{if } \phi \notin B' \\
\text{unify } (A_1 \to A_2)\ (B_1 \to B_2) \\
\quad = (\text{unify } SA_2\ SB_2) \circ S \\
\quad \text{where} \\
\qquad S = \text{unify } A_1\ B_1 \\
\text{unify } A\ B \\
\quad = \text{unify } B\ A
\end{array}
\qquad
\begin{array}{l}
\text{unifySeq } \epsilon\ \epsilon \\
\quad = \text{Id} \\
\text{unifySeq } [A]\Theta_1\ [B]\Theta_2 \\
\quad = (\text{unifySeq } S\Theta_1\ S\Theta_2) \circ S \\
\quad \text{where} \\
\qquad S = \text{unify } A\ B \\
\text{unifySeq } \Psi_1\ \Psi_1 \\
\quad = \text{Id} \\
\text{unifySeq } \Psi\ \Theta \\
\quad = (\Psi \mapsto \Theta) \qquad \text{if } \Psi \notin \Theta \\
\text{unifySeq } \Theta_1\ \Theta_2 \\
\quad = \text{unifySeq } \Theta_2\ \Theta_1
\end{array}
$$

Failure in the unification algorithm is implicit. For example, unification will not succeed between a structural modifier $[A]$ and the empty structural sequence ϵ in *unifySeq*, or between ϕ and $\phi \to \phi$ in *unify*.

Unification has two key properties detailed in the following lemmas.

Lemma 2.3.2. *If* $S = \text{unify}\, A\, B$ *is the successful unification of types* A *and* B*, then* $SA = SB$*.*

Proof. By straightforward induction on the structures of A and B. □

Lemma 2.3.3. $S = \text{unify}\, A\, B$ *is the most general unifier of* A *and* B*. Equivalently, for any substitution* S_1 *such that* $S_1 A = S_1 B$*, there exists a substitution* S_2 *such that* $S_1 = S_2 \circ S$*.*

We do not present a proof of this here. We argue that this holds for reasons similar to the equivalent result for Curry types (see, for example, [8]).

We are now in a position to present the principal types algorithm. As stated before, this returns a set of principal types. The size of this set depends on the number of successful unifications when typing applications, although it is bounded by 2^n, where n is the number of occurrences of F in the term.

Definition 2.3.4 (Principal Types Algorithm)**.**

$$
\begin{aligned}
&\mathrm{PT} : \mathrm{Term} \to \mathcal{P}(\mathcal{T}) \\
&\mathrm{PT}(S) \\
&\quad = \mathrm{fresh}\,\{\mathbb{T}_S\} \\
&\mathrm{PT}(F) \\
&\quad = \mathrm{fresh}\left\{\mathbb{T}_{F_{\mathrm{atom}}},\ \mathbb{T}_{F_{\mathrm{comp}}}\right\} \\
&\mathrm{PT}(MN) \\
&\quad = \{S\Gamma_1 \mid (A,B) \in S_1 \times S_2, S = \mathrm{unify}\ A\ \Psi_2(B \to \Gamma_1), \mathrm{fresh}\,\Psi_2, \mathrm{fresh}\,\Gamma_1, \Gamma_1 = \Psi_1\phi_1\} \\
&\quad \text{where} \\
&\qquad S_1 = \mathrm{PT}(M) \\
&\qquad S_2 = \mathrm{PT}(N)
\end{aligned}
$$

Type inference is decidable if this algorithm is both sound and complete. Soundness ensures that the resulting types of the algorithm correctly type the term; completeness that the algorithm will return the most general type of a typeable term.

Theorem 2.3.5 (Soundness of Type Inference)**.** *If $T = \mathrm{PT}\ M$ is the set of principal types for a term M, then $\forall A \in T \vdash M : A$.*

Proof. By induction on the structure of the term M.

1. S: $\mathrm{PT}\ S = \{\mathbb{T}_S\}$. Via the (S) inference rule, $\vdash S : \mathbb{T}_S$.

2. F: $\mathrm{PT}\ F = \left\{\mathbb{T}_{F_{\mathrm{atom}}},\ \mathbb{T}_{F_{\mathrm{comp}}}\right\}$. Through application of the (F_{atom}) and (F_{comp}) inference rules F is certainly typeable with $\mathbb{T}_{F_{\mathrm{atom}}}$ and $\mathbb{T}_{F_{\mathrm{comp}}}$.

3. PQ: Assume inductively the proposition holds for terms P and Q. Let $T = PT\ (PQ)$ and $A \in T$. We require $\vdash PQ : A$. As $A \in T$ there exists types $B \in \mathrm{PT}\ P$, $C \in \mathrm{PT}\ Q$ and $\Gamma_1 = \Psi_1\phi_1$ such that A is some substitution instance of Γ_1. This substitution, S, is the (successful) unification of B and $\Psi_2(C \to \Gamma_1)$ (for some Ψ_2). So $A = S\Gamma_1$. Additionally, $SB = S(\Psi_2(C \to \Gamma_1)) = (S\Psi_2)(SC \to S\Gamma_1)$, by lemma 2.3.2.

   Inductively, $\vdash P : B$, and therefore $\vdash P : SB$ (by 2.1.7). Hence $\vdash P : (S\Psi_2)(SC \to S\Gamma_1)$. Similarly, $\vdash Q : C$ and $\vdash Q : SC$. Finally, by the $(\to E)$ inference rule, $\vdash PQ : S\Gamma_1$.

□

Theorem 2.3.6 (Completeness of Type Inference)**.** *If $\vdash M : A$ then there exists $B \in PTM$ and substitution S such that $A = SB$.*

Proof. By induction on the structure of the term M.

1. S: If $\vdash S : A$, then the corresponding derivation tree must consist of a single instance of the (S) rule. Hence A is a substitution instance of the principal type for S, $\mathbb{T}_S$. By definition $\mathbb{T}_S \in \mathrm{PT}\,S$, so A is a substitution instance of a member of the set of principal types for S.

2. F: If $\vdash F : A$, then A must be derived from exactly one of (F_{atom}) and (F_{comp}). Therefore F is a substitution instance of either $\mathbb{T}_{F_{\text{atom}}}$ or $\mathbb{T}_{F_{\text{comp}}}$. These are the principal types for F and are exactly the types returned by $\text{PT}\, F$, so in all cases A is a substitution instance of some member of $\text{PT}\, F$.

3. PQ: Assume inductively terms P and Q satisfy the proposition. If $\vdash PQ : A$, then A must be derived using $(\to E)$. Therefore $\vdash P : \Theta(C \to A)$ and $\vdash Q : C$ for some type C and structural sequence Θ. Furthermore, by induction, there exists types D, E and substitutions S_1, S_2 such that $D \in \text{PT}\, P$, $S_1 D = \Theta(C \to A)$, $E \in \text{PT}\, Q$ and $S_2 E = C$.

   Consider the result of unifying D with $\Psi_2(E \to \Gamma_1)$ $(\Psi_2, \Gamma_1 = \Psi_1\phi_1$ fresh). This result must be successful; these types are certainly unifiable with the substitution $S = (\Psi_2 \mapsto \Theta) \circ (\Gamma_1 \mapsto A) \circ S_2 \circ S_1$. Therefore, $T = \text{unify}\ D\ \Psi_2(E \to \Gamma_1)$ is well defined. So $T\phi \in \text{PT}\, PQ$. Furthermore, as T is the most general such unifier, $S = S' \circ T$ for some substitution S'. Hence

$$\begin{aligned} \Theta(C \to A) &= SD \\ &= (S' \circ T)D \\ &= ((S' \circ T)\Psi_2)((S' \circ T)E \to (S' \circ T)\Gamma_1) \end{aligned}$$

   Consequently, $A = (S' \circ T)\Gamma_1$. As $T\Gamma_1 \in \text{PT}\, PQ$, A is the substitution instance of an element of the set of principal types.

□

Chapter 3

Self-Interpretation for a Calculus with Decidable Type Assignment

In this chapter we build on the work of Jay and Palsberg concerning typed self-interpretation with factorisation calculi (section 1.7). Their calculus, the blocking factorisation calculus, extends SF-calculus with a quote constructor B, a fixed-point combinator Y and the operator equality test E. This setting is sufficient to develop a self-recogniser `unquote` and self-enactor `enact` for quoted terms. Furthermore, the System F-like type system does not assign a universal type to quotations; quoting a term preserves its type. In this way `unquote` and `enact` are typed self-interpreters. Of course, the polymorphic type system does not have the property of decidable type assignment.

Our contribution lies in modifying the blocking factorisation calculus in order to eliminate the need for the combinator E. By doing this, we are able to extend the structural type systems from SF-calculus to this setting, preserving decidability of typeability. Slight modification to the construction of `unquote` and `enact` yield self-interpreters for a typed calculus with decidable type assignment.

We begin by introducing our modification to the blocking factorisation calculus. Note that we use the syntactic sugar of Jay and Palsberg freely here (section 1.7).

3.1 Reduced Blocking Factorisation Calculus

Our modification to the calculus of Jay and Palsberg, the blocking factorisation calculus, consists of eliminating the combinators E and K and removing variables from the language. Eliminating K does not reduce the expressiveness of the calculus, as K is behaviourly equivalent to FF or FS (section 1.3); it reduces the complexity of the type system at the expense of larger terms for the recogniser and enactor. Similarly, variables are unnecessary to construct the interpreters, so we remove them from the calculus for simplification. Note that our syntactic sugar (section 1.7) uses variables; these variables are not present after de-sugaring (since all terms are closed).

We delay the discussion of why we need to eliminate the operator equality tester E from the language until we introduce our structural type system in this context (section 3.3). Suffice to say, this is crucial to preserving the property of type assignment decidability in the type system. With this omission we present the definition of the modified calculus, which we refer to as the *reduced blocking factorisation calculus* or *$SFBY$-calculus*.

Definition 3.1.1. An operator O is given by

$$O ::= S \mid F \mid Y \mid B$$

S, F are the combinators of SF-calculus. Y is a fixed-point combinator, and B is the block or quote constructor.

Definition 3.1.2. Terms t_1, t_2 are defined by

$$t_1, t_2 ::= O \mid t_1 t_2$$

Our notions of matchable forms (definition 1.3.2) and operators and compounds (definition 1.3.3) carry over from SF-calculus.

Definition 3.1.3. The reduction rules for the language are those of the SF-calculus (definition 1.3.4), extended with the reduction rule for the fixed-point combinator

$$Yt \to t\,(Yt)$$

In particular, B has no reduction rule; it is a *constructor*.

K abbreviates to FS; I to $SKK = S(FS)(FS)$.

3.2 Structural Types for $SFBY$-Calculus

We extend our structural type system for the factorisation calculus to the blocking factorisation calculus. The grammar of types is unchanged (definition 2.1.1), as is our notion of substitution.

Definition 3.2.1. The principal types for the operators are as before (definition 2.1.5), extended with

$$Y : \mathbb{T}_Y = \Psi_2(\Gamma_1 \to \Gamma_1) \to \Gamma_1$$
$$B : \mathbb{T}_B = \Gamma_1 \to \Gamma_1$$

Definition 3.2.2. The inference rules for the type system extend those of SF-calculus (definition 2.1.6) with the following. Let $\mathcal{S}$ denote any substitution.

$$(Y): \quad \frac{}{\vdash Y : \mathcal{S}\mathbb{T}_Y}$$

$$(B): \quad \frac{}{\vdash B : \mathcal{S}\mathbb{T}_B}$$

In introducing the fixed-point combinator we lose a property of the type system. In SF-calculus, we could distinguish between operators and compounds by inspecting their types; a term is typeable with type of the form $(A \to B)$ (arrow type with no leading structural modifier) only if it is (β-equivalent to) an operator. This property does not hold for terms in $SFBY$-calculus. For example, $\vdash YI : \phi \to \phi$ is a term with an arrow type without a leading structural modifier, and is not equivalent to any operator.

Notice also the type for B is the identity type, the same as that of Jay and Palsberg (section 1.7). We have not introduced an Expr type constructor into the grammar of types in order to distinguish quotations in the type system. As noted in [4, 7], this construction is desirable assuming it satisfies the properties of representation and adequacy (section 1.6). Unfortunately, it is not possible for adequacy to hold using the same construction of quotations as in the blocking factorisation calculus. This is because we would require a term such as $B\,(YI)$ not to be typeable (it is not a quotation), but all terms BO to be typeable. However there is no way to distinguish operators with the type system here. Therefore we omit a distinct family of quotation types from the type system; our notion of quotation preserves the type of the term being quoted, as in the type system of section 1.7.

All of our fundamental results about structural types for SF-calculus extend to this setting. Most importantly, subject reduction carries over here. The proof follows that of the result for SF-calculus (theorem 2.2.1); the result clearly holds for the additional base case of Y (recalling B has no reduction rules).

The principal types algorithm extends naturally, too; the principal types for the new operators are simply fresh instances of their principal types. Soundness and completeness of type inference follow; the proofs are simple extensions of theorems 2.3.5 and 2.3.6 to deal with the new base cases.

3.3 Operator Equality in $SFBY$-calculus

The original formulation of the blocking factorisation calculus requires a combinator E which tests for equality of operators. It is of course necessary to be able to distinguish operators in the calculus for the purpose of interpretation. Indeed, with the factorisation combinator able to decompose compounds (and all quotations are compounds), these two combinators encapsulate the fundamental behaviour of interpretation. E, however, cannot have a principal type (or set of principal types), either with System **F** types [4] or with our structural type system. This is unsurprising; type systems identify behaviour of terms and cannot in general distinguish individual terms. Without a principal type for E type inference for arbitrary terms is not possible.

We have developed a term in $SFBY$-calculus which behaves identically to E. In this way we do not need to introduce it as a combinator and avoid the associated difficulties with typing, and yet are able to construct `unquote` and `enact` of the blocking factorisation calculus. Terms for testing operator equality in simpler calculi, such as SF- and SFB-calculi, are well documented in the original presentation of the factorisation calculus [5]. This forms the basis of our work here.

Definition 3.3.1. Define the terms

$$\begin{aligned}
\texttt{is(B or Y)} &= \lambda x.F(x(FK)IK)(KI)(K(KK)) \\
\texttt{is(B not Y)} &= \lambda x.F(x(KS))(KI)(K(KK)) \\
\texttt{is(F not S)} &= \lambda x.x(KI)(K(KI))K
\end{aligned}$$

`is(B or Y)` is identically '`is(C)`' of SFC-calculus [5], which maps B (the constructor C in this context) to K and S, F to KI. `is(F not S)` is '`is(F)`' of SF-calculus, and maps F to K, S to KI. We leave it to the reader to verify `is(B or Y)` $Y \to K$, `is(B not Y)` $B \to K$ and `is(B not Y)` $Y \to KI$.

These definitions are sufficient to construct terms which distinguish the operators of our calculus; for each operator O, `is(O)` maps O to K and all other operators to KI. In the following we use the `if-then-else` and `not` constructs whose first arguments are the truth values K and KI. $\texttt{if}\,(B)\,\texttt{then}\,(P)\,\texttt{else}\,(Q)$ de-sugars to BPQ and $\texttt{not}\,(B)$ to $\texttt{if}\,(B)\,\texttt{then}\,(KI)\,\texttt{else}\,(K)$.

Definition 3.3.2.

$$\begin{aligned}
\texttt{is(B)} &= \lambda x.\texttt{if}\,(\texttt{is(B or Y)}\,x)\,\texttt{then}\,(\texttt{is(B not Y)}\,x)\,\texttt{else}\,(KI)\\
\texttt{is(Y)} &= \lambda x.\texttt{if}\,(\texttt{is(B or Y)}\,x)\,\texttt{then}\,(\texttt{not}\,(\texttt{is(B not Y)}\,x))\,\texttt{else}\,(KI)\\
\texttt{is(F)} &= \lambda x.\texttt{if}\,(\texttt{is(B or Y)}\,x)\,\texttt{then}\,(KI)\,\texttt{else}\,(\texttt{is(F not S)}\,x)\\
\texttt{is(S)} &= \lambda x.\texttt{if}\,(\texttt{is(B or Y)}\,x)\,\texttt{then}\,(KI)\,\texttt{else}\,(\texttt{not}\,(\texttt{is(F not S)}\,x))
\end{aligned}$$

Finally, we can construct a term `eqatom` which decides equality of operators in $SFBY$-calculus.

Definition 3.3.3.

$$\begin{aligned}
\texttt{eqatom} = \lambda xy.&\texttt{if}\,(\texttt{is(S)}\,x)\,\texttt{then}\,(\texttt{is(S)}\,y)\,\texttt{else}\\
&\texttt{if}\,(\texttt{is(F)}\,x)\,\texttt{then}\,(\texttt{is(F)}\,y)\,\texttt{else}\\
&\texttt{if}\,(\texttt{is(B)}\,x)\,\texttt{then}\,(\texttt{is(B)}\,y)\,\texttt{else}\\
&(\texttt{is(Y)}\,y)
\end{aligned}$$

We can now build a term behaviourly equivalent to the E combinator of the blocking factorisation calculus (definition 1.7.2).

Definition 3.3.4. Define the operator equality term E for our calculus as

$$E = \lambda xyst.Fx\,(Fy\,(\texttt{if}\,(\texttt{eqatom}\;x\;y)\,\texttt{then}\,(s)\,\texttt{else}\,(t))\,(K\,(Kt)))\,(K\,(Kt))$$

The intended behaviour of E is to reduce to s when x and y are equal operators, and to t when x and y are any other matchable forms. The factorisation behaviour of F is used to test whether x and y are operators. If so, `eqatom` is used to test their equality, otherwise, t is returned. This is exactly the behaviour of the E combinator of Jay and Palsberg.

Lemma 3.3.5. *E is behaviourly equivalent to the E operator of the blocking factorisation calculus, i.e.*

$$EMN\,s\,t \to \begin{cases} s & \text{if } M = N \text{ are equal operators} \\ t & \text{otherwise, if } M, N \text{ factorable} \end{cases}$$

3.4 `unquote` and `enact` in *SFBY*-calculus

Given the E constructed in section 3.3, we are able to present a self-recogniser and self-enactor for our calculus using the construction of Jay and Palsberg (section 1.7). E is necessary when de-sugaring extensions (see [4]). The construction is in fact slightly simpler owing to the reduced number of terms and the lack of variables in the reduced blocking factorisation calculus.

Definition 3.4.1. Quotation is achieved by the following function

$$\begin{aligned} \mathit{quote}\,(O) &= BO \\ \mathit{quote}\,(PQ) &= \mathit{quote}\,(P)\ \mathit{quote}\,(Q) \end{aligned}$$

Definition 3.4.2. The self-recogniser `unquote` for the reduced blocking factorisation calculus is given by the term

$$\begin{aligned} &\texttt{let rec unquote} = \\ &\ B\,x \to x \\ &\mid \lambda x.F\,x\,S\,(\lambda yz.\texttt{unquote}\,(y)\,\texttt{unquote}\,(z)) \end{aligned}$$

Recalling the structure of a quotation, the first pattern in the definition of `unquote` extracts an operator from its quoted form. The second half of the pattern, an abstraction, will receive the quotation of an application, which is itself an application. The abstraction simply decomposes this application and computes `unquote` on the factors recursively. The choice of S as the second argument to F is arbitrary as it is never used; the first argument is always a compound so the atomic reduction rule for F is never applicable.

The behaviour of `unquote` is characterised by the following lemma [4]

Lemma 3.4.3. *For all terms t,* $\texttt{unquote}\ (\mathit{quote}\,(t)) \to t$.

The self-enactor is exactly that for the blocking factorisation calculus, with the cases for K and E removed. A detailed discussion of its operation of can be found in [4].

Definition 3.4.4. The self-enactor `enact` for the calculus is given by the term

$$\begin{aligned} &\texttt{let rec enact} = \\ &\quad \texttt{let unblock} = B\,x \to x \mid x \to x\ \texttt{in} \\ &\quad \texttt{let evalop} = x \to \texttt{unblock}\,(\texttt{enact}\ x)\ \texttt{in} \\ &\quad \texttt{let enact1} = \\ &\quad\ B\,Y\,x_1 \to \texttt{enact}\,(x_1\,(BY\,x_1)) \\ &\quad \mid B\,S\,x_1\,x_2\,x_3 \to \texttt{enact}\,(S\,x_1\,x_2\,x_3) \\ &\quad \mid B\,F\,x_1\,x_2\,x_3 \to \texttt{enact}\,(F\,(\texttt{evalop}\ x_1)\,x_2\,x_3) \\ &\quad \mid x_1 \to x_1 \\ &\quad \texttt{in} \\ &\quad\ x_1\,x_2 \to \texttt{enact1}\,(\texttt{enact}\ x_1\,x_2) \\ &\quad \mid x_1 \to x_1 \end{aligned}$$

The behaviour of this enactor is also characterised by Jay and Palsberg [4]

Lemma 3.4.5. *For all terms t, u such that $t \to u$,* $\texttt{enact}\ (\mathit{quote}\,(t)) \to \mathit{quote}\,(u)$.

Hence, the above constitutes a self-recogniser and self-enactor for $SFBY$-calculus. Given that the structural type system for this calculus preserves type assignment decidability (section 3.2), these are self-interpreters for a statically-typed language with decidable type assignment!

3.5 Typing `unquote` and `enact`

The next logical step is to try and type `unquote` and `enact` in the type system. Using System **F** types, Jay and Palsberg could very easily derive additional inference rules to type their syntactic constructs (abstraction, extensions etc.). These types are then preserved by the process of de-sugaring the constructs. The situation is not so straightforward using structural types, however. For example, it is not possible to derive a general typing rule for abstractions. Such a rule would behave as follows; note Γ denotes a typing context for assigning types for variables.

$$(\rightarrow I)_{\text{attempt}}: \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : \Theta(A \rightarrow B)}$$

But what is Θ? Actually, it is not possible to know the form of Θ; it depends on the structure of M, but this cannot be captured by a local typing rule. In some sense there is not enough information to rebuild the structural sequence of the functional type. This is not surprising, but it does mean that there is no way of analytically deriving the types for the sugared forms. In other words, there is no way of reasoning about the types for `unquote` and `enact` in their concise forms; the de-sugared terms, however, are built from hundreds of $SFBY$-combinators and are too large to type by hand.

The obvious alternative, therefore, is to leverage type inference in our type system in order to compute the types of `unquote` and `enact` for us. At the time of writing we have not been able to compute the types for the self-interpreters. We have not even computed the type for E! Our implementation cross-compiles the sugared terms into pure $SFBY$-terms, and then computes the principal types using straightforward implementations of the principal types and unification algorithms. We have found inference (with our implementation) to be computationally-intensive, surprisingly so. This is, in part, due to the growth of the set of principal types for a term, which grows exponentially, in the worst case, in the number of occurrences of F. Even our term E is build from several hundred $SFBY$-combinators, so practically is not yet typeable. Therefore, while this system shows there exists self-interpreters for a statically-typed calculus with decidable type assignment, actually typing terms is computationally expensive, which limits any practical use.

3.6 Implementation

As stated above, as we are unable to derive inference rules for our syntactic constructs, our implementation first cross-compiles a sugared term into a pure $SFBY$-term, and then attempts to type this term using the principal types algorithm. The cross-compiler, or parser, is generated using Happy, a parser generator for Haskell. The type inference implementation, also written in Haskell, represents substitutions as associative arrays, mapping type or structural variables to types in the grammar. Multiple, mutually-recursive methods are then required to apply substitutions to types, as dictated by the grammar of types. It is then straightforward to implement the unification and type inference algorithms of section 2.3.

With further thought, and an investigation into more efficient techniques for type inference implementation, it is likely better performance could be achieved beyond that which the current program exhibits. So far we have only been able to type the '`is(0)`' constructs (definition 3.3.2), whose principal types are as expected. Determining a principal type for `eqatom` (definition 3.3.3) has proved to be too computationally demanding.

Evaluation

Our structural type system is a natural extension to Curry types in order to type the factorisation combinator. Certainly, it is neither as simple in definition nor as powerful in expressiveness as polymorphic type systems; the expressiveness is, informally, just enough to type F. The grammar of types is complicated by the need for deterministic unification of structural sequences, and this complicates the principal types for the combinators, making them unwieldy. However, the theory of the type system is straightforward enough, and has the key property of decidable type assignment we desire.

The practical limitations of the type system become clear when we extend it to the reduced blocking factorisation calculus. By following the approach of Jay and Palsberg it is entirely necessary to eliminate the E combinator, as it has no principal type, either in the structural type system or a polymorphic type systems such as System **F**. We successfully show E's behaviour can be captured with an $SFBY$-term, although the corresponding term is built from several-hundred operators, bloating the de-sugared terms for `unquote` and `enact`. Really, our system shows that constructing self-interpreters for a statically-typed language with decidable type assignment is possible, although the construction is not yet practically useful. It is unclear how to efficiently type the interpreters (or even efficiently type SF-terms).

By using the same construction of quotations as Jay and Palsberg, we are unable to develop a system satisfying the adequacy property of Rendel et al., and as such do not introduce a desirable expression type constructor to distinguish quoted terms. This is a consequence of the construction, and not a consequence of any type system, however.

Conclusions and Future Work

The process of developing the type system, and its application to self-interpretation, has raised several interesting questions beyond the scope of the work here. It would be very insightful to characterise the expressiveness of the structural type system; equivalently, to investigate its corresponding logic. We know that this expressiveness is at least as great as the Curry type system, and strictly less than a polymorphic type system such as System **F**; a precise characterisation, however, would require techniques far different from any used here.

Simply working with the factorisation calculi yields questions not answered in the original presentation. For example, is SF-calculus with System **F** types strongly normalising? Is SF-calculus with the structural type system strongly normalising? We believe both these cases to be almost certainly true.

Certainly, aspects of our approach may be refined upon or explored further. Our restriction to the grammar of structural sequences is necessary to ensure deterministic unification of structural variables, but is it possible to be less restrictive, to have a more general grammar, while still preserving soundness and completeness results for the principal types algorithm? With respect to our work on self-interpretation, is it possible to implement the principal types algorithm for $SFBY$-calculus efficiently? Regardless, our type system does not work well with the syntactic sugar used to construct `unquote` and `enact`; does there exist a type system for the blocking factorisation calculus with decidable type assignment, where type inference is practically simple?

Finally, it is necessary to verify formally the most general unifier result for our type inference algorithm. While we would be surprised if this result didn't hold, it does underpin the completeness result for the type inference algorithm, which is a necessary result for our work on self-interpretation.

Bibliography

[1] A. Berarducci and C. Böhm. A self-interpreter of lambda calculus having a normal form. *Lecture notes in computer science*, 702, 1993.

[2] H.B. Curry and R. Feys. *Combinatory Logic*, volume 1. Amsterdam : North-Holland, 1958.

[3] R. Hindley and J. P. Seldin. *Introduction to Combinators and Lambda-calculus*. Cambridge University Press, 1986.

[4] B. Jay and J. Palsberg. Typed self-interpretation by pattern matching. *Acm Sigplan Notices*, 46, 2011.

[5] Barry Jay and Thomas Given-Wilson. A combinatory account of internal structure. *The Journal Of Symbolic Logic*, 76, 2011.

[6] T. Mogensen. Linear time self-interpretation of the pure lambda calculus. *Perspectives of System Informatics*, 1755, 2000.

[7] T. Rendel, K. Ostermann, and C. Hofer. Typed self-representation. *Acm Sigplan Notices*, 44, 2009.

[8] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12, 1965.

[9] Terese. *Term Rewriting Systems*, volume 53 of *Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[10] S. van. Bakel. Strict intersection types for the lambda calculus. *ACM Computing Surveys (CSUR)*, 43, 2011.

[11] J.B. Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98, 1999.