A Calculus of Transformation

B. Wolff, H. Shi , Universität Bremen[1]

DRAFT VERSION FROM 28. November 1994

Abstract: This paper presents the concepts and the semantics of a transformation-calculus TC that is generic wrt. concrete object languages. Built upon an object language description given by theory in higher-order logics (see [Andr 86]), TC provides context-sensitive rules in which requirements on the context of a redex can be imposed, and integrates a restricted form of extended rewriting. Furthermore, rules may be higher-order in order to represent tactical combinators and to model "parametric transformations". This work can be seen as a specification of transformation systems and a foundation for correctness-proofs of transformations.

I. Introduction

The notion of transformation is the core of transformational program development following the lines of CIP-L ([Bau+ 85] [Pepper 84]) and PROSPECTRA (see [HK□93]). The major goal of this paper is to get a better understanding of the concept "transformation" in form of a generic meta-language together with its formal semantics and its calculus.

Transformation in our sense is a form of deduction, applying *transformation rules* in a kind of term-rewriting process. A *rule* is a pair of terms l and r, written as l ⇒ r. If a term t *matches* l, then a term t' can be constructed from r and the match of t and l. In this case we say that the *rule application* (l ⇒ r) (t) *is reduced to* t'. This reduction is also called a *transformation step*. *Rule-terms* can be *higher-order*: e.g. (l□⇒□r)□⇒ (c l ⇒ c r) is a possible transformation rule, if c is a (first-order) constructor of a term language and l, r, t, t´ range over terms. This higher-order rule can transform a rule a□⇒□b into (c□a ⇒ c b). It will be demonstrated that higher-order rules allow a similar, explicit treatment of reduction contexts as in logical frameworks based on higher-order abstract syntax. The rule arrow binds to the right.

Context sensitive transformation rules allow the definition of "syntactical provisos" (or: applicability *conditions* Φ). Here, (c(l⟦Φ⟧)□⇒□r)(t) *is reduced to* t' if matching and reconstruction is possible and the applicability condition is fulfilled for the instance a of l and its context λx. c(x), i.e. Φ(λx□c(x),a) holds. Syntactically, applicability-conditions are formulas constructed over *annotation functions*, which assign to each pair of a term and its context a value such as *type* or the *set of bound variables*[2]. Note that higher-order transformations can be used to manipulate the contexts that applicability conditions are referring to.

---

[1] The authors address: Universität Bremen, Postfach 330440, FB 3. E-mail:{bu,shi}@informatik.uni-bremen.de

[2] Traditionally, the annotation function *boundVariables* maps an *occurrence* of a term to the desired set; the occurrence is repesented by *path's* in a term or the like. The pairs of "contexts and terms" in our approach can be seen as a more refined implementation of the concept of occurence.

The matching relation is defined by the usual instantiation of the variables with an appropriate substitution modulo the congruence induced by some set of first-order equations E (cf. E-matching [DJ 90], [Bach 93]). E-Matching is the vehicle for representing rewrite-relations traditionally presented by *rule schemata*. For example, let ++ denote the concatenation on lists described by the equations E={[] ++ X = X, (X::Y)++Z = X::(Y++Z)}, the rule $A{+}{+}[P[\![\Phi]\!]]{+}{+}B \Box\!\Rightarrow P$ picks from a list of syntactic elements (e.g. declarations) one P that fulfils the condition Φ. Symbols like ++ are called *pattern-generating functions* (or: *matching-combinators,* see [SW 94]).

In the line of our approach, explicit substitution (cf. [ACCL 91]) is also represented by particular theory E instead of integrating higher-order matching. This enables provisos referring to variables (*occurs* in, is *orthogonal* etc., or even the syntactic representation of the *admissibility* in LCF ([Paul 87])), allows rule-schemes produced by pattern generating functions and gives access to the induction-principle over abstract syntax terms.

Such as λ-terms in typed λ-calculi can be understood as functional proof-representations, rule-terms form a relational representation of transformational proofs. The very restricted non-determinism in the relational semantics is a consequence of the E-matching introduced by the pattern generators, that can be used to suppress "structural rules" in logics and to get a more "reusable" representation of classes of proofs than λ-terms.

The proposed meta-language can be seen as a Meta-logical Framework (cf. [HST 89], [BC 92], [Paul 89], [Bar 92], [MM 94], [CH 88]). In contrast to these approaches, transformation maintains a first-order, intensional view on the abstract syntax; from this point of view, our work is more related to *Combinatory Reduction Systems* (cf. [Klop 80]) or *Rewrite Logics* (cf. [MM 93]).

Rule-terms can be enriched by tactical combinators (cf. [KLSW 94]) to *tactical terms* or *specifications of deductions* . Such tactical terms can express the tactical dimension of proving - e.g. "apply a rule from left to right as long as possible". A major advantage of our approach is that the intensional, syntactic view on rule-terms opens the perspective of formally manipulating tactical terms within the framework, especially aiming at an executable specification denoting a tactical program. This aspect is deliberately excluded from this paper; an overview on syntactic criteria for tactical optimisations (in the sense of finding a "rewrite-strategy" that is "normalising") is given in [Klop 92], pp. 83.

This paper is organised as follows: In the first section, we will introduce the raw syntactic material of our generic calculus that will be restricted to well-formed expressions in the next section. The description of the semantics and the consistency conditions (requirements to the object-logics – consisting of function definitions and rules) leads to a calculus of meta-inferences inclusively induction. The last section discusses the different styles of toy-logics that can be encoded into our framework.

II. Raw Terms

Let C be an enumerable set of constant symbols, MV a set of symbols for meta-variables, F of symbols for "annotation functions", and π of predicate symbols (equality, Scott-ordering,...). Then we define recursively the sets of terms, formulas and rule-terms:

T	terms	$T := C \mid MV \mid T\ T$
Φ	formulas	$\Phi := \pi(R\|F...R\|F) \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \Phi \Rightarrow \Phi$
R	rule-terms	$R := C \mid MV \mid R\ R \mid R \Rightarrow R \mid [V]\ R \mid R\ [\![\Phi]\!]$

Due to our first-order intensional approach of representing object-languages, terms are simply applications over constants and variables. Rule-terms contain also these three variants as a basis for term-patterns, but enrich T with additional constructs. The variants "rule application" and "rule abstraction" have already been discussed in the introduction. The "annotation-constraint" $R\ [\![\Phi]\!]$ denotes a sub pattern R whose instances t and instance-contexts C[t] (see section IV) must satisfy a formula $\Phi(C,t)$. The scope-operator $[v]\ R$ binds v in the scope of R and can be seen as a universal quantification for v.

We will restrict the set of constants C (and hence the constructors of the abstract syntax of the object-language) to be first-order, and do not comprise λ-abstraction in the usual higher-order abstract syntax style. This enforces explicit representations of bindings in object-languages; the classical syntactical provisos for proper substitutions are preferably represented by annotation-constraints. On the level of rule-terms, however, it is possible to represent abstractions directly simply by making the two facets of the λ-abstraction, scope-formation and function-construction, explicit:

$$[x]\ (x \Rightarrow e)$$

This rule-term will be denoted by $\Lambda x.e$. Hence, on the level of R it is possible to represent contexts in the usual higher-order way. Remind that rule-terms do not necessarily denote functions, but relations.

Note that T is a subset of R. Hence, there exists an embedding $I:T \rightarrow R$. We usually omit the embedding functions and treat T-terms as R-terms. The function $V: R \rightarrow R$ ("forget annotations") is defined as the homomorphic extension over the projection $V(r[\![\Phi]\!]) = V(r)$.

III. Well-formedness

Terms and formulas are well-formed iff they are typed, rule-terms are well-formed iff they are typed. The typing-discipline is simply an adoption of conventional Milner-typing (see [DM□82]).

Typing

The syntactical categories of the type system are as follows:

α	type variables	β,γ,δ ...
χ	type constructors	bool, int, list, set
τ	(simple) types: $\tau ::= \alpha \mid \tau \mapsto \tau \mid \chi(\tau,..,\tau)$	$\tau_1,..,\tau_n$
σ	type schemes: $\sigma ::= \forall\alpha.\ \sigma \mid \tau$	

We chose $\mapsto$ to denote the relational abstraction in order to distinguish it from functional abstraction from the meta-language level (i.e. higher-order logics, see section 4).

As usual, a notion of signature Σ and type-assignment has to be introduced:

Σ	signature:	$\Sigma ::= ((\pi \mid F \mid C) : \sigma)^*$
Γ	type assignments:	$\Gamma ::= (x : \sigma)^*$

We define typing as usual by a mutual recursive system of inductive defined predicates. The 4-ary predicate $\Sigma, \Gamma \vdash t : \sigma$ states that t is typable with σ in the environment consisting of Σ and Γ. The predicate $\Sigma, \Gamma \Vdash_\tau \phi$ states that ϕ is a well-formed formula in the environment Σ,Γ and the type-context τ (the *type-context* assures that each annotation function symbol F, which can only occur in a constraint formula) have the proper typing according to the pattern they refer to). The predicate $\Sigma, \Gamma \Vdash\!\!\sim_\tau t_1 : \tau_1$ is a slightly generalised version of the $\vdash$-predicate.

A type is *essentially first order* if it has the form $\chi(...) \mapsto \ldots \mapsto \chi(...)$. Since we aim at first-order abstract syntax, our type system requires all constants to be essentially first-order. However, this does not exclude higher-order meta-variables in rule-terms.

With $\leq: \sigma \times \sigma \rightarrow$ bool we denote the usual subsumption ordering on type-schemes: $\forall\alpha.\Box\alpha\Box\mapsto\Box$int $\leq \forall\beta.\Box(\beta\mapsto\beta)\Box\mapsto\Box$int, for example. (See [DM 82] for a formal definition).

The core of the system consists of the typing axioms:

(*var*) $\Sigma, \Gamma \vdash x : \sigma$ $(x{:}\sigma \in \Gamma)$

(*const*) $\Sigma, \Gamma \vdash c : \sigma$ ($c{:}\sigma \in \Sigma$, σ essentiall first order)

(*⇒–elim*)
$$\frac{\Sigma, \Gamma \vdash r : \tau' \mapsto \tau \qquad \Sigma, \Gamma \vdash r' : \tau'}{\Sigma, \Gamma \vdash r\, r' : \tau}$$

(*⇒–intro*)
$$\frac{\Sigma, \Gamma \vdash r : \tau \qquad \Sigma, \Gamma \vdash r' : \tau'}{\Sigma, \Gamma \vdash r \Rightarrow r' : \tau \mapsto \tau'}$$

(*constr–intro*)
$$\frac{\Sigma, \Gamma \vdash r : \tau \qquad \Sigma, \Gamma \Vdash_\tau \phi}{\Sigma, \Gamma \vdash r\, [\![\phi]\!] : \tau}$$

(*scope–intro*)
$$\frac{\Sigma, (x{:}\sigma)::\Gamma \backslash x \vdash r : \tau}{\Sigma, \Gamma \vdash [x]\, r : \tau}$$

(*∀–elim*)
$$\frac{\Sigma, \Gamma \vdash r : \sigma}{\Sigma, \Gamma \vdash r : \sigma'} \quad (\sigma' \leq \sigma)$$

(*∀–intro*)
$$\frac{\Sigma, \Gamma \vdash r : \sigma}{\Sigma, \Gamma \vdash r : \forall \alpha.\sigma} \quad (\text{if } \alpha \notin \text{free}(\Gamma))$$

This system is extended by the rules for formulas:

(*predicate*)
$$\frac{\Sigma, \Gamma \Vdash_\tau t_1 : \tau_1 \ldots \Sigma, \Gamma \Vdash_\tau t_n : \tau_n}{\Sigma, \Gamma \Vdash_\tau \pi(t_1 \ldots t_n)} \quad (\text{if } \pi{:}\tau_1 \mapsto \ldots \mapsto \tau_n \mapsto \text{bool} \in \Sigma)$$

(*junctor*)
$$\frac{\Sigma, \Gamma \Vdash_\tau \phi \qquad \Sigma, \Gamma \Vdash_\tau \phi'}{\Sigma, \Gamma \Vdash_\tau \phi \ominus \phi'} \quad (\text{where } \ominus = \wedge, \vee, \Rightarrow)$$

$$(\textit{const-acc}) \qquad \frac{\Sigma, \Gamma \vdash t_1 : \tau_1}{\Sigma, \Gamma \vdash\!\!\sim_\tau t_1 : \tau_1}$$

$$(\textit{anno-acc}) \qquad \Sigma, \Gamma \vdash\!\!\sim_\tau f : \tau' \qquad (\text{if } (f:\forall\alpha.(\tau\mapsto\alpha)\mapsto\tau\mapsto\tau') \in \Sigma)$$

Theorem: **Existence of principal types is decidable.**

Proof: **Adaptation of [DM 82].**

With $T_\Sigma(\Gamma)$ (or, sloppily $T_\Sigma(X)$) we denote the set of well-typed rule-terms over signature Σ and type assumptions Γ. Instead of $\Sigma,\Gamma \vdash\!\!\sim : \sigma$ we will write r_σ (or r_τ) and assume an arbitrary, but fixed environment.

Free Variables

Let $\mathcal{V}(r) \in \wp(MV)$ denote the set of meta-variables occurring in r.
The sets of free and bound meta-variables are analogously defined to the λ-calculus are inductively defined by:

$$\mathcal{FV}(x) = \{x\}$$
$$\mathcal{FV}(c) = \mathcal{FV}(f) = \{\}$$
$$\mathcal{FV}(r\ r') = \mathcal{FV}(r \Rightarrow r') = \mathcal{FV}(r) \cup \mathcal{FV}(r')$$
$$\mathcal{FV}(r\ [\![\phi]\!]) = \mathcal{FV}(r) \cup \mathcal{FV}(\phi)$$
$$\mathcal{FV}([x]\ r) = \mathcal{FV}(r) \ - \{x\}$$

$$\mathcal{FV}(\pi(t_1 \ldots t_n)) = \mathcal{FV}(t_1) \cup \ldots \cup \mathcal{FV}(t_n)$$
$$\mathcal{FV}(\neg\phi) = \mathcal{FV}(\phi)$$
$$\mathcal{FV}(\phi \wedge \phi') = \mathcal{FV}(\phi \vee \phi') = \mathcal{FV}(\phi \Rightarrow \phi') = \mathcal{FV}(\phi) \cup \mathcal{FV}(\phi')$$

$$\mathcal{BV}(r) = \mathcal{V}(r) - \mathcal{FV}(r)$$

A rule term r is *closed* if its set of free variables is empty.

With $t\,\sigma$ we denote the simultaneous substitution for terms, formulas and r-terms of free variables as usual. We omit the definition here.

IV. Semantics

Higher-Order Logics as a Meta-Language

We intend to implement our calculus in a meta-logic. As a consequence, we can derive the meta-inference rules of the calculus out of the following elementary theory for the semantical functions of rule-terms. An implementation in a theorem prover could thus provide mechanical support for applying rules via the meta-inferences and for checking the applicability conditions.

Since an annotation constraint can be seen as a "context sensitive" boolean function, i.e. a function that depend on the pair of a term t and its context λx.C(x), the meta-logic should comprise higher-order terms.

Possible choices could be LCF (see [Paul 87]), Zermelo-Fränkel set theory or HOL ([Andr□86], which is based on [Chu 40] and goes back to the *Principia Mathematica*). LCF provides continuous, partial functions together with Scott-induction and is most appropriate for verification of functional programs. As a basis for a theoretical investigation, however, we feel that the necessary logical machinery is far too complicated. Zermelo-Fränkel set theory is preferable for logicians facing classical problems of mathematical foundations; it is untyped and is therefore leaving the type-checking of our rule-terms to active theorem proving. Hence, we chose HOL, a polymorphic typed classical logic with total functions and equality. On the basis of the very recent work of [Reg 94] which formulates classical domain-theory inside HOL, it is even possible to define within the framework of annotation functions a classical denotational semantics for an object-language and proving the soundness of rules with respect to this semantic function.

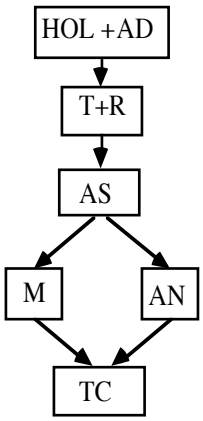
There are powerful support tools for HOL, which will not be discussed this paper. However, we adopt the notation presented in [Paul 94]. The type for function spaces is $\tau \Rightarrow \tau'$, the cartesian product is $[\tau, \tau']$ (notation ×, →). The type for formulas is *bool*. The abstraction in HOL-terms is written as %x.t (notation: λx.t), the application f(x). The quantifiers of HOL-formulas are All and Ex (notation: ∀, ∃), the junctors &, |, -->, not (notation ∧, ∨, →, ¬).

Theory architecture

HOL can be conservatively extended by a particular set theory, whose type constructor α set (notation $\wp(\alpha)$) is essentially the same as α → bool. The empty set is written by {}, none-empty sets are insert(a, insert(b, insert(c,{}))) (notation {a,b,c}). The comprehensions are denoted by {a . P[a]}, union by Un and intersection by Int (notation ∪, ∩). The union over a set is UNION: $\wp(\alpha) \times (\alpha \to \wp(\beta)) \to \wp(\beta)$ (notation $\bigcup_{a \in A} P(a)$). Membership is denoted by ∈.

More extensions possible, (e.g. by natural numbers Nat and by List; a number of axiomatizations for such structures can be found in [Paul 94]), which can be used as range of functions that *annotate* T-terms, for example, with a set of free variables, the depth of a term or

a "semantic value". We call this ensemble of theory-extensions as the *annotation domain* (*AD* for short) and HOL + AD as the *logical base*.



An instance of the transformation calculus with an object-language is built on top of the logical base. It consists of five theory-extensions: The definitions for generic terms and rule-terms (T+R), the axiomatic definition of the abstract syntax of the object-language (AS), pattern generating functions (called the *matching theory* M), annotation functions (called the *annotation theory* AN)and finally the full set of rule-terms.

Under the condition that certain the annotation functions are compatible with M and conservative w.r.t. data-types of the logical base, a semantics for a certain class of rule-terms is well defined.

Example: Encoding of an abstract syntax

Throughout this paper, we will build up a (fragment) of a logic and its calculi.

An encoding consists essentially of a set of type-assumptions for (sets of) constants C. Note that we can use a subset of C to represent the "variables" V of the object-language. These *object-level variables* underlay an own substitution.

Such a clear distinction between object-level variables V and the meta-variables MV is necessary in any framework that endeavours to encode object-languages with an own "binding-structure" (see [Talc 93]□) into an abstract syntax (let it be first-order or higher-order).

First, we have to state what type-constructors we want to use in the signature of the abstract syntax Σ_{AS}.

$$\{\text{seq, ide, iterm, wff, sequent }\} = \chi(\Sigma_{AS})$$

where χ is a function that yields all type constructors occurring in its argument (This statement is merely a comment). Then we state:

$$[<> : \text{seq } \alpha,\ \_::\_ : \alpha \mapsto \text{seq } \alpha \mapsto \text{seq } \alpha,\ x : \text{ide},\ \_' : \text{ide} \mapsto \text{ide},\ \_=\_ : \text{ide} \mapsto \text{ide} \mapsto \text{ide},$$
$$\text{Var} : \text{ide} \mapsto \text{iterm},\ \text{Con} : \text{ide} \mapsto \text{iterm},\ \text{Func}: \text{ide} \mapsto \text{seq iterm} \mapsto \text{iterm},$$
$$\text{Pred} : \text{ide} \mapsto \text{seq iterm} \mapsto \text{wff},\ \text{False} : \text{Wff},\ \text{And}: \text{wff} \mapsto \text{wff} \mapsto \text{wff},$$
$$\text{Or}: \text{wff} \mapsto \text{wff} \mapsto \text{wff},\ \text{Imp}: \text{wff} \mapsto \text{wff} \mapsto \text{wff},\ \text{All}: \text{ide} \mapsto \text{wff} \mapsto \text{wff},$$
$$□ : \text{sequent},\ \_\vdash\_ :\ \text{seq wff} \mapsto \text{seq wff} \mapsto\ \text{sequent}]\ = \Sigma_{AS}$$

We will employ lots of paraphrasing for this abstract syntax: instead of Func(var(x), var(x') :: var(x'') :: <>) we will simply write x(x',x'') or even f(a,b), since the indices to the x are an equivalent to anonymous deBruin-Variables. Imp(False,var(x)) is denoted by ff → x, sequences by the usual list notation.

The meaning of the empty sequent □ will become apparent later.

Pattern Generating Functions (Matching-Combinators)

For a transformation language that aims at a higher degree of abstraction and reusability of transformations, it is important to introduce a concept of "syntactical similarity" on terms of the object language. Moreover, our approach attempts to incorporate rule-schemata. Technically, "syntactical similarity" introduces some kind of congruence relation on the set of abstract syntax terms. This is done by a signature extension by symbols that do not represent a constructor, but a function.

Let *the symbol* ++ denote the concatenation on lists. A matching problem consists in finding a substitution σ, such that the equation $\langle X\ ++\ [c]\ ++\ Y\rangle\ \sigma = \langle a, c, b, c\rangle$ holds. An obstacle for finding this solution is the fact that ++ is not a free symbol. Hence, the resolution of the equation would be traditionally considered as a "E-matching"-problem , where E contains the equations defining the concatenation. Note that the choice of the substitution is not unique, such that we have another source of non-determinism in our framework.

"Extended rewriting" has been investigated for many theories, especially for theories with associative, commutative, idempotent operations or semi-groups and semi-lattices; a lot of ongoing research is still performed here. Nevertheless, the theories considered so far are very specific. Moreover, the matching-algorithms seem to be hard to combine and their complexity is nearly always at least NP-complete. But a close look at our problem reveals, that we are not interested in solving the matching-modulo-E problem

$$p\ \sigma\ \leftrightarrow_E^*\ t$$

it seems that with:

$$p\ \sigma\ \rightarrow_R^*\ t'\quad \leftarrow_R^*\ t$$

where R is a confluent and terminating term-rewriting system, we can achieve sufficient results (a detailed treatment of this kind of extended rewriting is contained in [SW 94]). The major advantage of matching modulo R (instead of E), is that the matching algorithm is now *parametrised* by R, such that R can be organised in a *modular, flexible* way.

Example: Combinators for a logic

We will specify R by the theory M. We will always require that all formulas in our matching theories are just conditional equations, and that M does not introduce new types wrt. AS, or put into different words: the matching theory must be an equational theory. Our example above is represented by:

$$\begin{aligned}
&[\ _{+}{+}_ && : \text{seq } \alpha \mapsto \text{seq } \alpha \mapsto \text{seq } \alpha,\\
&\ \ _[_\backslash_] && : \text{iterm} \mapsto \text{iterm} \mapsto \text{ide} \mapsto \text{wff},\\
&\ \ _[_\backslash_] && : \text{wff} \mapsto \text{iterm} \mapsto \text{ide} \mapsto \text{wff}] \in \Sigma_M;
\end{aligned}$$

Over $\Sigma_{AS} + \Sigma_M$, equations are formed as pairs of first-order terms (t_χ, t'_χ), denoted as $t_\chi \equiv t'_\chi$. We admit conditional equations: $t_1 \equiv t_1', \ldots, t_n \equiv t_n' \rightarrow t_{n+1} \equiv t_{n+1}'$. For our running example, we define for M:

$$
\begin{aligned}
&<> \mathbin{++} Y \quad \equiv \quad Y\\
&(a::X) \mathbin{++} Y \quad \equiv \quad a:: (X \mathbin{++} Y)\\
&x \neq y \rightarrow y[t \setminus x] \equiv y\\
&x[t \setminus x] \equiv t\\
&f(t_1,...,t_n)\ [t \setminus x] \equiv f(t_1\ [t \setminus x],...,t_n\ [t \setminus x])\\
&P(t_1,...,t_n)\ [t \setminus x] \equiv P(t_1\ [t \setminus x],...,t_n\ [t \setminus x])\\
&(p_1 \rightarrow p_2)\ [t \setminus x] \equiv p_1[t \setminus x] \rightarrow p_2[t \setminus x]\\
&x = y \equiv ff \rightarrow (\forall y.\ p)\ [t \setminus x] \equiv \forall y.\ (p\ [t \setminus x])\\
&(\forall x.\ p)\ [t \setminus x] \equiv \forall x.\ p
\end{aligned}
$$

As presented in [SW 94], these equations belong to a class for which the set of substitutions σ is finitary and can be constructed effectively by a narrowing technique. At the current stage, only first-order operation symbols are allowed (the semantic problems of higher-order constructors will become apparent in the next sections). However, higher-order matching-combinators are extremely appealing for program transformation. Let seqM be the list combinator with the type nat ↦ (nat ↦ term) ↦ seq term. We define seqM by:

$$
\begin{aligned}
&\mathrm{seqM}(0,f) \equiv []\\
&\mathrm{seqM}(\mathrm{succ}\ n,f) \equiv \mathrm{seqM}(n,f) \mathbin{++} [f(\mathrm{succ}\ n)]
\end{aligned}
$$

This combinator is particularly useful to describe sequences of terms matching the same pattern:

$$\mathrm{seqM}(n, \lambda i.\ C\ !\ i) \mathbin{++} \mathrm{seqM}(n,\ \lambda i.\ C\ !\ (n-i+1)) \Longrightarrow C!n$$

where the operator "!" is the selection in lists defined by $[x_1,...,x_n]\ !\ i = x_i$. This rule matches only palindromes with even length. Obviously, this combinator can be used to describe indexed rows of subpatterns. The rule-term could be embellished by the following notation:

$$[C_1 \ldots C_n, C_n \ldots C_1] \Longrightarrow C_n$$

Annotation Functions

Annotation constraints were formed over a set of functions that were associated to the set F. These functions map a "point in a term", i.e. a pair of a context-functions and sub term, to a value in AD. These functions were directly specified in HOL.

Since we are interested in the semantics and the semantical equivalence of rule-terms, we need proof-principles over annotation-constraints and, hence, annotation functions. Up to now, it is clearly impossible to have induction over "points" since the space of "context-functions" is too large to allow a generation principle.

A close look to the λ-abstractions we use to denote contexts reveals that we are not interested in "all" functions; projections into terms, for example, are excluded. The predicate $CXT:\Box(T \rightarrow T) \rightarrow bool$ together with the predicate CONST (is constant function) is a characterization for "context-functions" in our sense. The set of terms is implicitly restricted to $T_{\Sigma_{AS}}(X)$.

Definition: context-functions.

$$\mathrm{CONST}(C) \Leftrightarrow \exists c.\ \forall t.\ C(t) = c;$$

$$\begin{aligned}
&\mathrm{CXT}(C) \Leftrightarrow \neg\mathrm{CONST}(C) \wedge && \text{-- contexts must be nontrivial}\\
&\quad ((C = \lambda x.x) && \vee \quad \text{-- identity is empty context}\\
&\quad (C = \lambda x.\ (F\ x)\ (A\ x) && \Leftrightarrow\\
&\qquad (\mathrm{CXT}(F) \vee \mathrm{CXT}(A)))
\end{aligned}$$

Note, that CXT admits non-standard context such as $\lambda x.\ [x, x]$.

Over this data, we can define an induction principle, which has its roots (to our knowledge) in the context-induction by [Hen□88] and was refined to a noetherian context induction by [Bauer□94]. Bauers technique is based on the definition of a "maximal depth" of a context (written ML(C); defined analogously to CXT), i.e. the maximal length of the path from the "root to the replacement node". Since contexts can now be ordered in a well-founded ordering via ML, we can perform a Notherian induction over contexts: The induction step is described by:

Definition: Noetherian context-induction.

$$\begin{aligned}
&P(\lambda x.x) \quad \wedge\\
&\left[\forall C,C'.\ \mathrm{CXT}(C) \wedge \mathrm{CXT}(C') \wedge \mathrm{ML}(C') < n \wedge \mathrm{ML}(C) = n \wedge P(C') \quad \Rightarrow \quad P(C)\right]\\
\Rightarrow \quad &\forall\ C.\ \mathrm{CXT}(C) \Rightarrow P(C)
\end{aligned}$$

The proof of this context-induction principle can be done with a similar technique as the proof of the structural induction via fix-point-induction (see [Paul 84]), adapted to induction over typed terms.

According to our R-level typing requirement for annotation function symbols: $f\square:\square\forall\alpha.(\tau\mapsto\alpha)\mapsto\tau\mapsto\tau'$) where $f\in F$, the associated annotation functions are required to have the following HOL-type:

$$F_{AN} : (T \to T) \times T \to \tau_{AD}$$

The example in the next subsection will show, how annotation functions can be specified as a conventional algebraic, equational specification.

Lemma: Context-functions are injective.

$$\mathrm{CXT}(C) \Leftrightarrow (\forall x,x'.\ \neg x \equiv x' \Rightarrow\ \neg C(x) \equiv C(x')).$$

Fact: This only holds for Σ_{AS}-terms! Not every abstraction over R-terms is a context. Counterexample: $\lambda x.\ x$ ++ [a]...

Lemma: Contexts are compositional.

$$CXT(C) \wedge CXT(C') \rightarrow CXT(\lambda x.\ C(C'\ x))$$

Lemma: Finite composability into contexts:

$$\forall t.\ FIN\ (\{\ C.\ \exists t'.\ t = C\ t' \wedge CXT(C)\})$$

This second lemma suggests that the restriction of the function space by the CXT-predicate turns them to first-order objects.

Example: Annotation functions for a logic

The next two functions are classical, that have to be introduced for the control of logical deductions in an object-language that provides object-level variables and bindings. These functions are non-context-dependent functions, which is expressed by the fact that each variable for context-functions is independent and free.

Note that FV is overloaded w.r.t. R-level types.

$$FV(C,t) = FV_0(t)$$

where the auxiliary function FV_0 is defined by the equations:

$$FV_0(x) = \{x\}$$
$$FV_0(f(t_1,...,t_n)) = FV_0(t_1) \cup ... \cup FV_0(t_n)$$
$$FV_0(p(t_1,...,t_n)) = FV_0(t_1) \cup ... \cup FV_0(t_n)$$
$$FV_0(p_1 \rightarrow p_2) = FV_0(p_1) \cup ... \cup FV_0(p_2)$$
$$FV_0(\forall x.\ p) = FV_0(p) - \{x\}$$
$$FV_0(p :: S) = FV_0(p) \cup FV_0(S)$$

Analogously, we define the predicate FreeFor that decides when in all occurrences of x none of the variables in the variable set m is bound (hence, if m is assigned to the set of free variables $FV_0(t)$, this predicate decides if substitution can be carried out in a capture avoiding way). We present only the crucial equations:

$$FreeFor(C,t)(m,x) = FreeFor_0(m,x,t)$$

$$FreeFor_0(m,x,p(t_1,...,t_n)) = tt$$
$$FreeFor_0(m,x,p_1 \rightarrow p_2) = FreeFor_0(m,x,p_1 \rightarrow p_2) \wedge FreeFor_0(m,x,p_1 \rightarrow p_2)$$
$$FreeFor_0(m,x,\forall y.\ p) = x = y \ \vee\ x \neq y \ \wedge \neg(y \in m \wedge FreeFor_0(m,x,p))$$

The next function is a classical, really context-dependent function: the set of bound variables at an occurence in a term. Again, we present only the crucial equations:

$$BV(\lambda x.x,t) = \{\}$$
$$CXT(C) \rightarrow\ BV(C \circ \lambda x.p(t_1,...,x,...,t_n)),t) = BV(C,\ p(t_1,...,t,...,t_n))$$
$$CXT(C) \rightarrow\ BV(C \circ \lambda x.x \rightarrow p_2,\ p_1) = BV(C,\ p_1 \rightarrow p_2)$$
$$CXT(C) \rightarrow\ BV(C \circ \lambda x.\forall y.\ x,p) = BV(C,\ \forall y.\ p) - \{y\}$$

The next function, which will be used for a particular, context-sensitive logical calculus, collects the set of free variables that occur in the surrounding context in the premisses of implications.

$$\begin{array}{ll} & FVC(\lambda x.x,w) = \{\} \\ CXT(C) \rightarrow & FVC(C \circ \lambda x.p_1 \rightarrow x, p_2) = FVC(C,p_1 \rightarrow p_2) \cup FV(C,p_1) \\ CXT(C) \rightarrow & FVC(C \circ \lambda x.x \rightarrow p_2, p_1) = FVC(C,p_1 \rightarrow p_2) \\ CXT(C) \rightarrow & FVC(C \circ \lambda x.x \wedge p_2, p_1) = FVC(C,p_1 \wedge p_2) \\ CXT(C) \rightarrow & FVC(C \circ \lambda x.p_1 \wedge x, p_2) = FVC(C,p_1 \wedge p_2) \\ CXT(C) \rightarrow & FVC(C \circ \lambda d.\forall x.\ d, p) = FVC(C, \forall x.\ p) - \{x\} \end{array}$$

The style of these definitions resembles very much to the conventional "attribute grammars", where the context-sensitive functions correspond to "inherited attributes", while the non-context-dependent functions correspond to "synthesised attributes" (cf. [Wolff 94]).

Semantic interpretations[1]

Since the matching theory is restricted to an equational theory, there is always a Herbrand model ("initial algebra") for these equations, i.e. there is a type indexed family of carriersets:

Definition: Term carrier set[2] D.

$$D_S = A_{\chi 1} \oplus ... \oplus A_{\chi n} \qquad\qquad D = D_S \oplus (D_S \rightarrow D)$$

and an interpretation function $\mathsf{Sem}_0 : T \rightarrow D$ that gives any ground term $t_\tau \in T$ a value in D_τ. Sem_0 is extended to the non-ground interpretation $\mathsf{Sem} : T \rightarrow (MV \rightarrow D) \rightarrow D$ which is required to make all equations in M true in all environments γ: $MV \rightarrow D$. Because of the existence of the Herbrand-model, this Sem_0 and Sem exist.

The construction of the domain for our "higher-order-relation" rule-terms is performed in two steps: We introduce the pre-domain D_R and form the powerset over it. D_R is defined by:

Definition: rule-term pre-domain.

$$D_R = D_S \oplus (D_R \times D_R)$$

Definition: Injection I from D to $\wp(D_R)$.

$$\begin{array}{ll} I : D \rightarrow \wp(D_R) & \\ I(x) = \{x\} & \text{if } x \in D_S \\ I(x) = \bigcup_{d \in D_S} \{(d, m)\ .\ m \in I(x\ d)\} & \text{if } x \in D_S \rightarrow D \end{array}$$

---

1 Note that this presentation does not yet include a proper treatment of Milner-Polymorphism.

2 There is no solution for recursive universes like $D = D_S \oplus (D \rightarrow D)$ in HOL - this is our main motivation for using the significantly simpler universe of the essentially first-order functions. An alternative would be the restiction of $\rightarrow$ to the usual continuous function space; this would lead to a universe construction similar to [Reg 94], chapter 5.

In fact, even the inverse of I exists and the two domains D and $\wp(D_R)$ are isomorphic. The reason for the introduction of D was merely a psychological one: All constants can always be interpreted as (essentially first order) functions, and interpretations of T-terms can always be understood as functional interpretations and all arguments over existence of Herbrand-models are standard.

Why is $\wp(D_R)$ sufficiently large for higher-order relations in our sense? This is a consequence to the fact that we aim only to model a relation-space that we can syntactically represent. Higher-order objects have to be represented by arrow-abstractions, and the relation they induce have to be producible by substitutions (matchings).

The semantic interpretation for rule-terms and for formulas:

$$Sem_R : R \to (MV \to \wp(D_R)) \to "(T\to T)" \to \wp(D_R)$$

$$Sem_\Phi : \Phi \to (MV \to \wp(D_R)) \to "(T\to T)" \to "T" \to bool$$

where "(T→T)" and "T" are a type synonym for $\wp(D_R \times D_R)$ resp. D_R. We proceed as follows:

Definition: rule-term and formula interpretation.

$$Sem_R[\![c]\!]\gamma\, C = I\,(Sem_0[\![c]\!])$$
$$Sem_R[\![v]\!]\gamma\, C = \gamma\, v$$
$$Sem_R[\![r\ r']\!]\gamma\, C = \{\ t'\ .\ (t,t') \in Sem_R[\![r]\!]\gamma\,(\Lambda d.C(t\ d)) \wedge t \in Sem_R[\![r']\!]\gamma\,(\Lambda d.C(d\ t'))\ \}$$
$$Sem_R[\![r \Rightarrow r']\!]\gamma\, C = \{\ (t,t')\ .\ \ t \in Sem_R[\![r]\!]\gamma\,\Lambda d.d \wedge t' \in Sem_R[\![r']\!]\gamma\,\Lambda d.d\}$$
$$Sem_R[\![r\ [\![\Phi]\!]]\!]\gamma\, C = \{\ t\ .\ t \in Sem_R[\![r]\!]\gamma\, C\ \wedge\ Sem_\Phi[\![V\,\Phi]\!]\gamma\, C\ t = true\ \}$$
$$Sem_R[\![[v_\tau]\ r]\!]\gamma\, C = \bigcup_{d \in D_\tau} Sem_R[\![r]\!]\gamma^d_v\, C$$

$$Sem_\Phi[\![\pi(t_1,...,t_n)]\!]\gamma\, C\ t = \forall d_1...d_n.\ [\![\pi]\!]\,(d_1...d_n)$$
where $d_i \in Sem_R[\![t_i]\!]\gamma\, C$ if $t_i \in R$ and
$d_i = [\![t_i]\!]\,(C,t)$ otherwise

$$Sem_\Phi[\![\neg\Phi]\!]\gamma\, C\ t = \neg\ Sem_\Phi[\![\Phi]\!]\gamma\, C\ t$$
$$Sem_\Phi[\![\Phi_1 \wedge \Phi_2]\!]\gamma\, C\ t = Sem_\Phi[\![\Phi_1]\!]\gamma\, C\ t\ \wedge\ Sem_\Phi[\![\Phi_1 \wedge \Phi_2]\!]\gamma\, C\ t$$
$$Sem_\Phi[\![\Phi_1 \vee \Phi_2]\!]\gamma\, C\ t = Sem_\Phi[\![\Phi_1]\!]\gamma\, C\ t\ \vee\ Sem_\Phi[\![\Phi_2]\!]\gamma\, C\ t$$
$$Sem_\Phi[\![\Phi_1 \Rightarrow \Phi_2]\!]\gamma\, C\ t = Sem_\Phi[\![\Phi_1]\!]\gamma\, C\ t\ \Rightarrow\ Sem_\Phi[\![\Phi_2]\!]\gamma\, C\ t$$

The paraphrase "Λd.C(t d)", "Λd.C(d t')" and "Λd.d" are a notation for the set in $\wp(D_R)$ equivalent to these functions. This sets exist since contexts are restricted to second order. Moreover, we assume for any predicate symbol π a boolean predicate [[π]] with suitable arity. Analogously, we assume for any annotation-function symbol F an interpretation [[F]].

The treatment of contexts in higher-order rules deserves some attention. The definition for the application r r' simply appends the local context of the functor resp. the argument within the application to the surrounding context C. We say: *applications construct contexts*. However, for

the abstraction $r \Rightarrow r'$, the semantic value does not depend on the surrounding context at all; the value of the pattern and the target is interpreted in an empty context – which is due to the fact that the rewrite of a term in our approach is only defined at its root and rewrites in a sub term have to be represented by first transforming the rule. We say *rules form contexts*.

Furthermore, note that it is possible annotated rule-terms occur in annotation constraints: $r[\![r'[\![\Phi]\!] = r'']\!]$. The question arises, what is the context this inner constraint refers to. One possibility is to consider the constraint bracket $[\![\Phi]\!]$ as context-forming, another is to ignore them, i.e. giving them the semantics for $[\![true]\!]$. We chose the latter possibility, since we get a more general substitution theorem (see Theorem 4.3) this way.

Lemma 1: Annotation free r-terms are context independent.

$$Sem_R[\![V\ r]\!]\gamma\ C = Sem_R[\![V\ r]\!]\gamma\ C'$$

Lemma 2: Annotation-constraints decrease relations.

$$Sem_R[\![\ r\ ]\!]\gamma\ C \subseteq Sem_R[\![V\ r]\!]\gamma\ C$$

Lemma 3: Constraints are redundant in the context of constraints.

$$Sem_R[\![\ r\ [\![\Phi]\!]\ ]\!] = Sem_R[\![\ r\ [\![\ V\ \Phi]\!]\ ]\!]$$

Theorem 4.1: Constraint Approximation

$$Sem_R[\![\ r\ r']\!]\gamma\ C = \{\ t'\ |\ (t,t') \in Sem_R[\![\ r]\!]\gamma\ (\lambda d.C(t_0\ d)) \wedge t \in Sem_R[\![\ r']\!]\gamma\ (\lambda d.C(d\ t'_0)) \wedge (t_0,t'_0) \in Sem_R[\![V(R)]\!]\gamma\ (\Lambda d.d)\ \}$$

Theorem 4.2: Embedding compatibility.

$$Sem_R[\![I\ t]\!]\gamma\ C = I(Sem[\![t]\!]\gamma)$$

Theorem 4.3: Substitution theorem:

$$Sem_R[\![\ r\ \{V\ r' \setminus v\}\ ]\!]\ \gamma\ \ C = Sem_R[\![\ r\ ]\!]\ \gamma_v^d\ C \quad \text{where } d = Sem_R[\![V\ r']\!]\gamma\ C$$

$$Sem_\Phi[\![\Phi\{V\ r'\ \setminus v\}]\!]\ \gamma\ C\ t\ = Sem_\Phi[\![\Phi]\!]\ \gamma_v^d\ C \quad \text{where } d = Sem_R[\![V\ r']\!]\gamma\ C$$

The restriction of the substitution to annotation free terms is due to the fact that the substitution changes per definition the context of a constraint. Hence, we can not expect that this context sensitive construct is semantically invariant under substitution.

Since in the context of annotation-constraints $Sem_\Phi[\![\Phi]\!] = Sem_\Phi[\![V\ \Phi]\!]$ holds, the substitution theorem for Sem_Φ can be generalized to arbitrary substitutions in such contexts:

$$Sem_R[\![\ r\ [\![\Phi]\!]\ \{V\ r' \setminus v\}]\!] = Sem_R[\![\ r\ \{V\ r' \setminus v\}[\![\ \Phi\ \{r' \setminus v\}]\!]\ ]\!]$$

This gives rise to another interpretation (and a particular implementation) of annotation constraints: $r[\![\Phi]\!]$ could be represented by a triple (r, ϕ, σ), where $\phi\ \sigma = Sem_\Phi[\![\Phi\ \sigma]\!]$. The substitution σ can be interpreted as closure, allowing the redefinition of the substitution by:

$$(r, \phi, \sigma)\ \sigma' = (r\ \sigma', \phi, \sigma\ o\ \sigma')$$

This interpretation of annotation constraints makes the definition of a concrete syntax for formulas superfluous – it suffices to consider ϕ as context-dependent boolean function defined in the meta-logic. This way, however, it is difficult to motivate the complicated dependencies of ϕ, which has the type: $(MV \to R) \to (MV \to \wp(D_R)) \to (T \to \alpha) \to T \to bool$.

V. Consistency conditions

Conservativity of M w.r.t. AS

Let $[t]_M$ be the congruence class of t: $\{ t' . t \equiv t' \}$.

Definition: M is a conservative extension of T.

To prove: $\forall t. \exists! t' \in T. I(t') \in [t]_M$.

Criterion: This is the case if

1) M is an orthogonal convergent rewrite system and

2) All normal forms of M are AS-terms.

Conservativity of AN w.r.t. AD

"Stability wrt. matching theory".

Definition: Annotation functions F are compatible with the congruence induced by $\equiv_M$.

$\forall F \in \Sigma. \forall C,C', t,t'. (C \equiv_M C' \wedge t \equiv_M t') \Rightarrow F(C,t) = F(C',t')$

Example:

No Problem: FV over ++:

$FV_0(A ++ B) = FV_0(A) \cup FV_0(B)$

Relevant question here.

Problem: FV over substitutions.

$FV(C, t[t' / x]) = ???$

Possible, but difficult. And no one would yearn for such a mess. Methodology may be too rigid here. Individualise as in the case of Context-Coherence? Or another reason to incorporate the o-level substitution [t/x] into the framework.

Context-Coherence of Rules

Definition: Context coherence of an annotation in a rule-term.

$$r[r' [\![\Phi]\!]] \Rightarrow \forall \sigma.\ \forall\ C \in T \rightarrow T.\ \ I(C) \equiv \lambda x.r\ \sigma[x]\ \Rightarrow CXT(C)$$

Example:

$X[\![\Phi]\!]$ ++ Y is not coherent, since $\sigma=\{X \mapsto <>\}$ leads to constant functions, not contexts.

Similar: swap($X[\![\Phi]\!]$, [a,b]).

Fortunately: $[a[\![\Phi]\!]]$ ++ Y = $(a[\![\Phi]\!] :: <>)$ ++ X is a context coherent expression (and motivates, why an rule-dependent consistency condition is adequate here).

An object-language theory (i.e. AS + M + AN+ R) is consistent iff M is conservative w.r.t. AS, AN is conservative w.r.t. M and the collection of rules in R are context-coherent.

VI. The Calculus of Transformations

We are now able to present a conservative extension of HOL. A proof in it derives the validity of a statement t(T)t' under the theories M and AN. Applicability conditions are split off and have to be proved with HOL.

The predicate transformer ...$\langle\phi\rangle$: $(T \rightarrow T) \times T \rightarrow$ bool is defined by:

$$\langle\phi\rangle\ (C,t) = \forall\gamma.\ Sem_{\Phi}[\![\phi]\!]\ \gamma\ C\ t$$

Furthermore, note that the rule-terms $r[\![\Phi]\!]$ and $r[\![\Phi']\!]$ are considered equal iff V Φ and V Φ' are identical.

On this basis, the set of decomposition-meta-inferences of TC is defined as follows:

$$a\ (a \Rightarrow b)b \qquad \text{(Trafo-Appl)}$$

$$\frac{AN,\Box M\Box\vdash\Box t(R)t' \qquad AN,\Box M\Box\vdash\ u(C[t'])v}{\Box \qquad AN,\Box M\Box\vdash\ u(C[(R\Box t)])v} \qquad \text{(Beta)}$$

$$\frac{\Box\vdash_{M\Box\Box} u\Box\equiv\Box v,\Box AN,\Box M\Box\Box\vdash\Box t(C[u\Box\sigma])t'\Box\Box}{\Box AN,\Box M\Box\Box\vdash\Box t(C[v\Box\sigma])\Box t'\Box} \quad \text{(Eval-Rule)}$$

$$\frac{AN,\Box M\Box\Box\vdash\Box t(T\Box\sigma)t'}{\Box\Box\Box AN,\Box M\Box\Box\vdash\Box t(T)t'\Box} \quad \text{(Substitution)}$$

$$\frac{AN,\Box M\Box\vdash\Box t(C[T\sigma])t'}{\Box\Box\Box AN,\Box M\Box\vdash\Box t(C[[x]T])t} \quad \sigma=\{x/y\} \quad \text{(Scope-Elim)}$$

$$\frac{\vdash_{HOL+AN}\Box\langle\phi\rangle(C,t) \quad AN,M\Box\vdash\Box t(I\Box C[r])t'\Box\Box V(r)\Box=\Box I\Box t}{AN,\Box M\Box\vdash\quad t(I\Box C[r[\![\phi]\!]])t'\Box} \quad \text{(Split constraint)}$$

Theorem 6.1: Relative Conservativity.

TC is a conservative extension of HOL+AD+AS if the object-language theories are consistent.

Theorem 6.2: Correctness of TC.

$$\vdash_{TC} A(R)B \quad \rightarrow \quad \forall\gamma.\ \forall a,b.\ a\in Sem_R[\![A]\!]\ \gamma\ \lambda d.d\ \wedge\ b\in Sem_R[\![B]\!]\ \gamma\ \lambda d.d \Rightarrow (a,b)\in Sem_R[\![R]\!]\ \gamma\ \lambda d.d$$

Meta-Inference: Induction (cf. "Decomposition", [Pep 84]):

$$\frac{\begin{array}{l}AN,\ M\ \vdash t\{c_{i\chi}\setminus x_\chi\}(r\ \{c_{i\chi}\setminus x_\chi\})t'\{c_{i\chi}\setminus x_\chi\}\\ AN,M,u<<t,\{u_\chi\setminus x_\chi\}(r\{u_\chi\setminus x_\chi\})t'\{u_\chi\setminus x_\chi\}\Box\vdash\Box t\{u_\chi\setminus x_\chi\}(r\Box\{u_\chi\setminus x_\chi\})t'\{u_\chi\setminus x_\chi\}\end{array}}{\Box AN,\Box M\Box\Box\vdash\Box t(r)\Box t'\Box} \quad (\textit{NoetherInd})$$

Theorem 6.3: Correctness of transformation induction.

Meta-Inference: Constraint equivalence.

$$\frac{\begin{array}{l}AN \vdash_{HOL} CXT(C) \rightarrow \langle\phi\rangle(C,t)=\langle\phi'\rangle(C,t) \\ AN,\Box M\Box\vdash\Box t(I\Box C[r[\![\phi']\!]])t'V(r)\Box=\Box I\Box t\end{array}}{AN,\Box M\Box\vdash\quad t(I\Box C[r[\![\phi]\!]])t'\Box} \quad (\textit{ConstrEquiv})$$

Theorem: Correctness of constraint equivalence.

VII. Transformations as a Metalogical Framework

Encodings

It is well-known that a logic can have different proof-calculi. For first-order logic, for example, Hilbert-style and several Gentzen-style calculi (natual deduction NJ, or sequent-calculi). The following system is a Gentzen-Sequent-style system for intuitionistic first-order logic, called LJ (the presentation follows [BC 92]).

$$(\textit{axiom})\ \frac{}{\Gamma, A \vdash A} \qquad (\textit{falseE})\ \frac{}{\Gamma, ff \vdash G}$$

$$(\textit{AndI})\ \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A\wedge B} \qquad (\textit{AndE})\ \frac{\Gamma, A, B \vdash G}{\Gamma, A\wedge B \vdash G}$$

$$(\textit{OrIL})\ \frac{\Gamma \vdash A}{\Gamma \vdash A\vee B} \qquad (\textit{OrIR})\ \frac{\Gamma \vdash B}{\Gamma \vdash A\vee B} \qquad (\textit{OrE})\ \frac{\Gamma, A \vdash G \quad \Gamma, B \vdash G}{\Gamma, A\vee B \vdash G}$$

$$(\textit{ImpI})\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A\rightarrow B} \qquad (\textit{ImpE})\ \frac{\Gamma, A\rightarrow B \vdash A \quad \Gamma, B \vdash G}{\Gamma, A\rightarrow B \vdash G}$$

$$(\textit{AllI})\ \frac{\Gamma \vdash P}{\Gamma \vdash \forall x.P}\ (*) \qquad (\textit{AllE})\ \frac{\Gamma, P[t / x] \vdash G}{\Gamma, \forall x.P \vdash G}\ (**)$$

The last two rules have side-conditions. In the first, we insist that "x does not occur free in Γ" holds. In the second, we must have that "no substitutions of x take place into occurrences, where a free variables in t occurs bound". The latter condition is represented by the predicate FreeFor(t,x,P).

LJ is now represented in backward reasoning-style, called LJb. The calculus will produce the usual tree-like structure as a consequence of *AndI* and *OrI*. It is using sequences instead of multisets and ++ as matching combinator on them:

$$\Gamma{+}{+}[A]{+}{+}\Delta \vdash A \Rightarrow \square \qquad (\textit{axiom})$$

$$\Gamma{+}{+}[ff]{+}{+}\Delta \vdash \Gamma \Rightarrow \square \qquad (\textit{falseE})$$

$$(\Gamma \vdash A \Rightarrow \square) \Rightarrow \Gamma \vdash A \wedge B \Rightarrow \Gamma \vdash B \qquad (\textit{AndI1})$$

$$(\Gamma \vdash B \Rightarrow \square) \Rightarrow \Gamma \vdash A \wedge B \Rightarrow \Gamma \vdash A \qquad (\textit{AndI2 sym. case})$$

$$\ldots$$

$$\Gamma \vdash \forall x.P \Rightarrow \Gamma[\![\, x \notin FV]\!] \vdash P \qquad (\textit{AllI})$$

$$t[\![m = FV]\!] \Rightarrow \Gamma{+}{+}[\forall x.P[\![\, FreeFor(x,m,t)]\!]]{+}{+}\Delta \vdash G$$

$$\Rightarrow \Gamma{+}{+}[P[t / x]]{+}{+}\Delta \vdash G \qquad (\textit{AllE})$$

The condition, that no free V-variable must occur on the right hand side of a rule, forces the introduction of additional parameters in a rule. These parameters can be substitutions, or complete proofs. A comparison of the forward reasoning style version of axiom:

$$\Gamma \Rightarrow A \Rightarrow \square \Rightarrow \Gamma, A \vdash A$$

with the backward style reasoning demonstrates, that significantly more information has to be provided in this style (either by a unification process or by a user that controls the derivation). This set of rules has to be completed with:

$$(A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow A \Rightarrow C \qquad (\textit{composition})$$

which is usually left implicit (and implemented by the function composition on the meta-language level).

Since both side-conditions are built upon functions, that depend not on the context, the whole machinery for context-sensitivity looks overdone. This is a consequence of the design of this calculus, that incorporates the paradigm of inductive definitions instead of term-contexts.

We will present LJ in another, different calculus style, called TS:

$$(B \wedge C \rightarrow C) \Rightarrow tt \qquad (\textit{axiom})$$

$$(B \wedge ff \rightarrow C) \Rightarrow tt \qquad (\textit{falseE})$$

$$(C \rightarrow A \wedge B) \Rightarrow (C \rightarrow A) \wedge (C \rightarrow B) \qquad (\textit{AndI1})$$

$$(C \rightarrow A \vee B) \Rightarrow (C \rightarrow A) \qquad (\textit{OrIL})$$

$$(C \rightarrow A \vee B) \Rightarrow (C \rightarrow B) \qquad (\textit{OrIR})$$

$$(C \wedge (A \vee B)) \rightarrow G) \Rightarrow ((C \wedge A) \rightarrow G) \wedge ((C \wedge B) \rightarrow G) \qquad (\textit{OrE})$$

$$(C \rightarrow A \rightarrow B) \Rightarrow (C \wedge A \rightarrow B) \qquad (\textit{ImpI})$$

$$(C \wedge (A \rightarrow B)) \rightarrow G) \Rightarrow (C \wedge A \rightarrow B) \rightarrow A) \wedge (C \wedge B \rightarrow G) \qquad (\textit{ImpE})$$

$$C \rightarrow \forall x.P \quad \Rightarrow \quad C[\![\, x \notin FV \wedge x \notin FVC]\!] \rightarrow P \qquad (AllI)$$

$$t[\![m = FV]\!] \quad \Rightarrow \quad C \wedge \forall x.P[\![\, FreeFor(x, m \cup BV, t)]\!] \rightarrow G$$

$$\Rightarrow C \wedge [P[t / x]] \rightarrow G \qquad (AllE)$$

Furthermore, we need the structural rules:

$$B \wedge tt \Rightarrow B$$
$$B \wedge ff \Rightarrow ff$$
$$B \vee tt \Rightarrow tt$$
$$B \vee ff \Rightarrow B$$
$$B \wedge C \Rightarrow C \wedge B$$
$$B \vee C \Rightarrow C \vee B$$

which can be mimicked by rewriting-modulo-C1 or suitable matching-combinators (see [SW 94]). The most dramatical difference of this calculus to the previous, induction-oriented version consists in the combination-rule:

$$C \Rightarrow (A \Rightarrow B) \Rightarrow C\ A \Rightarrow C\ B$$

This congruence-rule (which is the gain for the additional, really context-sensitive constraint $x \square\!\!\notin FVC$ in *AllI*), that replaces the sequential composition, is the quintessence of a "transformation-style" calculus. It is a rewrite-oriented, single-threaded proof-style being the basis for an access to rewrite-theory, rewrite-strategies and rewrite implementation-techniques. Note that a derived calculus with more context-sensitive constraints can be developed for more powerful tactical programs formed over elementary rules, allowing the efficient construction of normal forms for particular classes of formulas (see [KLSW 94]).

Sample Proofs

Theorem: Equivalence of LJb and TS.

VIII. Summing up

Related Work

We'd like to relate our work to two major streams in logic and computer science — the one is the search for basic calculi in logics as a foundation of formal program development, the other is representing logical languages in some meta-logical framework.

With respect to the second issue, a comparison with meta-logical frameworks like LF, Isabelle, NuPRL, CC, ECC or DEVA (see introduction) is unavoidable here. These approaches are based on the "internalization of object-variables", i.e. the object-level variables were represented by the variables of the meta-language: A universal quantification $\forall x.P$ is represented by an expression in the meta-language

$$\text{all } (\lambda x.P)$$

where the function all : $(\alpha \rightarrow \text{bool}) \rightarrow \text{bool}$ can be interpreted as the *function* that yields true if its argument function denoted by the meta-level abstraction $\lambda x.P$ yields true for all values in α. Hence, the body of a quantification is not represented syntactically, but *extensionally*. Systems like Isabelle qualify functions like "all" as a "binder" and provide syntactical sugar for it. The advantage of this technique, which goes back to Church and is often referenced as "higher order abstract syntax", is tremendous: the encoding inherits the full machinery for representing binding, substitution and typing from the meta-language. This is the major reason for an extremely concise, compact representation of most "object-logics" in Isabelle, for example.

The problem is that it is not always adequate of abstracting away variables — sometimes it is necessary to describe that a variable "occurs in" an expression, and not only the function denoted by that expression is constant in that variable. Also the number of variables may be relevant — when deciding that a rule is "linear" or "destructive" (cf. [Klop 92]), which are of major importance in the context of computing normal forms and tactical optimisations, when deciding syntactical predicates like "admissability" in LCF or similar situations in MODAL logics (see [MM 93]), or when bridging the gap from logical deductions to program transformations that form together with tactical control a compiler (see [GS 90]).

The issue of explicit variables leads back to our first subject: the comparison of calculi in a similar style. The set of rule-terms in our calculus has a strong similarity to Oostroms Λ–calculus (cf. [Oost 90]). He extends the usual abstraction to $\Lambda E.E$-terms, interpreting the first expression as patterns. β- and η-reductions are generalised in the obvious way. In contrast to our work, Oostroms approach is untyped and has to impose therefore several restrictions in order to assure the consistency of the calculus.

The important connection to combinatorical reduction systems ([Klop 80]) deserves some attention. Klop's system incorporates a notion of rule with explicit binding structures, distinguishing Meta-Variables from (object)-variables like our framework. The encoding of the λ-calculus and several other calculi was demonstrated, and important results like "orthogonality

implies confluence" (leaving the "church-rosser theorem" of the λ-calculus as a corrollary!) could be proven for all object-languages in the framework. In a way, our work can be seen as an approach to combine CRS with an explicit treatment of contexts through higher-order rules, extended matching and context-sensitivity.

The work of Talcott [Talc 93] provides an abstract algebra for a theory of "binding structures", comprising a generic abstract syntax with object-and meta-variables and substitutions for both levels. The underlying data-structure can be used as an implementation for CRS, but a formal machinery for investigating rewriting and unification too.

Recently, there has been growing interest for calculi with explicit variables ([ACCL 91], [Lesc□94]). The idea is to enrich the abstract syntax of the λ-calculus by a substitution closure (written [s]) and additional shift -and composition combinators on them. Several first-order rewrite rule reducing substitutions were added to the (first-order) beta-reduction rule: $(\lambda a)b \rightarrow a[b \,.\, id]$. The main motivation here was originally an efficient implementation of the λ-calculus, yielding a foundation for "lazy evaluations of substitutions", but it turns out that the resulting system is also accessible for elegant mathematical investigation (Lescanne condensed the system to 7 orthogonal rules and offered termination proofs for the substitution theory).

Further Work

The relation between the λ-sub calculus and TC needs more investigation. Especially, we don't understand the connection between the (usually very rich) universes for typed λ-calculi and our domain for R for rule-terms. Moreover, our framework clearly needs simplification. One possibility might be the integration of "context-coherence" into type-inference.

The "raison d'être" for a logical framework is to establish a collection of common properties for a class of instance-logics out of a collection of notions and definitions. Obviously, there is a trade-off between the power of the common theory and the variety of logics that can be adequately represented - and the elegance of this representation.

We are not satisfied with the trade-off so far. We believe that both substitutions (for object-level variables C and meta-variables V) should be incorporated within the framework - requiring some analogue for "binding structures" as in Talcott's work.

Criteria for *executable* transformations are of major importance. While imposing some syntactic restrictions on the object language (M must be an ACB-theory, see [SW 94]), the annotation constraints (propositional) and AN-theories (strongly non-circular attributations, see [Wolff□94]), one gets transformations where applicability becomes decidable, and where powerful mechanisms for the construction of substitutions and evaluation of applicability conditions are available.

In classical transformation frameworks, it has been argued that transformations are by nature non-executable, because one wants to admit "applicability conditions as proof obligations". This is not necessarily true. If one designs object-languages such that transformations reproduce applicability conditions with the character of proof-obligations as premises of an impli-

cation in the target, further transformational deductions have to eliminate this premise (see [Wolff 94b]). This way, one gets a *uniform* formal framework, not splitting apart transformational deduction and theorem proving.

Conclusion

We have presented a notion of higher-order, context-sensitive and schematic transformation rules, that extends the notion of [CIP 85] and [HK 93] while still providing a simple formal framework. The framework is especially designed for instances of object languages in which variables are externalised. Our approach of the treatment of contexts (indispensable for languages with object-level bindings) resolves several shortcomings of the approach of CIP (cf. [Pep 84]) and PROSPECTRA. The framework supports a (to our knowledge) new, single-threaded, rewriting-oriented deduction style of object-logics, that seems to be more appropriate for the methodical construction of simplification procedures and the reuse of proofs.

Induction still has been integrated in the meta-inference system in order to allow equivalence proofs over rules.The power of the theory has still to be explored.

References

[ACCL 91] M. Abadi, L.Cardelli, J. Lévy: Explicit Substitutions. J. of Functional Programming, 1(4), pp. 375-416, 1991.

[Andr 86] P.B. Andrews: An Introduction to Mathematical Logic and Type Theory: To Truth through Proof. Academic Press, 1986.

[Bach 91] Bachmair, L.: Canonical Equational Proofs. Progress In Theoretical Computer Science, Birkhäuser, 1991.

[Bar 93] H, Barendregt: Introduction to Generalised Type Systems. To appear in the Journal of Functional Programming.

[Bas 94] D. Basin. Logic frameworks for logic programs. In 4th International Workshop on Logic Program Synthesis and Transformation, LOPSTR '94. Springer-Verlag, 1994. To appear.

[Bauer 94] B. Bauer: Attributed Term Induction - A Proof Principle for Attribute Grammars, Technical Report, Technische Universität München, TUM-I9403, 1994.

[BC 92] D. Basin, R. Constable: Metalogical frameworks. In: G. Huet, G. Plotkin and C. Jones (eds.): Proc. Second Workshop on Logical Frameworks, Edinburgh,pp. 47-72, 1991.

[BM 92] R.S. Bird, O. de Moor: Lecture Notes on the Calculus of Sets. Lecture Notes of the STOP-1992-Summerschool on Constructive Algorithmics, Ameland, 1992.

[Brück 94] B. Krieg-Brückner: Programmentwicklung durch Spezifikation und Transformation. Bremer Beiträge zum Verbundprojekt KORSO. Technischer Bericht 1/94, Universität Bremen. 1994.

[BW 84] F.L. Bauer, H. Wössner: Algorithmic Language and Program Development, Springer Verlag, 1984.

[BW 89] R.J.R Back, J. v. Wright: Refinement Calculus, Part I: Sequential Nondeterministic Programs. In: Stepwise Refinement of Distributed Systems, LNCS 430, pp 42–66, 1989.

[CH□88] T. Coquand, G. Huét: The Calculus of Constructions. Information and Computation, 76:95 – 120, 1988.

[Chu 40] A. Church: A Formulation of the Simple Theory of Types. J. of Symbolic Logic 5 (1940), pp. 56-68, 1940.

[CIP 85] Bauer et al. (The CIP Language Group): The Munich Project CIP. Volume I: The wide spectrum language CIP-L. LNCS 183.1985.

[CO 92] A. Cant, M.Ozols. A verification environment for ML programs. In Peter Lee, editor, Workshop on ML and its Applications, pages 151-156, San Francisco, California, ACM SIGPLAN. June 1992.

[DJ 90] N. Dershowitz, J-P. Jouannaud: Rewrite Systems. In: Handbook of theoretical Computer Science. Elsevier, 1990.

[DM 82] L. Damas, R. Milner: Principal Type Schemes for Functional Programs. POPL□8, Jan. 1982.

[GS 90] W. Grieskamp, W. Schulte: Termersetzung für den OPAL-Compiler. (in german, unpublished). 1990.

[Hen□88] R. Hennicker: Beobachtungsorientierte Spezifikationen. Dissertation, Fakultät für Mathematik und Informatik, Universität Passau, 1988.

[HK 93] B. Hoffmann, B. Krieg-Brückner (eds.): PROgram development by SPECification and TRAnsformation: Vol. I: Methodology, Vol. II: Language Family, Vol. III System. Prospectra Reports M.1.1.S3-R-55.2, -56.2, -57.2. LNCS 680. Universität Bremen, 1993.

[HST 89] R.Harper, D. Sanella, A. Tarlecki: Logic Representation in LF. In: Proc. 3rd Summer Conf. on Category Theory and Computer Science, Manchester, pp. 250-272 in LNCS 389, 1989.

[Klop 80] J. W. Klop: Combinatory Reduction Systems. Phd. Mathematical Centre Tracs 127, 1980.

[Klop 90] J.W. Klop: Term Rewriting Systems. Tech-Report CS-R9073, CWI, Amsterdam. A condensed version also appeared in: S. Abramski, Dov. M. Gabbay, T.S.E. Maibaum (eds): Handbook of Logic in Computer Science, Chapter 1. Oxford Science Publications, 1992.

[KLSW 94] Krieg-Brückner, B., Liu, J., Shi, H., Wolff, B.: Towards Correct, Efficient and Reusable Transformational Developments. in: Broy, M., Jähnichen, S. (eds.): KORSO, Correct Software by Formal Methods, *LNCS* (1994).

[Lesc 94] P. Lescanne: From λσ to λυ – a journey through calculi of explicit substitutions. POPL 94, pp. 60-69, 1994.

[Mese 92] J. Meseguer: Conditional Rewriting Logic as a Unified Model of Concurrency, Theoretical Computer Science 96, pp. 73-155, 1992.

[MM 93] N. Martí-Oliet, J. Meseguer: Rewriting-Logic as a Logic and Semantic Framework. Technical Report SRI-CSL-93-05, Comp. Sci. Lab., SRI Int., 1993.

[MM 94] N. Martí-Oliet, J. Meseguer: General Logics and Logical Frameworks.In: D. Gabbay (ed): What is a Logical System? Oxford University Press, 1994.

[Oost 90] V. van Oostrom: Lambda Calculus with patterns. CWI, Amsterdam. unpublished paper. 1990.

[Paul 84] L.C. Paulson: Deriving Structural Induction in LCF. LNCS 173, 1984.

[Paul 87] L.C. Paulson: Logic and Computation: Interactive proof with Cambridge LCF. Cambridge University Press, 1987.

[Paul 89] L. C. Paulson: The foundation of a generic theorem prover. Journal of Automated Reasoning, 5:363-397, 1989.

[Pep 84] P. Pepper: A Simple Calculus for Program Transformations (Inclusive of Induction), Technische Universität München, Technische Universität München, TUM-INFO-07-84-IO9-280/1-FMA, 1984.

[Reg 94] F. Regensburger: HOLCF: Eine konservative Einbettung von LCF in HOL. Phd thesis, Technische Universität München.1994.

[SW 94] Shi, H., Wolff, B.: A Finitary Matching Algorithm for Constructor Based Theories. KORSO Bericht, FB3 Informatik, Universität Bremen, 1994 (in: [Brück 94]).

[Talc 93] C. Talcott: A theory of binding structures and applications to rewriting. Theoretical Computer Science, 112 (1993), pp. 99—143, 1993.

[Wolff 94] Wolff, B.: LIMA - A Unified Approach to Modularisation, Parametrisation, and Views in Attributations, Technischer Bericht, FB3 Informatik, Universität Bremen *(this volume)*.

[WSL 93] M. Weber, M. Simons, C. Lafontaine: The Generic Development Language Deva: Presentation and Case Studies. LNCS 738, 1993.