Full reduction in the face of absurdity

Gabriel Scherer[1] and Didier Rémy[1]

INRIA,
{gabriel.scherer,didier.remy}@inria.fr

Abstract. Core calculi that model the essence of computations use full reduction semantics to be built on solid grounds. Expressive type systems for these calculi may use propositions to refine the notion of types, which allows abstraction over possibly inconsistent hypotheses. To preserve type soundness, reduction must then be delayed until logical hypotheses on which the computation depends have been proved consistent. When logical information is explicit inside terms, proposition variables delay the evaluation by construction. However, logical hypotheses may be left implicit, for the user's convenience in a surface language or because they have been erased prior to computation in an internal language. It then becomes difficult to track the dependencies of computations over possibly inconsistent hypotheses.
We propose an expressive type system with implicit coercions, consistent and inconsistent abstraction over coercions, and *assumption hiding*, which provides a fine-grained control of dependencies between computations and the logical hypotheses they depend on. Assumption hiding opens a continuum between explicit and implicit use of hypotheses, and restores confluence when full and weak reductions are mixed.

Extended version For reasons of page limits, the proofs have been omitted from this conference version. A full version with additional remarks and all proofs, is available electronically.[1]

1 Introduction

The Curry-Howard isomorphism trained generations of statically-typed-language designers to be able to instantly switch their point of view from *programs* to *proof terms*, and from *types* to *logic statements*. Proof assistants based on type theory let us use our functional programming intuitions to *program* proofs. One example of the merits of such a re-unification is the strikingly simple and natural treatment of *axioms* in the functional languages of those assistants: assuming an axiom P is just abstracting over a variable $(x : P)$ of the corresponding type, and using this assumption is done by applying or pattern matching this bound variable x. These languages generally allow *full reduction*, in particular reducing under unapplied λ-abstractions. For example, a Coq program abstracting over

[1] At http://gallium.inria.fr/~remy/coercions/

an axiom P is of the form $\lambda(x : P)\, a$, where a may be computed as usual, but the reduction of subterms depending on x will be blocked.

There is a subtle but important contrast with how logical assumptions have been dealt with in languages designed mostly for programming rather than proving, such as ML or Haskell. Typical examples are the reasoning on type equalities in the ML module system, or in Generalized Algebraic Data Types (GADTs). Consider the following example that implements application up to type equality, given in OCaml-like syntax:

$$\textbf{type}\ (\_,\ \_)\ \mathsf{eq} = \mathsf{Refl} : (\alpha,\ \alpha)\ \mathsf{eq}$$

$$\textbf{let}\ \mathsf{apply} : \forall\alpha_1\alpha_2\beta.\ (\alpha_1 \to \beta) \to \alpha_2 \to (\alpha_1,\ \alpha_2)\ \mathsf{eq} \to \beta$$
$$= \textbf{fun}\ \mathsf{f}\ \mathsf{x}\ \mathsf{Refl} \to \mathsf{f}\ \mathsf{x}$$

With GADTs, the equality assumption is present at the term level, marked by a λ-abstraction over the type (α_1, α_2) eq, but the *use* of this equality is *implicit*: equality assumptions introduced by abstraction or pattern-matching can be silently used in the corresponding term clauses. This implicitness can be explained away by translating source terms into an intermediate language, such as System FC [Vytiniotis and Jones, 2011] that marks uses of equality assumptions with explicit coercions – providing a treatment similar to logical assumptions in proof assistants. But it can also be formalized directly, as in the presentation of GADTs extended to arbitrary logic constraints by Simonet and Pottier [2007], or Dependent ML by Xi [2007].

It is well-known however that, with implicit use of potentially-absurd assumptions, it is no longer safe to use full reduction under those assumptions. Assuming ($\mathsf{fst} : (\alpha * \beta) \to \alpha$) and ($\mathsf{true} : \mathsf{bool}$), the term apply fst true reduces to

$$\textbf{fun}\ (\mathsf{Refl} : (\mathsf{bool},\ (\alpha * \beta))\ \mathsf{eq}) \to \mathsf{fst}\ \mathsf{true}$$

Reducing under this abstraction would mean computing fst true, *i.e.* the application of a destructor to a constructor of an incompatible type, which is called a *runtime error*. Interestingly, this issue does not happen with an *explicit* handling of logical assumptions. In System FC, the above example would reduce to the following normal form (assuming that the assumption γ has been used to convert the type of the argument true rather than the type of the function fst):

$$\textbf{fun}\ (\mathsf{Refl}\ (\gamma : \mathsf{bool}\ \tilde{}\#\ (\alpha * \beta)) \to \mathsf{fst}\ (\mathsf{true} \triangleright \gamma)$$

Here, bool ~# ('a * 'b) is the type of coercions that prove the equality between bool and ('a * 'b), and (true $\triangleright\ \gamma$) is the application of the coercion γ to true. This application cannot be reduced until the formal variable γ has been instantiated (that is, never, if we are in an empty context with a sound type system). Meaning, γ remains in between fst and true preventing the application. Although System FC is a weak calculus (abstracting on term or coercion variables blocks reduction), full reduction could be used in a similar, explicit system.

We are convinced that it is important to also study the implicit presentation directly. There is a convergence of designs that indicates that *implicit* use of assumptions is significantly more convenient to the programmer. For a less-obvious

example than GADTs in ML or Haskell, the book *Programming in Martin-Löf Type Theory* [Nordström et al., 1990] uses a type theory with *extensional* equality, which allows implicit use of equality assumptions, especially to simplify programming with quotient types. We want to study λ-calculi that match how users wish to program and define the operational behavior of programs directly at this level.

Besides, we deem unfortunate the absolute reign of *weak reduction* on λ-calculi designed for programming. We argue that while one could have a weak-reduction semantics for reasoning about runtime complexity, a more abstract *full reduction* understanding is better to reason about correctness – as an important step towards equational reasoning on open terms.

In fact, type systems of programming languages are designed for full reduction strategies, and left unchanged when restricting the semantics to weak reduction strategies. For example, the type systems of ML, System F and its derivatives (F_η [Mitchell, 1988], $\mathsf{F}_{<:}$ [Cardelli, 1993], MLF [Le Botlan and Rémy, 2003], *etc.*) are all sound for full reduction. While type systems are regularly improved to accept more well-typed programs, they do not try in general to take advantage of weak reduction strategies to accept nonsense under yet unapplied abstractions, *e.g.* $\lambda(x)\,1\,\mathsf{true}$, on the basis that these errors won't be reachable by a weak-reduction strategy.[2]

We claim that (pure) lambda-calculi for programming languages ought to strive to support *full reduction*; this design pressure should result in a better understanding of programming constructs. For example, soundness of full reduction subsumes soundness for any evaluation strategy such as call-by-name and call-by-value; and full reduction is used in practice in dependently-typed languages such as Coq or Agda, with significant efforts spent to make it practical [Grégoire and Leroy, 2002, Boespflug et al., 2011].

We could summarize the topic of this article with the following question. We know how to design calculi with *explicit* uses of logical assumptions and *full* reduction, or calculi with *implicit* uses of assumptions and *weak* reduction. Can we merge those apparently incompatible feature pairs into a single calculus, close to the non-encumbered terms the programmer wishes to write?

Consistent and inconsistent abstraction Intuitively, an abstraction on a type is *consistent* when we can prove at the point of abstraction that there always exists a possible instantiation for it in the current typing context; otherwise, we say it is *inconsistent*. A typical example of a consistent abstraction is an abstraction over a type variable α that has the kind $\star$ of concrete types, as we know that at least int has kind $\star$ so it is a valid instance for α. Abstraction over a type variable may also be constrained by a proposition that restricts the possible instances of the type variable. An example of an unsatisfiable proposition is the

[2] One interesting counterexample is typechecking record concatenation, which delays the resolution of typing constraints based on the evaluation strategy [Pottier, 2000] in order to avoid the heavy cost of early consistency checking.

inter-convertibility $\mathsf{int} \simeq (\mathsf{int} \to \tau)$, which is *absurd* for any type τ. Hence, an abstraction over a type variable α such that $\mathsf{int} \simeq (\mathsf{int} \to \alpha)$ is inconsistent.

Previous work by Cretin [2014] and Cretin and Rémy [2014] introduced the calculus F_{cc}, built around *consistent coercion abstraction*, a mechanism that allows implicit abstraction over coercions and uses typing-transforming coercions, provided we prove at their abstraction point that they are instantiable; such coercions are completely erasable – they are not at all present at the term level. This generalizes the traditional ML-style polymorphism in an expressive way, encompassing the type systems of System F, MLF, $\mathsf{F}_{<:}$, F_{η}, and F_{ω}. To be able to also abstract over hypotheses that may not be consistent, Cretin and Rémy added a distinct mechanism of *inconsistent polymorphism* that is present at the term level and blocks reduction.

If an F_{cc} term a has type τ in the context $\Gamma, \alpha : \kappa$, and we can prove that the kind κ is instantiable by producing some type $\Gamma \vdash \sigma : \kappa$, we will consider that a has type $\forall(\alpha : \kappa)\,\tau$ in context Γ. The Curry-style presentation, with no explicit syntax for type abstraction at the term level – we just write a, not $\Lambda(\alpha : \kappa)\,a$ – highlights that this form of polymorphism is erasable.

If on the contrary we do not know how to prove that κ is instantiable (or do not wish to do so), we may use *inconsistent* abstraction by building the term $\partial\, a$ at the distinct type[3] $\Pi(\alpha : \kappa)\,\tau$. This form blocks the reduction of a and is thus explicit at the term level.

To the explicit incoherent abstraction corresponds an explicit incoherent application, which unblocks computation: if the type σ has kind κ, then κ is in fact inhabited and $a \diamondsuit$ has type $\tau[\sigma/\alpha]$.

Even in full reduction (when reduction under λ's is allowed), reduction remains forbidden under ∂'s; an inconsistent abstraction is eliminated by the corresponding application, $(\partial\, a)\ \diamondsuit$, which reduces to a letting the evaluation of a be resumed. In contrast, consistent abstraction is erasable by construction: it is absent from the term itself, which alone determines reduction.

Issues with F_{cc} In the absence of inconsistency abstraction, the language F_{cc} has a full reduction semantics and is confluent. However, both properties break when introducing inconsistent abstraction, since inconsistent abstraction blocks the evaluation to maintain soundness. This amounts to introducing a form of weak reduction inside the language. While some reductions under ∂ are unsound and must be blocked, others may be harmless and could be safely reduced—but this is not allowed. This is going against our claim that core calculi ought to support full reduction to the largest possible extent.

Besides, it is well known that mixing weak and strong reductions may break confluence, and this problem affects F_{cc}. If b reduces to b', then $(\lambda(x)\,\partial\, x)\ b$ can reduce to either $(\lambda(x)\,\partial\, x)\ b'$ and then $\partial\, b'$, or to $\partial\, b$, which cannot be further reduced and, in particular, does not reduce to $\partial\, b'$ (or one of its reducts), as confluence would require.

[3] The notation $\Pi(\alpha : \kappa)\,\tau$ has nothing to do with dependent types.

These issues were well-understood by Cretin and Rémy [2014] and left for future work. We present an improved variant of F_{cc} that solves both problems simultaneously. In the course of doing so, we also encountered some more minor issues in the details of F_{cc}, which allowed us to also improve the system as a whole.

Propositional truths and hiding The language F_{cc} uses a blocking construct $\partial\, a$ to introduce the inconsistent abstraction $\Pi(\alpha : \kappa)\, \tau$. This does not match, however, the way potentially absurd assumptions are handled in dependent type theories, such as Coq, where reduction is blocked at the point of *use* of the assumption, not its point of *introduction*. This distinction is essential, in particular, to allow writing certified programs as Coq program terms: if the axioms (*e.g.* classical logic or proof irrelevance) are only used in logic parts of the formalization (under terms at type `Prop`), they get removed by extraction; a program whose correctness proof uses axioms can compute – while it would be blocked if we used our ∂ to introduce the axiom.

We therefore split inconsistent abstraction into two more atomic notions. First, an abstraction form *introduces* the assumption, but does not allow its implicit *use* yet; this does not block computation – the assumption is as frozen. Second, an *elimination* construct on frozen assumptions makes them available for implicit use – but blocks reduction.

Since the elimination construct blocks reduction, it needs to be present at the term level; it refers to assumption names introduced by the abstraction construct, which therefore also needs to be in terms – but without blocking reduction. In fact, we just reuse λ-abstraction for that purpose: locked assumptions are term variables at a new type $[P]$ of *propositional truths*, representing the assumption that the proposition P is true.

We write $\diamond$ for the introduction of propositional truths, and $\delta(a,\, \phi.b)$ for its elimination. Informally,[4] if the proposition P holds in the typing context Γ, then $\diamond$ is a witness of P at type $[P]$. The corresponding elimination rule, $\delta(a,\, \phi.b)$ computes a at type $[P]$, while blocking the reduction of b, type-checked under the assumption $\phi : P$, until a turns into a concrete witness $\diamond$. Then, $\delta(\diamond,\, \phi.b)$ can be reduced to the pseudo-substitution $b[\diamond/\phi]$ whose effect is to remove all occurrences of ϕ in b, and, finally, the reduction of b can proceed.

With these new constructions, we may use standard abstraction $\lambda(x : [P])\, a$ to abstract over a proposition P without blocking the evaluation of a, which means that a cannot use P yet. In particular, a may be of the form $a[\delta(x,\, \phi.b_1),\, \delta(x,\, \phi.b_2)]$, allowing the *implicit* use of the proposition P in subterms b_1 and b_2, which cannot be reduced, while full reduction is still allowed in a.

Propositional truth elimination allows the user to express the fact that an assumption P may not actually be used directly at its abstraction site, but only "at some later time". Conversely, there are situations where an elimination on P is needed to type-check parts of a term a and is no longer needed to typecheck some subterm b of a. To enable reduction of b in such a case, we introduce

[4] The language is formally defined in §2.

assumption hiding $\texttt{hide}\ \phi\ \texttt{in}\ b$, which enforces that the proposition variable ϕ will *not* be used implicitly in the subterm b. In exchange for losing this convenience, we regain the full reduction behavior for b.

While assumption hiding has been introduced for programming reasons, it is also instrumental in restoring confluence. The loss of confluence happens when a substitution places a reducible term in an irreducible context. We may now restore confluence by inserting appropriate hidings during substitution when traversing proposition eliminators so as to preserve reducibility.

Contributions The central, novel idea of our work is the interaction of the explicit and implicit modes of use of logical assertions in a programming calculus admitting full-reduction. From a theoretical point of view, implicitness was a somewhat-neglected design choice, and we propose a continuum between implicit and explicit uses thanks to *propositional truths* and *assumption hiding* (Section 2.2). It reveals, for example, that GADTs are fundamentally different from the usual algebraic datatypes. From a practical point of view, this gives the user flexible control over the scope of logical assumptions to prevent them from leaking into unrelated parts of his program—while retaining the convenience of their implicit invocation.

Another, more technical contribution is a new formal full-reduction calculus F_{th}, with inconsistent coercion abstraction that is confluent (Section 3.5). It is notable that the construction that regains confluence (*hiding*) was initially motivated by increasing the programmer's convenience.

Besides, there are several other contributions:

- We improve some details of the existing F_{cc} calculus, updating its mechanized soundness proof accordingly. Although coercion calculi in the spirit of F_{cc} are neither surface nor internal languages, they are good at exploring the design space; hence, even small improvements are valuable in the long term.
- We extend (3.5) the confluence proof technique of Takahashi [1995] so that it scales to larger calculi expressed in the Wright-Felleisen style, using reduction contexts to factor out common patterns and avoid a combinatorial increase in the number of cases.
- When translating between two given calculi, precisely establishing a bisimulation generally requires the use of an administrative variant of the target calculus; in our case, we need an administrative arrow type that is incompatible with the usual arrow type. While this is a common trick in the literature, its soundness proof is not as obvious as one would expect. We provide precise proofs that would be applicable to any calculus with several computational type constructors, *e.g.* arrows and products.

2 A calculus with propositional truths

In this section, we formally present our calculus, F_{th} – with propositional *truths* and *hiding*. As another instance of calculus based on erasable coercions, it is strongly inspired by the previous work on F_{cc} by Cretin and Rémy [2014] and

$$\textsc{TermVar}\quad \Gamma, x : \sigma, \Delta \vdash x : \sigma$$

$$\textsc{TermLam}\quad \frac{\Gamma \vdash \tau : \star \qquad \Gamma, x : \tau \vdash a : \sigma}{\Gamma \vdash \lambda(x)\, a : \tau \to \sigma}$$

$$\textsc{TermApp}\quad \frac{\Gamma \vdash a : \tau \to \sigma \qquad \Gamma \vdash b : \tau}{\Gamma \vdash a\ b : \sigma}$$

$$\textsc{TermProd}\quad \frac{\Gamma \vdash a : \tau_1 \qquad \Gamma \vdash b : \tau_2}{\Gamma \vdash (a, b) : \tau_1 * \tau_2}$$

$$\textsc{TermProj}\quad \frac{\Gamma \vdash a : \tau_1 * \tau_2}{\Gamma \vdash \pi_i\, a : \tau_i}$$

$$\textsc{TermCoerce}\quad \frac{\Gamma, \Sigma \vdash a : \tau \qquad \Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma}{\Gamma \vdash a : \sigma}$$

$$\textsc{TermWit}\quad \frac{\Gamma \vdash Q}{\Gamma \vdash \diamond : [Q]}$$

$$\textsc{TermAssume}\quad \frac{\Gamma \vdash a : [P] \qquad \Gamma, \phi : P \vdash b : \sigma}{\Gamma \vdash \delta(a,\, \phi.b) : \sigma}$$

$$\textsc{TermHide}\quad \frac{\Gamma \Vdash \Delta \qquad \Gamma \vdash \exists \Delta \qquad \Gamma, \Delta \vdash a : \tau}{\Gamma, \phi : P, \Delta \vdash \mathtt{hide}\, \phi\, \mathtt{in}\, a : \tau}$$

Fig. 1. F_{th} term typing judgment $\Gamma \vdash a : \tau$

$$\begin{array}{rcll}
a, b & ::= & x, y \ldots \mid \lambda(x)\, a \mid a\ a \mid (a, a) \mid \pi_i\, a & \text{Terms} \\
& \mid & \diamond \mid \delta(a,\, \phi.a) \mid \mathtt{hide}\, \phi\, \mathtt{in}\, a & \\
\tau, \sigma & ::= & \alpha, \beta \ldots \mid \tau \to \tau \mid \tau * \tau & \text{Types} \\
& \mid & \forall(\alpha : \kappa)\, \tau \mid (\tau, \sigma) \mid \pi_i\, \tau \mid () \mid [P] & \\
\kappa & ::= & \star \mid 1 \mid \kappa * \kappa \mid \{\alpha : \kappa \mid P\} & \text{Kinds} \\
P, Q & ::= & \top \mid P \wedge P \mid \forall(\alpha : \kappa)\, P \mid \exists \kappa \mid (\Sigma \vdash \tau) \Rightarrow \tau & \text{Prop.} \\
\Gamma, \Sigma, \Delta & ::= & \emptyset \mid \Gamma, x : \tau \mid \Gamma, \alpha : \kappa \mid \Gamma, \phi : P & \text{Contexts}
\end{array}$$

Fig. 2. Syntax of terms, types, kinds and propositions

follows the same global structure of judgments. Yet, we do not assume familiarity with F_{cc}.[5]

We first present the general structure of judgments and the constructs that are common to both F_{cc} and F_{th}, together with their typing rules (§2.1). We then detail the novel features of F_{th}, namely propositional truths and assumption hiding (§2.2). Last, we present the dynamic semantics of F_{th} (§2.3). In §2.4, we introduce a variant of F_{th} that is used to prove the soundness of F_{th} by translation to F_{cc} in several steps (§3).

2.1 Consistent coercion calculus

Cretin and Rémy [2014] use a general notion of *erasable coercions* with abstraction over consistent coercions to present different type system features, such as polymorphism, subtyping, and more in a common framework where these features can be easily composed together. The restriction that only *consistent* coercions can be abstracted over is key to *erasability*.

Our calculus has four syntactic categories: terms a, b; types τ, σ; kinds κ; and propositions P, Q. The syntax of each category and that of typing environments Γ, are described in Figure 2.

[5] F_{cc} also supports equi-recursive types; we left them out of this presentation as they are orthogonal to reduction under inconsistent assumptions. It is the only feature of F_{cc} as previously described that is absent from F_{th}.

The static semantics is given by four main judgments: a typing judgment $\Gamma \vdash a : \sigma$; a kinding judgment $\Gamma \vdash \sigma : \kappa$; a proposition satisfiability judgment $\Gamma \vdash P$; a coercion judgment $\Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$; plus a context consistency judgment $\Gamma \vdash \exists\Delta$ and well-formedness judgments $\Gamma \Vdash t$ where t may be an environment, a kind, a proposition, or a coercion.

Terms We first describe terms of the consistent subset of F_{th}, which are the terms of the untyped λ-calculus with products, extended with one additional construct for coercions. Other constructs for manipulating inconsistent assumptions, namely, propositional truth and assumption hiding will be presented in §2.2.

The term typing judgment is defined by the rules in Figure 1. The introduction and elimination rules for arrows (TermVar, TermLam, TermApp) and products (TermProd, and TermProj) are standard.

A remarkable feature of coercion calculi is that there is exactly one rule that does not change the term (and thus does not influence the dynamic semantics): the coercion rule TermCoerce. All runtime-irrelevant typing constructions, such as subtyping conversion and polymorphism introduction and elimination, are factorized into coercions. To express polymorphism, these coercions are *typing* coercions $(\Sigma \vdash \tau) \triangleright \sigma$ rather than *type* coercions $\tau \triangleright \sigma$: they also affect the typing environment Γ in which the coercion is used, by extending Γ with Σ when typechecking the premise of type τ, as described by Rule TermCoerce.

This factorization has been explained in previous works of Cretin and Rémy [2014] and is orthogonal to our point of interest in the present paper, namely, the interplay between *program types* and *logical propositions* in a programming system. We thus focus our presentation on propositions in general rather than coercions, and propositional truths would naturally extend to many other program logics, such as arithmetic reasoning or general refinement types. Still, by maintaining a crisp separation between (Curry-style) *program terms* that compute and *derivations* on which we statically reason, consistent coercion calculi are good systems in which to think about implicit versus explicit uses of logic reasoning in program terms.

Coercions Despite the fact that coercions are included in the syntactic class of propositions, there are still two separate judgments $\Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$ and $\Gamma \vdash P$.

The coercion judgment is defined in Figure 3. Besides structural rules of reflexivity (CoerRefl) and transitivity (CoerTrans), coercions have rules for polymorphism (type abstraction CoerGen and type application CoerInst), and rules CoerArrow, CoerProd, and CoerWit for distributivity of coercions under *computational* type constructors (those that describe the shape of terms and appear in the term typing judgment). Formulating rules for both polymorphism and distributivity under computational type constructors as coercions let us easily compose them: the F_η rules for instantiation of polymorphism under constructors naturally fall out as derived rules in consistent coercion calculi. Finally, Rule CoerProp injects any propositional proof of a coercion (seen as a proposition) into the coercion judgment – when the coercion context is consistent. We refer the reader to Cretin and Rémy [2014] for a detailed presentation.

$$\textsc{CoerRefl}\ \ \Gamma \vdash \tau \triangleright \tau \qquad \textsc{CoerTrans}\ \ \frac{\Gamma, \Sigma_1 \vdash (\Sigma_2 \vdash \tau_3) \triangleright \tau_2 \quad \Gamma \vdash (\Sigma_1 \vdash \tau_2) \triangleright \tau_1}{\Gamma \vdash (\Sigma_1, \Sigma_2 \vdash \tau_3) \triangleright \tau_1} \qquad \textsc{CoerGen}\ \ \frac{\Gamma \vdash \exists \kappa}{\Gamma \vdash (\alpha : \kappa \vdash \tau) \triangleright \forall (\alpha : \kappa) \tau}$$

$$\textsc{CoerInst}\ \ \frac{\Gamma \vdash \sigma : \kappa}{\Gamma \vdash \forall (\alpha : \kappa) \tau \triangleright \tau[\sigma/\alpha]} \qquad \textsc{CoerArrow}\ \ \frac{\Gamma, \Sigma \vdash \tau' \triangleright \tau \quad \Gamma \vdash \tau' : \star \quad \Gamma \vdash (\Sigma \vdash \sigma) \triangleright \sigma'}{\Gamma \vdash (\Sigma \vdash (\tau \to \sigma)) \triangleright (\tau' \to \sigma')}$$

$$\textsc{CoerProd}\ \ \frac{\Gamma \vdash (\Sigma \vdash \tau) \triangleright \tau' \quad \Gamma \vdash (\Sigma \vdash \sigma) \triangleright \sigma'}{\Gamma \vdash (\Sigma \vdash \tau * \sigma) \triangleright \tau' * \sigma'} \qquad \textsc{CoerWit}\ \ \frac{\Gamma, \phi : P \vdash Q}{\Gamma \vdash [P] \triangleright [Q]} \qquad \textsc{CoerProp}\ \ \frac{\Gamma \vdash (\Sigma \vdash \tau) \trianglerighteq \sigma \quad \Gamma \vdash \exists \Sigma}{\Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma}$$

Fig. 3. Coercion judgment $\Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$

$$\textsc{PropVar}\ \ \Gamma, \phi : P, \Delta \vdash P \qquad \textsc{PropAnd}\ \ \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \wedge P_2} \qquad \textsc{PropProj}\ \ \frac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_i} \qquad \textsc{PropGen}\ \ \frac{\Gamma \Vdash \kappa \quad \Gamma, \alpha : \kappa \vdash P}{\Gamma \vdash \forall (\alpha : \kappa)\, P}$$

$$\textsc{PropInst}\ \ \frac{\Gamma \vdash \forall (\alpha : \kappa)\, P \quad \Gamma \vdash \tau : \kappa}{\Gamma \vdash P[\tau/\alpha]} \qquad \textsc{PropTrue}\ \ \Gamma \vdash \top \qquad \textsc{PropConv}\ \ \frac{\Gamma \vdash P \quad P =_\beta P' \quad \Gamma \Vdash P'}{\Gamma \vdash P'}$$

$$\textsc{PropKind}\ \ \frac{\Gamma \vdash \tau : \{\alpha : \kappa \mid P\}}{\Gamma \vdash P[\tau/\alpha]} \qquad \textsc{PropInh}\ \ \frac{\Gamma \vdash \sigma : \kappa}{\Gamma \vdash \exists \kappa} \qquad \textsc{PropCoer}\ \ \frac{\Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma \quad \Gamma, \Sigma \vdash \tau : \star}{\Gamma \vdash (\Sigma \vdash \tau) \trianglerighteq \sigma}$$

Fig. 4. Proposition satisfiability judgment $\Gamma \vdash P$

Notice how the introduction rule for polymorphism CoerGen requires the quantified-over kind κ to be inhabited—the proposition $\exists \kappa$ denoting kind inhabitation. This is the cornerstone of the distinction between *consistent* and *inconsistent* polymorphism: to abstract over a potentially-absurd kind or proposition (you have no inhabitation proof at hand), one must instead use the inconsistent abstraction, which changes the term as it blocks the reduction.

Kinding, satisfiability, and consistency Figures 5, 4, and 6 present those three related judgments.

The proposition satisfiability judgment $\Gamma \vdash P$ is defined in Figure 4. Besides coercions $(\Sigma \vdash \tau) \trianglerighteq \sigma$, the propositional features inherited from F_{cc} are relatively limited: there are the features used to subsume existing System F variants with some form of constrained quantification (F_η, $\mathsf{F}_{<:}$, MLF *etc.*), but more propositions could be added. The trivial true proposition, conjunction of propositions, and type-polymorphic propositions have obvious introduction and elimination rules.

The kind inhabitation proposition $\exists \kappa$ is true whenever kind κ is inhabited by some type σ; we use the judgment $\Gamma \vdash \exists \kappa$ instead of $\Gamma \vdash \sigma : \kappa$ when only

$$\textsc{KindVar}\quad \Gamma, \alpha : \kappa, \Delta \vdash \alpha : \kappa$$

$$\textsc{KindArrow}\quad \frac{\Gamma \vdash \tau : \star \quad \Gamma \vdash \sigma : \star}{\Gamma \vdash \tau \to \sigma : \star}$$

$$\textsc{KindProd}\quad \frac{\Gamma \vdash \tau : \star \quad \Gamma \vdash \sigma : \star}{\Gamma \vdash \tau * \sigma : \star}$$

$$\textsc{KindWit}\quad \frac{\Gamma \Vdash P}{\Gamma \vdash [P] : \star}$$

$$\textsc{KindAll}\quad \frac{\Gamma \Vdash \kappa \quad \Gamma, \alpha : \kappa \vdash \tau : \star}{\Gamma \vdash \forall(\alpha : \kappa)\, \tau : \star}$$

$$\textsc{KindUnit}\quad \Gamma \vdash () : 1$$

$$\textsc{KindPair}\quad \frac{\Gamma \vdash \tau : \kappa_1 \quad \Gamma \vdash \sigma : \kappa_2}{\Gamma \vdash (\tau, \sigma) : \kappa_1 * \kappa_2}$$

$$\textsc{KindProj}\quad \frac{\Gamma \vdash \tau : \kappa_1 * \kappa_2}{\Gamma \vdash \pi_i\, \tau : \kappa_i}$$

$$\textsc{KindRefine}\quad \frac{\Gamma \vdash \tau : \kappa \quad \Gamma, \alpha : \kappa \Vdash P \quad \Gamma \vdash P[\tau/\alpha]}{\Gamma \vdash \tau : \{\alpha : \kappa \mid P\}}$$

$$\textsc{KindForget}\quad \frac{\Gamma \vdash \tau : \{\alpha : \kappa \mid P\}}{\Gamma \vdash \tau : \kappa}$$

$$\textsc{KindConv}\quad \frac{\Gamma \vdash \tau : \kappa \quad \kappa =_\beta \kappa' \quad \Gamma \Vdash \kappa'}{\Gamma \vdash \tau : \kappa'}$$

Fig. 5. Kinding judgment $\Gamma \vdash \tau : \kappa$

$$\textsc{ContEmpty}\quad \Gamma \vdash \exists \emptyset$$

$$\textsc{ContTerm}\quad \frac{\Gamma \vdash \exists \Delta \quad \Gamma, \Delta \vdash \tau : \star}{\Gamma \vdash \exists(\Delta, x : \tau)}$$

$$\textsc{ContType}\quad \frac{\Gamma \vdash \exists \Delta \quad \Gamma, \Delta \vdash \exists \kappa}{\Gamma \vdash \exists(\Delta, \alpha : \kappa)}$$

$$\textsc{ContProp}\quad \frac{\Gamma \vdash \exists \Delta \quad \Gamma, \Delta \vdash P}{\Gamma \vdash \exists(\Delta, \phi : P)}$$

Fig. 6. Context consistency judgment $\Gamma \vdash \exists \Delta$

consistency matters. It is defined in Figure 4. Inhabitation is lifted to whole contexts in Figure 6, as the judgment $\Gamma \vdash \exists \Delta$.

Kinding rules are defined in Figure 5. Kinding rules for base types are standard. The unit kind 1 is inhabited by the type-level trivial value $()$. Refinement kinds are the only construction introducing propositions in kinds—and thus in types: a refinement kind $\{\alpha : \kappa \mid P\}$ is inhabited by the types τ of kind κ such that the proposition $P[\tau/\alpha]$ holds. For example, the bounded quantification $\forall(\alpha \leq \tau)\, \sigma$ can be expressed as $\forall(\alpha : \{\alpha : \star \mid \alpha \geqslant \tau\})\, \sigma$.

Product kinds allow quantifying over several kinds at once; in combination with refinement kinds, this gives an expressive and convenient way to use refinement conditions $P(\alpha, \beta)$ that depend on several variables. In particular, $\forall(\gamma : \{\gamma : \kappa_1 * \kappa_2 \mid P(\pi_1\, \gamma, \pi_2\, \gamma)\})\, \tau$ cannot be expressed in the general case as a double abstraction of the form $\forall(\alpha : \kappa_1)\, \forall(\beta : \{\beta : \kappa_2 \mid P(\alpha, \beta)\})\, \tau$; the consistency proof in the former case requires a witness $\gamma : \kappa_1 * \kappa_2$ that satisfies P, while the consistency proof of the second abstraction in the latter case requires to provide a witness $\beta : \kappa_2$ for *any* fixed (rigid) variable $\alpha : \kappa_1$. Depending on P, the first form may be consistent and the second inconsistent; to split bindings while keeping consistency, one has to constrain the domain of α by writing $\forall(\alpha : \{\alpha : \kappa_1 \mid \exists\{\beta : \kappa_2 \mid P(\alpha, \beta)\}\})\, \forall(\beta : \{\beta : \kappa_2 \mid P(\alpha, \beta)\})\, \tau$, which inconveniently duplicates the proposition.

The reader may have recognized in refinement kinds a restricted form of (kind-level) dependent product. Indeed, this would exactly be a dependent product if the propositions were included into the kinds – dependent products would

$$\Gamma \Vdash \star \qquad \Gamma \Vdash 1 \qquad \frac{\Gamma \Vdash \kappa_1 \quad \Gamma \Vdash \kappa_2}{\Gamma \Vdash \kappa_1 * \kappa_2} \qquad \frac{\Gamma \Vdash \kappa \quad \Gamma, \alpha : \kappa \Vdash P}{\Gamma \Vdash \{\alpha : \kappa \mid P\}} \qquad \Gamma \Vdash \top$$

$$\frac{\Gamma \Vdash \kappa}{\Gamma \Vdash \exists \kappa} \qquad \frac{\Gamma \Vdash P \quad \Gamma \Vdash Q}{\Gamma \Vdash P \wedge Q} \qquad \frac{\Gamma \Vdash \kappa \quad \Gamma, \alpha : \kappa \Vdash P}{\Gamma \Vdash \forall(\alpha : \kappa)\, P}$$

$$\frac{\Gamma \Vdash \Sigma \quad \Gamma, \Sigma \vdash \tau : \star \quad \Gamma \vdash \sigma : \star}{\Gamma \Vdash (\Sigma \vdash \tau) \triangleright \sigma} \qquad \Gamma \Vdash \emptyset \qquad \frac{\Gamma \Vdash \Delta \quad \Gamma, \Delta \vdash \tau : \star \quad x \notin \Gamma, \Delta}{\Gamma \Vdash \Delta, x : \tau}$$

$$\frac{\Gamma \Vdash \Delta \quad \Gamma, \Delta \Vdash \kappa \quad \alpha \notin \Gamma, \Delta}{\Gamma \Vdash \Delta, \alpha : \kappa} \qquad \frac{\Gamma \Vdash \Delta \quad \Gamma \Vdash P \quad \phi \notin \Gamma, \Delta}{\Gamma \Vdash \Delta, \phi : P}$$

Fig. 7. Well-formedness judgments

then unify product kinds, refinement kinds, and conjunction of propositions. However, F_{cc}'s irrelevant handling of proposition proofs gives us very simple, clutter-free elimination rules for the refinement kind, which do not have to appear in the syntax of types. We occasionally benefit from that convenience.

The presence of type-level data structures (in our case product kinds) implies a need for type-level computation and identification of computationally-equal objects, in particular in rules PropConv and KindConv. The conversion rules for kinds and propositions allow to interchange well-formed objects equal upto β-reduction of projections, $\pi_i\,(\tau_1, \tau_2) =_\beta \tau_i$, and is closed by congruence and equivalence to all types, propositions, and kinds.

Well-formedness Figure 7 presents the well-formedness judgments of F_{cc} for contexts, kinds and propositions, which are all standard.

2.2 Propositional truths and hiding

The type $[P]$ represents the type of dynamic witnesses that P is satisfied. The type-checking rule for types of the form $[P]$ are listed in Figure 1. This type is introduced by the token $\diamond$, a ground value that inhabits $[P]$ exactly when the proposition P is satisfied in the current typing environment (Rule TermWit). It is eliminated by the construction $\delta(a,\, \phi.b)$, where a must have a propositional truth type $[Q]$, and b is type-checked in an extended context where the assumption $\phi : Q$ is implicitly available (Rule TermAssume) – until it is hidden again in some subterm of the form $\mathtt{hide}\,\phi\,\mathtt{in}\,a'$.

As any other computational type, there is a distributivity coercion for propositional truths, Rule CoerWit (Figure 3), which following F_{cc} design principle, can be derived and justified from the context $\delta(\Box,\, \phi.\diamond)$, as an η-expansion of the identity context $\Box$. Rule CoerWit tells us that a witness for a proposition P of type $[P]$ can be coerced into a witness for a proposition Q of type $[Q]$ whenever P implies Q as a proposition.

Finally, the typing rule TermHide (Figure 1) for $\mathtt{hide}\,\phi\,\mathtt{in}\,a$ in context $\Gamma, \phi : P, \Delta$ is a form of weakening of ϕ. It is only valid under the condition that

$$(\lambda(x)\, a)\; b \circ\!\!\rightarrow a[b/x]_\emptyset \qquad \delta(\diamond,\, \phi.b) \circ\!\!\rightarrow b[\diamond/\phi] \qquad \pi_i\,(a_1, a_2) \circ\!\!\rightarrow a_i$$

$$\textsc{Context}\ \frac{a \circ\!\!\rightarrow b \qquad \mathtt{unguarded}(E)}{E[a] \longrightarrow E[b]}$$

$$\begin{aligned} c_{th} &::= \lambda(x)\, a \mid (a, b) \mid \diamond \\ d_{th} &::= \Box\; b \mid \pi_i\, \Box \mid \delta(\Box,\, \phi.b) \\ \mathcal{E}_{th} &\triangleq \{E[a] \mid\ \mathtt{unguarded}(E),\ a = d_{th}[c_{th}],\ a \not\circ\!\!\rightarrow\} \end{aligned}$$

Fig. 8. Dynamic semantics of F_{th}

$$\begin{aligned} \mathtt{unguarded}(E) &\triangleq (\mathtt{guard}_\emptyset(E) = \emptyset) \\ \mathtt{guard}_S(\lambda(x)\, E) &\triangleq \mathtt{guard}_S(E) \\ \mathtt{guard}_S(a\ E) &\triangleq \mathtt{guard}_S(E) \\ \mathtt{guard}_S(E\ a) &\triangleq \mathtt{guard}_S(E) \end{aligned} \qquad \begin{aligned} \mathtt{guard}_S(\delta(E,\, \phi.b)) &\triangleq \mathtt{guard}_S(E) \\ \mathtt{guard}_S(\delta(a,\, \phi.E)) &\triangleq \mathtt{guard}_{S\cup\{\phi\}}(E) \\ \mathtt{guard}_S(\mathtt{hide}\,\phi\,\mathtt{in}\,E) &\triangleq \mathtt{guard}_{S\setminus\{\phi\}}(E) \\ \mathtt{guard}_S(\Box) &\triangleq S \end{aligned}$$

Fig. 9. Guards

$\Gamma \vdash \exists\Delta$ holds. This does not mean that Δ must be consistent (it can depend on variables in Γ that were introduced by blocking elimination), but that it is consistent *relative to* Γ.

Kind-level propositions. Propositional truths are named as such because they are constructed and abstracted over in terms, with an explicit elimination construction; by contrast with the *definitional* judgment $\Gamma \vdash P$ which only lives in typing derivations. Note that it is possible to see propositions as kinds: the kind $\{\alpha : 1 \mid P\}$, which could be abbreviated as $\langle P \rangle$, is inhabited by $()$ exactly when the proposition P is satisfiable.

2.3 Dynamic semantics

The dynamic semantics of F_{th} is defined in figures 8, 9 and 10. Because of assumption hiding, the notion of elimination contexts is non-standard: irreducible terms may have reducible subterms. In fact, the head β-reduction steps are also non-standard, because of the way hiding constructions are added during substitution of reducible values, so as to preserve confluence.

Reduction and head reduction We define a β-reduction relation ($\longrightarrow$) that is congruent to reduction contexts, and a *head* β-reduction relation ($\circ\!\!\rightarrow$) that only applies to head β-redexes (Figure 8). Distinguishing head reductions is important for the confluence proof (Section 3.5). Those reductions are fairly standard, except for the use of non-standard notions of substitution, and a side-condition on contexts described below.

$$
\begin{array}{rcll}
x[b/x]_S & \triangleq & \mathtt{hide}\, S\, \mathtt{in}\, b & \\
y[b/x]_S & \triangleq & y & (\text{if } y \neq x) \\
\diamond[b/x]_S & \triangleq & \diamond & \\
(\lambda(y)\, a)[b/x]_S & \triangleq & \lambda(y)\, a[b/x]_S & (\text{if } y \neq x) \\
(a\ a')[b/x]_S & \triangleq & a[b/x]_S\ a'[b/x]_S & \\
\delta(a,\, \phi.a')[b/x]_S & \triangleq & \delta(a[b/x]_S,\, \phi.a'[b/x]_{S\cup\{\phi\}}) & (\text{if } \phi \notin S) \\
(\mathtt{hide}\, \phi\, \mathtt{in}\, a)[b/x]_S & \triangleq & \mathtt{hide}\, \phi\, \mathtt{in}\, a[b/x]_{S\setminus\{\phi\}} &
\end{array}
$$

$$
\begin{array}{rcll}
x[\diamond/\phi] & \triangleq & x & \\
(\lambda(x)\, a)[\diamond/\phi] & \triangleq & \lambda(x)\, a[\diamond/\phi] & \\
(\mathtt{hide}\, \phi\, \mathtt{in}\, a)[\diamond/\phi] & \triangleq & a & \\
(\mathtt{hide}\, \psi\, \mathtt{in}\, a)[\diamond/\phi] & \triangleq & \mathtt{hide}\, \psi\, \mathtt{in}\, a[\diamond/\phi] & (\text{if } \psi \neq \phi)
\end{array}
$$

Fig. 10. Hiding and unhiding substitutions

Reduction contexts Full reduction is meant to allow any reduction path, so in general all one-hole term contexts E are reduction contexts. In F_{th}, subterms that are in the scope of an implicit assumption, or equivalently of a proposition variable, must still be blocked. We use an auxiliary function $\mathtt{guard}_S(E)$ to compute the set of proposition variables, called the *guards*, under which the hole $\Box$ of the single-hole context E is blocked, extended with an initial set S. The predicate $\mathtt{unguarded}(E)$ is then an abbreviation for the emptyness of $\mathtt{guard}_\emptyset(E)$. *Reduction contexts* are the *unguarded* one-hole contexts E. For example, $\delta(a,\, \phi.\Box)$ is not a reduction context, whereas $(\lambda(x)\, \Box)$ and $\delta(w_1,\, \phi.\delta(w_2,\, \psi.\mathtt{hide}\, \psi\, \mathtt{in}\, \mathtt{hide}\, \phi\, \mathtt{in}\, \Box))$ are. Unguardedness is checked by an additional premise in Rule Context.

Hiding substitution $a[b/x]_S$ In order to preserve confluence it is essential that β-reduction preserves reducibility of subterms. A counter-example for confluence in F_{cc}, translated in F_{th}, is the term $(\lambda(x)\, \delta(y,\, \phi.x))\ b$. The problem is that b appears in a reducible position but would become irreducible after one head reduction step, *i.e.* in the term $\delta(y,\, \phi.b)$—with the usual notion of reduction.

Our solution is to define the reduction of λ-redexes using a non-standard notion of substitution, $a[b/x]_\emptyset$ that inserts assumption hidings as necessary for substituted terms to remain reducible. For instance, $\delta(y,\, \phi.x)[b/x]_\emptyset$ is equal to $\delta(y,\, \phi.\mathtt{hide}\, \phi\, \mathtt{in}\, b)$. In general, this hiding substitution can be indexed by any guard, which is the list of logical assumptions made so far during term traversal.

The hiding substitution is defined on Figure 10 where $\triangleq$ stands for definition equality and $\mathtt{hide}\, S\, \mathtt{in}\, b$ is syntactic sugar for repeated hiding for all variables in the set S. Some cases that are simple traversals have been omitted.

Un-hiding substitution $b[\diamond/\phi]$ When the witness a of a propositional elimination $\delta(a,\, \phi.b)$ is blocked over reduces to $\diamond$, we know that the proposition witnessed by a is true, and the reduction of b can proceed. We remove each occurrence of $\mathtt{hide}\, \phi$ in each subterm of b, as it is not only unnecessary, but could also block now-reducible β-redexes if it remained: $a[\diamond/\phi]$ removes all occurrences of "$\mathtt{hide}\, \phi$" in the term b while traversing b. It is also defined in Figure 10 – we give only a few representative cases. Note that the typing rule for assumption hiding

guarantees that ϕ cannot appear in the subterm of $\texttt{hide}\,\phi$ – and this property is preserved by reduction.

Errors Head reduction occurs when a destructor of some computational type meets a constructor of the same type. An *immediate error* is a term whose head is a destructor applied on a constructor of a different type. Figure 8 defines destructor contexts d_{th} and constructor terms c_{th}; the set $\mathcal{E}_{th}$ of *errors* is then defined as immediate errors occurring in a reduction context. Note that being stuck on a free variable is not an error, that errors may still further reduce (if it contains other reducible positions with valid redexes), and that a non-error term may contain an immediate error blocked under a propositional elimination, such as $\lambda(x)\,\delta(x,\,\phi.\pi_1\,\mathsf{true})$ in our introductory example of abstracting over an equality between `int` and `bool`.

Below, we define errors for variants of our calculus in the same way, generated from a definition of constructor terms, destructor contexts, and the head reduction relation. Given a language of terms with a reduction relation and a set of errors $\mathcal{E}$, we say that a term a is *sound* if no reduction sequence starting from a ends in $\mathcal{E}$.

2.4 Two variants of F_{th}: F_t and F_{cc}

The soundness of F_{th} is proved by translation into F_{cc}, which has been proved sound [Cretin and Rémy, 2014]. In fact, the translation is in two steps, using an intermediate calculus F_t. Below, we formally define the calculi F_t and F_{cc}.

Removing assumption hiding. The language F_t is obtain from F_{th} by restricting to terms without hiding and by modifying the semantics of β-reduction so that it does not introduce hiding: the λ-reduction rule is $(\lambda(x)\,a)\;b \circ\!\!\rightarrow a[b/x]$. (As a consequence, F_t is not confluent.)

The rest of the definition is unchanged; in absence of hiding, unguarded contexts $\texttt{unguarded}(E)$ degenerate to a simpler, context-free definition that includes $\lambda(x)\,\Box$ and $\delta(\Box,\,\phi.b)$, but not $\delta(a,\,\phi.\Box)$; and unhiding substitutions $b[\diamond/\phi]$ leave terms unchanged. Error terms $\mathcal{E}_t$ are the subset of $\mathcal{E}_{th}$ of terms without hiding.

Primitive inconsistent abstraction. F_{cc} uses a different primitive of *inconsistent abstraction* to work with inconsistent propositions, or rather potentially-uninhabited kinds. Its construction $\partial\,a$, mentioned in the introduction, blocks reduction immediately and has a type of the form $\Pi(\alpha : \kappa)\,\tau$ stating that it assumes a type $\alpha : \kappa$ while κ may be uninhabited. Conversely, $a \diamondsuit$ unblocks a computation of type $\Pi(\alpha : \kappa)\,\tau$, whenever the kind κ can be shown inhabited. The head reduction rule is $(\partial\,a) \diamondsuit \circ\!\!\rightarrow a$, and reduction contexts are as before, if we consider that $\texttt{guard}_S(\partial\,E)$ is defined as $\texttt{guard}_{S\cup\phi}(E)$ where ϕ is a fresh propositional variable. The typing rules for those constructs are as follows:

$$\textsc{IncohIntro}\ \frac{\Gamma \Vdash \kappa \qquad \Gamma, \alpha : \kappa \vdash a : \tau}{\Gamma \vdash \partial\,a : \Pi(\alpha : \kappa)\,\tau} \qquad \textsc{IncohElim}\ \frac{\Gamma \vdash a : \Pi(\alpha : \kappa)\,\tau \qquad \Gamma \vdash \sigma : \kappa}{\Gamma \vdash a \diamondsuit : \tau[\sigma/\alpha]}$$

$$\textsc{KindIncoh}\quad \frac{\Gamma \Vdash \kappa \qquad \Gamma, \alpha : \kappa \vdash \tau : \star}{\Gamma \vdash \Pi(\alpha : \kappa)\, \tau : \star} \qquad \textsc{CoerIncoh}\quad \frac{\Gamma, \alpha : \kappa', \Sigma \vdash \sigma : \kappa \qquad \Gamma \vdash \exists \Sigma \qquad \Gamma, \alpha : \kappa' \vdash (\Sigma \vdash \tau[\sigma/\alpha]) \triangleright \tau'}{\Gamma \vdash (\Sigma \vdash \Pi(\alpha : \kappa)\, \tau) \triangleright \Pi(\alpha : \kappa')\, \tau'}$$

The set $\mathcal{E}_{cc}$ of F_{cc} error terms is generated from its head reduction, its constructor terms (as in F_{th}, but without $\diamond$ and with $(\partial\, a)$) and its destructor contexts (as in F_{th}, but without $\delta(\Box,\, \phi.b)$ and with $(\Box\ \diamondsuit)$).

3 Soundness and confluence

In this section, we prove our two technical results, confluence and soundness of F_{th}. The proof proceeds by a series of translations, proving that the source language is sound if the target language is sound as well. In §3.1, we recall the F_{cc} soundness result from previous work. In §3.2, we show a translation from the sublanguage F_t to (an administrative variant of) F_{cc}. This establishes soundness of F_t. In §3.5, we prove confluence of F_{th} using parallel reductions. For this we precisely define F_{th} multi-hole contexts, which give convenient tools to reason on its dynamic semantics. Finally, §3.6 proves soundness of F_{th}, using the tools introduced for the confluence proof.

3.1 Soundness of F_{cc}

F_{cc} comes with a (computer-checked) soundness proof for its (non-deterministic) reduction: starting from a well-typed term, no reduction path can lead to an erroneous stuck term. For deep reasons detailed in previous work, subject reduction (preservation of typing by reduction) does not hold for F_{cc}. Therefore, the soundness proof uses more semantic tools, building a model of the type system where types are sets of terms.

Theorem 1 (Previous work, Cretin and Rémy [2014] Soundness of F_{cc}). *Terms that are well-typed in F_{cc} in a consistent environment are sound. That is, if $\emptyset \vdash \exists \Gamma$ and $\Gamma \vdash a : \tau$, then a is sound.*

3.2 Translating propositional truths to F_{cc}

We now define a translation $[\![\_]\!]$ of terms, types and judgment derivations into F_{cc}. Informally, the idea of the translation is a form of CPS-encoding: we can translate a witness of type $[P]$ into a continuation consuming any inconsistent abstraction $\Pi(\alpha : \langle P \rangle)\, \tau$ to return a τ. Witness construction $\diamond$ would become the elimination continuation $\lambda(x)\, (x\ \diamondsuit)$, while propositional elimination $\delta(a,\, \phi.b)$ uses the translation of a as a continuation: $[\![a]\!]\ (\partial\ [\![b]\!])$.

The actual translation on terms and types is close to the informal description above, with an important difference. The informal translation gives the expected computational behavior to well-typed terms, but has the defect of mapping some terms that are errors in F_t to terms in F_{cc} that may still further reduce: for

example, $\delta((\lambda(x)\,x),\,\phi.y)$ is a stuck F_t term, but its translation $(\lambda(x)\,x)\;(\partial\,y)$ can be further reduced.

Because the soundness proof of F_{cc} is done semantically, and subject reduction does not hold for this calculus, it is important that our translation of F_t terms be well-behaved even on ill-typed terms. Indeed, we want to translate whole reduction paths starting from a known F_t term which, even if well-typed, may reduce to ill-typed terms (but, as we prove in this section, not an error). We also want to reason about the translation of those sound, ill-typed reducts.

To get a translation of $\delta((\lambda(x)\,x),\,\phi.y)$ that is stuck, we use a slight variant of F_{cc}, called $\mathsf{F}_{cc}^\flat$, for the target language. It is equipped with an "administrative" copy of the arrow type $(\tau \to^\flat \sigma)$, of λ-abstraction $(\lambda^\flat(x)\,a)$ and application $(a^\flat\;b)$. The type system and reduction semantics are exactly those of F_{cc}, with each rule (in the static and dynamic semantics) about λ-abstractions duplicated into an identical "administrative" variant.

The administrative $\lambda^\flat$ is entirely separate from the usual λ, and in particular $(\lambda(x)\,a)^\flat\;b$ and $(\lambda^\flat(x)\,a)\;b$ do not reduce and thus are both errors.

We can now formally define the translation from F_t to $\mathsf{F}_{cc}^\flat$, which makes judicious use of administrative constructions to preserve stuck terms. It is defined below on the F_t-specific constructions; it just preserves the structure of other constructions and translate their subterms (we use $\_$ for unused variable bindings):

$$\begin{aligned}
[\![[P]]\!] &\triangleq \forall(\beta:\star)\,(\Pi(_:\{_:1 \mid [\![P]\!]\})\,\beta) \to^\flat \beta \\
[\![\delta(a,\,\phi.b)]\!] &\triangleq [\![a]\!]^\flat\;(\partial\;[\![b]\!]) \\
[\![\diamond]\!] &\triangleq \lambda^\flat(x)\,(x\;\diamondsuit) \\
[\![\Gamma, \phi : P]\!] &\triangleq [\![\Gamma]\!], \alpha : \{_:1 \mid [\![P]\!]\}
\end{aligned}$$

For example, the translation of the error $\delta((\lambda(x)\,x),\,\phi.y)$ is now $(\lambda(x)\,x)^\flat\;(\partial\,y)$, which is also an error. One cannot build a counter-example of the form $\delta((\lambda^\flat(x)\,a),\,\phi.b)$ as the administrative variants are not part of the input language F_t. One can show by induction that the translation preserves errors and typing.

Lemma 1 (Error preservation of F_t).
A term a is an error in F_t if and only if $[\![a]\!]$ is an error in $\mathsf{F}_{cc}^\flat$.

Lemma 2 (Typing preservation of F_t).
If $\Gamma \vdash a : \tau$ in F_t, then $[\![\Gamma]\!] \vdash [\![a]\!] : [\![\tau]\!]$ in $\mathsf{F}_{cc}^\flat$.

Independently of typing preservation, we also prove a bisimulation property between F_t and $\mathsf{F}_{cc}^\flat$. It is not quite the case that any single-reduction step in F_t is turned into a single-reduction step of $\mathsf{F}_{cc}^\flat$, because the reduction of the translation of $\delta(\diamond,\,\phi.b)$, that is $(\lambda^\flat(x)\,(x\;\diamondsuit))^\flat\;(\partial\;[\![b]\!])$, does an extra administrative $\lambda^\flat$-reduction step before the expected ∂-reduction. We define the relation $(\circ\!\!\to_\flat)$ of *administrative* head β-reductions as the subset, in $\mathsf{F}_{cc}^\flat$, of reductions in $(\circ\!\!\to)$ of the form $(\lambda^\flat(x)\,a)^\flat\;b \circ\!\!\to a[b/x]$, and $(\longrightarrow_\flat)$, the administrative β-reductions, its closure $(\circ\!\!\to_\flat)$ under reduction contexts.

For any relation ($\mathcal{R}$), we define the relation $\mathcal{R}^?$ by $a\ \mathcal{R}^?\ b$ if and only if $(a\ \mathcal{R}\ b) \vee (a = b)$.

Lemma 3 (Bisimulation of F_t by $\mathsf{F}_{cc}^\flat$).
For any $a \longrightarrow a'$ in F_t, we have $[\![a]\!] \longrightarrow_\flat^? b \longrightarrow [\![a']\!]$ for some b in $\mathsf{F}_{cc}^\flat$.

Conversely, if $[\![a]\!] \longrightarrow b$ in $\mathsf{F}_{cc}^\flat$, then $a \longrightarrow a'$ for some a' such that either $b = [\![a']\!]$ or $[\![a]\!] \longrightarrow_\flat b \longrightarrow [\![a']\!]$.

Corollary 1. *If $[\![a]\!]$ is sound in $\mathsf{F}_{cc}^\flat$, then a is sound in F_t.*

Corollary 2. *If $\mathsf{F}_{cc}^\flat$ is sound, then so is F_t.*

We note that we do not need the bisimulation result to establish soundness (relative to $\mathsf{F}_{cc}^\flat$), but only the forward simulation and the forward translation of errors.

The backward simulation shows that besides having the same soundness property, F_t and $\mathsf{F}_{cc}^\flat$ are also the same in term of number of reductions up to administrative steps: reasoning on program efficiency can therefore also be transposed from one to the other. In this respect, it may be important to remark that the one computation step we allowed to neglect, the administrative $\lambda^\flat$-reduction, never performs arbitrary duplication of its argument: whenever it appears in the translation, the $\lambda^\flat$-variable appears exactly once in the body. We could better enforce this invariant by using a linear type for this administrative construction, but this would require invasive changes to the type system.

3.3 Translating F_{cc} into F_t

Just as we presented a translation from F_t into (an administrative variant of) F_{cc} to prove F_t's soundness, it is possible and enlightening to translate F_{cc} back into (an administrative variant of) F_t – after fixing a minor defect of F_{cc} as previously presented. By lack of space, we have not included this translation in the conference version of this article, but it is available in the full version.

3.4 Soundness of the administrative arrow

To conclude, from the two previous sections, that F_{cc}'s soundness implies F_t's soundness and conversely, we need to prove the soundness of the administrative variants relative to their base calculus. While this is a common technique, its soundness proof is actually not as obvious as one would expect. By lack of space, the proof is only available in the full version.

This result proves, in particular, the soundness of $\mathsf{F}_{cc}^\flat$ relative to F_{cc}. Along with Corollary 2, establishing the soundness of F_t relative to $\mathsf{F}_{cc}^\flat$, and the already established soundness of F_{cc} (Theorem 1) this concludes the soundness proof of F_t.

$$\Box_i : S \vdash \Box_i : S \qquad \emptyset \vdash x : S \qquad \frac{\Gamma \vdash E : S \backslash \{\phi\}}{\Gamma \vdash \mathtt{hide}\,\phi\,\mathtt{in}\,E : S} \qquad \frac{\Gamma \vdash E_1 : S \qquad \Delta \vdash E_2 : S \cup \{\phi\}}{\Gamma, \Delta \vdash \delta(E_1,\, \phi.E_2) : S}$$

$$\frac{\Gamma \vdash E_1 : S \qquad \Delta \vdash E_2 : S}{\Gamma, \Delta \vdash (E_1\ E_2), (E_1, E_2) : S} \qquad \frac{\Gamma \vdash E : S}{\Gamma \vdash (\lambda(x)\, E), (\pi_i\, E) : S}$$

$$\frac{a = E[x]^i \qquad x \notin E \qquad (\Box_i : S_i) \vdash E : S}{a[b/x]_S \triangleq E[\mathtt{hide}\, S_i\, \mathtt{in}\, b]^i} \qquad \frac{a = E[\mathtt{hide}\,\phi\,\mathtt{in}\,b_i]^i \qquad \phi \notin E}{a[\diamond/\phi] \triangleq E[b_i]^i}$$

Fig. 11. Guard analysis of multi-hole contexts

3.5 Confluence of F_{th}

Multi-hole contexts Figure 11 introduces a new judgment $(\Box_i : S_i)^{i \in I} \vdash E : S$, that is a simple syntactic analysis of the *guards* of a multi-hole context, that is the set of propositional variables that block the reduction of each hole. The judgment can be read as "if the whole term is guarded by S, then the i-th hole $\Box_i$ is guarded by S_i". A multi-hole context is just a term whose variables are, by convention, named $\Box_i$ for some i in I, and which appear only once in the term; we enforce that latter invariant by using disjoint union for the context union Γ, Δ, which corresponds to a simple linear typing discipline. The notation $E[\_]^{i \in I}$ corresponds to a context with a family of holes indexed by i, and in contexts $\amalg^{i \in I} \Delta_i$ is the disjoint union of a family of contexts $(\Delta_i)^{i \in I}$. For sake of brevity, we often leave I implicit and just write i instead of $i \in I$.

Notice that $\mathtt{guard}_S(E)$ for a single-hole context is uniquely defined by $\Box : \mathtt{guard}_S(E) \vdash E : S$. We also use multi-contexts to re-define the hiding substitution $a[b/x]_S$ defined in §2.3, and the hide-removing substitution $a[\diamond/\phi]$ used in the reduction rule for $\delta(\diamond,\, \phi.a)$.

Finally, a multi-context E is a *prefix* of E' (or a term, if E' has no holes) if E' can be obtained by substituting sub-contexts into the holes of E.

Parallel reductions We prove confluence using the Tait-Martin-Löf technique of parallel reductions, with a simple proof argument inspired by Takahashi [1995]. The idea of Takahashi is that parallel reduction (noted $a \Longrightarrow b$) for the simple λ-calculus with only arrows can be made deterministic by adding a redex-avoidance rule (the $a \neq (\lambda(\_)\ \_)$ hypothesis below meaning that a does not start with an abstraction) to the parallel reduction of application:

$$\frac{a \neq (\lambda(\_)\ \_) \qquad a \Longrightarrow a' \qquad b \Longrightarrow b'}{a\ b \Longrightarrow a'\ b'} \qquad \frac{a \Longrightarrow a' \qquad b \Longrightarrow b'}{(\lambda(x)\, a)\ b \Longrightarrow a'[b'/x]}$$

Without the redex-avoiding condition $a \neq \lambda(\_)\ \_$ in the application reduction rule, two reduction paths are available to each β-redex, performing the β-reduction or not. This gives a parallel condition that *may* reduce each redex in one step, and can thus subsume the usual single-step reduction relation by choosing to reduce exactly one redex. Takahashi remarks that the condition forces all redexes of the

$$\frac{E \not\to \quad (\Box_i : \emptyset)^i \vdash E : \emptyset \quad (a_i \circ\!\!\Rightarrow b_i)^i}{E[a_i]^i \Longrightarrow E[b_i]^i} \qquad \frac{(a_i \Longrightarrow a'_i)^i \quad R[a'_i]^i \stackrel{?}{\circ\!\!\to} b}{R[a_i]^i \circ\!\!\Rightarrow b}$$

$$R ::= (\lambda(x)\,\Box_1)\;\Box_2 \mid \pi_i\,(\Box_1, \Box_2) \mid \delta(\diamond,\, \phi.E_\phi[\mathtt{hide}\,\phi\,\mathtt{in}\,\Box_i]^i)$$
$$\text{(with } \mathtt{unguarded}(E_\phi) \text{ and } \phi \notin E_\phi)$$

Fig. 12. Parallel reduction

term to be reduced (Gross-Knuth reduction), and that this modified relation trivially forces confluence of the parallel reduction, of which it is a special case.

Adapting Takahashi's idea to a Wright-Felleisen setting of head reduction and elimination contexts suggests a new formulation, which is to decompose a reducible term a into the form $E[b_i]^i$ where the multi-context E *is not reducible* when seen as a term – a generalization of the redex-avoiding condition. For the same reason that Takahashi's reduction was deterministic, the decomposition of a into $E[b_i]^i$ where E is not reducible and the b_i are head redexes is unique, since E is the largest head context that does not contain redexes. This decomposition still let us define parallel reduction, rather than only the Gross-Knuth reduction.

Figure 12 gives the definition of our parallel reduction $a \Longrightarrow b$, mutually defined with the *head* parallel reduction $a \circ\!\!\Rightarrow b$ that reduces head redexes . The notation $E \not\longrightarrow$ can be understood in term of the single-step reduction relation, when E is seen as a term as any other: $\neg(\exists E',\, E \longrightarrow E')$.

The parallel reduction of $E[a_i]^i$ only happens when the a_i are all redexes, as they must be related to some b_i by the head parallel reduction ($\circ\!\!\Rightarrow$) that only starts from head redexes $R[\Box_i]^i$. Not all these redexes need to be reduced, however, as the head beta-reduction step $R[a'_i]^i \circ\!\!\to^? b$ is optional. In particular, taking $R[a'_i]^i = b$ for each redex shows that the relation ($\Longrightarrow$) is reflexive.

The restriction that the substituted terms a_i are redexes is crucial to modularly reason about reducibility; for if we substituted the non-redex $\lambda(x)\,a$ into the context $(\Box\; b)$, we would get a reducible result while neither the term nor the context were. No such situation can happen when the plugged terms are head redexes themselves, as redexes do not overlap.

Lemma 4 (Orthogonality).
Redexes do not overlap: If $(\Box_i : \emptyset)^i \vdash E : \emptyset$ is a one-hole irreducible context distinct from $\Box$, then for any redex contexts $R[\Box_j]^j$ and $(R'_i[\Box_k]^k)^i$ and families of terms $(a_j)^j$ and $(b_{i,k})^{i,k}$ we have $R[a_j]^j \neq E[R'_i[b_{i,k}]^k]^i$.

The other lemma we need, to prove the unicity of the decomposition by irreducible contexts, is about the structure of reducible positions in a term or context.

Lemma 5 (Reducible positions).
For any guard S, any term a has a minimal non-empty prefix F such that $(\Box_k : \emptyset)^k \vdash F[\Box_k]^k : S$. For any non-empty prefix F' of a with $(\Box_{k'})^{k'} \vdash F'[\Box_{k'}]^{k'} : S$, F is a prefix of F'. Furthermore, F is irreducible.

Lemma 6 (Unique decomposition of irreducible contexts).
If two parallel reductions have the same source, then they use the same context-redexes decomposition.

In the general case of filling a context E with subterms that are not necessarily head redexes, we may still reason on reducibility of the subterms:

Lemma 7 (Composability of parallel reduction).
The following rule, which does not constrain the $(a_i)^i$ to be head redexes or E to be irreducible, is admissible:

$$\frac{(\Box_i : \emptyset)^i \vdash E : \emptyset \qquad (a_i \Longrightarrow b_i)^i}{E[a_i]^i \Longrightarrow E[b_i]^i}$$

The last technical lemma we need closes a commutative diagram between parallel reduction and one-step head reduction.

Lemma 8 (Commutation $\Longrightarrow$ and $\circ\!\!\rightarrow$).
If $R[a_i]^i \Longrightarrow R[a'_i]^i$ and both $R[a_i]^i \circ\!\!\rightarrow b$ and $R[a'_i]^i \circ\!\!\rightarrow b'$, then $b \Longrightarrow b'$.

Note that it is precisely that last lemma that failed with F_t or F_{cc} without a hiding construct. Indeed, with $b \Longrightarrow b'$, reducing $(\lambda(x)\,\delta(y,\,\phi.x))\ b$ to $\delta(y,\,\phi.b)$ does not allow closing the diagram to $\delta(y,\,\phi.b')$, while reducing to $\delta(y,\,\phi.\texttt{hide}\,\phi\,\texttt{in}\,b)$ allows closing the diagram by reducing to $\delta(y,\,\phi.\texttt{hide}\,\phi\,\texttt{in}\,b')$.

Theorem 2. *The parallel reduction relation $(\Longrightarrow)$ is confluent.*

Corollary 3 (Confluence). *The relation $(\longrightarrow^*)$ is confluent.*

3.6 Soundness of F_{th}

The soundness proof of F_{th} is again a translation from F_{th} to F_t with a forward simulation. Before getting to the translation proper, we need to study two transformations used to define it. *Hide-extrusion* (3.6) removes hiding from a F_{th} term, and its correctness property let us simulate forward reductions of the form $\delta(\diamond,\,\phi.b) \circ\!\!\rightarrow b[\diamond/\phi]$. *Hide-normalization* (3.6) strengthens the structure of hiding in a F_{th} term, in such a way that we can forward-simulate the other F_{th} reductions, despite the mismatch between F_{th}'s hiding substitution $a[b/x]_S$ and F_t's natural substitution. We finally prove F_{th}'s soundness (Theorem 3).

Hide-extrusion In a language without hiding such as F_t, it is possible for the programmer to emulate the effects of hiding by extruding terms out of a blocking construction. Instead of $\delta(a,\,\phi.E[\texttt{hide}\,\phi\,\texttt{in}\,b])$, one can write $\texttt{let}\ x_b = b\ \texttt{in}\ \delta(a,\,\phi.E[x_b])$, where b appears in reducible position; we call this transformation *hide-extrusion*. In the general case, E may bind variables or block over other proposition variables, and the translation needs to be refined to preserve b's typing environment; for example, $\delta(a,\,\phi.\lambda(y)\,(f\ (\texttt{hide}\,\phi\,\texttt{in}\,b)))$ is hide-extruded into $\texttt{let}\ x_b = \lambda(y)\,b\ \texttt{in}\ \delta(a,\,\phi.\lambda(y)\,(f\ (x_b\ y)))$.

$$\delta(b,\, \phi.C[\texttt{hide}\,\phi\,\texttt{in}\,a]) \;\hookrightarrow\; \texttt{let}\; x = \texttt{abs}(C,a)\;\texttt{in}\;\delta(b,\, \phi.C[\texttt{app}(x,C)])$$

$$N \;::=\; \Box\, b \mid a\, \Box \mid (\Box, b) \mid (a, \Box) \mid \pi_i\, \Box \mid \sigma_i\, \Box \mid \delta(\Box,\, \phi.b) \qquad \text{Non-binding contexts}$$

$$\begin{aligned}
\texttt{abs}(\Box, a) &\triangleq a & \texttt{app}(a, \Box) &\triangleq a \\
\texttt{abs}(N[C], a) &\triangleq \texttt{abs}(C, a) & \texttt{app}(a, N[C]) &\triangleq \texttt{app}(a, C) \\
\texttt{abs}(\lambda(y)\, C, a) &\triangleq \lambda(y)\, \texttt{abs}(C, a) & \texttt{app}(a, \lambda(y)\, C) &\triangleq \texttt{app}(a\; y, C) \\
\texttt{abs}(\delta(w,\, \psi.C), a) &\triangleq \lambda(x_w)\, \delta(x_w,\, \psi.\texttt{abs}(C, a)) & \texttt{app}(a, \delta(x_w,\, \psi.C)) &\triangleq \texttt{app}(a\; \diamond, C)
\end{aligned}$$

Fig. 13. Hide-extrusion: translating `hide` back into plain F_t

Figure 13 gives a formal definition of the hide-extruding rewrite $a \hookrightarrow a'$ by defining two functions $\texttt{abs}(C, a)$, which abstracts a over all the variables bound in the context C, and $\texttt{app}(b, C)$, which closes such an abstracted b (when applied from under the context C) by applying it to the appropriate variables, accordingly. These definitions are factorized by a grammar N of context frames that do not bind any variable.

Lemma 9 (Typing preservation of hide-extrusion).
If a is well-typed in F_{th} and $a \hookrightarrow b$, then b is well-typed, at the same type.

Lemma 10 (Hide-extrusion of errors).
If $a \hookrightarrow b$, then a is an error if and only if b is an error.

Lemma 11 (Extrusion Reduction).
For all term a, we have $\texttt{app}(\texttt{abs}(C, a), C) \longrightarrow^* a[\diamond/\texttt{guard}_\emptyset(C)]$.

Each hide-extrusion rewrite removes exactly one $\texttt{hide}\,\phi$ from the source term; in particular, iterating hide-extrusion terminates, and gives a term without any hiding construct. We prove in 3.6 that this gives a forward simulation of F_{th} by F_t. This is easy to see in the simple case of extrusion through a reduction context E without any other $\texttt{hide}\,\phi$:

$$\delta(b,\, \phi.E[\texttt{hide}\,\phi\,\texttt{in}\,a]) \;\hookrightarrow\; \texttt{let}\; x = \texttt{abs}(E,a)\;\texttt{in}\;\delta(b,\, \phi.E[\texttt{app}(x,E)])$$

The only reducible subterms of the source term are a and b. The subterms b and a are still reducible in the target term, since in particular $\texttt{guard}_\emptyset(\texttt{abs}(E, \Box))$ is empty. If b eventually reduces to $\diamond$, then the source becomes $E[a]$ which may reduce further. The target can reduce to $\texttt{let}\; x = \texttt{abs}(E, a)\;\texttt{in}\; E[\texttt{app}(x, E)]$, which itself reduces to $E[\texttt{app}(\texttt{abs}(E, a), E)]$, then to $E[a]$ by the previous lemma 11.

Hide-normalization The remaining issue for a forward simulation of F_{th} by F_t is the difference between the substitutions used in β-reductions. If $(\lambda(x)\, a)\; b$ is related to some $(\lambda(x)\, a')\; b'$ by hide-extrusion, $a[b/x]_\emptyset$ may not be related to $a'[b'/x]$ in the general case, as the substitution in F_{th} may introduce new hiding constructs that have to be extruded again.

The idea of hide-normalization is to rewrite a term so that both substitutions *coincide*, by establishing the invariant that the guard of each bound variable

occurrence is equal to the guard of its binder. For example, in $\lambda(x)\,\delta(y,\,\phi.x)$ the guard of x's binding site is $\emptyset$, while its occurrence has guard $\{\phi\}$. β-reducing this λ-abstraction would introduce a $\texttt{hide}\,\phi$. We can statically rewrite it into $\lambda(x)\,\delta(y,\,\phi.\texttt{hide}\,\phi\,\texttt{in}\,x)$, which is equivalent (unblocking free variables doesn't affect reduction), and whose β-reduction doesn't introduce hiding.

In the general case, we define the hide-normalization function $\texttt{H}(a)$ from F_{th} to F_{th}. It recursively traverses all subterms and is a direct mapping, except :

$$\textsc{HideNormLam}\ \frac{(\Box_i : S_i)^i \vdash C : \emptyset \qquad x \notin C}{\texttt{H}(\lambda(x)\,C[x]^i) \triangleq \lambda(x)\,\texttt{H}(C[\texttt{hide}\,S_i\,\texttt{in}\,x]^i)}$$

Lemma 12 (Type preservation of hide-normalization).
If a is well-typed in F_{th}, then $\texttt{H}(a)$ is also well-typed, at the same type.

Lemma 13 (Error preservation of hide-normalization).
A F_{th} term a is an error if and only if $\texttt{H}(a)$ is an error.

Lemma 14 (Hide-normalization is stable by reduction).
If $\texttt{H}(a) \longrightarrow b'$, then b' is equal to $\texttt{H}(b)$ for some b.

Lemma 15 (Hide-normalization is a forward simulation).
If $a \longrightarrow b$ hen $\texttt{H}(a) \longrightarrow \texttt{H}(b)$.

Soundness Given a well-typed F_{th} term a, its hide-normalized form $\texttt{H}(a)$ is still well-typed and has the same reduction behavior – errors included. We can compute the maximal hide-extrusion a' of $\texttt{H}(a)$; this term is well-typed in both F_t and F_{th}. All that remains, to establish that the original term a is sound, is to forward-simulate any reduction path starting from a' in F_t. This should be done carefully, however, as it is *not* the case that the hide-extrusion of $\texttt{H}(a)$ is itself hide-normal; it is, except on the subterms created by hide-extrusion. Hide-extrusion introduces linearly-used variables to preserve scoping, and of course does insert the appropriate hiding constructs, as its goal is to remove hiding. Fortunately, we do not need to hide-normalize the terms produced by hide-extrusion: they remain well-separated from other subterms during reduction, and are not affected by β-reduction from other parts of the term.

Theorem 3 (Soundness of F_{th}). *Every well-typed F_{th} term is sound.*

4 Related and Future Work

Related Work

Confluence and weak reduction It appears to be folklore that there are three ways to get confluence in a weak reduction setting. One solution is to allow reduction under weak binders of subterms that do not use the bound variables [Çağman and Hindley, 1998]; we cannot apply this method in F_{th} as *uses* of propositions

are not traced in terms. Another solution is to introduce explicit weakening when substituting under a binder, so as to preserve the non-dependency with bound variables. This corresponds to our hiding substitution. Finally, one may use explicit substitutions and forbid them from going through weak bindings, so that the substituted terms remain reducible. Interestingly, this happens to be precisely the computational behavior of terms used in our final soundness proof (from F_{th} to F_t), as a result of hide-normalization followed by hide-extrusion.

Some explicit substitution calculi [Kesner, 2007] also have explicit weakening for the purpose of understanding reduction behavior of substructural systems (*e.g.* linear proof nets) where weakening must be applied *maximally* and this invariant is preserved by reductions and substitutions. This gives a reduction semantics that is different from our more relaxed system.

Another system with explicit weakening is Adbmal by Hendriks and van Oostrom [2003]. Their weakening construct enforces a well-parenthesized order between introduction and weakening by removing not just one variable from scope, but also all variables introduced afterwards. Our hiding construct allows non-bracketed introduction-hiding sequences, which is more convenient for the programmer. Interestingly, we also considered a construct `hide` $\star$ `in` a to hide *all* propositional variables in scope and simplify the definition of hiding substitutions, but the local use of hide-normalization in the soundness proof suffices to get a similar effect. The *scope-extrusion* performed before Adbmal's β-reductions, which extrudes the weakening above a bound variable to also weaken its binder is also related to our hide-normalization technique.

System FC The family of works on System FC [Sulzmann et al., 2007] is related to consistent coercion calculi in general, but also to our specific focus on implicit *v.s.* explicit use of potentially-inconsistent propositions. Sulzmann et al. [2007, §3.8] argue that explicit coercions often simplify understanding of compiler transformations by turning semantically incorrect hard-to-debug optimizations into scope-breaking transformations that are immediately detected. Implicit use of logical hypotheses is for user's convenience, and is not necessary in a compilter intermediate language. Yet, we claim that F_{th} retain some advantages in an explicit setting. The explicit reduction-blocking elimination reifies the semantic boundary into the syntax, which simplifies reasoning for both users and compiler designers. Another relation to our work is the march towards richer kind systems. F_{cc} includes a small set of features to demonstrate its usefulness, but the features studied for System FC, which moves towards a fully dependent type and kind sublanguage [Weirich et al., 2013], would also make sense in our setting. In particular, dependent kinds would make it natural to include propositions directly as kinds and merge product kinds, refinement kinds, and proposition conjunction as a single dependent product constructor. Consistency is known to be a pain point in the metatheory of System FC. It is neither needed nor traced in arbitrary coercion abstractions – they are not quite erasable as coercion abstraction blocks reduction. Yet, it is required for the axioms introduced at the toplevel – *e.g.* to model type families. An F_{cc}-inspired, more explicit treatment of consistency may structure System FC and provide optimization opportunities.

We know that the mode of use of coercions corresponding to bounded quantification is consistent and can be erased; but the practical question of how to decide consistency is not answered in our work.

Future Work

Completing consistent coercion calculi In the process of our work we have encountered small glitches in F_{cc}: rules that we would expect to be derivable, and that were not in the current system. We have fixed them as necessary for F_{th}'s need, but some aspects could still be improved – adding η-expansions in the kind equality, and understanding whether the context consistency requirement of coercions could be removed, and recovered by a semantics argument.

Extraction Coq's extraction process [Letouzey, 2004] compiles a language with full reduction and explicit uses of hypotheses into OCaml, a language with weak reduction and implicit uses of hypotheses; F_{th} might be a good intermediate language in which to express and study some of the optimizations happening during the translation—which is known to be difficult. More generally, the dependent type community is aware that computation is very different under arbitrary contexts [Brady et al., 2003]. We suspect that context consistency could be a good generalization of the "empty context" assumption. A distinction between *propositional* and *definitional* truths naturally arises in our framework and, interestingly, we have a use for abstracting over definitional truths – while dependent systems don't generally consider abstracting over definitional equalities.

Conclusion

We have introduced F_{th}, a consistent coercion calculus that blocks reductions under implicit inconsistent assumptions in a fine-grained manner. This solves both practical issues (user control over reducibility) and theoretical issues (confluence) with a previous calculus of erasable coercions, F_{cc}, and opens interesting perspectives on the study of full-reduction calculi for programming language design, the interplay between type systems and weak reduction strategies, and an explicit handling of consistency in dependent type systems.

Acknowledgments

Julien Cretin made many helpful remarks on our work; in particular, he suggested to introduce incoherent abstraction on *propositions* instead of *kinds*, which simplifies the presentation. We had fruitful discussions with Luc Maranget and Thibaut Balabonski about weak reduction; Thibaut Balabonski suggested the use of multi-hole contexts to unify guards (Figure 9) and hiding substitution (Figure 10), an idea we used in the proof of confluence.

Bibliography

M. Boespflug, M. Dénès, and B. Grégoire. Full reduction at full throttle. In *Certified Programs and Proofs (CPP)*, 2011.

E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In *TYPES*, pages 115–129, 2003.

N. Çağman and J. R. Hindley. Combinatory weak reduction in lambda calculus. *Theoretical Computer Science*, 198(1):239–247, 1998.

L. Cardelli. An implementation of FSub. Research Report 97, , 1993.

J. Cretin. *Erasable coercions: a unified approach to type systems.* PhD thesis, Université Paris-Diderot, Paris 7, 2014.

J. Cretin and D. Rémy. System F with Coercion Constraints. In *Logics In Computer Science (LICS)*. ACM, July 2014.

B. Grégoire and X. Leroy. A compiled implementation of strong reduction. *ACM SIGPLAN Notices*, 37(9):235–246, 2002.

D. Hendriks and V. van Oostrom. adbmal. In *CADE*, 2003.

D. Kesner. The theory of calculi with explicit substitutions revisited. In *Computer Science Logic*, pages 238–252. Springer, 2007.

D. Le Botlan and D. Rémy. MLF: Raising ML to the power of System-F. In *ICFP'80*, Aug. 2003.

P. Letouzey. A New Extraction for Coq. In *TYPES 2002*, Feb. 2004.

J. C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 2/3(76), 1988.

B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löfs type theory*, volume 200. Oxford University Press Oxford, 1990.

F. Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, Nov. 2000.

V. Simonet and F. Pottier. A constraint-based approach to guarded algebraic data types. *TOPLAS*, 29(1), Jan. 2007.

M. Sulzmann, M. M. T. Chakravarty, S. L. P. Jones, and K. Donnelly. System f with type equality coercions. In *TLDI*, pages 53–66, 2007.

M. Takahashi. Parallel reductions in lambda-calculus. *Inf. Comput.*, 118(1): 120–127, 1995.

D. Vytiniotis and S. P. Jones. Practical aspects of evidence-based compilation in system FC. , 2011.

S. Weirich, J. Hsu, and R. A. Eisenberg. System FC with explicit kind equality. In *ICFP*, pages 275–286, 2013.

H. Xi. Dependent ml an approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.