Factoring Proofs

Adam Poswolsky and Carsten Schürmann *

Yale University
New Haven, CT, USA
email: {poswolsky|carsten}@cs.yale.edu

Abstract. Logic programs, when total, realize proofs via a realizability interpretation similar to the Curry Howard isomorphism. The challenge in statically deciding totality lies in checking for termination and for coverage. For Elf logic programs, sophisticated algorithms exist for both problems. However, those solutions only apply to a much smaller subset of logic programs than a programmer is typically inclined to write. Often programs have to be rewritten to accommodate the idiosyncrasies of the coverage checker that cannot handle logic programs that backtrack. Factoring is a semantic preserving program transformation technique that removes backtracking from logic programs and renders them coverage checkable.

1 Introduction

Logic programs, when total, realize proofs via a realizability interpretation similar to the Curry Howard isomorphism. The "mode" of a logic program, which specifies which arguments are inputs and which are outputs, stands for a logical formula that relates universally and existentially quantified assumptions. When logic programs become proofs, running them inside an abstract machine is far less interesting than checking their properties — foremost totality, which consists of checking for termination, progress, and mode-correctness. A large portion of programs written in the logic programming language Elf [10] (part of the meta-logical framework Twelf [11]) actually formalize proofs.

The challenge in statically deciding progress lies in checking for coverage. Coverage checking [12] determines if a set of patterns covers all cases and relies on the ability to locate sets of applicable cases at any point in the abstract execution of a logic program. However, the declarative nature of logic programming conflicts directly with this goal since sets of alternative patterns are not locally definable within a clause.

Local case analysis is mirrored in Twelf by "deep backtracking". Deep backtracking is contrasted to "shallow backtracking" by its need to go back in the search tree to signal that a rule is not applicable.

```
run true 1.                  run false 0.
run (if E E1 E2) V           run (if E E1 E2) V
 :- run E 1, run E1 V.        :- run E 0, run E2 V.
```

* This work was supported in part by the National Science Foundation under grants CCR-0133502.

This example depicts a total function that maps a simple language of *if* expressions into {0,1} using deep backtracking in the first subgoal of the `if` cases to decide which of the clauses to take. In the current design, the coverage checker would reject any logic program whose implementation relies on deep backtracking [12]. If one still wants to use it, one has to rewrite `run` as:

```
run true 1.
run false 0.
run (if E E1 E2) V
 :- run E V',
    run' V' E1 E2 V.

run' 1 E1 E2 V
 :- run E1 V.
run' 0 E1 E2 V
 :- run E2 V.
```

In practice, large programs require frequent rewrites that become quickly an unacceptable nuisance. *Factoring* is a solution-preserving program transformation that removes deep backtracking from logic programs and hence greatly extends the set of logic programs that can be automatically checked for coverage.

The technique of factoring proceeds in three steps. First, the proof, an Elf mode-correct logic program (whose arguments play an a priori fixed input/output role), is converted into a specifically designed λ-calculus called $\mathcal{T}_\omega$. This functional λ-calculus remotely resembles the Elf logic programming language. $\mathcal{T}_\omega$ supports dependent types and embedded implications, has an operational semantics that does backtracking, computes complete sets of solutions, and even permits local case analysis, but does not support existential logic variables subject to instantiation by unification. The second step merges compatible cases within different branches of a term. And finally, the program may be translated back into a corresponding Elf program where every non-local case analysis is converted back into a program fragment of an auxiliary predicate. All Elf logic programs are pure, which means that the algorithm presented in this paper does not give answers to dealing with non-logical constructs, including asserts, retracts and cut.

All but the last of the three steps outlined above have been implemented and extensively tested with respect to coverage checking. The largest Twelf project done yet is Crary's implementation of foundational typed assembly language with about thirty-thousand lines of Twelf code [3] with more than a thousand theorems. Unfortunately, many of Crary's theorems are presently factored by hand because previous versions of the coverage checking algorithm could only verify input coverage and the trivial form of output coverage (there can be only one case for each subgoal). The current working version of the Twelf system takes advantage of factoring for coverage checking, rendering this factoring by hand unnecessary.

This paper is organized as follows. In Section 2 we briefly describe the logic programming language Elf. In Section 3, we define the λ-calculus $\mathcal{T}_\omega$ that is designed to capture mode-correct logic programs and describe the translation of logic programs. Next, in Section 4, we present the factoring algorithm which works on all mode-correct logic programs and show in Section 5 the main result of the paper– that any factored program is semantic preserving. After Section 6 on related work, we sketch further applications of factoring in Section 7. We then conclude and assess results in Section 8.

```
exp : type.
z    : exp.
s    : exp -> exp.
case : exp -> exp -> (exp -> exp) -> exp.


eval : exp -> exp -> type.
ev_z      : eval z z.
ev_s      : ΠE:exp.ΠV:exp.
              eval (s E) (s V)
              ← eval E V.
ev_case_z : ΠE1:exp.ΠE2:exp.ΠE3:exp → exp.ΠV:exp.
              eval (case E1 E2 E3) V
              ← eval E1 z
              ← eval E2 V.
ev_case_s : ΠE1:exp.ΠE2:exp.ΠE3:exp → exp.ΠV:exp.ΠV1:exp.
              eval (case E1 E2 E3) V
              ← eval E1 (s V1)
              ← eval (E3 V1) V.
```

Fig. 1. Example

2 The logical framework LF

An Elf logic program is a signature in the LF logical framework [6]. It consists of type families and constant declarations. Figure 1 depicts a small logic program that encodes the relevant fragment of an operational semantics for the case construct of Mini-ML [8]. For illustration purposes we keep the example simple. However, the techniques described in this paper scale to large example as well, e.g. the development of foundational typed assembly language [3].

2.1 Basics

In LF we follow the judgments-as-types representation paradigm. The LF type constant `exp` encodes the domain of expressions. Similarly, the evaluation judgment is represented as a type `eval` $E\ \ V$ encoding a big-step operational semantics that E evaluates to V. We write M for LF objects and A for LF types that are defined as follows and omit the definition of kinds K.

$$\begin{array}{lll} \text{Objects} & M & ::= c \mid x \mid \lambda x : A.\, M \mid M_1\ M_2 \\ \text{Types} & A & ::= a \mid A\ M \mid \Pi x : A_1.\, A_2 \end{array}$$

Here, c refers to object level constants and a to type level constants that are declared of their respective types and (here omitted) kinds in the signature Σ. A context Γ assigns types to variables and can be seen as a list of declarations $\cdot, x_1 : A_1 \ldots x_n : A_n$. We denote the empty context as $\cdot$ and choose to omit the leading "$\cdot$," from non-empty contexts.

As usual, we identify α-equivalent terms. We take $\beta\eta$-conversion as the notion of definitional equality [6, 2], for which we write $M \equiv M'$ and $A \equiv A'$.

$$\textit{Substitutions } \sigma ::= \cdot \mid \sigma, M/x$$

Substitutions are capture-avoiding. Point-wise application is written as $M[M'/x]$ and $A[M'/x]$. Simultaneous application is written as $M[\sigma]$ and $A[\sigma]$. We denote an empty substitution also as " $\cdot$ ", noting the ambiguous meaning of $\cdot$. As for contexts, we omit the leading "$\cdot$," from non-empty substitutions. Composition of substitutions is defined by $(\cdot) \circ \sigma = (\cdot)$ and $(M/x, \theta) \circ \sigma = (M[\sigma]/x, \theta \circ \sigma)$. If we write ϱ, then we refer to a renaming substitution.

Signatures, contexts, and substitutions may not declare a variable or constant more than once, and renaming of bound variables may be applied tacitly to ensure this condition. Besides equality, the main typing judgments are $\Gamma \vdash M : A$ for objects, $\Gamma \vdash A : K$ for types, and $\Gamma' \vdash \sigma : \Gamma$ for a well-typed substitution σ with domain Γ and free variables among Γ'. All typing judgments are indexed by the signature Σ, omitted in the interest of clarity. Furthermore, we assume throughout the rest of this paper that both substitution application (to terms and types) and substitution composition are defined for valid terms, types, and substitutions only. Since it is perhaps not so well-known, we will give only the typing rules for substitutions.

$$\frac{}{\Gamma' \vdash \cdot : \cdot} \qquad \frac{\Gamma' \vdash \sigma : \Gamma \qquad \Gamma' \vdash M : A[\sigma]}{\Gamma' \vdash (\sigma, M/x) : (\Gamma, x{:}A)}$$

For a context $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$, we define the identity substitution $\mathsf{id}_\Gamma = (x_1/x_1, \ldots, x_n/x_n)$ so that $\Gamma \vdash \mathsf{id}_\Gamma : \Gamma$.

2.2 Logic Programming in Elf

Depending on the polarity of their occurrence, the Elf logic programming language interprets LF types as definite clauses D or goals G, arrows as subgoals or embedded implications, and Π's as binders for existential variables or parameters. We write H for heads.

Heads	$H ::= a\ M_1\ \ldots\ M_n$
Goals	$G ::= H \mid \Pi x : A.\, G \mid D \rightarrow G$
Definite clauses	$D ::= H \mid \Pi x : A.\, D \mid G \rightarrow D$

The reader can easily check that the logic program depicted in Figure 1 falls into this fragment. In a signature, type families (e.g. `eval`) are the names of logic programs, and constant declarations form its implementation. Out of convenience, we use the left associative backarrow $\leftarrow$ for subgoals. To conform with standard logic programming notation, $(H \leftarrow G_1) \leftarrow G_2$ stands for $G_2 \rightarrow (G_1 \rightarrow H)$.

2.3 Mode-Correct Logic Programs

A logic program embodies a proof iff it is a total function and the corresponding type family encodes the theorem. Mode-correctness, termination, and progress are the necessary conditions for totality. A logic program is *mode-correct* if its arguments are assigned fixed roles as either inputs or outputs. The role of an argument must not switch from output to input, or vice versa. We follow standard practice and assign signs to argument position reflecting the meta theoretic property that all results returned in "$-$" positions are ground (do not contain free variables) assuming that all input arguments in "$+$" position are ground as well. For example, the mode-correct type family `eval` in Figure 1 has mode `eval +E -V`.

Mode analysis has been widely studied for various logics, including Horn logic [17], hereditary Harrop formulas, and Elf [14], and therefore we do not hesitate to refer the reader to the literature for syntactic algorithms and decision procedures to decide if a logic program is mode-correct. Mode checking requires that all other type families that may be called during the execution of the program must be mode-correct as well.

In Elf, the operational semantics for logic programming employs a higher-order pattern unification algorithm [9] with a constraint store for unification equations that need to be postponed when a unification problem lies outside the pattern fragment. Higher-order pattern unification is decidable as long as all existential variables have at least one occurrence in a rigid position (following Huet's conventions). Mode-correct logic programs require that all variables in the head of a clause are instantiated during matching, and thus need to have at least one occurrence in a rigid position.

Example 1. A non-trivial example of a mode-correct logic program, which also illustrates the benefits of factoring, is the strong normalization theorem described in Appendix A.

```
sn : ΠS:state.ΠT:tp.ofs S T
       -> ΠV:val.S =>* ($ ! V) -> type
%mode +{S:state} +{T:tp} +{DS:ofs S T}
        -{V:val} -{T: S =>* ($ ! V)} sn DS T.
```

This theorem relates typing derivations of states to traces in a small-step operational semantics. S is a state, T its type, and `ofs` encodes the typing judgment for states. In the second argument type, `=>*` stands for the small step semantics and (`$ !` V) stands for the final state, where V is a return value. The type of `sn` states that every well-typed state will reduce to a value given that the first three arguments are inputs (universally quantified) and the remaining two are outputs (existentially quantified). A logic program may be examined for totality only when the modes of the individual arguments are known ahead of time.

3 Functional Calculus $\mathcal{T}_\omega$

Before we give a precise characterization of the factoring algorithm, we describe a functional calculus called $\mathcal{T}_\omega$, into which mode-correct logic programs can be encoded while preserving their meaning.

Logic programs feature existential variables, failure, and backtracking. Although $\mathcal{T}_\omega$-programs may still fail and use backtracking, there are no existential variables. Since we consider only mode-correct logic programs, its conversion yields a program that computes the instantiation of the existential variables of the query.

Figure 2 describes the syntactic categories of $\mathcal{T}_\omega$. We write $\mathcal{F}$ for the types of $\mathcal{T}_\omega$-programs where universal types can be read as dependent Π types and existential types

Types	$\mathcal{F}$	$::= \forall x : A.\, \mathcal{F} \mid \exists x : A.\, \mathcal{F} \mid \top$
Values	V, W	$::= \{\eta; \Lambda x : A.\, P\} \mid \langle M; V \rangle \mid \langle\rangle$
Programs	P, F	$::= \{\eta; P\} \mid \Lambda x : A.\, P \mid P\ M \mid \langle M; P \rangle \mid \text{let } \langle x : A; \alpha \in \mathcal{F} \rangle = P_1 \text{ in } P_2$ $\mid \langle\rangle \mid \alpha \mid \text{case }\ \Omega \mid \mu\alpha \in \mathcal{F}.\, P$
Patterns	σ	$::= \mathsf{id}_\Psi \mid \sigma, M/x \mid \sigma, \alpha/\alpha$
Cases	Ω	$::= .\ \mid \Omega, (\Psi \rhd \sigma \mapsto P)$
Contexts	Ψ	$::= \cdot \mid \Psi, x : A \mid \Psi, \alpha \in \mathcal{F}$
Environments	η	$::= \mathsf{id}_\Psi \mid \eta, M/x \mid \eta, P/\alpha$

Fig. 2. $\mathcal{T}_\omega$ and program syntax

as dependent Σ types. Ψ stands for contexts that declare LF level variables of type A, and $\mathcal{T}_\omega$-level variables of formula $\mathcal{F}$. P stands for programs, which is the level at which factoring begins (Section 4). We also use F to stand for programs which have undergone the factoring algorithm. We write η for environments and $\{\eta; P\}$ for closures, $\Lambda x : A.\, P$ are functions that bind LF objects, and $\langle M; P \rangle$ represent pairs. This presentation omits treating pairs of two programs necessary to encode mutually recursive logic programs, but they can be added without problems [13]. As usual, $\langle\rangle$ is the the unit object, x are variables, and $P\ M$ is application. The most noteworthy construct is a case statement. Because of dependencies, we cannot consider cases locally over individual variables, but must consider cases over *all* variables simultaneously. Each individual case $(\Psi \rhd \sigma \mapsto P)$ consists therefore of three parts: a pattern that is simply a substitution σ, with its co-domain Ψ, and a program P that is to be executed when the case matches. In this formulation, matching η reduces to finding a substitution η' such that $\sigma \circ \eta' = \eta$. As usual, we write $\mu x \in \mathcal{F}.\, P$ for recursion and use let $(x \in \mathcal{F}) = P_1$ in P_2.

Because it is intuitively clear, we omit the details of the conversion of mode-correct logic programs from LF into $\mathcal{T}_\omega$. We do note that the conversion is defined purely *syntactically*. It follows the syntactic structure of each constant type and converts heads into recursive goals. For presentation purposes we do not show the treatment of embedded implications and universal parameters. However, the results presented in this paper scale directly to this fragment. We comment on these extensions when appropriate. The version of $\mathcal{T}_\omega$ that is designed to treat both concepts requires a formulation of LF with Σ types and type level λ's, contexts to distinguish between parametric and non-parametric assumptions $x : A$ and $x\#A$, and a program construct that permits dynamic introduction of parametric assumptions, called $\nu x\#A.P$ [16].

Example 2. Figure 3 gives the result of converting the logic program `eval` from Figure 1 into $\mathcal{T}_\omega$. For improved readability we omit type and formula labels from the let, Λ, and context. Observe that the backtracking behavior of the logic program corresponds to the non-determinacy of having *match redundant* cases in the converted program.

In the interest of further meta-theoretical study of factoring, we specify a big-step operational semantics of $\mathcal{T}_\omega$ that is given in Figure 4 via the judgment $\eta \vdash P \hookrightarrow$

$$
\begin{array}{l}
\text{eval} \in \forall e : \text{exp}.\, \exists v : \text{exp}.\, \top \\
\quad = \mu f.\, \Lambda e.\, \text{case } \; (f \rhd (\text{id}, \text{z}/e) \mapsto \langle \text{z}; \langle\rangle \rangle), \\
\qquad\qquad\qquad\quad (f, e \rhd (\text{id}, \text{s } e/e) \mapsto \\
\qquad\qquad\qquad\qquad \text{let } w = f\ e \text{ in} \\
\qquad\qquad\qquad\qquad\quad \text{case } (f, e, v \rhd (\text{id}, \langle v; \langle\rangle \rangle / w) \mapsto \langle \text{s } v; \langle\rangle \rangle)) \\
\qquad\qquad\qquad\quad (f, e_1, e_2, e_3 \rhd (\text{id}, \text{case } e_1\ e_2\ e_3 / e) \mapsto \\
\qquad\qquad\qquad\qquad \text{let } w_1 = f\ e_1 \text{ in} \\
\qquad\qquad\qquad\qquad\quad \text{case } (f, e_1, e_2, e_3 \rhd (\text{id}, \langle \text{z}; \langle\rangle \rangle / w_1) \mapsto \\
\qquad\qquad\qquad\qquad\qquad \text{let } w_2 = f\ e_2 \text{ in} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{case } (f, e_1, e_2, e_2, v_2 \rhd (\text{id}, \langle v_2; \langle\rangle \rangle / w_2) \mapsto \langle v_2; \langle\rangle \rangle))), \\
\qquad\qquad\qquad\quad (f, e_1, e_2, e_3 \rhd (\text{id}, \text{case } e_1\ e_2\ e_3 / e) \mapsto \\
\qquad\qquad\qquad\qquad \text{let } w_1 = f\ e_1 \text{ in} \\
\qquad\qquad\qquad\qquad\quad \text{case } (f, e_1, e_2, e_3, v_1 \rhd (\text{id}, \langle \text{s } v_1; \langle\rangle \rangle / w_1) \mapsto \\
\qquad\qquad\qquad\qquad\qquad \text{let } w_3 = f\ (e_3\ v_1) \text{ in} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{case } (f, e_1, e_2, e_2, v_1, v_3 \rhd (\text{id}, \langle v_3; \langle\rangle \rangle / w_2) \mapsto \langle v_3; \langle\rangle \rangle)))
\end{array}
$$

Fig. 3. Converted version of logic program for eval in $\mathcal{T}_\omega$.

V, which defines the operational meaning of P as the result V in an environment η. The rules are more or less standard and we will explain only the non-standard ones. The non-standard rule is ev_case, which matches patterns contained in Ω against the environment — not only a few variables in the environment, but the environment as a whole. Matching is expressed by a judgment on its own: $\vdash \eta \sim \Omega \hookrightarrow V$. A necessary and sufficient condition for a match to exist is a new η' s.t. $\sigma \circ \eta' = \eta$, expressed by the two rules ev_yes and ev_no.

Definition 1 (Semantics). *The meaning of a program,* $[\![P]\!]$*, is defined as all possible outcomes of the computation:* $[\![P]\!] = \{V \mid \cdot \vdash P \hookrightarrow V\}$.

The deep backtracking of logic programs is captured by the rule ev_no. This rule allows it to non-deterministically pick any matching case out of Ω. Backtracking is a feature that is essential when programming declaratively. It is this property that has popularized the use of logic programming languages to tackle problems in AI. In our setting, where logic programs are used to encode proofs, the use of non-determinism often leads to declarative, concise, and hence clear encodings. However, by using deep backtracking, the otherwise local choice over which branch to take next was splattered all over the logic program, rendering the currently available technique of coverage checking inapplicable. In many cases, this form of deep backtracking can be removed by *factoring* the program.

4 Factoring

The idea behind factoring relies on searching for case analysis within a logic program, exposing the abstract computation paths that lead to different cases, and collapsing multiple cases into one. In logic programming, there are neither local definitions nor explicit case constructs, but in $\mathcal{T}_\omega$ there are. This is ultimately the reason for considering $\mathcal{T}_\omega$, and so we define factoring only in this setting.

$$\frac{\eta \vdash \eta(\mathbf{x}) = V}{\eta \vdash \mathbf{x} \hookrightarrow V}\,\textsf{ev\_var} \qquad \frac{}{\eta \vdash \langle\rangle \hookrightarrow \langle\rangle}\,\textsf{ev\_unit}$$

$$\frac{\eta \vdash P \hookrightarrow V}{\eta \vdash \langle M; P\rangle \hookrightarrow \langle M[\eta]; V\rangle}\,\textsf{ev\_pair} \qquad \frac{\eta \vdash P_1 \hookrightarrow V_1 \qquad \eta, V_1/\mathbf{x} \vdash P_2 \hookrightarrow V}{\eta \vdash \text{let } (\mathbf{x} \in \mathcal{F}) = P_1 \text{ in } P_2 \hookrightarrow V}\,\textsf{ev\_let}$$

$$\frac{}{\eta \vdash \Lambda x : A.\, P \hookrightarrow \{\eta; \Lambda x : A.\, P\}}\,\textsf{ev\_}\Lambda \qquad \frac{\eta, \mu\mathbf{x} \in F.\, P/\mathbf{x} \vdash P \hookrightarrow V}{\eta \vdash \mu\mathbf{x} \in F.\, P \hookrightarrow V}\,\textsf{ev\_rec}$$

$$\frac{\eta \vdash P \hookrightarrow \{\eta'; \Lambda\mathbf{x} : A.\, P'\} \qquad \eta', M/\mathbf{x} \vdash P' \hookrightarrow V}{\eta \vdash P\ M \hookrightarrow V}\,\textsf{ev\_app}$$

$$\frac{\vdash \eta \sim \Omega \hookrightarrow V}{\eta \vdash \text{case } \Omega \hookrightarrow V}\,\textsf{ev\_case}$$

$$\frac{\sigma \circ \eta' = \eta \qquad \eta' \vdash P \hookrightarrow V}{\vdash \eta \sim (\Omega, (\Psi \triangleright \sigma \mapsto P)) \hookrightarrow V}\,\textsf{ev\_yes} \qquad \frac{\vdash \eta \sim \Omega \hookrightarrow V}{\vdash \eta \sim (\Omega, (\Psi \triangleright \sigma \mapsto P)) \hookrightarrow V}\,\textsf{ev\_no}$$

Fig. 4. $\mathcal{T}_\omega$ operational semantics.

Thus, our algorithm factors $\mathcal{T}_\omega$-programs that may contain cases embedded into cases as can be seen in our running example in Figure 3. We stipulate that two (or more) cases can be merged only if (1) the patterns contained within are redundant (modulo renaming); e.g., the third and fourth case in the main case analysis in our example both have (id, case $e_1\ e_2\ e_3/e$), and (2) the path inside the cases to the next *embedded* case statement are equivalent. Two (or more) cases of this kind form an ω-*redex*.

Every ω-redex can be reduced by merging the bodies of the respective cases. During merging the two cases are replaced by one where the set of cases defining the two embedded case are merged into one. Consequently, merging may generate new ω-redices, although those redices are moving deeper and deeper inside the term with every factoring step. The innermost case statements, however cannot contain any more ω-redices, which means that any $\mathcal{T}_\omega$-program can be factored only finitely often before there are no ω-redices left. Thus, repeated applications of the factoring algorithm will eventually terminate. By design, our merge judgment will recursively factor any new ω-redices that result. We say that a program is *factored* iff it does not contain any ω-redices. Similarly, a list of cases is called *factored* iff it does not contain any ω redices. Subsequently, a logic program $\Sigma|a$ is *factored* iff the image of $\Sigma|a$ in $\mathcal{T}_\omega$ is factored.

Our factoring algorithm removes all ω-redices, in one go, from a program. By design, factoring is an idempotent operation. Once a program is factored, not surprisingly, repeated factoring will yield exactly the same result. The factoring algorithm is defined in terms of four judgments each of which we explain intuitively and present formally during the remainder of this section. We write $\Omega \rightsquigarrow\!\!\!\!\rightsquigarrow \Omega'$ for the judgment reducing an

$$\frac{}{\cdot \rightsquigarrow\!\!\!\!\rightsquigarrow \cdot}\ \textsf{case\_empty} \qquad \frac{\Omega \rightsquigarrow\!\!\!\!\rightsquigarrow \Omega' \quad P \leadsto F \quad \Omega' \oplus (\Psi \rhd \sigma \mapsto F) = \Omega''}{\Omega, (\Psi \rhd \sigma \mapsto P) \rightsquigarrow\!\!\!\!\rightsquigarrow \Omega''}\ \textsf{case\_nonempty}$$

$$\frac{\Omega \rightsquigarrow\!\!\!\!\rightsquigarrow \Omega' \quad P \leadsto F \quad \Omega' \oplus (\Psi \rhd \sigma \mapsto F) \uparrow}{\Omega, (\Psi \rhd \sigma \mapsto P) \rightsquigarrow\!\!\!\!\rightsquigarrow \Omega', (\Psi \rhd \sigma \mapsto F)}\ \textsf{case\_new}$$

Fig. 5. ω-reduction.

$$\frac{}{\mathbf{x} \leadsto \mathbf{x}}\ \textsf{fact\_var} \qquad \frac{}{\langle\rangle \leadsto \langle\rangle}\ \textsf{fact\_unit} \qquad \frac{P \leadsto F}{\lambda x : A.P \leadsto \lambda x : A.F}\ \textsf{fact\_}\Lambda$$

$$\frac{P \leadsto F}{\mu\mathbf{x} \in \mathcal{F}.\, P \leadsto \mu\mathbf{x} \in \mathcal{F}.\, F}\ \textsf{fact\_rec} \qquad \frac{P \leadsto F}{\langle M; P\rangle \leadsto \langle M; F\rangle}\ \textsf{fact\_pair} \qquad \frac{P \leadsto F}{P\ M \leadsto F\ M}\ \textsf{fact\_app}$$

$$\frac{P_1 \leadsto F_1 \quad P_2 \leadsto F_2}{\text{let } (\mathbf{x} \in \mathcal{F}) = P_1 \text{ in } P_2 \leadsto \text{let } (\mathbf{x} \in \mathcal{F}) = F_1 \text{ in } F_2}\ \textsf{fact\_let} \qquad \frac{\Omega \rightsquigarrow\!\!\!\!\rightsquigarrow \Omega'}{\text{case } \Omega \leadsto \text{case } \Omega'}\ \textsf{fact\_case}$$

Fig. 6. Congruence closure.

ω-redex, $P \leadsto P'$ for its congruence closure on programs, $\Omega \oplus (\Psi \rhd \sigma \mapsto P) = \Omega'$ for folding cases, and $P_1 \star P_2 = P_3$ for merging programs.

For more details on the algorithm and its meta-theory, the interested reader is directed to [13].

ω-reduction. We write $\Omega \rightsquigarrow\!\!\!\!\rightsquigarrow \Omega'$ if a list of cases Ω factors into Ω'. A list of cases is factored left to right, by folding in case by case to the already factored list. The three rules defining the traversal are shown in Figure 5. Notice, as a design choice, before we attempt to fold a case into a factored Ω, we factor inside the case (via $\leadsto$). If $\Omega \oplus (\Psi \rhd \sigma \mapsto P) \uparrow$, then the case $(\Psi \rhd \sigma \mapsto P)$ cannot be merged into Ω and the rule case_new applies. Otherwise, case_nonempty applies. In fact, if Ω does not contain on ω-redices, still $\Omega \rightsquigarrow\!\!\!\!\rightsquigarrow \Omega$.

Congruence closure. We write $P \leadsto F$ for the congruence closure of ω-reduction to programs as displayed Figure 6. We also say that F results from factoring P. It is clear that this congruence relation will extend to richer systems than $\mathcal{T}_\omega$, in particular the version which is necessary to capture universal parameters and embedded implications.

Folding Cases. We write $\Omega \oplus (\Psi \rhd \sigma \mapsto P) = \Omega'$ to fold a case into Ω, which is assumed to be factored already. The rules are given in Figure 7 and compare the case $(\Psi \rhd \sigma \mapsto P)$ to every single case in Ω.

A question arises regarding what to do if three (or anything more than two) cases are all mutually redundant. One approach is to perform two folds, one for the first two, and

$$\frac{\sigma_1 = \sigma_2 \circ \varrho \qquad P_1 \star P_2[\varrho] = P_3 \qquad \Omega \oplus (\Psi_2 \rhd \sigma_2 \mapsto P_2) = \Omega'}{\Omega, (\Psi_1 \rhd \sigma_1 \mapsto P_1) \oplus (\Psi_2 \rhd \sigma_2 \mapsto P_2) = \Omega', (\Psi_1 \rhd \sigma_1 \mapsto P_3)}\ \textsf{fold\_yes\_succeed\_more}$$

$$\frac{\sigma_1 = \sigma_2 \circ \varrho \qquad P_1 \star P_2[\varrho] = P_3 \qquad \Omega \oplus (\Psi_2 \rhd \sigma_2 \mapsto P_2) \uparrow}{\Omega, (\Psi_1 \rhd \sigma_1 \mapsto P_1) \oplus (\Psi_2 \rhd \sigma_2 \mapsto P_2) = \Omega, (\Psi_1 \rhd \sigma_1 \mapsto P_3)}\ \textsf{fold\_yes\_succeed\_nomore}$$

$$\frac{\sigma_1 = \sigma_2 \circ \varrho \qquad (P_1 \star P_2[\varrho]) \uparrow \qquad \Omega \oplus (\Psi_2 \rhd \sigma_2 \mapsto P_2) = \Omega'}{\Omega, (\Psi_1 \rhd \sigma_1 \mapsto P_1) \oplus (\Psi_2 \rhd \sigma_2 \mapsto P_2) = \Omega', (\Psi_1 \rhd \sigma_1 \mapsto P_1)}\ \textsf{fold\_yes\_fail}$$

$$\frac{\sigma_1 \neq \sigma_2 \circ \varrho \qquad \Omega \oplus (\Psi_2 \rhd \sigma_2 \mapsto P_2) = \Omega'}{\Omega, (\Psi_1 \rhd \sigma_1 \mapsto P_1) \oplus (\Psi_2 \rhd \sigma_2 \mapsto P_2) = \Omega', (\Psi_1 \rhd \sigma_1 \mapsto P_1)}\ \textsf{fold\_no}$$

$$\frac{}{\cdot \oplus (\Psi \rhd \sigma \mapsto P) \uparrow}\ \textsf{fold}\!\uparrow\!\textsf{\_empty} \qquad \frac{\sigma_1 \neq \sigma_2 \circ \varrho \qquad \Omega \oplus (\Psi_2 \rhd \sigma_2 \mapsto P_2) \uparrow}{\Omega, (\Psi_1 \rhd \sigma_1 \mapsto P_1) \oplus (\Psi_2 \rhd \sigma_2 \mapsto P_2) \uparrow}\ \textsf{fold}\!\uparrow\!\textsf{\_nonempty\_no}$$

$$\frac{\sigma_1 = \sigma_2 \circ \varrho \qquad (P_1 \star P_2[t]) \uparrow \qquad \Omega \oplus (\Psi_2 \rhd \sigma_2 \mapsto P_2) \uparrow}{\Omega, (\Psi_1 \rhd \sigma_1 \mapsto P_1) \oplus (\Psi_2 \rhd \sigma_2 \mapsto P_2) \uparrow}\ \textsf{fold}\!\uparrow\!\textsf{\_nonempty\_yes}$$

Fig. 7. Folding cases.

the other for folding the result with the third. The other approach, which we take in this paper, is to fold the first case with both the second and the third, and then fold the two results together. If all three cases are mutually disjoint (with respect to their semantics), then both approaches return the same result. However, in the case that they are all not mutually disjoint, the first approach may make subsequent merges less possible than the second approach.

The rules traverse Ω from right to left trying to merge $(\Psi \rhd \sigma \mapsto P)$ into each case. The case is left alone if there is either no renaming substitution between the two contexts (fold_no) or if the actual merge, via judgment $\star$, fails (fold_yes_fail). If the merge succeeds then it is possible that it can fold $(\Psi \rhd \sigma \mapsto P)$ into more cases (fold_yes_succeed_more) or not (fold_yes_succeed_nomore).

Two patterns should be considered equal modulo reordering of variables declared in their co-domains, which is established by a renaming substitution $\varrho ::= \cdot \mid \varrho, x/y$. Thus, the ϱ used in the premise of all the rules satisfies $\Psi_1 \vdash \varrho : \Psi_2$. While traversing Ω, the merging operation ($\star$ judgment) is guarded by the condition that σ_1 and σ_2 are equal modulo renaming. Even if it finds a compatible σ_1 and σ_2, the merge can still fail which is represented as $(P_1 \star P_2[\varrho]) \uparrow$. If it cannot merge $(\Psi \rhd \sigma \mapsto P)$ with any cases in Ω, then we write $\Omega \oplus (\Psi \rhd \sigma \mapsto P) \uparrow$.

Merging Factored Programs. We write $P_1 \star P_2 = P_3$ to express that P_3 is the result of merging P_1 and P_2. If no such P_3 exists, then we write $(P_1 \star P_2) \uparrow$. The rules are shown in Figure 8. A necessary precondition for merging two already factored programs is that both programs compute exactly the same values up to the point when

$$\frac{}{\langle\rangle \star \langle\rangle = \langle\rangle}\ \textsf{merge\_unit} \qquad \frac{}{(\lambda x : A.P) \star (\lambda x : A.P) = \lambda x : A.P}\ \textsf{merge\_}\Lambda$$

$$\frac{}{(\mu \mathbf{x} \in \mathcal{F}.\, P) \star (\mu \mathbf{x} \in \mathcal{F}.\, P) = \mu \mathbf{x} \in \mathcal{F}.\, P}\ \textsf{merge\_rec} \qquad \frac{}{(P\ M) \star (P\ M) = P\ M}\ \textsf{merge\_app}$$

$$\frac{P_1 \star P_2 = P_3}{\langle M; P_1\rangle \star \langle M; P_2\rangle = \langle M; P_3\rangle}\ \textsf{merge\_pair} \qquad \frac{\Omega \oplus (\Psi \rhd \sigma \mapsto P) = \Omega'}{\text{case } \Omega \star \text{case } (\Psi \rhd \sigma \mapsto P) = \text{case } \Omega'}\ \textsf{merge\_fold}$$

$$\frac{\Omega \oplus (\Psi \rhd \sigma \mapsto P) \uparrow}{\text{case } \Omega \star \text{case } (\Psi \rhd \sigma \mapsto P) = \text{case } (\Omega, (\Psi \rhd \sigma \mapsto P))}\ \textsf{merge\_new}$$

$$\frac{P_2 \star P_2' = P_3}{(\text{let } (\mathbf{x} \in \mathcal{F}) = P_1 \text{ in } P_2) \star (\text{let } (\mathbf{x} \in \mathcal{F}) = P_1 \text{ in } P_2') = \text{let } (\mathbf{x} \in \mathcal{F}) = P_1 \text{ in } P_3}\ \textsf{merge\_let}$$

Fig. 8. Merging factored programs.

$$\begin{array}{l}
\text{eval} \in \forall e : \text{exp}.\, \exists v : \text{exp}.\, \top \\
\quad = \mu f.\, \Lambda e.\, \text{case } \ (f \rhd (\text{id}, \text{z}/e) \mapsto \langle \text{z}; \langle\rangle\rangle), \\
\qquad\qquad\qquad\quad (f, e \rhd (\text{id}, \text{s } e/e) \mapsto \\
\qquad\qquad\qquad\qquad \text{let } w = f\ e \text{ in} \\
\qquad\qquad\qquad\qquad\quad \text{case } (f, e, v \rhd (\text{id}, \langle v; \langle\rangle\rangle/w) \mapsto \langle \text{s } v; \langle\rangle\rangle)) \\
\qquad\qquad\qquad\quad (f, e_1, e_2, e_3 \rhd (\text{id}, \text{case } e_1\ e_2\ e_3/e) \mapsto \\
\qquad\qquad\qquad\qquad \text{let } w_1 = f\ e_1 \text{ in} \\
\qquad\qquad\qquad\qquad\quad \text{case } \ (f, e_1, e_2, e_3 \rhd (\text{id}, \langle \text{z}; \langle\rangle\rangle/w_1) \mapsto \\
\qquad\qquad\qquad\qquad\qquad\quad \text{let } w_2 = f\ e_2 \text{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{case } (f, e_1, e_2, e_2, v_2 \rhd (\text{id}, \langle v_2; \langle\rangle\rangle/w_2) \mapsto \langle v_2; \langle\rangle\rangle))), \\
\qquad\qquad\qquad\qquad\qquad (f, e_1, e_2, e_3, v_1 \rhd (\text{id}, \langle \text{s } v_1; \langle\rangle\rangle/w_1) \mapsto \\
\qquad\qquad\qquad\qquad\qquad\quad \text{let } w_3 = f\ (e_3\ v_1) \text{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{case } (f, e_1, e_2, e_2, v_1, v_3 \rhd (\text{id}, \langle v_3; \langle\rangle\rangle/w_2) \mapsto \langle v_3; \langle\rangle\rangle))))
\end{array}$$

Fig. 9. Factored version of Figure 3.

another case statement is encountered. Thus, in merge_app and merge_let the first arguments to the application and let can be assumed to be ω-normal (contain no reducible ω-redices). Thus, unless they are syntactical equal, nothing can be said about their semantic properties, and merging is doomed to failure. For an extended version of $\mathcal{T}_\omega$ that captures universal parameters and embedded implications, additional rules can be easily added to Figure 8.

Example 3. A factored version of the $\mathcal{T}_\omega$-program from Figure 3 is given in Figure 9. The difference between the two programs is that a deep case analysis distributed over two branches has been collapsed into one case analysis.

5 Meta Theory

We begin now with the discussion of the main technical result of this work. First we show that the idempotence property, holds. The proof goes by induction over the factoring relation and is available at [13].

Lemma 1 (Idempotency).

1. *If $P \rightsquigarrow F$ then $F \rightsquigarrow F$.*
2. *If $\Omega \rightsquigarrow\!\!\!\!\rightsquigarrow \Omega'$ then $\Omega' \rightsquigarrow\!\!\!\!\rightsquigarrow \Omega'$.*

Second, we remark that the following meta theoretical investigation cannot be carried out without the proper definition of semantic equivalence modulo factoring between values and environments, which is denoted as $\stackrel{*}{\Rightarrow}$. This informally states that the computational behavior of both programs are the same, but if we are just looking at a λ then we only care if it is equivalent when it is *computed*, i.e. applied to some argument.

Definition 2 (Equivalence modulo factoring).

- *$V \stackrel{*}{\Rightarrow} W$ is defined recursively as:*
  - *If $V = \{\eta; \Lambda x : A.P\}$ and $W = \{\eta'; \Lambda x : A.F\}$ then*
    - $\eta \stackrel{*}{\Rightarrow} \eta'$
    - *and $P \rightsquigarrow F$*
  - *If $V = \langle M; V\rangle$ and $W = \langle M; W\rangle$ then $V \stackrel{*}{\Rightarrow} W$*
  - *Otherwise, $V = W$*
- *$\eta \stackrel{*}{\Rightarrow} \eta_1$ is defined recursively as:*
  - *$(\eta, \mu\mathbf{x} \in \mathcal{F}.\, P/\mathbf{x}) \stackrel{*}{\Rightarrow} (\eta_1, \mu\mathbf{x} \in \mathcal{F}.\, F/\mathbf{x})$ if the following is true:*
    - $\eta \stackrel{*}{\Rightarrow} \eta_1$
    - $P \rightsquigarrow F$
  - *$(\eta, V/\mathbf{x}) \stackrel{*}{\Rightarrow} (\eta, W/\mathbf{x})$ iff $V \stackrel{*}{\Rightarrow} W$*
  - $\cdot \stackrel{*}{\Rightarrow} \cdot$

It follows immediately from this definition that for Λ-free terms, $\stackrel{*}{\Rightarrow}$ implies $=$ (syntactical equality), a property that is essential in the proof of the main theorem. We extend the notion of equivalence modulo factoring to define the meaning of programs: we say that P and F are semantically equal and write $[\![P]\!] \equiv [\![F]\!]$ if for all $V \in [\![P]\!]$ there exists a $W \in [\![F]\!]$ such that $V \stackrel{*}{\Rightarrow} W$, and vice versa.

Let $\Sigma | a$ be a logic program, P be its conversion in $\mathcal{T}_\omega$, and F a factored program such that $P \rightsquigarrow F$. We are going to show that both P's and F's evaluation under $\mathcal{T}_\omega$ compute the same (syntactically equal) result. To this end, we study in the first step the properties of $\star$ and $\oplus$ that behave as one would expect.

Lemma 2 (Properties of $\star$ and $\oplus$).

1. *If $P_1 \star P_2 = P_3$, then*
   (a) if $\eta \vdash P_1 \hookrightarrow V$ then $\eta \vdash P_3 \hookrightarrow V$,
   (b) if $\eta \vdash P_2 \hookrightarrow V$ then $\eta \vdash P_3 \hookrightarrow V$,
   (c) if $\eta \vdash P_3 \hookrightarrow V$ then either $\eta \vdash P_1 \hookrightarrow V$ or $\eta \vdash P_2 \hookrightarrow V$.

2. *Let* $\Omega \oplus (\Psi \rhd \sigma \mapsto P) = \Omega'$ *be given.*
   (a) *If* $\sigma \circ \eta' = \eta$ *and* $\eta' \vdash P \hookrightarrow V$*, then* $\vdash \eta \sim \Omega' \hookrightarrow V$.
   (b) *If* $\vdash \eta \sim \Omega' \hookrightarrow V$ *then either*
      – $\sigma \circ \eta' = \eta$ *and* $\eta' \vdash P \hookrightarrow V$*, or*
      – $\vdash \eta \sim \Omega \hookrightarrow V$.

In this work [13] we showed the relation between the evaluation of programs and their factored cousins.

Lemma 3 (Embedding). *Let* $P \leadsto F$ *and* $\eta \overset{*}{\Rightarrow} \eta_1$ *be given.*

1. *If* $\eta \vdash P \hookrightarrow V$ *then there exists a* W *s.t.* $V \overset{*}{\Rightarrow} W$ *and* $\eta_1 \vdash F \hookrightarrow W$.
2. *If* $\eta_1 \vdash F \hookrightarrow W$ *then there exists a* V *s.t.* $V \overset{*}{\Rightarrow} W$ *and* $\eta \vdash P \hookrightarrow V$.

And we specialize this particular result to state one of the main results of factoring.

Theorem 1 (Main). $[P] \equiv [F]$.

Proof. *Follows directly from Lemma 3.*

The other main result is that a factored program does not have any cases which can be merged together.

Theorem 2 (ω-normal form). *Let* $P \leadsto F$*. For all cases* Ω *in* F *we have the property that: If* $(\Psi_1 \rhd \sigma_1 \mapsto P_1) \in \Omega$ *and* $(\Psi_2 \rhd \sigma_2 \mapsto P_2) \in \Omega$ *and* $(\Psi_1 \rhd \sigma_1 \mapsto P_1) \neq (\Psi_2 \vartriangle \sigma_2 \mapsto P_2)$ *then there does not exist a renaming substitution,* ϱ*, such that* $\sigma_1 = \sigma_2 \circ \varrho$ *and*

– *there exists a* P_3 *such that* $P_1 \star P_2[\varrho] = P_3$.
– *there exists a* P_3 *such that* $P_2 \star P_1[\varrho^{-1}] = P_3$.

Finally, to demonstrate that this is a technique that works on all programs, we must show termination.

Theorem 3 (Termination). *For all* P*, there exists an* F *such that* $P \leadsto F$.

Proof. By induction on the structure of of $P \leadsto F$.

6 Related Work

Factoring and checking for determinancy are closely related problem areas. With a slight modification to our factoring algorithm, we can check determinacy of a program. This new factoring algorithm will succeed if and only if there are no redundant cases [13]. In this respect, factoring provides a sufficient condition to check for determinacy. However, intuitively it should be clear that not every logic program can be converted into a semantically equivalent one that is deterministic. In fact, the problem of converting any inherently deterministic logic program to a semantically equivalent one is undecidable as shown by Sawamur and Takeshima [15]. However, it is decidable for a smaller subclass. The Mercury Project [7], for example, provides two different ways of running logic programs; it provides both a fast deterministic and a somewhat slower nondeterministic algorithm. The Mercury programmer can signal the logic programming engine, via a pragma (command), which algorithm to use. Mercury subsequently checks if the given logic program is really deterministic before it grants the request.

Similar work has also been done with respect to Prolog [4, 5], which shows very promising results but restricts itself to evaluating programs in Prolog. There work also undergoes program transformation for the sake of enhancements in unification, but has the result of making the result more complex. On the other hand, we argue that the result of our factoring algorithm returns an elegant, clear, and simple factored cousin.

At first it would appear that one way of strengthening factoring would be by supporting admissible permutations on subgoals that do not alter the meaning of the logic program. But, due to dependencies, it is not easy to reorder subgoals even if those reorders are meaning preserving. Furthermore, work focusing on Prolog [4, 5], have shown that relaxing the order-preserving strategy makes the problem NP-complete. We therefore constrain the factoring algorithm to maintain the order of subgoals.

7 Applications

As already discussed, factoring is the essential transformation to prepare logic programs for coverage checking. Previously, since factoring was not available in Twelf, one was forced to do it by hand. The largest project undertaken so far is Crary's implementation of foundational typed assembly language with about thirty-thousand lines of Twelf code [3] with more than a thousand theorems, of which several dozens were factored manually. The current working version of the Twelf system takes advantage of factoring for coverage checking, rendering this factoring by hand unnecessary.

Furthermore, as already alluded to in Section 6, we have also worked on using factoring to check for determinacy. In this realm, factoring lends itself as a preprocessor for running an optimized abstract machine, just in the sense of Mercury. We have implemented the operational semantics for $\mathcal{T}_\omega$ in Twelf (including all higher-order functionality) and offer it to the user in form of a `%fquery` command, whose syntax is similar to that of `%query`. Despite the fact that we use Twelf's slow yet powerful higher-order unification algorithm instead of a fast matching algorithm possibly employing pattern compilation, $\mathcal{T}_\omega$-programs run approximately as fast as the Twelf's logic program. The semantics of $\mathcal{T}_\omega$ refers to the operational semantics of the programming language Delphin [16]. Finally, the result of factoring can be compiled further down to languages like ML, OCaml, or Haskell, and then compiled further into machine code. This part of the project has not been implemented yet, but might be necessary for real world applications, where proofs are actually executed. Applications are plentiful [1, 3].

A factored logic program in $\mathcal{T}_\omega$ can be converted back into an Elf program using auxiliary type families to capture local definitions and case analysis in the $\mathcal{T}_\omega$-program. The important insight is that although cases have been folded and programs have been merged, the original program remains largely intact. The *only* change is that cases may now contain more than one branch. Note that each of those embedded cases may depend on all variables of the context.

For example, converting the factored function for `eval` from Figure 9 requires (1) the context of all variables that occur free inside the case subject, i.e. $f \in \mathcal{F}, e_1$:

exp, e_2 : exp, e_3 : exp $\to$ exp, w_1 : exp, (2) the return type of the case, i.e. exp, and (3) the list of embedded cases, i.e.:

$$(f, e_1, e_2, e_3 \rhd (\text{id}, \langle \text{z}; \langle\rangle\rangle / w_1) \mapsto \ldots)$$
$$(f, e_1, e_2, e_3 \rhd (\text{id}, \langle \text{s } v_1; \langle\rangle\rangle / w_1) \mapsto \ldots)$$

With this information, a converter can derive the definition of an auxiliary set of clauses, omitting f because it is known to encode the induction hypothesis:

```
eval' : exp -> exp -> (exp -> exp) -> exp -> exp -> type.
eval'1 : eval' E1 E2 E3 z X
            <- eval E2 X.
eval'2 : eval' E1 E2 E3 (s V1) X
            <- eval (E3 V1) X.
```

and the main set of clauses as:

```
eval : exp -> exp -> type.
eval1 : eval z z.
eval2 : eval (s E) (s X).
            <- eval E X.
eval3 : eval' (case E1 E2 E3) X
            <- eval E1 W1
            <- eval' E1 E2 E3 W1) X.
```

And it is clear that the set of solutions computed by this `eval` and that of Figure 1 are the same.

8 Conclusion

Logic programs that implement total functions can be interpreted as proofs. The challenge in statically deciding totality lies in checking for termination and coverage. Coverage checking is a complex procedure that requires mode-correct logic programs to be converted to functional form. Without automatically factoring logic programs first, many logic programs that employ deep backtracking would simply be rejected from the coverage checker, and therefore cannot be verified as proofs.

In this paper we have defined factoring as a reduction relation on programs that reduces so called ω-redices. Normalizing these redices eventually terminates and returns an ω-normal program that we also call factored. Factoring is semantics preserving and removes deep backtracking from logic programs, thus greatly expanding the domain of logic programs (i.e. proofs) that can be coverage checked.

References

1. A. W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258, Boston, USA, June 2001.

2. T. Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.
3. K. Crary. Toward a foundational typed assembly language. In G. Morrisett, editor, *Proceedings of the 30th Annual Symposium on Principles of Programming Languages*, pages 198–212, New Orleans, Louisiana, Jan. 2003. ACM Press.
4. S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, K. F. Sagonas, S. Skiena, T. Swift, and D. S. Warren. Unification factoring for efficient execution of logic programs. In *Symposium on Principles of Programming Languages*, pages 247–258, 1995.
5. S. Dawson, C. R. Ramakrishnan, S. Skiena, and T. Swift. Principles and practice of unification factoring. *ACM Transactions on Programming Languages and Systems*, 18(5):528–563, 1996.
6. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.
7. F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the mercury compiler, 1996.
8. S. Michaylov and F. Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, Jan. 1991. Springer-Verlag LNAI 596.
9. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
10. F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
11. F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.
12. F. Pfenning and C. Schürmann. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of 16th International Conference on Theorem Proving in Higher Order Logics*. Springer Verlag, 2004.
13. A. Poswolsky. Factoring report. yale university tech report (tr-1256). `ftp://ftp.cs.yale.edu/pub/tr/tr1256.ps`, 2003.
14. E. Rohwedder and F. Pfenning. Mode and termination checking for higher-order logic programs. In H. R. Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, Apr. 1996. Springer-Verlag LNCS 1058.
15. H. Sawamura and T. Takeshima. Recursive unsolvability of determinacy, solvable cases of determinacy and their applications to prolog optimization., 1985.
16. C. Schürmann. Delphin: Toward programming with logical frameworks. Technical Report YALEU/DCS/TR-1272, Yale University, Department of Computer Science, New Haven, CT, USA, 2004.
17. Z. Somogyi. A system of precise modes for logic programs. In *In Proceedings of the Fourth International Conference on Logic Programming*, pages 769–787, Melbourne,Australia, May 1987.

A Appendix

A.1 An Extended Example

```
% A continuation based machine for z, s, and case.
% Author: Carsten Schuermann

% %%%%%%%%%%%%%%%%%%%% Expressions, Values and Instructions.

exp  : type.                          %name exp E x.
inst : type.
val : type.

z : exp.
s : exp -> exp.
case : exp -> exp -> (val -> exp) -> exp.
` : val -> exp.                       %prefix 14 `.

z* : val.
s* : val -> val.

! : val -> inst.
case1: val -> exp -> (val -> exp) -> (val -> inst) -> inst.

% %%%%%%%%%%%%%%%%%%%% States

state : type.
# : (val -> inst) -> exp -> state.    %infix right 12 #.
$ : inst -> state.                    %prefix  12 $.

% %%%%%%%%%%%%%%%%%%%% Single step semantics

=> : state -> state -> type.          %infix right 11 =>.
%mode => +S -S'.
s_z      : K # z => $ K z*.
s_s      : K # (s E) => ([v] K (s* v)) # E.
s_case   : K # (case E E1 E2) => ([v] (case1 v E1 E2 K)) # E.
s_lift   : K # (` V) => $ (K V).
s_casez : $ (case1 z* E1 E2 K) => K # E1.
s_cases : $ (case1 (s* V) E1 E2 K) => K # (E2 V).
%worlds () (=> _ _).

% %%%%%%%%%%%%%%%%%%%% Multi step semantics

=>* : state -> state -> type.         %name =>* C.
%mode (=>* +S -S').                   %infix none 5 =>*.
```

```
stop : S =>* S.
<< : S =>* S''
      <- S => S'
      <- S' =>* S''.                        %infix left 5 <<.
%worlds () (=>* _ _).
%covers (=>* +S -S').

% %%%%%%%%%%%%%%%%%%%% Types

tp : type.                                  %name tp T.
nat : tp.
==> : tp -> tp -> tp.                       %infix right 10 ==>.

% %%%%%%%%%%%%%%%%%%%% Static semantics.

ofe : exp -> tp -> type.
ofv : val -> tp -> type.
ofi : inst -> tp -> type.
ofk : (val -> inst) -> tp -> type.
ofs : state -> tp -> type.

of_z : ofe z nat.
of_s : ofe (s E) nat
        <- ofe E nat.
of_case: ofe (case E E1 E2) T
          <- ofe E nat
          <- ofe E1 T
          <- ({x} ofv x nat -> ofe (E2 x) T).
of_`      : ofe (` V) T
             <- ofv V T.

of_z* : ofv z* nat.
of_s* : ofv (s* V) nat
        <- ofv V nat.

of_!    : ofi (! V) T
           <- ofv V T.
of_case1 : ofi (case1 V E1 E2 K) T
            <- ofv V nat
            <- ofe E1 T1
            <- ({x} ofv x nat -> ofe (E2 x) T1)
            <- ofk K (T1  ==> T).

of_k  : ofk K (T1 ==> T2)
           <- ({v:val} ofv v T1 -> ofi (K v) T2).
```

```
of_#  : ofs (K # E) T1
         <- ofk K (T2 ==> T1)
         <- ofe E T2.
of_$  : ofs ($ I) T
         <- ofi I T.
%block l1 : some {T:tp} block {x:val}{u:ofv x T}.
%worlds (l1) (ofe _ _) (ofv _ _) (ofi _ _) (ofk _ _) (ofs _ _).
%covers (ofe +E -T) (ofv +V -T) (ofi +I -T) (ofk +K -T) (ofs +S -T).


% %%%%%%%%%%%%%%%%%%%% Meta theory

% %%%%%%%%%%%%%%%%%%%% Type preservation.

tps : ofs S T -> S => S' -> ofs S' T -> type.

tps_z : tps (of_# of_z (of_k DE')) (s_z) (of_$ (DE' z* of_z*)).
tps_s : tps (of_# (of_s DE1) (of_k [x][u] DE2 x u))
         (s_s)
         (of_# DE1 (of_k [v] [u] DE2 (s* v) (of_s* u))).
tps_case : tps (of_# (of_case DE3 DE2 DE1) (of_k [v][u] DE' v u))
            (s_case)
            (of_# DE1 (of_k [v] [u] of_case1
 (of_k [v'][u']  DE' v' u') DE3 DE2 u)).
tps_` : tps (of_# (of_` DV) (of_k DE)) (s_lift) (of_$ (DE _ DV)).
tps_casez: tps (of_$ (of_case1 DK DE2 DE1 of_z*))
     (s_casez)
     (of_# DE1 DK).
tps_cases: tps (of_$ (of_case1 DK DE2 DE1 (of_s* DV)))
             (s_cases)
             (of_# (DE2 _ DV) DK).


%mode tps +DS +R -DS'.
%worlds () (tps _ _ _).

% %%%%%%%%%%%%%%%%%%%% Validity of States

valid : state -> type.
v_val   : valid ($ ! V).
v_noval : valid S
           <- S => S'.

%worlds () (valid _).
```

```
%covers (valid +S).

% %%%%%%%%%%%%%%%%%%% Progress lemma

progress : ofs S T -> valid S -> type.
p_z : progress (of_# of_z DK) (v_noval s_z).
p_s : progress (of_# (of_s DE) DK) (v_noval s_s).
p_case : progress (of_# (of_case DE3 DE2 DE1) DK)
  (v_noval s_case).
p_` : progress (of_# (of_` DV) DK) (v_noval s_lift).
p_case1  : progress (of_$ (of_case1 DK ([v][u] DE3 v u)
     DE2 of_z*))
    (v_noval s_casez).
p_case2  : progress (of_$ (of_case1 DK ([v][u] DE3 v u)
     DE2 (of_s* DE1)))
    (v_noval s_cases).
p_!      : progress (of_$ (of_! DV)) (v_val).

%mode progress +DS -VD.
%worlds () (progress _ _).
%covers (progress +DE -VD).

% %%%%%%%%%%%%%%%%%%% Strong normalization

sn : ofs S T -> S =>* ($ ! V) -> type.
%mode +{S:state} +{T:tp} +{DS:ofs S T}
         -{V:val} -{T: S =>* ($ ! V)} sn DS T.
done     : sn DS stop
            <- progress DS v_val.
step     : sn DS (R << S)
            <- progress DS (v_noval S)
            <- tps DS S DS'
            <- sn DS' R.
```