Adding Equations to System F Types

Neelakantan R. Krishnaswami[1] and Nick Benton[2]

[1] Max Planck Institute for Software Systems <neelk@mpi-sws.org>
[2] Microsoft Research <nick@microsoft.com>

Abstract. We present an extension of System F with types for term-level equations. This internalization of the rich equational theory of the polymorphic lambda calculus yields an expressive core language, suitable for formalizing features such as Haskell's rewriting rules mechanism or Extended ML signatures.

1 Introduction

Abstraction, modularity and information hiding are fundamental principles of software engineering and language design. Yet programming against an interface is often difficult, as conventional type systems can express only the most basic of the many assumptions and guarantees made by a module. The problem is that *too much* information is hidden, or only present in informal, ambiguous documentation. Dependent types allow much richer interfaces but come with their own costs, including a significant increase in the complexity of the language. Hoare-style program logics also allow much more expressive interfaces, but as well as being highly complex are arguably too decoupled from the underlying programming language; logics have their own syntax and typing cannot exploit logical specifications.

Compilers face problems similar to those of developers. If an optimizing compiler respects the modular structure of a program, it loses vital information that it could use to generate better code. But looking through abstractions to implementations is expensive and non-modular. Some compilers produce enriched interfaces for compiled modules that summarize the results of static analyses but the connection between this metadata and the program is often somewhat ad hoc, making it hard to soundly exploit the extra information in non-trivial transformations of client code.

In this paper, we address the question of how to incorporate more precise module specifications into a language via an extension of System F, the paradigmatic calculus for studying data abstraction. We restrict attention to specifications of a particular form, viz. equations between terms, understood as contextual equivalences. Equations are made part of the type system, making precise how they may be scoped, passed around and exploited. In this regard, our system resembles a restricted form of dependent types. However, equations are not actually proved within the language. A denotational model makes precise what semantic conditions must be verified in order to establish equalities; various techniques, from automatic analyses to interactive manual proofs, could be soundly used to check the conditions. This aspect resembles the treatment of entailment checking as a delegated semantic side-condition in Hoare-style program logics. The contributions of the paper may be summarized as follows:

– We extend the type system of F with a type of term-level equations.
– We illustrate how expressive this language is by encoding some of patterns of programming with dependent types and GADTs.
– We give a very simple parametric [20, 22] relational semantics to our extended language.
– We prove that our extended language is type-safe and terminating.
– We illustrate how the addition of equations enables useful new reasoning patterns in parametricity proofs, such as proofs of the equivalence of existential ML-style module interfaces with Church-style datatype encodings.

2 Informal Overview

This section describes our language, $F_{\equiv}$, semi-formally, and gives examples of its use. The grammar is shown in Figure 1. The types of $F_{\equiv}$ are the usual ones of System F, extended with a new type former $e \equiv_A e'$, which asserts the equality of e and e' at type A. Since types now contain terms, and in particular term-level variables, we change the syntax of the function type from $A \to B$ to $(x : A) \to B$, so that function arguments of type A can be referenced within B. This resembles the dependent product of dependent type theory, and ensures that the argument of a function can be mentioned in any returned equality types.

The terms of the language include those of System F, including variables x; type abstractions $\Lambda\alpha.\ e$ and type applications $e\ [A]$; and term abstractions $\lambda x.\ e$ and term application $e\ e'$. We defer giving the precise typing rules until Section 3, but they very closely resemble their counterparts in System F.

There are also two new term-level constants: $\bullet$ and abort. The term $\bullet$ is the sole inhabitant of the equality type $e \equiv_A e'$ when the equation holds (the type is uninhabited otherwise). We remark that $\bullet$ does not provide any intrinsic evidence of the equality — the right to introduce a $\bullet$ arises as a semantic side-condition. The absence of evidence keeps the term language very simple, since the $\bullet$ is merely a placeholder for the reflection of a fact in the semantic model back into the language's type system. The abort constant allows equational information to influence typing: it has arbitrary type, but only if the context is (semantically) inconsistent. One can thus do deep semantic proofs to justify complicated equations, inject those into the types, and then use simple syntactic means to handle the plumbing which pushes facts around to other parts of the program.

Figure 2 presents the notational abbreviations that we use in the remainder of the paper. These are mostly well-known Church encodings, but note particularly the weak sum type $\exists x : A.\ B$. We'll also assume standard syntactic sugar (e.g. projections, case analysis) associated with these abbreviations in examples. (The model we present in Section 3 can be used to show that these suggestive abbreviations actually have the intended semantics.)

$$
\begin{array}{ll}
\text{Types} & A ::= \alpha \mid \forall\alpha.\ A \mid (x : A) \to B \mid e \equiv_A e' \\
\text{Terms} & e\ ::= \Lambda\alpha.\ e \mid e\ [A] \mid \lambda x.\ e \mid e\ e' \mid \bullet \mid \mathsf{abort} \mid x \\
\text{Values} & v\ ::= \Lambda\alpha.\ e \mid \lambda x.\ e \mid \bullet \\
\text{Contexts} & \Gamma ::= \cdot \mid \Gamma, \alpha \mid \Gamma, x : A
\end{array}
$$

Fig. 1. Syntax

$A \to B$	$= (x : A) \to B$, when $x \notin \mathrm{FV}(B)$
$\forall x : A.\ B$	$= (x : A) \to B$
$\forall x_1 : A_1, \ldots, x_n : A_n.\ B$	$= \forall x_1 : A_1.\ \ldots \forall x_n : A_n.\ B$
$A \times B$	$= \forall\alpha.\ (A \to B \to \alpha) \to \alpha$
1	$= \forall\alpha.\ \alpha \to \alpha$
$\exists x : A.\ B$	$= \forall\alpha.\ ((x : A) \to B \to \alpha) \to \alpha$
$\exists x_1 : A_1, \ldots, x_n : A_n.\ B$	$= \exists x_1 : A_1.\ \ldots \exists x_n : A_n.\ B$
$\exists\alpha.\ A$	$= \forall\beta.\ (\forall\alpha.\ A \to \beta) \to \beta$
$A + B$	$= \forall\alpha.\ (A \to \alpha) \to (B \to \alpha) \to \alpha$
$\bot$	$= \forall\alpha.\ \alpha$
$\neg A$	$= A \to \bot$

Fig. 2. Type Abbreviations

2.1 Examples

Refined Typings Equations allow extra constraints to be imposed on arguments and extra guarantees to be given for results. For example, a function that should only be called with commutative binary operations on a type A might be given a type like

$$(f : A \to A \to A) \to (\forall a : A, a' : A. f\ a\ a' \equiv_A f\ a'\ a) \to A$$

For producing values together with assertions of their equational properties, one makes use of existential packages. For example, a function yielding idempotent unary operations on A from arguments of type B could be typed as

$$B \to (\exists f : A \to A. \forall a : A. f\ (f\ a) \equiv_A f\ a)$$

Clients can project both the underlying value and the equational information from such packages and use them to justify their own equations.

Datatype Encodings More useful examples of $F_=$ involve enriching the signatures of modules, encoded using second-order existential types in the standard way [17, 21]. We now give a few examples of how one can type abstract datatypes in $F_=$, encapsulating types, operations on those types and algebraic properties satisfied by those operations.

Booleans We begin by giving an $F_=$ encoding of an interface to the Booleans.

```
$\exists \mathsf{bool},$
  $\mathit{true} : \mathsf{bool},$
  $\mathit{false} : \mathsf{bool},$
  $\mathit{if} : \forall\alpha.\ \mathsf{bool} \to \alpha \to \alpha \to \alpha.$
  $\forall\alpha, a : \alpha, a' : \alpha.\ \mathit{if}\ [\alpha]\ \mathit{true}\ a\ a' \equiv_\alpha a\ \times$
  $\forall\alpha, a : \alpha, a' : \alpha.\ \mathit{if}\ [\alpha]\ \mathit{false}\ a\ a' \equiv_\alpha a'$
```

The signature exposes constructors and eliminators for the boolean type, together with their β-theory as equational properties. A natural question is how this module type relates to the traditional Church encoding of booleans. One can certainly give an implementation of the abstract datatype in terms of that encoding, in which bool is instantiated with $\forall\alpha.\ \alpha \to \alpha \to \alpha$. More surpisingly perhaps, as we will show more formally in Section 4, the extended module type uniquely characterizes the booleans, and $F_=$ allows clients to exploit the consequences of this semantic fact. Note that the interface does not explicitly state any of the properties which ordinarily characterize datatypes, such as the disjointness of *true* and *false*, or that they are the only constructors for the boolean type. However, the presence of the eliminator and its equational theory, plus parametricity, are sufficient to derive these properties: parametricity ensures that *true* and *false* are the only way to construct the booleans, and furthermore, they must be disjoint, or else we could use the equational theory of *if* to derive a contradiction.

Natural Numbers As an example of a non-finite type, we can give an interface for the type of natural numbers:

```
$\exists \mathsf{nat},$
  $z : \mathsf{nat},$
  $s : \mathsf{nat} \to \mathsf{nat},$
  $\mathit{iter} : \forall\alpha.\ \alpha \to (\alpha \to \alpha) \to \mathsf{nat} \to \alpha.$
  $\forall\alpha, i, f.\ \mathit{iter}\ [\alpha]\ i\ f\ z \equiv_\alpha i\ \times$
  $\forall\alpha, i, f, x.\ \mathit{iter}\ [\alpha]\ i\ f\ (s\ x) \equiv_\alpha f(\mathit{iter}\ [\alpha]\ i\ f\ x)$
```

This interface says that we have an abstract type nat, with constructors z and s. We have an eliminator *iter*, and two equations explaining how to eliminate zero and successor.

This signature does *not* expose the representation of nat, nor does it specify the implementation of *iter*. We are free to implement the natural numbers in any way we like – for example, with a binary (rather than unary) representation. Furthermore, in Section 4 we will prove that this signature is isomorphic to the Church encoding of the natural numbers, which means that any implementation of this type actually is an implementation of the natural numbers.

Lists Here is a possible interface for the type of lists of booleans.

```
$\exists \mathsf{List}_{\mathsf{bool}},$
  $\mathit{nil} : \mathsf{List}_{\mathsf{bool}},$
```

$$
\begin{aligned}
&\mathit{cons} : \mathsf{bool} \to \mathsf{List}_{\mathsf{bool}} \to \mathsf{List}_{\mathsf{bool}},\\
&\mathit{map} : (\mathsf{bool} \to \mathsf{bool}) \to \mathsf{List}_{\mathsf{bool}} \to \mathsf{List}_{\mathsf{bool}},\\
&\mathit{fold} : \forall \alpha.\ \alpha \to (\mathsf{bool} \to \alpha \to \alpha) \to \mathsf{List}_{\mathsf{bool}} \to \alpha,\\
&\mathit{map}\ \mathit{id}_{\mathsf{bool}} \equiv_{\mathsf{List}_{\mathsf{bool}}} \mathit{id}_{\mathsf{List}_{\mathsf{bool}}} \times\\
&\forall f.\ \mathit{map}\ f\ \mathit{nil} \equiv_{\mathsf{List}_{\mathsf{bool}}} \mathit{nil} \times\\
&\forall f, b, bs.\ \mathit{map}\ f\ (\mathit{cons}\ b\ bs) \equiv_{\mathsf{List}_{\mathsf{bool}}} \mathit{cons}\ (f\ b)\ (\mathit{map}\ f\ bs) \times\\
&\forall f, g : \mathsf{bool} \to \mathsf{bool}.\ (\mathit{map}\ f) \circ (\mathit{map}\ g) \equiv_{\mathsf{List}_{\mathsf{bool}}} \mathit{map}\ (f \circ g) \times\\
&\forall \alpha, a, f.\ \mathit{fold}\ [\alpha]\ a\ f\ \mathit{nil} \equiv_{\alpha} a \times\\
&\forall \alpha, a, f, b, bs.\ \mathit{fold}\ [\alpha]\ a\ f\ (\mathit{cons}\ b\ bs) \equiv_{\alpha} f\ b\ (\mathit{fold}\ [\alpha]\ a\ f\ bs)
\end{aligned}
$$

This example illustrates that the interface does not have to precisely match the constructors of the Church encoding for lists — in this interface, we include the *map* operation. However, since the interface tells us what the behavior of *map* is, we can still prove that all values of list type can be built up just from *nil* and *cons*.

This lets us greatly extend the interface to a module, without having to give up natural reasoning principles for it. For example, suppose we extend lists with a left fold operator, characterized by the following signature:

$$
\begin{aligned}
&\mathit{foldl} : \forall \alpha.\ \alpha \to (\mathsf{bool} \to \alpha \to \alpha) \to \mathsf{List}_{\mathsf{bool}} \to \alpha\\
&\forall \alpha, i, f.\ \mathit{foldl}\ [\alpha]\ i\ f\ \mathit{nil} \equiv_{\alpha} i\\
&\forall \alpha, i, f, b, bs.\ \mathit{foldl}\ [\alpha]\ i\ f\ (\mathit{cons}\ b\ bs) \equiv_{\alpha} \mathit{foldl}\ [\alpha]\ (f\ b\ i)\ f\ bs
\end{aligned}
$$

With this definition in hand, clients can establish things like the fact that if a function is associative and commutative, then the left and right folds coincide — a fact which we can encode in types:

$$
\begin{aligned}
&\mathit{assoc}_A(f) \triangleq \forall a_1, a_2, a_3.\ f\ (f\ a_1\ a_2)\ a_3 \equiv_A f\ a_1 (f\ a_2\ a_3)\\
&\mathit{comm}_A(f) \triangleq \forall a_1, a_2.\ f\ a_1\ a_2 \equiv_A f\ a_2\ a_1
\end{aligned}
$$

$$
\forall \alpha, f, i, bs.\ \mathit{assoc}_{\alpha}(f) \to \mathit{comm}_{\alpha}(i) \to \mathit{fold}\ f\ i\ bs \equiv_{\alpha} \mathit{foldl}\ f\ i\ bs
$$

Here we exploit the usual Curry-Howard pun, using the function type to do the duty of a logical implication. The proof of the equation above goes by induction on the nil/cons structure of lists, which is only possible because we can prove that this structure exists via parametricity.

2.2 Applications

Deforestation of Higher-Order Programs Consider the following sequence of equational rewrites.

$$
\begin{aligned}
(\mathit{map}\ \mathit{not}) \circ (\mathit{map}\ \mathit{not}) &= \mathit{map}\ (\mathit{not} \circ \mathit{not})\\
&= \mathit{map}\ \mathit{id}_{\mathsf{bool}}\\
&= \mathit{id}_{\mathsf{List}_{\mathsf{bool}}}
\end{aligned}
$$

This is a standard example of deforestation [15], in which two intermediate data structures (a negated list and a double-negated list) are not generated, in order to save memory usage. Deforestation offers many interesting examples, since it appeals to equational properties which go well beyond simple inlining and other forms of compile-time β-reduction.

However, now consider the following expression h:

$$h \triangleq \lambda map.\ (map\ not) \circ (map\ not)$$

The question is, can the body of h be simplified? In general, the answer will be no — unless we can prove that only *map* functionals satisfying the right equational properties flow into the lambda, we are not justified in rewriting this expression. Hence we are in the somewhat ironic position that deforestation — an optimization invented to make higher-order programming more efficient — is often inapplicable in client programs which make use of higher-order functions.

One way around this difficulty would be if we could prove the soundness of these transformations in *open* contexts, where we don't know exactly which lambda-terms might flow into a higher-order program. By adding the necessary information as type data to the programming language, we can rely on the necessary equational properties to hold without having to make the concrete bindings of terms like *map* and *not* visible.

So by rewriting h to pass in the desired properties, we can ensure that a rewriting is performable *in an open context*:

$$\begin{array}{ll}
h\ :\ & ((map : \mathsf{bool} \to \mathsf{bool}) \to \mathsf{List}_{\mathsf{bool}} \to \mathsf{List}_{\mathsf{bool}}) \\
& \quad \to map\ id_{\mathsf{bool}} \equiv_{\mathsf{List}_{\mathsf{bool}} \to \mathsf{List}_{\mathsf{bool}}} id_{\mathsf{List}_{\mathsf{bool}}} \\
& \quad \to \forall f, g : \mathsf{bool} \to \mathsf{bool}.\ (map\ f) \circ (map\ g) \equiv_{\mathsf{List}_{\mathsf{bool}} \to \mathsf{List}_{\mathsf{bool}}} map\ (f \circ g) \\
& \quad \to \mathsf{List}_{\mathsf{bool}} \to \mathsf{List}_{\mathsf{bool}}
\end{array}$$

$$h \triangleq \lambda map.\ \lambda pf.\ \lambda pf'.\ (map\ not) \circ (map\ not)$$

Here, we do not need to know what the actual implementation of *map* is, since we have stipulated that the function h must be called only with functions which have the equational properties we need them to satisfy, and hence we can conclude that h is equivalent to id.

GADT-style Encodings and Making Unsafe Operations Safe Generalized algebraic data types [11] extend ordinary algebraic datatypes with index information permitting static types to contain information about the specific data constructors used to build them. This lets programmers use dynamic run-time tests to gain additional information about the static type of terms. Since our type system lets us directly place information about terms into types, many of these encodings can be fit into our framework.

Concretely, consider the following specification of an option type.

$$\begin{array}{l}
\exists \mathsf{option}_A, \\
\quad none : \mathsf{option}_A, \\
\quad some : A \to \mathsf{option}_A, \\
\quad case : \forall \alpha.\ \mathsf{option}_A \to \alpha \to (A \to \alpha) \to \alpha.
\end{array}$$

$$\forall \alpha.\ (v : \alpha) \to (f : A \to \alpha) \to \mathit{case}\ [\alpha]\ \mathit{none}\ v\ f \equiv_\alpha v \quad \times$$
$$\forall \alpha.\ (a : A) \to (v : \alpha) \to (f : A \to \alpha) \to \mathit{case}\ [\alpha]\ (\mathit{some}\ a)\ v\ f \equiv_\alpha f\ a$$

This specification follows the pattern of the boolean and list types earlier, containing the *none* and *some* constructors as well as the *case* eliminator for them, plus equations describing the β-theory of *case*.

We can use this specification to write refined case programs which return type-level evidence of equalities. First, we define a variant case function that returns not only a value, but also a proof that the returned value is equal to the input.

$$\begin{aligned} &\mathit{case}' \ :\ (x : \mathsf{option}_A) \to (x \equiv_{\mathsf{option}_A} \mathit{none}) + (\exists a : A.\ x \equiv_{\mathsf{option}_A} \mathit{some}\ a) \\ &\mathit{case}' \triangleq \lambda x.\ \mathit{case}\ [\ldots]\ (\mathit{inl}\ \bullet)\ (\lambda a.\ \mathit{inr}\ (a, \bullet)) \end{aligned}$$

As can be seen from the type, case' takes an option and returns a sum type. If the argument is *none*, it returns the left branch containing the static fact that the argument is *none*, and if the argument is a *some*, then it returns a value a such that *some* a is equal to the argument of *case*. All of the equality type terms are witnessed by $\bullet$ terms.

$$\begin{aligned} &\mathit{valOf}\ :\ (x : \mathsf{option}_A) \to \neg(x \equiv_{\mathsf{option}_A} \mathit{none}) \to A \\ &\mathit{valOf} \triangleq \lambda x.\ \lambda pf.\ \mathit{case}\ [A]\ \mathsf{abort}\ (\lambda a.\ a) \end{aligned}$$

This operator takes an option and a proof that it is not equal to *none*. This lets us pass abort as an argument in the untaken branch, since we know the *case* can only be reduced when it has a proof that its argument is not *none*.

As before, these operations are provably correct only when injectivity and disjointness hold, and again, these properties are provable from the interface specification. As a result, we can define these apparently-unsafe operators *outside* the body of the module, since our program *valOf* only relies on the equational properties specified in the interface, and not on the specifics of the implementation.

3 Syntax and Semantics

Before proceeding to the metatheory, we give a high-level overview of the structure of this section.

1. We give the syntax of terms and types, and an *untyped* operational semantics for our programming language. This language contains an abort construct which can get stuck.
2. We define a "pre-typing" relation, which judges whether terms and types are syntactically well-formed. Unlike a true type system, our pretyping system is (by design) unsound: there are no restrictions on the use of the abort connective.
3. However, the pretyping relation offers enough structure that we can define a binary logical relation giving semantics to each of the type constructors, by structural induction on the pretyping relation.

$$\frac{}{v \Downarrow v} \qquad \frac{e_0 \Downarrow \lambda x.\ e_0' \qquad [e_1/x]e_0' \Downarrow v}{e_0\ e_1 \Downarrow v} \qquad \frac{e_0 \Downarrow \Lambda\alpha.\ e_0' \qquad [A/\alpha]e_0 \Downarrow v}{e_0\ [A] \Downarrow v}$$

Fig. 3. Operational Semantics

$$\boxed{\Gamma \rhd \mathrm{ok}} \qquad \boxed{\Gamma \rhd A} \qquad \boxed{\Gamma \rhd e : A}$$

$$\frac{}{\cdot \rhd \mathrm{ok}} \qquad \frac{\Gamma \rhd \mathrm{ok}}{\Gamma, \alpha \rhd \mathrm{ok}} \qquad \frac{\Gamma \rhd \mathrm{ok} \qquad \Gamma \rhd A}{\Gamma, x : A \rhd \mathrm{ok}}$$

$$\frac{\Gamma \rhd \mathrm{ok} \qquad \alpha \in \Gamma}{\Gamma \rhd \alpha} \qquad \frac{\Gamma, \alpha \rhd A}{\Gamma \rhd \forall\alpha.\ A} \qquad \frac{\Gamma \rhd A \qquad \Gamma, x : A \rhd B}{\Gamma \rhd (x : A) \to B}$$

$$\frac{\Gamma \rhd A \qquad \Gamma \rhd e : A \qquad \Gamma \rhd e' : A}{\Gamma \rhd e \equiv_A e'}$$

$$\frac{\Gamma, x : A \rhd e : B}{\Gamma \rhd \lambda x.\ e : (x : A) \to B} \qquad \frac{\Gamma \rhd e : (x : A) \to B \qquad \Gamma \rhd e' : A}{\Gamma \rhd e\ e' : [e/x]B}$$

$$\frac{\Gamma, \alpha \rhd e : A}{\Gamma \rhd \Lambda\alpha.\ e : \forall\alpha.\ A} \qquad \frac{\Gamma \rhd e : \forall\alpha.\ B \qquad \Gamma \rhd A}{\Gamma \rhd e\ [A] : [A/\alpha]B} \qquad \frac{\Gamma \rhd \mathrm{ok} \qquad x : A \in \Gamma}{\Gamma \rhd x : A}$$

$$\frac{\Gamma \rhd e \equiv_A e'}{\Gamma \rhd \bullet : e \equiv_A e'}\ (\textsc{Danger1}) \qquad \frac{\Gamma \rhd A}{\Gamma \rhd \mathsf{abort} : A}\ (\textsc{Danger2})$$

Fig. 4. Pretyping Relation

4. Then, we give the true typing relation, which refines the pretyping relation to include semantic side-conditions on equality formation and the use of abort.
5. Finally, we prove the identity extension lemma for the true typing relation.

Readers familiar with PER models for System F (e.g., [6]) will find this proof structure quite familiar. We begin with an untyped model of computation as a universe, and then define a semantics of types as relations on the universe by induction on the derivation of the pretyping relation. The main technical novelty in our approach is that our types may contain terms, and we thus need to interpret types in a context containing interpretations of the terms.

The operational semantics for our programming language is given in Figure 3, and is a standard call-by-name semantics. There is no evaluation rule for the constant •, since it has no explicit elimination form. There is no reduction rule for abort — this term creates a stuck computation, since it indicates unreachable code.

In Figure 4, we give the pretyping rules. We have three judgements:

- The $\Gamma \rhd \text{ok}$ judgement judges whether a context is well-formed, and
- the $\Gamma \rhd A$ judgement judges whether a type A is well-formed in context Γ, and
- the $\Gamma \rhd e : A$ judgement judges whether a term e is well-formed with respect to pretype A in context Γ.

All three of these judgements are mutually-recursive, since the equality type $e \equiv_A e'$ contains terms, and its well-formedness rule asserts that e and e' must have pretype A. The rules mostly resemble F's rules, with the variation that both term and type applications need to perform a substitution (rather than solely type application, as in ordinary System F).

The two surprising rules of the system are Danger1 and Danger2, which are the rules for introducing the equality type and the abort keyword. As a result, the pretype system is obviously unsound, since we can freely introduce the stuck term abort wherever we like.

Of course, we will eventually refine these two rules so that equalities can only be used to introduce true equalities, and abort can only be used in contexts under which we can prove that evaluation can never reach that point. Do note, however, that in this setting, it is the existence of abort which gives the equality type its force. There are no elimination rules for equality types, and so the only way that programs can make use of equalities is to exploit the equations in context to write abort at certain places.

Now, we state the basic syntactic substitution properties of the calculus.

Theorem 1. *(Syntactic Substitution) Suppose $\Gamma \rhd A$ and $\Gamma \rhd e : B$. Then*

- *If $\Gamma, \alpha, \Gamma' \rhd ok$ then $\Gamma, [A/\alpha]\Gamma' \rhd ok$.*
- *If $\Gamma, \alpha, \Gamma' \rhd B$ then $\Gamma, [A/\alpha]\Gamma' \rhd [A/\alpha]B$.*
- *If $\Gamma, \alpha, \Gamma' \rhd e' : C$ then $\Gamma, [A/\alpha]\Gamma' \rhd [A/\alpha]e' : [A/\alpha]C$.*
- *If $\Gamma, x : A, \Gamma' \rhd ok$ then $\Gamma, [e/x]\Gamma' \rhd ok$.*
- *If $\Gamma, x : A, \Gamma' \rhd B$ then $\Gamma, [e/x]\Gamma' \rhd [e/x]B$.*
- *If $\Gamma, x : A, \Gamma' \rhd e' : C$ then $\Gamma, [e/x]\Gamma' \rhd [e/x]e' : [e/x]C$.*

The proofs of these theorems are a routine structural induction.

To add semantic side-conditions to the Danger1 and Danger2 rules, we need to give a relational semantics of types, since we need to be able to compare terms for equality. In Figure 6, we give the logical relation defining the relational interpretation of types, as a structural recursion over the pretyping derivations $\Gamma \rhd A$. For each type constructor, we give the relation defining equality at that type. Furthermore, since we are defining our relations by induction on the structure of the pretyping derivation $\Gamma \rhd A$, we also parameterize this relation by a grounding substitution γ.

The two key judgements in this relation begin with $Env(\Gamma \rhd \text{ok})$, which defines the set of well-formed grounding substitutions for the environment Γ. As a context consists of a sequence of type variables α and term variables $x : A$, the grounding substitutions consist of sequences of triples (A, A', R) of closed types and the relations between the terms of those types, which ground the type

$$\begin{array}{ll}
(\cdot)_0 & = \cdot \\
(\gamma, (e, e')/x)_0 & = \gamma_0, (e/x) \\
(\gamma, (A, A', R)/\alpha)_0 & = \gamma_0, (A/\alpha) \\
\\
(\cdot)_1 & = \cdot \\
(\gamma, (e, e')/x)_1 & = \gamma_1, (e'/x) \\
(\gamma, (A, A', R)/\alpha)_1 & = \gamma_1, (A'/\alpha) \\
\\
\gamma(e) & = (\gamma_0(e), \gamma_1(e)) \\
\gamma(A) & = (\gamma_0(e), \gamma_1(A))
\end{array}$$

Fig. 5. Operations on Relational Substitutions

variables α, and pairs of expressions (e, e') which lie in the relation for A to ground each term variable.

As γ is a relational substitution, we also need operations to extract ordinary substitutions from it. These operations are defined in Figure 5. Given γ, the substitution γ_0 takes the left components of the relational substitution, and γ_1 takes the right components. We write $\gamma(e)$ as shorthand for the pair $(\gamma_0(e), \gamma_1(e))$, and similarly we write $\gamma(A)$ for $(\gamma_0(A), \gamma_1(A))$.

The environment relation is used mutually-recursively to define the relation $Val(\Gamma \rhd A)(\gamma)$, which relates pairs of closed values of type A in the context Γ closed by the substitution γ. This definition follows the usual pattern of logical relations: type variables α look up the appropriate relation in the context γ, and the value relation for function space $(x : A) \to B$ relates two functions f and g if they take related arguments to related results.

The interpretation of the universal quantifier $\forall\alpha.\ B$ says that two terms are related if for all value relations between pretypes A and A' the type application preserves the relation. By quantifying over relations between arbitrary values, we avoid recursively mentioning the definition of the logical relation, and thereby avoid circularity. This is a syntactic version of the techniques used in PER models of polymorphism: fixing a universe ahead of time lets us consider the intersection of all relations on that universe, without running afoul of the apparent circularity of impredicative quantification.

Finally, we define the value relation for equality types $Val(\Gamma \rhd e \equiv_A e')(\gamma)$ as the pair $(\bullet, \bullet)$, but *only if* the pair $(\gamma_0(e), \gamma_1(e'))$ is in the relation for A. Otherwise the relation is empty. This gives the semantic sense in which the equality type is an equality type: it is a type containing a single unit value when the equality is true, and is the empty type when it is not.

We also include the definition $Exp(\Gamma \rhd A)(\gamma)$, which are pairs of expressions reducing to values related by $Val(\Gamma \rhd A)(\gamma)$. This is an auxiliary definition simplifying the definitions of values and environments.

Having fixed the semantics of types, we give the true typing rules in Figure 7. As before, we have three mutually-recursive judgements, $\Gamma \vdash \text{ok}$, for well-formed contexts, $\Gamma \vdash A$, for well-formed types, and $\Gamma \vdash e : A$ for well-typing. All of the typing rules precisely mirror the pretyping rules, with the exception of the equality and abort rules.

$$
\begin{array}{l}
\mathit{Val}(\varGamma \rhd \alpha)(\gamma) = \text{let } (A, B, R) \;=\; \gamma(\alpha) \text{ in } R \\
\mathit{Val}(\varGamma \rhd (x : A) \to B)(\gamma) = \\
\quad \left\{ \langle \lambda x.\ e, \lambda x.\ e' \rangle \;\middle|\; \begin{array}{l} \cdot \rhd \lambda x.\ e : \gamma_0((x : A) \to B) \text{ and} \\ \cdot \rhd \lambda x.\ e' : \gamma_1((x : A) \to B) \text{ and} \\ \forall e_0, e_0' \in \mathit{Exp}(\varGamma \rhd A)(\gamma). \\ \quad \langle [e_0/x]e, [e_0'/x]e' \rangle \in \mathit{Exp}(\varGamma, x : A \rhd B)(\gamma, \langle e_0, e_0' \rangle / x) \end{array} \right\} \\
\mathit{Val}(\varGamma \rhd \forall \alpha.\ B)(\gamma) = \\
\quad \left\{ \langle \varLambda \alpha.\ e, \varLambda \alpha'.\ e' \rangle \;\middle|\; \begin{array}{l} \cdot \rhd \varLambda \alpha.\ e : \gamma_0(\forall \alpha.\ B) \text{ and} \\ \cdot \rhd \varLambda \alpha.\ e' : \gamma_1(\forall \alpha.\ B) \text{ and} \\ \forall A, A', R. \\ \quad \cdot \rhd A \text{ and } \cdot \rhd A' \text{ and} \\ \quad R \subseteq \{\langle v, v' \rangle \mid \cdot \rhd v : A \text{ and } \cdot \rhd v' : A'\} \text{ and} \\ \quad \langle [A/\alpha]e, [A'/\alpha']e' \rangle \in \mathit{Exp}(\varGamma, \alpha \rhd B)(\gamma, (A, A', R)/\alpha) \end{array} \right\} \\
\mathit{Val}(\varGamma \rhd e_0 \equiv_A e_1)(\gamma) = \{\langle \bullet, \bullet \rangle \mid \langle \gamma_0(e_0), \gamma_1(e_1) \rangle \in \mathit{Exp}(\varGamma \rhd A)(\gamma)\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Exp}(\varGamma \rhd A)(\gamma) = \\
\quad \left\{ \langle e_0, e_1 \rangle \;\middle|\; \begin{array}{l} \cdot \rhd e_0 : \gamma_0(A) \text{ and } \cdot \rhd e_1 : \gamma_1(A) \text{ and} \\ \exists v_0, v_1.\ \begin{array}{l} \gamma_0(e_0) \Downarrow v_0 \text{ and } \gamma_1(e_1) \Downarrow v_1 \text{ and} \\ \langle v_0, v_1 \rangle \in \mathit{Val}(\varGamma \rhd A)(\gamma) \end{array} \end{array} \right\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Env}(\cdot \rhd \text{ok}) = \{\langle\rangle\} \\
\mathit{Env}(\varGamma, x : A \rhd \text{ok}) = \{(\gamma, \langle e, e' \rangle / x) \mid \gamma \in \mathit{Env}(\varGamma \rhd \text{ok}) \text{ and } (e, e') \in \mathit{Exp}(\varGamma \rhd A)(\gamma)\} \\
\mathit{Env}(\varGamma, \alpha \rhd \text{ok}) = \\
\quad \left\{ (\gamma, (A, A', R)/\alpha) \;\middle|\; \begin{array}{l} \gamma \in \mathit{Env}(\varGamma \rhd \text{ok}) \text{ and } \cdot \rhd A \text{ and } \cdot \rhd A' \text{ and} \\ R \subseteq \{\langle v, v' \rangle \mid \cdot \rhd v : A \text{ and } \cdot \rhd v' : A'\} \end{array} \right\}
\end{array}
$$

Fig. 6. Relational Semantics

Each of these has a semantic side-condition controlling when they can be used. These side-conditions mean that the type-checking problem is not decidable, since potentially arbitrary mathematical reasoning may be needed to show that the rule applies. However, the soundness theorem for the language ensures that once the side-conditions are discharged, then evaluation cannot alter the typability of of the program under reduction.

The premise of the equality rule contains the non-syntactic premise that $\varGamma \models e = e' : A$. This means that in all semantic environments $\gamma \in \mathit{Env}(\varGamma \rhd \text{ok})$, the pair $(\gamma_0(e), \gamma_1(e'))$ must lie in the expression relation for the type A. This means that the two expressions must be equivalent to introduce an equality type.

Similarly, the premise of the abort rule is that $\varGamma \models \bot$ must hold, which means that there are *no* environments in $\mathit{Env}(\varGamma \rhd \text{ok})$. This means that the context $\varGamma$ is a contradictory one, with no environments that can satisfy it.

Now, we can prove a semantic version of the substitution theorem.

Theorem 2. *(Semantic Substitution)*

- *Suppose* $\Gamma \rhd A$ *and* $(\gamma, \gamma(A)/\alpha, \gamma') \in \mathit{Env}(\Gamma, \alpha, \Gamma' \rhd ok)$. *Then*
  - $(\gamma, \gamma') \in \mathit{Env}(\Gamma, [A/\alpha]\Gamma' \rhd ok)$
  - *If* $(v, v') \in \mathit{Val}(\Gamma, \alpha, \Gamma' \rhd e)(B)(\gamma, \gamma(A)/\alpha, \gamma')$, *then* $(v, v') \in \mathit{Val}(\Gamma, [A/\alpha]\Gamma' \rhd [A/\alpha]B)(\gamma, \gamma')$.
  - *If* $(e, e') \in \mathit{Exp}(\Gamma, \alpha, \Gamma' \rhd e)(B)(\gamma, \gamma(A)/\alpha, \gamma')$, *then* $(e, e') \in \mathit{Exp}(\Gamma, [A/\alpha]\Gamma' \rhd [A/\alpha]B)(\gamma, \gamma')$.
- *Suppose* $\Gamma \rhd e : A$ *and* $(\gamma, \gamma(e)/x, \gamma') \in \mathit{Env}(\Gamma, x : A, \Gamma' \rhd ok)$. *Then*
  - $(\gamma, \gamma) \in \mathit{Env}(\Gamma, [e/x]\Gamma' \rhd ok)$.
  - *If* $(v, v') \in \mathit{Val}(\Gamma, x : A, \Gamma' \rhd e)(A)(\gamma, \gamma(e)/x, \gamma')$, *then* $(v, v') \in \mathit{Val}(\Gamma, [e/x]\Gamma' \rhd [e/x]A)(\gamma, \gamma')$.
  - *If* $(e, e') \in \mathit{Exp}(\Gamma, x : A, \Gamma' \rhd e)(A)(\gamma, \gamma(e)/x, \gamma')$, *then* $(e, e') \in \mathit{Exp}(\Gamma, [e/x]\Gamma' \rhd [e/x]A)(\gamma, \gamma')$.

These theorems follow from induction on the context and type pre-well-formedness judgements. We can use these theorems to prove Reynolds' abstraction theorem for our language.

Theorem 3. *(Abstraction Theorem) If* $\Gamma \vdash e : A$, *then* $\Gamma \models e = e : A$.

This theorem follows from a structural induction on the typing derivation, making use of the semantic substitution principles. Normalization and type-preservation follow immediately.

Corollary 1. *(Normalization) If* $\cdot \vdash e : A$, *then* $\exists v$ *such that* $e \Downarrow v$.

Corollary 2. *(Type Preservation) If* $\cdot \vdash e : A$ *and* $e \Downarrow v$, *then* $\cdot \vdash v : A$.

It is worth noting that the type preservation lemma is *exact* — the type of the result is exactly the same as the type of the original. We do not need any notion of type equality beyond the same syntactic equality (modulo α) that System F needed.

4 Existential Representations of Inductive Datatypes

A surprising feature of the examples in Section 2 is that we gave an apparently *existential* encoding of inductive datatypes such as the booleans. This is a little surprising, since the Church encodings of these types in System F are *universal.*

As a concrete example, recall the Church encoding of the boolean type.

- The type of Church booleans $\mathsf{cbool} = \forall\alpha.\ \alpha \to \alpha \to \alpha$.
- Truth is defined as $\Lambda\alpha.\ \lambda a.\ \lambda a'.\ a$.
- Falsity is defined as $\Lambda\alpha.\ \lambda a.\ \lambda a'.\ a'$.
- The conditional is $\mathit{if} : \mathsf{cbool} \to \forall\alpha.\ \alpha \to \alpha \to \alpha \triangleq \lambda b.\ b$.

Contrast this with the interface we gave for the boolean type:

$$\boxed{\Gamma \models e = e' : A} \qquad \boxed{\Gamma \models \bot}$$

$$\begin{array}{lll}
\Gamma \models e_0 = e_1 : A & \iff & \forall \gamma \in \mathit{Env}(\Gamma \rhd \mathrm{ok}).\ (\gamma_0(e_0), \gamma_1(e_1)) \in \mathit{Exp}(\Gamma \rhd A)(\gamma) \\
\Gamma \models \bot & \iff & \mathit{Env}(\Gamma \rhd \mathrm{ok}) = \emptyset
\end{array}$$

$$\boxed{\Gamma \vdash \mathrm{ok}} \qquad \boxed{\Gamma \vdash A} \qquad \boxed{\Gamma \vdash e : A}$$

$$\frac{}{\cdot \vdash \mathrm{ok}} \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma, \alpha \vdash \mathrm{ok}} \qquad \frac{\Gamma \vdash \mathrm{ok} \qquad \Gamma \vdash A}{\Gamma, x : A \vdash \mathrm{ok}}$$

$$\frac{\Gamma \vdash \mathrm{ok} \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \forall \alpha.\ A} \qquad \frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash B}{\Gamma \vdash (x : A) \to B}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash e : A \qquad \Gamma \vdash e' : A}{\Gamma \vdash e \equiv_A e'}$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.\ e : (x : A) \to B} \qquad \frac{\Gamma \vdash e : (x : A) \to B \qquad \Gamma \vdash e' : A}{\Gamma \vdash e\ e' : [e/x]B} \qquad \frac{\Gamma, \alpha \vdash e : A}{\Gamma \vdash \Lambda \alpha.\ e : \forall \alpha.\ A}$$

$$\frac{\Gamma \vdash e : \forall \alpha.\ B \qquad \Gamma \vdash A}{\Gamma \vdash e\ [A] : [A/\alpha]B} \qquad \frac{\Gamma \vdash e \equiv_A e' \qquad \Gamma \models e = e' : A}{\Gamma \vdash \bullet : e \equiv_A e'} \qquad \frac{\Gamma \vdash A \qquad \Gamma \models \bot}{\Gamma \vdash \mathsf{abort} : A}$$

$$\frac{\Gamma \vdash \mathrm{ok} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}$$

Fig. 7. Typing

$$\begin{array}{l}
B \equiv \\
\quad \exists \mathsf{bool} \\
\qquad \mathit{true} : \mathsf{bool}, \\
\qquad \mathit{false} : \mathsf{bool}, \\
\qquad \mathit{if} : \forall \alpha.\ \mathsf{bool} \to \alpha \to \alpha \to \alpha. \\
\qquad \forall \alpha, a : \alpha, a' : \alpha.\ \mathit{if}\ [\alpha]\ \mathit{true}\ a\ a' \equiv_\alpha a\ \times \\
\qquad \forall \alpha, a : \alpha, a' : \alpha.\ \mathit{if}\ [\alpha]\ \mathit{false}\ a\ a' \equiv_\alpha a'
\end{array}$$

Unlike the Church encoding, the interface completely conceals the representation type of the booleans, as well as the implementations of truth, falsity and if-then-else. The only constraint we place in the interface is to require the β-theory of the booleans to hold.

Now we will show that these two implementations of the booleans are actually the same. To do this, first note that we somehow need to compare an *arbitrary* element of the existential type to a particular set of elements of the Church type. Luckily, we have precisely the tools we need with the equality types of our calculus. The Church booleans can be represented as elements of the type

```
$B' \equiv$
    $\exists\ \mathit{true} : \mathsf{cbool},$
      $\mathit{false} : \mathsf{cbool},$
      $\mathit{true} \equiv_{\mathsf{cbool}} \Lambda\alpha.\ \lambda a.\ \lambda a'.\ a\ \ \times$
      $\mathit{false} \equiv_{\mathsf{cbool}} \Lambda\alpha.\ \lambda a.\ \lambda a'.\ a'$
```

By using equality types, we ensure that we have a tuple whose first element is Church truth and whose second element is Church falsity.

This gives us the material we need to prove the following theorem:

Theorem 4. *(Equivalence of Boolean Types) We have an isomorphism between the types B and B'.*

Proof. To show this holds, we wil give explicit maps $i : B \to B'$ and $j : B' \to B$. Then we will show that $\cdot \models i \circ j = id : B'$ and that $\cdot \models j \circ i = id : B$. We give the definitions below, using the syntax for tuples and existentials for clarity.

$$i\ : B \to B' \triangleq \lambda b.\ \langle \Lambda\alpha.\ \lambda a.\ \lambda a'.\ a, \Lambda\alpha.\ \lambda a.\ \lambda a'.\ a', \bullet, \bullet\rangle$$

$$j : B' \to B \triangleq \lambda b'.\ \begin{array}{l} \text{let } t\ =\ \Lambda\alpha.\ \lambda a.\ \lambda a'.\ a \text{ in} \\ \text{let } f\ =\ \Lambda\alpha.\ \lambda a.\ \lambda a'.\ a' \text{ in} \\ \text{let } \mathit{if}\ =\ \Lambda\alpha.\ \lambda x.\ \lambda y.\ \lambda b.\ b\ [\alpha]\ x\ y \text{ in} \\ \mathit{pack}\ \langle \mathsf{cbool}, t, f, \mathit{if}, \bullet, \bullet\rangle \end{array}$$

The $B \to B'$ direction ignores its argument, and simply returns the obvious tuple inhabiting B'. The $B' \to B$ direction also ignores its argument, and returns an instance of the existential representation which uses the Church booleans as the representation type.

Therefore, each composition is a constant function, and so showing that it is equivalent to the identity function means showing that all elements of B are equivalent, and similarly for B'. The case for B' is easy, and the interesting case fo B reduces to the problem of showing that any element of the existential boolean type is equivalent to element using the Church booleans as its representation type.

This follows from unwinding the definitions. To do this, we introduce a relation that (unsurprisingly) relates Church truth to the true value of the hidden existential type, and Church falsity to the false value of the hidden existential type. Then, the equations for the hidden existential implementation of B can be used to show that the hidden implementation of *if* is equivalent to the Church implementation.

Similarly, we can relate (an extended version of) the existential natural number interface given in Section 2 with the Church encoding $\mathsf{churchnat} = \forall\alpha.\ \alpha \to (\alpha \to \alpha) \to \alpha$.

```
$N \equiv$
    $\exists\mathsf{nat}$
      $z : \mathsf{nat},$
      $s : \mathsf{nat} \to \mathsf{nat},$
      $pred : \mathsf{nat} \to \mathsf{option}_{\mathsf{nat}},$
      $iter : \forall \alpha.\ \mathsf{nat} \to \alpha \to (\alpha \to \alpha) \to \alpha.$
      $pred\ z \equiv_{\mathsf{option}_{\mathsf{nat}}} none\ \ \times$
      $\forall n.\ pred\ (s\ n) \equiv_{\mathsf{option}_{\mathsf{nat}}} some\ n\ \ \times$
      $\forall \alpha, i, f.\ iter\ [\alpha]\ z\ i\ f \equiv_{\alpha} i\ \ \times$
      $\forall n, \alpha, i, f.\ iter\ [\alpha]\ (s\ n)\ i\ f \equiv_{\alpha} f\ (iter\ [\alpha]\ n\ i\ f)$
```

```
  $N' \equiv$
    $\exists z : \mathsf{churchnat},$
      $s : \mathsf{churchnat} \to \mathsf{churchnat},$
      $- : z \equiv_{,}\ \Lambda\alpha.\ \lambda i.\ \lambda f.\ i$
      $- : s \equiv_{\mathsf{churchnat} \to \mathsf{churchnat}} \lambda n.\ \Lambda\alpha.\ \lambda i.\ \lambda f.\ f\ (n\ \alpha\ i\ f)$
```

We can then prove the equivalence of these two types.

Theorem 5. *(Equivalence of Natural Number Types) There exists an isomorphism between N and N'.*

Proof. The proof of this theorem follows exactly the same pattern as for the booleans. Ultimately we will end up showing that arbitrary elements of N are equivalent to the representation using the Church natural numbers. To do this, we will also need to define the predecessor function *pred* on the Church naturals, which is a linear time operation.

The most interesting thing about this theorem is not the proof, which is standard, but rather the fact that we extended the natural number interface with the predecessor *pred*. The fact that the representation of natural numbers is completely hidden in the existential style means that we can (for example) use a representation of the natural numbers in which the predecessor is cheap to compute. This contrasts with the explicit unary representation of the Church encoding, in which the predecessor is necessarily linear time. As a result, we can relate this slow implementation to fast ones without any difficulties.

This all relies critically on the equations. In the absence of equations specifying the behavior of the predecessor, there is no way to have this constructor while ensuring that the type really does represent the natural numbers object, since there could be many implementations which are type-correct (in F) but lack the necessary equational properties. However, with equations we can add operations for efficiency without ruining the reasoning properties of the datatype, by cutting down the set of reasonable implementations until only ones equivalent to the intended datatype are possible. (We made extensive use of this in our list example in Section 2.)

This is why we have not proven a general representation theorem for all polynomial datatypes. While a representation theorem does not seem hard to come by in the case where the constructors and fold-style eliminators constitute the interface, it seems that we should consider a representation theorems in the more interesting case in which the interfaces are augmented with extra operations that improve the computational efficiency of implementation. However, it remains unclear to us what such interfaces should be, in general.

5 Related Work

5.1 System R and Plotkin-Abadi Logic

The two most prominent systems for reasoning about parametricity are System R [1, 4] and Plotkin-Abadi logic [19]. These logics can be viewed as program logics a la Hoare logic, in that they fix a programming logic (System F), and then give a logical system for reasoning about terms in that language.

Our language can be understood as an attempt to take a small fragment of these logics, and then reflect them back into the types of F. This naturally suggests two directions. First, might it be worthwhile to add more of these logics to the type system of $F_=$? In this first paper, we wished to illustrate just how much is achievable with a very modest addition to the type theory of System F, but the extension is a very natural question.

In particular, all of the semantic side-conditions have been discharged by working directly with the relational semantics. By building a logic for parametricity, we could potentially use it to give a proof system for equalities and aborts. However, the presence of abort in our language means that such a logic could not be a simple replay of the developments of [1] or [19], though.

5.2 Dependent Types

The appearance of terms in types in our calculus is rather reminiscent of systems of dependent types, such as Martin-Löf type theory [16] or the calculus of constructions [10]. Indeed, the realizability semantics we use for $F_=$ is quite similar to the semantics of extensional type theories such as Nuprl [9]. Furthermore, we share with extensional type theory the property that typechecking is not syntax-directed: our proof term for equality, $\bullet$, does not contain the evidence of equality. This is similar to the equality reflection property of extensional type theory, in which proof terms for introducing equalities may depend on propositional equality proofs not evident in the proof term.

However, the semantics of our equality type is a bit different from the equality of dependent type theory. In type theory the elimination form for the equality type $e \equiv_A e'$ is used to cast terms of type $B[e]$ into ones of type $B[e']$. As a result, actually deriving a contradiction (i.e., a terms of type $\bot$) from an impossible equality (e.g., a proof of $0 \equiv_{\mathbb{N}} 1$) requires using a large elimination to turn contradictions into proofs of falsity.

In our setting, we instead admit the use of the abort keyword in any inconsistent context, which allows us to make use of contradictions without having to explicitly support large eliminations.

5.3 The Haskell Rules Mechanism

The Glasgow Haskell compiler contains a mechanism called *rules* [12], which allow programmers to specify equational rewrite rules (such as $(map\ f) \circ (map\ g) \mapsto$

map $(f \circ g)$) as part of library interfaces. However, these rewrite rules are restricted to referring to top-level module identifiers, and rewriting cannot be applied to an expression unless the term in question refers to *exactly* the same variables as the rules definition referred to. This restriction means that rules — which were a feature whose purpose is to lower the cost of good higher-order style — are much less effective when applied to higher-order code (where operators such as *map* may flow in as arguments to functions).

Our type theory illustrates that it is possible to integrate Haskell-style rules into a simple type theory treating rewrite rules as first-class types. One particularly interesting direction to investigate is adding rules to type classes, which would permit stating the equational assumptions about polymorphic terms. E.g., Haskell's Functor typeclass has a method with type

$$\mathit{fmap} : \mathsf{Functor}\ F \Rightarrow (\alpha \to \beta) \to F\ \alpha \to F\ \beta$$

It is intended that *fmap* is functorial — that is, that $\mathit{fmap}\ \mathit{id} = \mathit{id}$, and that $(\mathit{fmap}\ f) \circ (\mathit{fmap}\ g) = \mathit{fmap}\ (f \circ g)$. By placing these equations into the Functor interface and verifying the typeclass instances, they could even be used to drive optimizations of client code.

5.4 Extended ML

One of the earliest serious attempts to extend a functional language with equational specifications was the Extended ML [13] project. In this work, SML module signatures were extended with algebraic signatures stating the intended equational properties of the abstract datatypes.

This work was quite ambitious, and it involved a rather large fragment of ML including features such as exceptions and non-termination. Furthermore, the concept of algebraic signature was generalized well beyond equational properties to include full logical predicates. However, the technical ambition of this approach meant that its semantics were never fully settled (the question of polymorphism was especially vexing, as was the specification of imperative ML code).

In this paper, we have avoided effects to maximize the force of parametricity. This lets us specify quite sophisticated properties (e.g., initiality) with a bare minimum of additional syntactic and semantic machinery. One especially nice feature of our work is that the presence of equation makes it very natural to connect Church-style datatype encodings with the existential style of data abstraction more common in ML (and exploited by EML).

These days, there are quite well-developed semantic frameworks in place to model polymorphic languages with features like nontermination, recursive types, and higher order state [18, 7]. However, in spite of this machinery, it is simply an unavoidable fact that fewer equations hold when effects are present. To what extent the reduced of validity equational reasoning limits the use of equality types is unclear. One approach to this problem may be to encapsulate effects in a monadic type, and then use other techniques (such as Hoare logic [14]) to reason about the monadic code.

6 Future Work

There are two strands of future work. First, there is the theoretical strand. The first question is whether our termination result can be strengthened into a strong normalization result, which would require a more sophisticated logical relation [2].

Second, it may be possible to give a logic for this calculus along the lines of Plotkin-Abadi logic, and then use the rules of that calculus to give proof terms for the equality type. This would make typechecking decidable, and might make an interesting basis for a dependent type theory with parametricity, along the lines of [5]. While this is a challenging problem, the extreme simplicity of our semantics offers reasonable grounds for hope.

ML-style modules support the "strong" dot-notation elimination form [8], whereas our existential encoding uses F-style existentials with a"weak" let-binding eliminator. Recently, Rossberg, Russo and Dreyer have shown [21] how to translate ML-style modules into System F, and it would be interesting to study if a similar translation could take ML signatures extended with equations and translate into $F_=$.

On a practical note, how can equation types be profitably employed in optimizations? Connecting equations to optimizations is an intriguing problem.

Finally, our type system emits proof obligations at each introduction of an equality or use of an abort. It would be useful to ship these proof obligations off to a theorem prover such as Coq. Doing so will require a certain amount of care, since parametricity is essential to the arguments we make, and we will need to make use of recent work [3] on representing the semantics of polymorphism in type theory.

References

1. Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. In *Principles of Programming Languages*, pages 157–170, 1993.
2. Andreas Abel. Weak beta-theta-normalization and normalization by evaluation for system f. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference*, volume 5330 of *Lecture Notes in Computer Science*, pages 497–511. Springer, 2008.
3. Robert Atkey. Syntax for free: Representing syntax with binding using parametricity. In Pierre-Louis Curien, editor, *TLCA*, volume 5608 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2009.
4. Roberto Bellucci, Martín Abadi, and Pierre-Louis Curien. A model for formal parametric polymorphism: A PER interpretation for system R. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *Second International Conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 1995.
5. Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010*, pages 345–356. ACM, 2010.

6. Lars Birkedal, Rasmus Ejlers Møgelberg, and Rasmus Lerchedahl Petersen. Domain-theoretical models of parametric polymorphism. *Theoretical Computer Science*, 388(1-3):152–172, 2007.
7. Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. Realisability semantics of parametric polymorphism, general references and recursive types. *Mathematical Structures in Computer Science*, 20(4):655–703, 2010.
8. Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In M. Broy and C. B. Jones, editors, *Proceedings IFIP TC2 working conference on programming concepts and methods*, pages 479–504. North-Holland, 1990.
9. Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice Hall, 1986.
10. Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.
11. Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In John H. Reppy and Julia L. Lawall, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 50–61. ACM, 2006.
12. Simon Peyton Jones, Andrew Tolmach, and Tony Hoare;. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, 2001.
13. Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173(2):445–484, 1997.
14. Neelakantan R. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic.* PhD thesis, Carnegie Mellon University, 2011.
15. Simon Marlow and Philip Wadler. Deforestation for higher-order functions. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 154–165. Springer, 1992.
16. Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.
17. John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10:470–502, 1988.
18. Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming*, pages 135–148. ACM, 2009.
19. Gordon D. Plotkin and Martín Abadi. A logic for parametric polymorphism. In Marc Bezem and Jan Friso Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer, 1993.
20. John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.
21. Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '10. ACM, 2010.
22. Philip Wadler. The Girard-Reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1-3):201–226, 2007.