1-24-2012

Equational Reasoning about Programs with General Recursion and Call-by-value Semantics

Garrin Kimmell
University of Iowa

Aaron Stump
University of Iowa

Harley D. Eades III
University of Iowa

Peng Fu
University of Iowa

Tim Sheard
Portland State University

See next page for additional authors

Recommended Citation

Garrin Kimmell, Aaron Stump, Harley D. Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjoberg, Nathan Collins, and Ki Yung Ahn, "Equational Reasoning about Programs with General Recursion and Call-by-value Semantics", . January 2012.

Garrin Kimmell, Aaron Stump, Harley D. Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins, and Ki Yung Ahn. **Equational Reasoning about Programs with General Recursion and Call-by-value Semantics**. In *Sixth ACM SIGPLAN Workshop Programming Languages meets Program Verification (PLPV '12)*, 2012.

Equational Reasoning about Programs with General Recursion and Call-by-value Semantics

Abstract
Dependently typed programming languages provide a mechanism for integrating verification and programming by encoding invariants as types. Traditionally, dependently typed languages have been based on constructive type theories, where the connection between proofs and programs is based on the Curry-Howard correspondence. This connection comes at a price, however, as it is necessary for the languages to be normalizing to preserve logical soundness. Trellys is a call-by-value dependently typed programming language currently in development that is designed to integrate a type theory with unsound programming features, such as general recursion, Type:Type, and others. In this paper we outline one core language design for Trellys, and demonstrate the use of the key language constructs to facilitate sound reasoning about potentially unsound programs.

Disciplines
Computer Sciences

Comments
Garrin Kimmell, Aaron Stump, Harley D. Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins, and Ki Yung Ahn. **Equational Reasoning about Programs with General Recursion and Call-by-value Semantics**. In *Sixth ACM SIGPLAN Workshop Programming Languages meets Program Verification (PLPV '12)*, 2012.

Author(s)
Garrin Kimmell, Aaron Stump, Harley D. Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjoberg, Nathan Collins, and Ki Yung Ahn

Equational Reasoning about Programs with General Recursion and Call-by-value Semantics

Garrin Kimmell
University of Iowa
garrin-kimmell@uiowa.edu

Aaron Stump
University of Iowa
astump@acm.org

Harley D. Eades III
University of Iowa
harley-eades@uiowa.edu

Peng Fu
University of Iowa
peng-fu@uiowa.edu

Tim Sheard
Portland State University
sheard@cis.pdx.edu

Stephanie Weirich
University of Pennsylvania
sweirich@cis.upenn.edu

Chris Casinghino
University of Pennsylvania
ccasin@cis.upenn.edu

Vilhelm Sjöberg
University of Pennsylvania
vilhelm@cis.upenn.edu

Nathan Collins
Portland State University
nathan.collins@gmail.com

Ki Yung Ahn
Portland State University
kya@cs.pdx.edu

Abstract

Dependently typed programming languages provide a mechanism for integrating verification and programming by encoding invariants as types. Traditionally, dependently typed languages have been based on constructive type theories, where the connection between proofs and programs is based on the Curry-Howard correspondence. This connection comes at a price, however, as it is necessary for the languages to be normalizing to preserve logical soundness. Trellys is a call-by-value dependently typed programming language currently in development that is designed to integrate a type theory with unsound programming features, such as general recursion, `Type:Type`, and others. In this paper we outline one core language design for Trellys, and demonstrate the use of the key language constructs to facilitate sound reasoning about potentially unsound programs.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs

General Terms Design, Languages, Verification, Theory

PLPV'12, January 24, 2012, Philadelphia, PA, USA.
Copyright © 2012 ACM 978-1-4503-1125-0/12/01...$10.00

1. Introduction

When writing verified programs, there are two traditional approaches. Theorem provers like ACL2 and Isabelle are used to perform verification *externally* [7, 15]. Programs are defined independently of the desired invariants, and then those invariants are verified after the fact. In addition to external verification, languages based on constructive type theory, such as Coq [18] and Agda [2] also support encoding program invariants *internally*, using dependent types. Specifications are tightly connected to code, and the burden of external proof can be reduced.

For both approaches, general recursion (or the definition of partial functions) poses challenges. Constructive type theories require functions to terminate on all inputs, to preserve soundness of the logic under the Curry-Howard isomorphism. Sophisticated techniques, such as encoding possibly-diverging computations as coinductive data, are required to define truly partial functions [4]. Alternatively, one can formulate a domain of definition for which the functions are, in fact, total, using an accessibility predicate [1]. This basic idea has also been used for higher-order logics [8]. Relatively few theories have been proposed for direct reasoning about general recursion; examples are LCF, LTC and VeriFun [3, 19**?**].

The Trellys project is a collaborative research initiative to design a dependently typed programming language with direct support for general recursion and other features such as Type:Type, which, like general recursion, are unsound under the Curry-Howard isomorphism. The goal of Trellys is to bridge the gap between dependently typed languages and program logics, allowing a programmer to utilize both internal and external verification techniques in the presence of these unsound features.

Trellys is also intended as a practical programming language. By removing the constraint that all programs terminate, we are forced to consider details such as evaluation strategy, since the termination behavior of a term in a language with general recursion can vary if the reduction strategy is changed. For Trellys, we have chosen call-by-value reduction because of the simplicity of the cost model it provides. This choice has far reaching consequences on the design of the logic used for verification in Trellys. In this paper, we address and propose solutions to a number of the issues encountered at this particular point in the design space: a call-by-value dependently typed language with general recursion.

We identify several problems encountered when trying to integrate dependent types with general recursion:

- How do we exploit the fact that inside programs, variables bound in those programs range over values, while still allowing quantification over all programs,[1] including ones which diverge?
- The theory of call-by-value β-equality is fairly weak, given the restriction that the argument to β-reduction be a value. How can we strengthen this equality to include all arguments that provably have values, but are not syntactically values?
- The natural way to prove theorems like associativity of append is by induction on the structure of a value. How can we strengthen such theorems to apply to all programs, including possibly diverging ones?

In solving these problems, we develop the main contributions of this paper:

- We define a **judgmental notion of value** that distinguishes variables contextually. This is accomplished by marking variables introduced during quantification by whether or not they should be treated as values. The syntactic notion of value is then changed to be a judgmental notion, which classifies those variables as ranging over values or expressions, depending on how they are marked in the context.
- We integrate a notion of **termination cast** from our previous work (see Section 3) with call-by-value reduction, so that programs which are proven to be terminating can be considered to be values, for purposes of β-reduction.
- We introduce a non-computational **termination case** form, which allows us to case-split during reasoning on whether a program terminates or diverges. Using this, many theorems can be generalized to hold for all programs, not just terminating ones. This generalization is quite useful in practice, as it means that the theorems can be applied without needing to prove that their arguments are terminating.

This paper concerns the *expressiveness* of dependently-typed core languages. Its purpose is to explain these novel features through examples, describing the problems that non-termination and call-by-value reasoning bring to full-spectrum dependent type systems, and informally describing how these features can provide solutions to these problems. This paper discusses these ideas in the context of the Sep^3, one of several core language designs that we are developing in the Trellys project. The Sep^3 core language is a work-in-progress, and still under development. As a result it has not been subject to metatheoretic study and proofs of standard type system results (such as type soundness) are beyond the scope of this paper.

The remainder of this paper is structured as follows. Section 2 provides a brief overview of a core language design for Trellys, which we call Sep^3, outlining the key design principles we followed. In Section 3 we identify the connections between Trellys and prior work. In Sections 4 through 6 we detail the problems identified above and present their associated solutions. Section 7 gives a brief overview of our experiences using a prototype implementation of Sep^3 and discusses an example proof. We conclude and identify directions for future work in Section 8.

$\Gamma \vdash \mathtt{p} : \mathtt{P}$	Proof p shows proposition P
$\Gamma \vdash \mathtt{t} : \mathtt{T}$	Term t has type T
$\Gamma \vdash \mathrm{val}\ \mathtt{t}$	Term t is a value
$\mathtt{t}_1 \ \rightarrow^{<m} \ \mathtt{t}_2$	Term t1 evaluates to t2 within m steps

Figure 2. Basic judgment forms

2. Language Overview

The Sep^3 language is one core-language that we are developing to explore the design space of dependently typed languages. This name is short for “separation of proof and program”, indicative of the syntactic separation in the language between proofs and programs (and similarly, between propositions and types). Proofs can mention programs, without invoking them. We dub this capability “Freedom of Speech”. Conversely, programs can use proofs to help demonstrate to the type checker that invariants expressed using dependent types hold.

In the exposition and judgments that follow, we use a few conventions to make the syntactic distinction clear. The metavariable p refers to proofs; the metavariable P refers to propositions. Propositions classify proofs (i.e. p : P), in the sense that types classify terms in the programming language. However, the programming language has a collapsed syntax where terms and types are on the same level, and are represented by the metavariables t and T. We write t : T, even though t and T live in the same syntactic level. Context distinguishes between t and T. We will use the syntactic metavariable e to represent either proofs (p) or programs (t) and the metavariable A to represent either propositions (P) or program types (T).. The metavariables x, y, f, and q range over both program and proof variables.

Below, we identify a number of major features of Sep^3 that we use in the following sections. The syntax of the features of Sep^3 that we discuss in this paper is summarized in Figure 1. The semantics of Sep^3 is specified by the judgments listed in Figure 2. We introduce the rules associated with these judgements as we explain the relevant features of Sep^3.

Although Sep^3 is quite verbose, the language is designed as a core type theory, and we are willing to sacrifice programmer overhead toward the goal of simplifying the pending meta-theoretical study. As the examples later in this paper make apparent, writing proofs and programs directly in the core language can be quite burdensome. However, we plan to eventually reduce this load with surface-language features – such as proof tactics and integration with automated reasoning tools – that can automate the insertion of core-language annotations.

Annotations and Reduction As a core language, Sep^3 requires a large number of annotations to ensure that typing is algorithmic. Moreover, these annotations do not have computational content. For example, changing the type of a term requires a programmer to explicitly insert the cast into the program. The difficulty this introduces is that such casts – which are only necessary for typing – could potentially impact reduction. To simplify the definition of reduction on the programming language, we define a separate *unannotated* language (Figure 3) in which many of these annotations are erased. Reduction for the program fragment is defined only over this unannotated language. The unannotated language is largely a simplification of the annotated language, so we use the same names

[1] We write “program” for terms in the non-logical fragment of the language, and write “proof” for terms in the logical fragment.

```
variables          x,y, f, q
natural numbers    m,n
expressions        e      ::=  p | t
classifiers        A      ::=  P | T

proofs             p      ::=  x | \(x:A) => p | p e                        abstraction and application
                           |   join m n                                     equality axiom
                           |   valax t                                      value axiom
                           |   termcase t {y} of abort → p1 | ! → p2        termination case
                           |   ord                                          ordering axiom
                           |   ordtrans p1 p2                               ordering transitivity
                           |   ind f(x : T) [q]. p                          induction
                           |   case t {y} p of C1 → p1 ... Cn → pn          case analysis

propositions       P      ::=  forall (x : A). p                            quantification
                           |   t1 = t2                                      equality
                           |   t!                                           termination
                           |   t1 < t2                                      ordering

programs and types t,T    ::=  x | \(x:A) -> t | t e                        abstraction and application
                           |   (x:A) -> T                                   dependent function type
                           |   Type                                         type of types
                           |   rec f(x:T1). t2                              recursion
                           |   conv t by p at x.T                           conversion
                           |   \[x:A] -> t | t [p]                          implicit abstraction and application
                           |   [x:A] -> T                                   implicit function type
                           |   C                                            data constructor
                           |   case t {y} of C1 → t1 ... Cn → tn            case analysis
                           |   T t1 ... tn                                  datatype
                           |   abort T                                      failure
                           |   tcast t by p                                 termination cast

contexts           Γ      ::=  ∅ | Γ, x:A | Γ, x:A^val
```

Figure 1. Sep^3 basic syntax

```
unannotated programs         u,U  ::=  x | \x -> u | u1 u2                    abstraction and application
                                   |   (x:A) -> U                            dependent function type
                                   |   Type                                  type of types
                                   |   rec f(x). u                           recursion
                                   |   C                                     data constructor
                                   |   case u {y} of C1 → u1 ... Cn → un     case analysis
                                   |   T u1 ... un                           datatype
                                   |   abort                                 failure
                                   |   tcast t                               termination cast

unannotated program values   v    ::=  \x -> u |                             abstraction
                                   |   (x:A) -> U                            dependent function type
                                   |   Type                                  type of types
                                   |   rec f(x). u                           recursion
                                   |   C v1 ... vn                           constructions
                                   |   T v1 ... vn                           datatype
                                   |   tcast u                               termination cast
```

Figure 3. Sep^3 unannotated syntax

```
|x|                                        =   x
|C|                                        =   C
|\(x:T) -> t|                              =   \x -> |t|
|\[x:A] -> t|                              =   |t|
|\(x:P) -> t|                              =   |t|
|rec f(x:t). t|                            =   rec f(x). |t|
|t1 t2|                                    =   |t1| |t2|
|t1 [t2]|                                  =   |t1|
|t1 p|                                     =   |t1|
|conv t by p at x.T|                       =   |t|
|case t {y} of C1 → t1 ... Cn → tn|        =   case |t| {y} of C1 → |t1| ... Cn → |tn|
|tcast t by p|                             =   tcast |t|
|abort t|                                  =   abort
|(x:A) -> T|                               =   (x:|A|) -> |T|
|[x:A] -> T|                               =   (x:|A|) -> |T|
|Type|                                     =   Type
```

Figure 4. Erasing annotated programs to unannotated programs.

to represent similar constructs in both languages, using the context to distinguish between annotated and unannotated terms.

The erasure function $|\cdot|$ (Figure 4) maps annotated programs to unannotated programs. Note that, in addition to erasing type annotations on abstractions and casts (`conv`), the erasure function also drops all proofs. This is because proofs in Sep^3 are purely specificational, and do not have any run-time behavior. Nevertheless, we define reduction on the proof language because it is relevant to the (future work) meta-theoretical study of the proof language, e.g. a consistency proof.

Erasure drops *all* proofs from the annotated language, but Sep^3 also provides a mechanism for a programmer to specify which *programs* should be preserved across erasure, by allowing abstractions and applications to be annotated as compile time or run time. For example, type arguments to polymorphic functions generally do not contain computational content and are annotated as compile time. In the concrete syntax of the language presented in this paper, compile time abstractions are marked by wrapping the abstraction variable and type annotation with square brackets, as shown in the definition of `id` below. Similarly, applications to compile-time arguments are marked with square brackets, as in the `Nat` type argument to `id` in the definition of `zero` below.[2]

```
id = \[a:Type] -> \(x:a) -> x
zero = id [Nat] Z
```

Applications and abstractions marked as compile time are erased before executing a program, including when proving terms equal using `join`. Erasing compile-time annotations allow proofs of equality to be constructed without reasoning about specificational data. For example, equality between two constructions of `VCons 'a' (VCons 'b' VNil)` indexed by different lengths (e.g. `plus (S Z) Z` and `(S Z)`) can be proved without reasoning about equalities of addition if the length index is marked compile-time. This adapts ideas on erasure from several previous works [10, 12, 17].

Equality Formulas The Sep^3 proof language includes a primitive formula representing the propositional equality of two programs, written $\mathtt{t}_1 = \mathtt{t}_2$. A proof of an equality is given by the term `join n m` where `n` and `m` are meta-level natural numbers serving as upper bounds on the number of reductions steps that the respective terms will take when deciding equality. The typing rule for `join` is shown as TJoin in Figure 5. This rule makes use of the function $|\cdot|$ that erases type annotations and implicit arguments. If a term in an equality proved by `join` reduces to a normal form or a stuck term in fewer steps than the bound given as an argument to `join`, then we compare that stuck term or normal form with the term resulting from reducing the other side of the equality in a similar manner. Equalities can be proved between terms that are non-terminating, provided that the two terms are joinable in a number of steps less than or equal to the given bounds. Furthermore, equalities can be proved between open terms; in this sense the joinability relation relies on the partial evaluation of terms.

$$\frac{|\mathtt{t}_1| \to^{<\mathtt{m}} \mathtt{t}_1' \quad |\mathtt{t}_2| \to^{<\mathtt{n}} \mathtt{t}_2' \quad \mathtt{t}_1' =_\alpha \mathtt{t}_2'}{\Gamma \vdash \mathtt{join}\,\mathtt{m}\,\mathtt{n} : \mathtt{t}_1 \;=\; \mathtt{t}_2}\ \text{TJoin}$$

$$\frac{\Gamma \vdash \mathtt{p} : \mathtt{t}_1 = \mathtt{t}_2 \quad \Gamma \vdash \mathtt{e}_1 : [\mathtt{t1}/\mathtt{x}]\mathtt{e}_2}{\Gamma \vdash \mathtt{conv}\;\mathtt{e1}\;\mathtt{by}\;\mathtt{p}\;\mathtt{at}\;\mathtt{x}.\;\mathtt{e2} : [\mathtt{t2}/\mathtt{x}]\mathtt{e2}}\ \text{TConv}$$

Figure 5. Join and Conversion Typing Rules

Conversion Sep^3 programs and proofs can utilize equalities to change the type of a given proof or program. The type rule for conversion is shown in Figure 5.

Syntactically, a conversion has the form `conv e by p at x.e'`, where `x` is a variable possibly occurring free in `e'`, and `p` is a proof of an equality `t1 = t2`. This will cast the expression `e` from the type `[t1/x]e'` to `[t2/x]e'`.

Consider the example of a length-indexed vector, `v` of length `plus Z n`. Using `join`, we can construct a proof that `plus Z n = n` in a bounded number of steps. Using a `conv`, we can then cast the type of `v` from `Vector [a] [plus Z n]` to `Vector [a] [n]` by changing the index from `plus Z n` to `n` using the supplied equality proof.

```
conv v by (join 10 0 : plus Z n = n)
  at x. Vector a x
```

Conversion in Sep^3 is not automatically inferred. Casts must be supplied by the programmer. This is because conversion is based on equality between programs and it is not possible to decide equality of programs, since they may not terminate.

Termination Reasoning Termination of programs is expressed with the termination formula, `t`!, indicating a term `t` is total. Termination proofs can be constructed in two ways. First, terms which can be judged (see Section 4) to be values, can trivially be proved terminating using the `valax` (pronounced "value axiom") form. The typing rule for `valax` is shown in Figure 7. This includes terms

[2] For abstraction over proofs, and application of programs to proofs, the square brackets are unnecessary, as these arguments will always be erased

which are syntactic values. Furthermore, termination proofs can be introduced using a termination case proof construct, `termcase`, that non-constructively case-splits on the termination behavior of a term (see Section 6).

Recursion and Induction The Sep^3 programming language includes a `rec` form for defining general recursive functions. This construct does not constrain the arguments to recursive calls, potentially allowing diverging computation. The proof language, in contrast, provides an `ind` form for induction over programs. This form requires the argument to recursive calls to be strictly decreasing in size. Recursive invocations of the formula must provide a proof of this, written `t' < t`, constructed using the `ord` form (Figure 6). An ordering can only be constructed between a known terminating data structure and one of its immediate sub-terms, ensuring that the induction is well-founded. A `ordtrans` construct allows induction to proceed on data that is not an immediate sub-term of its parent. The typing rules for `rec`, `ind`, `ord`, and `ordtrans` are shown in Figure 6.

Effects In the current Sep^3 design, we do not provide a primitive language mechanism for handling effects such as imperative state and exceptions. We instead assume that these effects can be encoded monadically. This design decision may lead readers to question why we handle non-termination in a special manner, when it can be encoded as a monadic effect as well. Our response is one of intent – in this design we're interested in allowing non-termination not necessarily because we want to define non-terminating computations, but rather because general recursion is often the most straightforward way to define functions of interest, regardless of whether they terminate. Type theories that require termination of all functions in contrast take the approach of requiring all functions to terminate, and then allowing an encoding of non-termination. In the Sep^3 design, we allow a programmer to define functions directly with general recursion, and then later prove termination separately. These positions occupy two different points in the design space; we believe that the Sep^3 design offers the advantage of incrementality.

3. Related Work

It is surprising that relatively few works and systems are concerned with external reasoning for call-by-value, general-recursive higher-order functional programs. Indeed, we are aware of no prior theorem proving systems which exactly address this very natural problem! NuPRL might be the closest, since it supports external reasoning about higher-order functional programs with general recursion, but it appears that the semantics is lazy rather than call-by-value [5]. The logics of theorem provers like Coq and Isabelle require all functions to be terminating, and then need not (and do not) include a particular reduction strategy as part of their semantics [15, 18]. ACL2 also requires totality of functions [11]. As mentioned above, there are methods for defining and reasoning about general-recursive functions, but these require a non-trivial encoding, for example, using co-inductive data types, domain predicates, or domain theory [4, 8**?** , 9]. Systems or theories for direct reasoning about general-recursive functions seem to be less widely used or known. LTC supports explicit reasoning about totality, conversion, and typing for (untyped) PCF programs (for a recent work on LTC, see [3]). Equality is based on conversion, rather than reduction, and hence no reduction strategy is privileged in the axiomatization of the theory. VeriFun supports reasoning about general-recursive, possibly underdefined functions [19]. The language of VeriFun does have call-by-value semantics and polymorphic types, but only first-order functions. Feferman's System W is a logical theory intended for the formalization of mathematics [6, Chapter 13]. Its language for function definition uses a (generally non-computable) search operator in place of general recursion, and its theory, like LTC, is based on conversion rather than reduction. The CFML tool automatically extracts a formula from an OCaml program that can be used for verification in Coq. It's used for external verification only, and not for a dependently-typed language [**?**].

The Ynot system, based on Hoare Type Theory, is a generalization of Hoare Logic to higher-order functional programs with general recursion, state, and call-by-value semantics [13, 14]. Thus, Ynot provides an internal verification solution to the problem of interest in this paper, and indeed to the further difficult matter of reasoning about state. But, to our knowledge, Ynot is not intended for external reasoning about programs; rather, it uses a monad indexed by pre- and post-conditions on the imperative state in order to perform internal verification of programs. Previous work of Stump and co-authors on Guru has similar goals as Sep^3, with a similar language design separating proofs and programs, and using termination casts with a judgmental notion of value [16, 17]. But in those works, quantifiers range only over values, rather than arbitrary programs; there is no construct for termination case; and the issue of call-by-value β-reduction is not addressed. Indeed, the Guru implementation unsoundly allows β-reduction with non-value arguments.[3]

4. A Value Judgment

In Sep^3, we syntactically classify some programs as values, as is usual when defining a language. Typically, in a call-by-value language like Sep^3 variables are identified as values, since the operational semantics of the language dictates that when instantiating a quantification the term used for the instantiation be reduced to a value before the instantiation takes place. Hence, inside the body of the quantification, the quantified variable can be assumed to range over values, since it will only be instantiated by values.

However, in Sep^3 we must distinguish between quantification in the program fragment and in the proof fragment: variables introduced by quantification in programs will only be instantiated with values, while logical quantification of programs is over program *expressions*, not values. The distinction is important, because it enables our "freedom of speech" principle. Proofs can *mention* programs without the expectation that those programs will be reduced (which would be dangerous if they diverged). If it were necessary to reduce programs to values to instantiate a proof quantifying over programs, then one of two strategies would be required.

1. The operational semantics of the proof language defined for meta-theoretical study would use a call-by-value β-reduction rule. Then the soundness of the proof fragment would depend on termination of the program fragment, but allowing non-termination of the program fragment is an explicit goal of the Sep^3 language design.
2. The second possibility is to only instantiate theorems about known terminating programs, perhaps by requiring a syntactic value restriction on applications of theorems quantified over terminating programs. This restriction, while sound, reduces the expressiveness of the language, as there are many theorems that are true regardless of whether it is possible to reduce programs those theorems quantify over to values. For example `plus x Z = x` regardless of whether `x` terminates. If `x` diverges then both sides of the equality diverge, and are hence still equal.

For Sep^3, we modify the definition of the set of terms which are values. Rather than a simple syntactic definition, we instead utilize a judgmental definition of value, allowing the context to be used when determining whether a term is a value. Figure 7 shows the

[3] This issue with the Guru implementation, discovered in the course of the current research on Sep^3, is currently being repaired.

$$\frac{\Gamma, \mathtt{x} : \mathtt{T}, \mathtt{f} : \mathtt{forall}(\mathtt{y} : \mathtt{T})(\_ : \mathtt{y} < \mathtt{x}).[\mathtt{y}/\mathtt{x}]\mathtt{P} \vdash \mathtt{p} : \mathtt{P}}{\Gamma \vdash \mathtt{ind}\ \mathtt{f}\ (\mathtt{x} : \mathtt{T})\ [\mathtt{u}].\mathtt{p} : \mathtt{forall}(\mathtt{x} : \mathtt{T})(\mathtt{u} : \mathtt{x}!).\mathtt{P}}\ \text{TInd} \qquad \frac{\Gamma \vdash \mathtt{p}_1 : \mathtt{t}_1 < \mathtt{t}_2 \quad \Gamma \vdash \mathtt{p}_2 : \mathtt{t}_2 < \mathtt{t}_3}{\Gamma \vdash \mathtt{ordtrans}\ \mathtt{p}_1\ \mathtt{p}_2 : \mathtt{t}_1 < \mathtt{t}_3}\ \text{TOrdTrans}$$

$$\frac{\Gamma, \mathtt{x} : \mathtt{T}, \mathtt{f} : (\mathtt{x} : \mathtt{T}_1) \to \mathtt{T}_2 \vdash \mathtt{t} : \mathtt{T}_2}{\Gamma \vdash \mathtt{rec}\ \mathtt{f}\ (\mathtt{x} : \mathtt{T}_1).\mathtt{t} : (\mathtt{x} : \mathtt{T}_1) \to \mathtt{T}_2}\ \text{TRec} \qquad \frac{\Gamma \vdash \mathbf{val}\ \mathtt{t}_1 \quad \Gamma \vdash \mathbf{val}\ \mathtt{t}_2 \quad \Gamma \vdash \mathtt{p} : \mathtt{t}_2 = \mathtt{C}_\mathtt{i}\ \mathtt{u}_0 \ldots \mathtt{t}_1 \ldots \mathtt{u}_\mathtt{n}}{\Gamma \vdash \mathtt{ord}\ \mathtt{p} : \mathtt{t}_1 < \mathtt{t}_2}\ \text{TOrd}$$

Figure 6. Induction and Recursion Typing Rules

$$\frac{\Gamma, \mathtt{x} : \mathtt{T}^{\mathbf{val}} \vdash \mathtt{t} : \mathtt{T}'}{\Gamma \vdash \backslash(\mathtt{x} : \mathtt{T}) \to \mathtt{t} : (\mathtt{x} : \mathtt{T}) \to \mathtt{T}'}\ \text{TLamProg}$$

$$\frac{\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{p} : \mathtt{P}}{\Gamma \vdash \backslash(\mathtt{x} : \mathtt{T}) \Rightarrow \mathtt{p} : \mathtt{forall}(\mathtt{x} : \mathtt{T}).\mathtt{P}}\ \text{TLamProof}$$

$$\frac{\Gamma \vdash \mathbf{val}\ \mathtt{t}}{\Gamma \vdash \mathtt{valax}\ \mathtt{t} : \mathtt{t}!}\ \text{TValAx}$$

Figure 7. Typing rules for quantification and valax

$$\frac{\mathtt{x} : \mathtt{A}^{\mathbf{val}} \in \Gamma}{\Gamma \vdash \mathbf{val}\ \mathtt{x}}\ \text{ValVar}$$

$$\frac{}{\Gamma \vdash \mathbf{val}\ \backslash \mathtt{x} : \mathtt{A}.\mathtt{t}}\ \text{ValLam}$$

$$\frac{\forall i. \Gamma \vdash \mathbf{val}\ \mathtt{t}_\mathtt{i}}{\Gamma \vdash \mathbf{val}\ \mathtt{C}\ \mathtt{t}_0\ \ldots\ \mathtt{t}_\mathtt{n}}\ \text{ValCons}$$

$$\frac{}{\Gamma \vdash \mathbf{val}\ \mathtt{tcast}\ \mathtt{t}\ \mathtt{by}\ \mathtt{p}}\ \text{ValTCast}$$

Figure 8. Selected Value Judgment Rules
.

rules for quantification over programs in both the proof and program languages. In the case of quantification in the proof language (rule TLamProof), the quantification variable is added to the context without a value annotation. Conversely, in the program fragment (rule TLamProg) the quantification is added with an additional `val` annotation on the typing assumption.

The value judgment for programs (Figure 8) includes a rule, ValVar, which identifies a variable as a value if it occurs in the context with a `val` annotation. The value judgment includes a number of axioms for each syntactic value form. We also show a representative subset of syntactic forms that are axiomatically judged values; others not depicted include dependent products, recursive functions, and the classifier `Type`. Programs that can be judged to be values can be proved to be terminating using the `valax` construct. The typing rule for `valax` is shown in Figure 7. As an example, we can construct a proof `p : S Z !` for a ground constructor value `S Z`, which is judgmentally a value. More interestingly, if we have $\mathtt{x} : \mathtt{Nat}^{\mathbf{val}}$ in context, then we can prove `S x !` using `valax`.

The ValCons rule identifies a constructor application to arguments, each judged as values, to be judged as a value. We utilize this rule when typing case expressions in proofs scrutinizing program values. When case-splitting on a program value in a proof, it is necessary to supply a proof to the case expression that the scrutinee terminates, to ensure that divergence of programs does not leak into the proof language. We are guaranteed (by virtue of the termination proof) that the scrutinee will terminate, so we are also guaranteed that the normal form of the scrutinee will be a construction with arguments that are values. Thus, when performing reasoning within a case branch, the pattern variables for the case branch will necessarily be instantiated with values. Consequently, when adding those variable to the context when type checking a case branch, we mark those variables as values, as shown in the ProofCase rule in Figure 9.

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{p} : \mathtt{t}! \\ \Gamma, \mathtt{y} : \mathtt{t} = \mathtt{Z} \vdash p_1 : P \\ \Gamma, \mathtt{x} : \mathtt{Nat}^{\mathbf{val}}, \mathtt{y} : \mathtt{t}\ =\ \mathtt{S}\ \mathtt{x} \vdash \mathtt{p}_2 : \mathtt{P} \\ \Gamma \vdash \mathtt{t} : \mathtt{Nat}\end{array}}{\Gamma \vdash \mathtt{case}\ \mathtt{t}\ \{\mathtt{y}\}\ \mathtt{p}\ \mathtt{of}\ \mathtt{Z} \to \mathtt{p}_1 \mid \mathtt{S}\ \mathtt{x} \to \mathtt{p}_2 : P}\ \text{TCaseProofNat}$$

Figure 9. Typing for Case in Proofs, Specialized to Nat

5. Evaluation with Termination Cast

In Sep^3, propositional equalities are proved using the `join` construct, which forms equalities by evaluating the erasure of each side of the equality to a normal form, a stuck term, or a maximum number of steps and then comparing the resulting terms modulo α-equality. This means that `join` depends on call-by-value reduction of programs, where β-reduction can only be performed in a context where the function argument is a value. However, reasoning in proofs is often over open terms, where variables occurring in the propositional equalities range over programs which need not be values. Variables representing quantification over programs are not treated as values due to the freedom-of-speech principle. This principle was designed to allow proofs to quantify over programs, including those that diverge.

Unfortunately, quantifying over all programs, not just values, causes difficulty. Evaluating expressions which include free program variables introduced by quantifiers in proofs will result in a stuck term whenever such a variable occurs in the argument position of a β-redex. We want to reason about terms, but our most powerful tool (reasoning about equality under β-reduction), is restricted because the programs we wish to reason about might possibly diverge.

Consider the following theorem, that expresses that the polymorphic identity function is, in fact, an identity.

```
type id : [a:Type] -> (y:a) -> a
prog id = \[a:Type](y:a) -> y

theorem id_is_id :
  forall (a:Type)(x:a).id [a] x = x
proof id_is_id =
  \(a:Type)(x:a) => join 100 0
```

This proof fails because the two terms are not joinable. The reduction sequence shown below demonstrates the problem.

$$\begin{array}{ll} \mathtt{id\ [a]\ x} & \{\text{by erasure}\} \\ \mathtt{id\ x} & \{\text{by def. of } \mathtt{id}\} \\ \mathtt{(\backslash y.y)\ x} & \not\rightsquigarrow \end{array}$$

The β-redex `(\y.y) x` is blocked because the Sep^3 programming language has a call-by-value semantics, and the variable `x` is not a value, as it was introduced by a proof quantification. The equality proved by `join` is too fine, because it can only equate terms

$$\frac{\Gamma \vdash \mathtt{t : T} \quad \Gamma \vdash \mathtt{p : t\,!}}{\Gamma \vdash \mathtt{tcast\ t\ by\ p : T}}\ \text{TTCast}$$

$$\frac{}{\mathtt{E[tcast\ v]} \to \mathtt{E[v]}}\ \text{ETCastVal}$$

$$\frac{}{\mathtt{E[(\lambda\ x.u)\ v]} \to \mathtt{E[[v/x]u]}}\ \text{EBeta}$$

Figure 10. Termination cast typing and reduction rules

based on joinability using the call-by-value reduction semantics of the programming language.

Consider the above example if a call-by-name evaluation strategy were used when proving equalities using `join`. The blocked β-redex `(\y.y) x` would step, because it is not necessary to reduce the argument to an application to a value prior to β-reduction. Making such a change would be unsafe, however, because it could allow us to observe different termination behaviors of the program, depending on whether the program is being reduced inside a proof (using call by name) or during actual execution (using call by value).

To illustrate, consider the term `(\x:Nat.Z) loop`, where `loop` stands for any diverging computation. Inside a proof, using a call-by-name semantics we could prove `(\x:Nat.Z) loop = Z`, while at run-time using call-by-value the program would diverge.

On the other hand, if the argument `t` to an application is known to be terminating, because a proof `p:t !` is available, then the termination behavior of such an application will remain the same, regardless of whether it is reduced using call-by-name or call-by-value β reduction.

Sep^3 provides a *termination cast* construct, `tcast`, that allows a programmer to mark expressions as known to be terminating. The `tcast` construct takes a program and a proof that the program has a value. The typing rule for `tcast` is shown in Figure 10.

To allow reasoning over expressions including `tcast` terms, the semantics of the programming language is augmented to allow an application with a `tcast` argument to step, despite the subject of the termination cast not necessarily being reduced to a value. In effect, `tcast` allows a programmer to posit a hypothetical value that the expression will reduce to, and then continue reduction based on that hypothesized value. This allows the language to prove more equalities than would be possible if `tcast` were not included.

Using `tcast`, a weaker form of the `id_is_id` theorem can be proved. The theorem is weakened to only hold for terminating arguments, by adding an additional parameter to the theorem that proves `x` is terminating. The proof is shown below.

```
theorem id_is_id_term :
  forall (a:Type)(x:a)(x_term:x!).id [a] x = x
proof id_is_id_term =
  \(a:Type)(x:a)(x_term:x!) =>
      join 100 0 : id [a] (tcast x by x_term) = x
```

In Section 4 we introduced a value judgment to differentiate between variables introduced by proof abstractions from those introduced by program abstractions. Using the value judgment, it is possible to redefine the β_v to use the value judgment.

$$\frac{\Gamma \vdash \mathbf{val}\ \mathtt{u'}}{\mathtt{E[(\lambda\ x.u)\ u']} \to \mathtt{E[[u'/x]u]}}\ \text{EBeta}$$

This rule allows the reduction rule to reuse the value judgment, but it comes at the cost of complicating the reduction relation. No longer is reduction defined syntactically, but now it is a contextual relation. The alternative approach, and the one we take in Sep^3, is to use the syntactic view of evaluation. We define a syntactic class of values for reduction purposes that contains all of the syntactic forms judged axiomatically to be values by the value judgment, represented by the production `v` in Figure 3. Also included in this syntactic category are programs wrapped with termination casts. Variables are not classified as values, as before. If a term can be judged a value, then it is possible to extract a term in this class of syntactic values using a termination cast. For example, if a variable `x` is judged a value (but would not be syntactically classified a value), then `tcast x by valax x` is the associated syntactic value. In this way, we simplify the reduction relation to use standard β_v over a class of syntactic values including `tcast` terms, while still using the value judgment for typing.

When constructing a proof of equality using `join`, the terms being equated are erased, as described previously. However, the `tcast` terms are preserved (although the termination proof can be dropped, as it has no computational content). Additionally, the programming language semantics are extended with the ETCastVal rule shown in Figure 10 that allows a `tcast` to be dropped when the expression being cast is a syntactic value. This prevents `tcast` from blocking a redex, as would be the case with `(tcast \x.x by p) v`.

Preserving termination casts during `join` reduction is necessary to allow `join` to construct equalities between terms involving expressions that do not normalize to values. After terms are reduced with `join`, they are compared modulo termination casts and renaming of bound variables.

Preserving termination casts when reasoning about programs using `join` may cause a term to take more reduction steps inside proofs, because the EBeta reduction step may duplicate `tcast` terms requiring a non-zero number of reduction steps to normalize to values. This need not introduce inefficiencies in compiled programs, because termination casts only occur in proofs, which are erased during compilation.

6. Termination Case

Sep^3 includes a `termcase` (Figure 11) construct that allows a proof to case-split on whether a scrutinized program terminates or diverges, with `y` bound as a proof of the corresponding termination assumption in each branch. In the terminates (`!`) branch, `y` is a proof that the scrutinee terminates. In the diverges (`abort`) branch, `y` is a proof that the scrutinee is equal to `abort`, signifying divergence. The typing rule TAbort, for `abort` in the annotated language, requires an annotation `t` providing the type of the `abort` term. Reduction for `abort` is defined by the rule EAbort. If `abort` appears in evaluation position, then the term immediately steps to `abort`. This means that all provably diverging terms (identified by equivalence to `abort`) are contextually equivalent.

The `termcase` construct is non-constructive, as it axiomatizes excluded middle for termination, an undecidable property. The construct is not computational, as the proof language reduction rules for `termcase` defined by the ETermCaseAbort and ETermCaseTerm rules listed in Figure 11 show.

Reduction of `termcase` relies on an unimplementable oracle (represented by the premises of ETermCaseAbort and ETermCaseEval) that can determine whether any given term normalizes to a value. The construct can be viewed as an internalization of the theorem of type soundness for the program fragment, relaxed to partial correctness: if $\vdash$ `t` : `T`, then either `t` $\to^*$ `v` (where `v` is a syntactic value) and $\vdash$ `v` : `T!` , or else `t` diverges. This has already been observed in Wright and Felleisen's classic paper [20].

Using `termcase` we can strengthen proofs of algebraic theorems, which are subject to termination preconditions, to stronger theorems that do not require termination preconditions. As a simple example, we can return to the proof of `id_is_id` from Section 5. To prove the theorem above, it is necessary to have a proof of termination available to `tcast` the variable `x`, so that the `join` proof can succeed. Nevertheless, the theorem is valid for all inputs, irre-

$$\frac{\Gamma, \mathtt{y} : \mathtt{t} = \mathtt{abort}\ \mathtt{t}' \vdash \mathtt{p}_{\mathtt{a}} : \mathtt{P} \quad \Gamma, \mathtt{y} : \mathtt{t}! \vdash \mathtt{p}_{!} : \mathtt{P} \quad \Gamma \vdash \mathtt{t} : \mathtt{t}'}{\Gamma \vdash \mathtt{termcase}\ \mathtt{t}\ \{\mathtt{y}\}\ \mathtt{of}\ \mathtt{abort}\ \rightarrow\ \mathtt{p}_{\mathtt{a}} \mid !\ \rightarrow \mathtt{p}_{!} : \mathtt{P}}\ \text{TTermCase}$$

$$\frac{\Gamma \vdash \mathtt{t}\ :\ \mathtt{Type}}{\Gamma \vdash \mathtt{abort}\ \mathtt{t}\ :\ \mathtt{t}}\ \text{TAbort}$$

$$\frac{\mathtt{if}\ \mathtt{t}\ \mathtt{diverges}}{\mathtt{P}[\mathtt{termcase}\ \mathtt{t}\ \{\mathtt{t}_{\mathtt{eq}}\}\ \mathtt{of}\ \mathtt{abort}\ \rightarrow \mathtt{p}_1 | ! \rightarrow \mathtt{p}_2] \rightarrow \mathtt{P}[\mathtt{p}_1]}\ \text{ETermCaseAbort}$$

$$\frac{\mathtt{if}\ \mathtt{t}\ \mathtt{terminates}}{\mathtt{P}[\mathtt{termcase}\ \mathtt{t}\ \{\mathtt{t}_{\mathtt{eq}}\}\ \mathtt{of}\ \mathtt{abort}\ \rightarrow \mathtt{p}_1 | ! \rightarrow \mathtt{p}_2] \rightarrow \mathtt{P}[\mathtt{p}_2]}\ \text{ETermCaseTerm}$$

$$\frac{}{\mathtt{E}[\mathtt{abort}] \rightarrow \mathtt{abort}}\ \text{EAbort}$$

Figure 11. Termination Case Typing and Reduction Rules

spective of termination behavior. Using termcase, this theorem can be strengthened.

```
theorem id_is_id :
  forall (a:Type)(x:a).id [a] x = x
proof id_is_id =
 \(a:Type)(x:a) =>
   termcase x {x_term} of
    abort ->
     let u1 [u1_eq] = join 100 100
        : id [a] (abort a) = (abort a)
     in conv u1 by x_term
          at hole. (id [a] hole = hole)
    | ! -> id_is_id_term a x x_term
```

The proof does not require a separate lemma proving id to be a total function on terminating inputs. This capability – proving theorems about programs without proving those programs total – is an important proof technique enabled by termcase. Although id_is_id amounts to such a proof for this particular example, in general termcase allows us to do algebraic reasoning without resorting to proving termination of the functions we are reasoning over.

7. Implementation and Experience

To gauge the feasibility of the Sep^3 language design, we have implemented a prototype type checker and evaluator. This implementation has been useful in guiding the language design, and the design of Sep^3 and the implementation have continued to evolve in tandem.

Sep^3 is designed as a core language, and requires a large number of programmer annotations to get the type checker to accept a program. Requiring such a large number of annotations simplifies the design of the language and the implementation of the tools, but it complicates *using* the language, as the annotation burden is quite great. A second goal of the Trellys project is to design a surface language that allows many of the necessary core annotations to be synthesized by a program analysis algorithm. Through use of the core language implementation, we have identified some initial approaches to automating proof and program construction, reducing the programmer burden.

The use of explicit conversion in Sep^3 requires a large number of programmer annotations to change the type of programs and proofs. The process of proving an equality often involves constructing a proof relating the desired left-hand side of an equality to some intermediate term, changing the equality using a conversion, building another equality with the converted intermediate term and the desired right-hand side, and then linking the two using transitivity of equality. While a laborious and verbose task, the process is relatively straightforward. We have implemented a small proof tactic, morejoin that allows a programmer to specify a number of equality proofs. The morejoin tactic behaves in the same way as join, but when a term is stuck it attempts to rewrite the term using one of the supplied equalities and continue evaluating. This simple approach elaborates directly to the manual process described above, yet has the potential to vastly simplify proofs. A small enhancement to morejoin allows the programmer to also specify termination proofs, which will then be used to automatically insert termination casts to step blocked β-redexes.

Figures 12-15 show an example proof of associativity of append for lists using the prototype Sep^3 implementation. The proofs liberally use morejoin to automatically insert conversions and termination casts when constructing equalities, rather than using the more verbose join.

The proof of associativity of is by induction on the structure of the list argument. Because the lists are programs, it is necessary to provide proofs that the arguments to append terminate. The weak form of the theorem is shown in Figure 14. The proof proceeds by induction on xs, so it requires a proof that xs is terminating. Within the body of the proof, we prove equalities involving the variables ys and zs. Because these variables are quantified by a proof, they range over expressions, so it is necessary to use termination casts, inserted by morejoin using the supplied termination proofs, to reduce terms involving these variables.

The proof uses an additional lemma (Figure 13) that proves append total on terminating inputs. This is a convenience, to simplify presentation, and could be avoided with additional reasoning using termcase. For more complex functions, where the proof of totality is not so straightforward, using termcase may be preferable.

Because the programs xs, ys, and zs are all used in strict positions on both sides of the equality, the formula can be strengthened to an equality over all program terms producing lists, regardless of whether they terminate. Figure 15 shows the generalization of the proof of associativity of append to potentially non-terminating arguments.

The proof uses termcase to consider the termination behavior of each argument in turn. In each abort branch, the EAbort rule allows us to join an application of append to the diverging argument with abort, demonstrating that both sides of the associativity formula join with abort. In the final terminates branch, the context contains proofs xs_term, ys_term, and zs_term that prove the associated arguments are terminating. With these proofs available, the weaker append_assoc lemma can be invoked.

```
data List : (a:Type) -> Type where
   Nil : List a
 | Cons : (x:a) -> (xs:List a) -> List a

type append : [b:Type] -> (xs:List b) -> (ys:List b)  -> List b
prog rec append [b:Type] (xs:List b) = \ (ys:List b) ->
    case xs {xs_eq} of
       Nil -> ys
     | Cons x xs' -> Cons [b] x (append [b] xs' ys)
```

Figure 12. List Append

```
theorem append_term : forall(a:Type)(xs:List a)(xs_term:xs!)(ys:List a)(ys_term:ys!).(append [a] xs ys)!
proof ind append_term [a:Type](xs:List a){xs_term} = \(ys:List a)(ys_term:ys!) =>
   case xs {xs_eq} xs_term of
      Nil -> let u1 [u1_eq] = morejoin {sym xs_eq, xs_term, ys_term} : append [a] xs ys = ys
             in conv ys_term by (sym u1) at hole. hole !
    | Cons x xs' -> let ih [ih_eq] = append_term [a] xs' (ord xs_eq : xs' < xs) ys ys_term;
                        x_term [x_term_eq] = valax x : x!;
                        unroll_app [unroll_app] = morejoin {sym xs_eq,xs_term,ys_term }
                                  :  append [a] xs ys = Cons [a] x (append [a] xs' ys);
                        u1 [u1_eq] = value (Cons [a] ~x_term ~ih)
                    in conv u1 by (sym unroll_app) at hole. hole !
```

Figure 13. Proof that append terminates on terminating inputs

```
theorem append_assoc_term :
  forall (a:Type) (xs:List a)(xs_term:xs!) (ys:List a)(ys_term:ys!)(zs:List a)(zs_term:zs!) .
    append [a] xs (append [a] ys zs) = append [a] (append [a] xs ys) zs
proof ind append_assoc_term [a:Type](xs:List a){xs_term} =
 \(ys:List a)(ys_term:ys!) (zs:List a)(zs_term:zs!) =>
   let term_xs_ys [txy_eq] = append_term [a] xs xs_term ys ys_term;
       term_ys_zs [tyz_eq] = append_term [a] ys ys_term zs zs_term
   in case xs {xs_eq} xs_term of
       Nil ->
        let u1 [u1_eq] = morejoin {sym xs_eq, ys_term, xs_term} :
            ys = append [a] xs ys;
            u2 [u2_eq] = morejoin {sym xs_eq, xs_term} :
             append [a] xs (tcast (append [a] ys zs) by term_ys_zs)
              =  append [a] ys zs;
            u3 [u3_eq] = morejoin {sym xs_eq, xs_term, ys_term} :
            ys = append [a] xs ys
        in conv u2 by u3
            at hole. append [a] xs (append [a] ys zs) =  append [a] hole zs

     | Cons x xs' ->
        let unroll_app [ua]  = morejoin {sym xs_eq,xs_term, term_ys_zs} :
              append [a] xs (append [a] ys zs)
              = Cons [a] x (append [a] xs' (append [a] ys zs));
            ih [ih_eq] = append_assoc_term [a] xs' (ord xs_eq) ys ys_term zs zs_term;
            u1 [u1_eq] = conv unroll_app by ih at hole.
              append [a] xs (append [a] ys zs) = Cons [a] x hole;
            u2 [u2_eq] = morejoin {sym xs_eq, xs_term, ys_term} :
              append [a] xs ys = Cons [a] x (append [a] xs' ys);
            term_xs'_ys [tx_eq] = append_term [a] xs' (valax xs') ys ys_term;
            u3 [u3_eq] = morejoin {sym xs_eq, xs_term,ys_term} :
              (append [a] (append [a] xs ys) zs)
               = (append [a] (Cons [a] x (append [a] xs' ys)) zs);
            u4 [u4_eq] = morejoin {zs_term, ys_term, valax x, term_xs'_ys} :
              append [a] (Cons [a] x (append [a] xs' ys)) zs
              = Cons [a] x (append [a] (append [a] xs' ys) zs);
            u5 [u5_eq] = trans u3 u4 :
                append [a] (append [a] xs ys) zs
                = Cons [a] x (append [a] (append [a] xs' ys) zs)
        in conv u1 by (sym u5) at hole.
                append [a] xs (append [a] ys zs) = hole
```

Figure 14. Associativity of List Append

```
theorem append_assoc :
  forall (a:Type) (xs:List a)(ys:List a)(zs:List a).
    append [a] xs (append [a] ys zs) = append [a] (append [a] xs ys) zs
proof append_assoc =
 \(a:Type) (xs:List a) (ys:List a) (zs:List a) =>
  termcase xs {xs_term} of
     abort ->
      let aleft [al_eq] = join 100 100 :
           (append [a] (abort (List a)) (append [a] ys zs)) = (abort (List a));
          aright [ar_eq] = join 100 100 :
           (abort (List a)) = (append [a] (append [a] (abort (List a)) ys) zs);
          u1 [u1_eq] =  trans aleft aright
      in conv u1 by x_term at hole.
          append [a] hole (append [a] ys zs) =
          append [a] (append [a] hole ys) zs
   | ! ->
      termcase ys {ys_term} of
         abort ->
          let aleft [al_eq] = morejoin {xs_term} :
               (append [a] xs (append [a] (abort (List a)) zs)) = abort (List a) ;
              aright [al_eq] = morejoin {xs_term} :
                 (abort (List a)) = (append [a] (append [a] xs (abort (List a))) zs);
              u1 [u1_eq] = trans aleft aright
          in conv u1 by ys_term at hole.
                append [a] xs (append [a] hole zs) =
                append [a] (append [a] xs hole) zs
       | ! ->
        termcase zs {zs_term} of
          abort -> let aleft [al_eq] = morejoin {xs_term,ys_term} :
                        (append [a] xs (append [a] ys (abort (List a)))) = abort (List a);
                       a_x_y_term [axy_eq] = append_term a xs xs_term ys ys_term;
                       aright [ar_eq] = morejoin {xs_term,ys_term,a_x_y_term} :
                        (abort (List a)) = (append [a] (append [a] xs ys) (abort (List a)));
                       u1 [u1_eq] = trans aleft aright
                   in conv u1 by zs_term at
                       hole.  append [a] xs (append [a] ys hole) =
                              append [a] (append [a] xs ys) hole
       | ! -> append_assoc_term [a] xs xs_term ys ys_term zs zs_term
```

Figure 15. Generalizing associativity of list append to non-terminating arguments.

8. Conclusion

Trellys is a research project investigating the design of a dependently typed programming language with call-by-value semantics and general recursion. Sep^3 is a core language design for Trellys, and occupies, to the best of our knowledge, a unique position in the language design space. Sep^3 supports internal and external verification, while not requiring a programmer to resort to indirect encodings to implement general recursive functions.

Sep^3 uses a syntactic distinction between the proof and programming languages to isolate non-termination in the programming language from the proof language. Despite the syntactic distinction proofs can *mention* programs in a sound way, a capability dubbed “Freedom of Speech”.

Reasoning about programs with general recursion in a dependently typed language requires a number of modifications to the logic to ensure soundness while maintaining expressiveness. Variables can range, depending on context, over values or expressions, so Sep^3 includes a value judgment to differentiate the two. Equality proofs are constructed using partial evaluation of open terms, so termination casts are added to allow the programming language to soundly extend call-by-value reduction over non syntactic values. Many theorems are valid regardless of the termination behavior of the terms the theorems quantify over, so a termination case expression allows us to express those theorems, and furthermore allows us to reason about possibly diverging programs without proving termination.

Trellys remains a work in progress, and Sep^3 represents one attempt at defining a core language to support the desired goal of combining dependent types and a call-by-value language including general recursion. While the principal language design includes the concepts presented here as well as many other features, much work remains, most importantly the analysis of the meta-theoretical properties of the language design. Other future work involves the following topics.

- Sep^3 depends on a syntactic separation between the proof and programming fragments of the language. The Trellys team continues investigation into methods to remove this syntactic distinction, including an internalized type representing the proof/program classification of a term. This allows terms to safely be migrated from the proof language to the programming language, and vice-versa.
- Sep^3 is designed as a core language, intended as an internal source for meta-theoretical study and to ease implementation of language processing tools. Core language proofs and programs are verbose, requiring a programmer to add a large number of explicit annotations. The Trellys project is investigating methods to automate the insertion of these annotations. We intend to collect these methods to form the basis of a surface language more amenable to programmer use.

A prototype implementation of the language design presented in this paper is available at `http://code.google.com/p/trellys/`

lib/sep. A type checker and evaluator for the core language are in place, and continue to evolve to track minor changes to the language design. The tool also contains a number of enhancements to automate and simplify the use of the language.

Acknowledgments

This work was supported by the National Science Foundation (NSF grant 0910500).

References

[1] A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

[2] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda - a functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009.

[3] A. Bove, P. Dybjer, and A. Sicard-Ramírez. Embedding a logical theory of constructions in agda. In T. Altenkirch and T. D. Millstein, editors, *PLPV*, pages 59–66. ACM, 2009.

[4] V. Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.

[] A. Charguéraud. Program verification through characteristic formulae. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 321–332, New York, NY, USA, 2010. ACM.

[5] R. Constable and the PRL group. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.

[6] S. Feferman. *In the Light of Logic*. Oxford University Press, 1998.

[7] M. Kaufmann, P. Manolios, and J. Moore. *Computer-aided reasoning: an approach*. Springer Netherlands, 2000.

[8] A. Krauss. Partial and nested recursive function definitions in higher-order logic. *Journal of Automated Reasoning*, 44(4):303–336, 2010.

[9] A. Krauss. Recursive definitions of monadic functions. In A. Bove, E. Komendantskaya, and M. Niqui, editors, *Workshop on Partiality and Recursion in Interactive Theorem Provers (PAR)*, volume 43 of *EPTCS*, pages 1–13, 2010.

[10] N. Linger. *Irrelevance, Polymorphism, and Erasure in Type Theory*. PhD thesis, Portland State University, 2008.

[] J. Longley and R. Pollack. Reasoning about cbv functional programs in isabelle/hol. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *TPHOLs*, volume 3223 of *Lecture Notes in Computer Science*, pages 201–216. Springer, 2004.

[11] Matt Kaufmann and J Moore. A Precise Description of the ACL2 Logic, 1998. Available from the ACL2 web site.

[] R. Milner. LCF: A way of doing proofs with a machine. In J. Becvr, editor, *Mathematical Foundations of Computer Science 1979*, volume 74 of *Lecture Notes in Computer Science*, pages 146–159. Springer Berlin / Heidelberg, 1979.

[12] N. Mishra-Linger and T. Sheard. Erasure and polymorphism in pure type systems. In R. M. Amadio, editor, *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2008.

[13] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 62–73. ACM, 2006.

[14] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In J. Hook and P. Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming (ICFP)*, pages 229–240, 2008.

[15] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[16] A. Stump and E. Austin. Resource Typing in Guru. In J.-C. Filliâtre and C. Flanagan, editors, *Proceedings of the 4th ACM Workshop Programming Languages meets Program Verification, PLPV 2010, Madrid, Spain, January 19, 2010*, pages 27–38. ACM, 2010.

[17] A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson. Verified Programming in Guru. In T. Altenkirch and T. Millstein, editors, *Programming Languges meets Program Verification (PLPV)*, 2009.

[18] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.3*. INRIA, 2010. Available from http://coq.inria.fr/V8.3/refman/.

[19] C. Walther and S. Schweitzer. Automated Termination Analysis for Incompletely Defined Programs. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference (LPAR), Montevideo, Uruguay*, pages 332–346, 2005.

[20] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.