ħ: A Plank for Higher-order Attribute Contraction Schemes
Extended Abstract

Cynthia Kop
University of Copenhagen

Kristoffer H. Rose
Two Sigma Investments, LP

Abstract

We present and formalize ħ, a core (or "plank") calculus that can serve as the foundation for several compiler specification languages, notably CRSX (Combinatory Reductions Systems with eXtensions) HACS (Higher-order Attribute Contraction Schemes), and TransScript. We discuss how the ħ typing and formation rules introduce the necessary restrictions to ensure that rewriting is well-defined, even in the presence of ħ's powerful extensions for manipulating free variables and environments as first class elements (including in pattern matching).

1 Introduction

Several systems that manipulate programs, so-called *meta-programming* systems, have emerged over the years, ranging from generic specification languages, where the goal is to derive implementations from formal specifications of the program language semantics, all the way to tools that support specific aspects of program execution or compiler generation.

One direction has been to combine *higher order rewriting* [5] with *higher order abstract syntax* (HOAS) [8]. This approach is used by CRSX (Combinatory Reduction Systems with eXtensions) [11], developed for writing industrial compilers at IBM Research [12], and the derived systems HACS (Higher-order Attribute Contration Schemes) [13], developed to teach compiler construction at NYU [9], and TransScript [], developed to support industrial strength fully automatic compiler generation. The implementation of the full CRSX language [10] turned out to be quite complex, and over time we have developed notions of what the "core" elements of such languages should be.

ħ is our attempt at providing such an extended foundation. ħ is specifically based on Aczel's *Contraction Schemes* [1] (where the restriction we shall see of only permitting abstractions as constructor arguments is first found) with the substitution notion from Klop's CRSs, *Combinatory Reduction Systems* [6] and imposing a typing discipline similar to the the second-order limitation of Blanqui's *Inductive Data Type Systems* [2]. The extension of ħ is the mechanism used to add first class *environments* and *free variables* in a formal way, and integrate those with the type discipline and pattern matching; these features are derived from the CRSX system [12], where they were never provided a formal foundation.

1.1 Example (βη)**.** The canonical higher order rewriting example, λ-calculus with β and η rewrite rules over closed terms, is expressed as follows in ħ:

```
L scheme Lam([L]L);
L scheme Ap(L,L);
L rule Ap(Lam([x]#M(x)), #N) → #M(#N);
L rule Lam([x]Ap(#M(), x)) →  #M();
```

The example shows CRS-style higher order matching, where patterns match all occurrences of bound variables, allowing rules to do *substitution* of bound variables. In the first rule, #M(*x*) in the left-hand side matches a term containing the bound variable x, while in the right-hand side #M(#N) achieves the substitution "#M[#N/*x*]" of the usual β rule. CRS notably also makes it possible to test for the *absence* of binder arguments to meta-variables: in the last rule, #M() is not the same as #M(*x*), and expresses that *x* can *not* occur in subterms matching #M.

1.2 Example. Here is a traditional "call-by-value" λ calculus evaluator, using HOAS and an environment.

```
L data Lam([L]L);                                   1
L data Ap(L, L);                                    2
L variable;                                         3

L scheme Eval(L, {L:L});                            5
L rule Eval(Lam([x]#B(x)), {#env})                  6
  → Lam([x]#B(x));                                  7
L rule Eval(Ap(#F, #A), {#env})                     8
  → Apply(Eval(#F, {#env}),                         9
          Eval(#A, {#env}), {#env});                10
L rule Eval(x, {#env; x : #V}) →#V;                 11

L scheme Apply(L, L, {L:L});                        13
L rule Apply(Lam([x]#B(x)), #V, {#env})             14
  → Eval(#B(z), {#env, z : #V});                    15
```

The variable declaration ensures that variables in the L sort are syntactic, so they can be matched in line 11 (free) and line 13 (bound), where we in the latter substitute it with a fresh variable, *z* in line 14, in the term and environment.

2 The Calculus

2.1 Definition ($\hbar$ syntax)**.** The $\hbar$ syntax is summarized as follows (with overbars denoting vectors):

$$
\begin{array}{rll}
H & ::= \overline{D} & (\hbar\text{Script})\\
D & ::= S\ \textbf{data}\ d\,(\,\bar{F}\,)\,; & (\text{Declaration})\\
 & \mid S\ \textbf{scheme}\ f\,(\,\bar{F}\,)\,; & \\
 & \mid S\ \textbf{variable}\,; & \\
 & \mid S\ \textbf{rule}\ T \to T\,; & \\
F & ::= [\,\bar{S}\,]\,S \mid \{\,S : S\,\} & (\text{Form})\\
S & ::= s\,\langle\,\bar{S}\,\rangle \mid \alpha & (\text{Sort})\\
T & ::= c\,(\,\bar{P}\,) \mid v \mid m\,(\,\bar{T}\,) & (\text{Term})\\
P & ::= [\,\bar{v}\,]\,T \mid \{\,\bar{A}\,\} & (\text{Piece})\\
A & ::= v : T \mid \neg v{:} \mid m\,(\,\bar{v}\,) & (\text{Association})
\end{array}
$$

The top level of a $\hbar$ script is H and the grammar assumes that we have three categories of identifier terminals defined (with representative exemplars from the examples):

- $c, s, d, f \in \mathcal{C}$ stand for *constructor* terminals (Lam, Ap).
- $v, \alpha \in \mathcal{V}$ stand for *variable* terminals (*x*, *z*).
- $m \in \mathcal{M}$ stands for *meta-variable* terminals (#M, #*env*).

The $\hbar$ formalism includes traditional "constructor" term rewriting systems, where there is a distinction between *defined* **scheme** symbols and **data** symbols. However, unlike functional programming, $\hbar$ allows for normal forms that include (incompletely) defined symbols.

3 Sorting

In this Section we define $\hbar$ script formation formally by restricting the terms of the grammar in Definition 2.1 further to only allow well-formed "sortable" scripts. Informally, sorting ensures that

- pattern and contraction restrictions are obeyed;
- our special "syntactic variables" are used correctly;
- binders are used in the shape declared for constructors;
- subterms (including variable and meta-variable occurrences) have the right sort;
- association keys and values have the proper sorts.

3.1 Definition. A *global sort environment* Γ is a structure that combines $\Gamma_{\text{rank}} : \mathcal{C} \to \mathcal{N}$: the rank of each sort constructor (all sorts must be fully applied, there are no higher kinds); Γ_{hasvar}: a set of sort names (the sorts that allow variables); $\Gamma_{\text{con}} : \mathcal{C} \to S \times F^*$: a map from constructor names to pairs of a sort and a list of forms (the pair consists of the construction's sort and the shape of the arguments); and Γ_{fun}: a set of constructor names (those declared with **scheme**).

3.2 Definition. A *rule environment* Δ is a structure that combines $\Delta_{\text{var}} : \mathcal{V} \to S$: a mapping from variable names to sorts (the sort assigned to each variable for the rule) and $\Delta_{\text{meta}} : \mathcal{M} \to MF$: a mapping from meta-variable names to "meta-forms" defined by

$$MF ::= \bar{S} \Rightarrow S \mid \bar{S} \Rightarrow \{S{:}S\} \qquad (\text{MetaForm})$$

MF captures the difference between regular meta-variables and "catch-all" ones: The shape $\bar{S} \Rightarrow S$ is used for meta-variables that need to be meta-applied to arguments with the sorts $\bar{S}$ to then form a term of the sort S. The shape $\bar{S} \Rightarrow \{S_1{:}S_2\}$ is used for meta-variables that catch all the associations in an association list from S_1 to S_2.

3.3 Definition. A $\hbar$ script H is *well formed* if we can prove $\vdash H$ with the rule

$$\frac{\Gamma \vdash D_1 \quad \cdots \quad \Gamma \vdash D_n}{\vdash D_1 \ldots D_n} \quad (\exists\Gamma) \qquad (\text{SH})$$

using the rules below for sorting each declaration.

Thus sorting relies on a sort environment "witness" to establish that a script is well sorted. In practice, Γ will be assembled from the constraints of the component declarations. Because all top level symbols are explicitly sorted, sort assignment is not problematic.

3.4 Definition. A $\hbar$ declaration D is well-sorted for a sort environment Γ if we can prove $\Gamma \vdash D$ with the SD-* rules:

$$\frac{\Gamma \vdash s\langle\bar{\alpha}\rangle}{\Gamma \vdash s\langle\bar{\alpha}\rangle\ \textbf{data}\ d(\bar{F})\,;} \quad \Gamma_{\text{con}}(d) = \big\langle s\langle\bar{\alpha}\rangle, \bar{F}\big\rangle,\ d \notin \Gamma_{\text{fun}} \qquad (\text{SD-Data})$$

$$\frac{\Gamma \vdash S}{\Gamma \vdash S\ \textbf{scheme}\ f(\bar{F})\,;} \quad \Gamma_{\text{con}}(f) = \big\langle S, \bar{F}\big\rangle,\ f \in \Gamma_{\text{fun}} \qquad (\text{SD-Fun})$$

$$\frac{\Gamma \vdash s\langle\bar{\alpha}\rangle}{\Gamma \vdash s\langle\bar{\alpha}\rangle\ \textbf{variable}\,;} \quad s \in \Gamma_{\text{hasvar}} \qquad (\text{SD-Var})$$

$$\frac{\Gamma \vdash S \quad \Gamma, \Delta, V, \text{Pat}, \epsilon \vdash T_1 : S \quad \Gamma, \Delta, V, \text{Con}, \epsilon \vdash T_2 : S}{\Gamma \vdash S\ \textbf{rule}\ T_1 \to T_2\,;} \quad \begin{cases} (\exists\Delta) \\ V_i = \text{NonAssocVars}(T_i) \end{cases} \qquad (\text{SD-Rule})$$

Rules (SD-Data,SD-Fun) express that constructors are recorded correct in the environment; note that data constructors must belong to a unique named sort. (SD-Var) records sorts with syntactic varables, and (SD-Rule) checks the sorts and well-formedness of the terms in rules; the "NonAssocVars" function returns all variables in a term that are *not* inside an association.

3.5 Definition. A sort denotation S is well-sorted for a sort environment Γ if we can prove $\Gamma \vdash S$ with the SS-* rules:

$$\frac{\Gamma \vdash S_1 \quad \cdots \quad \Gamma \vdash S_n}{\Gamma \vdash s\langle S_1, \ldots, S_n\rangle} \quad \Gamma_{\text{rank}}(s) = n \qquad (\text{SS-Cons})$$

$$\frac{}{\Gamma \vdash \alpha} \qquad (\text{SS-Var})$$

Essentially, sorts are well-formed when they have consistent rank.

Sorting of Term $\boxed{\Gamma, \Delta, V, TC, \bar{v} \vdash T : S}$

$$\frac{\Gamma, \Delta, V, \mathrm{InPat}, \bar{v} \vdash P_1 : F_1 \quad \cdots \quad \Gamma, \Delta, V, \mathrm{InPat}, \bar{v} \vdash P_n : F_n}{\Gamma, \Delta, V, \mathrm{Pat}, \bar{v} \vdash f\,(\,P_1, \ldots, P_n\,) : S} \qquad \begin{cases} f \in \Gamma_{\mathrm{fun}} \\ \Gamma_{\mathrm{con}}(f) = \langle S, (F_1, \ldots, F_n) \rangle \end{cases} \qquad \text{(SMP-Fun)}$$

$$\frac{\Gamma, \Delta, V, \mathrm{InPat}, \bar{v} \vdash P_1 : F_1 \quad \cdots \quad \Gamma, \Delta, V, \mathrm{InPat}, \bar{v} \vdash P_n : F_n}{\Gamma, \Delta, V, \mathrm{InPat}, \bar{v} \vdash d\,(\,P_1, \ldots, P_n\,) : S} \qquad \begin{cases} d \notin \Gamma_{\mathrm{fun}} \\ \Gamma_{\mathrm{con}}(d) = \langle S, (F_1, \ldots, F_n) \rangle \end{cases} \qquad \text{(SMP-Data)}$$

$$\frac{}{\Gamma, \Delta, V, \mathrm{InPat}, \bar{v} \vdash m\,(\,w_1, \ldots, w_n\,) : S} \qquad \begin{cases} \Delta_{\mathrm{meta}}(m) = ((S_1, \ldots, S_n) \Rightarrow S) \\ \forall i\colon w_i \in \bar{v} \\ \forall i\colon \Delta_{\mathrm{var}}(w_i) = S_i \\ \text{All the } w_i \text{ are different} \end{cases} \qquad \text{(SMP-Meta)}$$

$$\frac{}{\Gamma, \Delta, V, \mathrm{InPat}, \bar{v} \vdash w : s\langle \bar{S} \rangle} \qquad \begin{cases} \Delta_{\mathrm{var}}(w) = s\langle \bar{S} \rangle \\ s \in \Gamma_{\mathrm{hasvar}} \end{cases} \qquad \text{(SMP-Var)}$$

$$\frac{\Gamma, \Delta, V, \mathrm{Con}, \bar{v} \vdash P_1 : F_1 \quad \cdots \quad \Gamma, \Delta, V, \mathrm{Con}, \bar{v} \vdash P_n : F_n}{\Gamma, \Delta, V, \mathrm{Con}, \bar{v} \vdash c\,(\,P_1, \ldots, P_n\,) : S} \qquad \Gamma_{\mathrm{con}}(c) = \langle S, (F_1, \ldots, F_n) \rangle \qquad \text{(SMC-Cons)}$$

$$\frac{\Gamma, \Delta, V, \mathrm{Sub}, \bar{v} \vdash T_1 : S_1 \quad \cdots \quad \Gamma, \Delta, V, \mathrm{Sub}, \bar{v} \vdash T_n : S_n}{\Gamma, \Delta, V, \mathrm{Con}, \bar{v} \vdash m\,(\,T_1, \ldots, T_n\,) : S} \qquad \Delta_{\mathrm{meta}}(m) = ((S_1, \ldots, S_n) \Rightarrow S) \qquad \text{(SMC-Meta)}$$

$$\frac{}{\Gamma, \Delta, V, \mathrm{Con}, \bar{v} \vdash w : s\langle \bar{S} \rangle} \qquad \begin{cases} \Delta_{\mathrm{var}}(w) = s\langle \bar{S} \rangle \\ s \in \Gamma_{\mathrm{hasvar}} \end{cases} \qquad \text{(SMC-Var)}$$

$$\frac{\Gamma, \Delta, V, \mathrm{Con}, \bar{v} \vdash c\,(\,P_1, \ldots, P_n\,) : s\langle \bar{S} \rangle}{\Gamma, \Delta, V, \mathrm{Sub}, \bar{v} \vdash c\,(\,P_1, \ldots, P_n\,) : s\langle \bar{S} \rangle} \qquad s \notin \Gamma_{\mathrm{hasvar}} \qquad \text{(SMS-Cons)}$$

$$\frac{\Gamma, \Delta, V, \mathrm{Con}, \bar{v} \vdash m\,(\,T_1, \ldots, T_n\,) : s\langle \bar{S} \rangle}{\Gamma, \Delta, V, \mathrm{Sub}, \bar{v} \vdash m\,(\,T_1, \ldots, T_n\,) : s\langle \bar{S} \rangle} \qquad s \notin \Gamma_{\mathrm{hasvar}} \qquad \text{(SMS-Meta)}$$

$$\frac{}{\Gamma, \Delta, V, \mathrm{Sub}, \bar{v} \vdash w : S} \qquad \Delta_{\mathrm{var}}(w) = S \qquad \text{(SMS-Var)}$$

Sorting of Piece $\boxed{\Gamma, \Delta, V, TC, \bar{v} \vdash P : F}$

$$\frac{\Gamma, \Delta \cup \Delta', V, TC, (\bar{v}\,\overline{w}) \vdash T : S}{\Gamma, \Delta, V, TC, \bar{v} \vdash [\,\overline{w}\,]\,T : [\bar{S}]S} \qquad (\exists \Delta'), \Delta'_{\mathrm{var}}(\overline{w}) = \bar{S} \qquad \text{(SP-Bind)}$$

$$\frac{\Gamma, \Delta, V, TC, \bar{v} \vdash A_1 : \{S{:}S'\} \quad \cdots \quad \Gamma, \Delta, V, TC, \bar{v} \vdash A_n : \{S{:}S'\}}{\Gamma, \Delta, V, TC, \bar{v} \vdash \{\,A_1, \ldots, A_n\,\} : \{S{:}S'\}} \qquad \text{(SP-Assoc)}$$

Sorting of Association $\boxed{\Gamma, \Delta, V, TC, \bar{v} \vdash A : \{S{:}S\}}$

$$\frac{\Gamma, \Delta, V, TC, \bar{v} \vdash w : S \quad \Gamma, \Delta, V \cup V', TC, \bar{v} \vdash T' : S'}{\Gamma, \Delta, V, TC, \bar{v} \vdash w{:}T' : \{S{:}S'\}} \qquad \begin{cases} w \in V \\ V' = \mathrm{NonAssocVars}(T') \end{cases} \qquad \text{(SA-Map)}$$

$$\frac{\Gamma, \Delta, V, \mathrm{InPat}, \bar{v} \vdash w : S}{\Gamma, \Delta, V, \mathrm{InPat}, \bar{v} \vdash \neg w{:} : \{S{:}S'\}} \qquad \text{(SAP-Not)}$$

$$\frac{}{\Gamma, \Delta, V, \mathrm{InPat}, \bar{v} \vdash m\,(\,w_1, \ldots, w_n\,) : \{S{:}S'\}} \qquad \begin{cases} \Delta_{\mathrm{meta}}(m) = ((S_1, \ldots, S_n) \Rightarrow \{S{:}S'\}) \\ \forall i\colon w_i \in \bar{v} \\ \forall i\colon \Delta_{\mathrm{var}}(w_i) = S_i \end{cases} \qquad \text{(SAP-All)}$$

$$\frac{\Gamma, \Delta, V, \mathrm{Sub}, \bar{v} \vdash T_1 : S_1 \quad \cdots \quad \Gamma, \Delta, V, \mathrm{Sub}, \bar{v} \vdash T_n : S_n}{\Gamma, \Delta, V, \mathrm{Con}, \bar{v} \vdash m\,(\,T_1, \ldots, T_n\,) : \{S{:}S'\}} \qquad \Delta_{\mathrm{meta}}(m) = ((S_1, \ldots, S_n) \Rightarrow \{S{:}S'\}) \qquad \text{(SAC-All)}$$

Figure 1: $\hbar$ term sorting rules.

3.6 Definition. Given a sort environment Γ, a rule environment Δ, and a variable set V. A term T is *well-formed of sort* S in term context TC with

$$TC ::= \text{Pat} \mid \text{InPat} \mid \text{Con} \mid \text{Sub} \qquad \text{(TermContext)}$$

and bound variables $\bar{v}$ if we can prove

$$\Gamma, \Delta, V, TC, \bar{v} \vdash T : S$$

using the rules in Figure 1.

The rules in Figure 1 give the primary term rules per term context TC, which indicates the context of our term fragment:

- "Pat" indicates the pattern (left side) of a rule, at the outermost level.
- "InPat" is used inside a piece of the pattern of a rule, not the outermost level.
- "Con" denotes any location in the contraction (right side) of a rule except a substitution.
- "Sub" denotes a substitution location (an immediate child of a meta-application) in the contraction (right side) of a rule.

The (SMP-*) rules handle patterns. The first rule (SMP-Fun) is the entry rule for patterns, with TC = Pat, which must be function constructions, and have pieces that are well-sorted. Fragments of patterns are then handled by the three following rules, (SMP-Data,SMP-Meta,SMP-Var), with TC = InPat, which capture the sort propagation as well as the special constraints for patterns: (SMP-Data) sort checks that only data constructors are allowed; (SMP-Meta) verifies that pattern meta-application arguments are restricted to distinct bound variables; (SMP-Var) verifies that other instances of variables only occur where a **variable** declaration explicitly permits it.

The (SMC-*) rules handle contraction terms. (SMC-Cons) verifies that each construction (function or data) is well sorted. (SMC-Meta) verifies that every meta-application is sorted consistently with the rule environment, and checks that the implied substitutions are well formed. (SMC-Var) checks that variables are used at the right sort, and are permitted (by a **variable** declaration for the sort).

The (SMS-*) rules handle contraction substitution arguments. They really just wrap the corresponding (SMC-*) rules except that (SMS-Cons,SMS-Meta) checks that the non-trivial substitution is permitted by the sort *not* having a **variable** declaration, and (SMS-Var) just checks the sort of the replacement variable *without* any constraints on whether the sort has a **variable** declaration.

The (SP-*) rules handle "pieces," *i.e.*, parameters of constructions (which at the outermost level have a different structure than other subterms). Note that these do *not* depend on the term context, which is merely passed to the premises. (From the other rules we can see that the term context will always be InPat or Con.) (SP-Bind) handles scopes. It creates a *locally extended* version of the rule environment, Δ', which extends the variable bindings part of Δ with the binders in the scope; the sorts of these are fixed by the parent construction. (SP-Assoc) handles collections of associations. These are checked by separate individual rules for association, below.

The (SA*) rules handle associations. These are categorized in a slightly different ways than the others above, as associations have different rules in patterns and contractions. (SA-Map) gives the rule for a simple mappings. The only unusual requirement is the side condition that the variable, w, must also occur elsewhere in the term as a non-key, a requirement that ensures that matching is deterministic (in essence avoiding the "axiom of choice" for the set of keys). The two (SAP-*) rules express the constraints on the other two forms that can occur in patterns, and (SAC-All) that environment copy is unconstrained in cntractions.

4 Implementation

The $\hbar$ language has been implemented in the CRSX project from scratch, and is available from `http://github.com/crsx/core/plank`. At the time of writing, it still has errors, but these are all minor and we expect to root them out by the time of the workshop.

5 Conclusion

With $\hbar$, we have presented a rather small calculus that can serve as the underlying formalism for reasoning about, as well as implementing, serious compiler generation languages that support native higher-order abstract syntax. We have a full implementation.

Related work. We would like to give credit to SIS [7], which shares with $\hbar$ the use of *simplification* using a λ-calculus based formalism.

The most prominent system that supports implementation of compilers in formal (rewriting and attribute grammar) form is ASF+SDF [4], which is based on first order rewriting. While modules have been added for symbol table management, these lack the full integration and easy way to handle scoped intermediate languages. The successor, Rascal [3] adds a module for HOAS, but Rascal specifications operate in a world of side effects, which we find hard to reconcile with higher-order term structures (with scopes).

The notion of "higher-order" used by $\hbar$ is similar to but not quite the same as in higher-order attribute grammars (HAG) [14]. Like HAGs, $\hbar$ specifications permit constructing and passing of abstract syntax fragments in attributes but the "higher order" aspect of $\hbar$ also covers the rewriting side, where we can build parameterized abstractions over any part of a specification, including with attributes. Indeed, one can use substitution inside attributes, and have absence of attributes and substitution block rewriting.

Future work. For the full version of this work, we plan to complete the implementation by porting the old CRSX pattern matching compiler and possibly looking to integrate with the TransScript ANTLR-based parser frontend. On the theory side we still owe the world a proper formal development of the properties of matching in ℏ.

Acknowledgements. The authors are grateful to Lionel Villard, Maria Schett, and Julian Nagele, for continuous valuable discussions in the CRSX project during the development of this work. The project is supported by the Marie Skłodowska-Curie action "HORIP", program H2020-MSCA-IF-2014, 658162.

References

[1] Peter Aczel. A general Church-Rosser theorem. `http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGeneralChurch-RosserTheorem.pdf`, July 1978. Corrections at `http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGRT_corrections.pdf`.

[2] Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. Inductive-data-type systems. *Theor. Computer Science*, 272(1-2):41–68, 2002. Corrected version in `http://arxiv.org/abs/cs/0610063`.

[3] J. van den Bos, M. A. Hills, P. Klint, T. van der Storm, and J. J. Vinju. Rascal: From algebraic specification to meta-programming. In V. Rusu and F. Durán, editors, *AMMSE 2011—Second International Workshop on Algebraic Methods in Model-based Software Engineering*, volume 56 of *Electronic Proceedings in Theoretical Computer Science*, pages 15–32, 2011.

[4] M.G.J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[5] Jean-Pierre Jouannaud. Higher-order rewriting: Framework, confluence and termination. In *Processes, Terms and Cycles: Steps on the road to infinity—Essays Dedicated to Jan Willem Klop on the occasion of his 60th Birthday*, volume 3838 of *Lecture Notes in Computer Science*, pages 224–250. Springer Verlag, 2005.

[6] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theor. Computer Science*, 121:279–308, 1993.

[7] Peter Mosses. SIS – semantics implementation system, reference manual and user's guide. Technical Report MD-30, DAIMI (Computer Science Department), Aarhus University, August 1979.

[8] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 199–208, New York, NY, USA, 1988. ACM.

[9] Eva Rose and Kristoffer H. Rose. Compiler construction. The graduate school computer science Compiler Construction class (CSCI-GA.2130) at the Courant Institute for the Mathematical Sciences, New York University, 2015.

[10] Kristoffer Rose. Combinatory reduction systems with extensions. GitHub, `https://github.com/crsx`, 2014.

[11] Kristoffer H. Rose. *Operational Reduction Models for Functional Programming Languages*. PhD thesis, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 København Ø, February 1996. `http://krisrose.net/thesis.pdf`.

[12] Kristoffer H. Rose. CRSX – combinatory reduction systems with extensions. In Manfred Schmidt-Schauß, editor, *RTA '11—22nd International Conference on Rewriting Techniques and Applications*, volume 10 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 81–90, Novi Sad, Serbia, June 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[13] Kristoffer H. Rose. Higher-order attribute contraction schemes. Technical Report TSTR-2016-1, Two Sigma Investments, LP, March 2016.

[14] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 131–145, New York, NY, USA, 1989. ACM.