Equational reasoning about programs with general recursion and call-by-value semantics

Garrin KIMMELL¹, Aaron STUMP², Harley D. EADES³, Peng FU⁴, Tim SHEARD⁵, Stephanie WEIRICH⁶, Chris CASINGHINO⁷, Vilhelm SJÖBERG⁸, Nathan COLLINS⁹ and Ki YUNG AHN¹⁰

¹,²,³,⁴ University of Iowa
⁶,⁷,⁸ University of Pennsylvania
⁵,⁹,¹⁰ Portland State University

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 arbitrary data types. 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 diverging programs.

KEYWORDS
design, languages, verification, theory

1 Introduction
When writing verified programs, there are two traditional approaches. Theorem provers like ACL2 and Isabelle are used to perform verification externally [13], [24]. 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 [34] and Agda [3] 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 co-inductive data, are required to define truly partial functions [5]. Alternatively, one can formulate a domain of definition for which the functions are, in fact, total, using an accessibility predicate [2]. This basic idea has also been used for higher-order logics [15].

Relatively few theories have been proposed for direct reasoning about general recursion; examples are LCF, LTC and VeriFun [4], [20], [35].

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 log-
ics, allowing a programmer to utilize both internal and external verification techniques in the presence of these logically 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 allowing proofs abstractions to range over all programs, including ones that diverge?
- The theory of call-by-value β-equality is fairly weak, given the restriction that the argument to β-reduction must 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 achieved 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 8) with call-by-value reduction, so that programs that are proved 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 Sep³¹, one of several core language designs that we are developing in the Trellys project. The Sep³ core language is a work-in-progress, and still under development. As a result it has not been subject to meta-theoretic study and proofs of standard type system results (such as type soundness) are beyond the scope of this paper. In this regard, it is similar to works like that on ΠΣ, which explore a novel language design without conducting any substantial metatheoretic investigations¹. Indeed, tools as important for research in dependent type theory as Coq and Agda lack in their current as opposed to historic forms a complete metatheory, and soundness bugs have been found in recent years in both tools [12], [25].² The goal of achieving highly reliable proof assistants by stringent metatheoretic analysis, and even verification of their implementations, is one we strongly endorse. But to warrant the tremendous theoretical and engineering investment required to realize such a goal, it is necessary first to investigate language designs carefully, to ensure they are adequate for their applications in practice. It is in this exploratory spirit that we study our proposed language design from the perspective of expressivity and applicability, and defer its metatheoretic analysis to future work.

This article is an extension of an earlier paper [14]. The organization of the paper has changed slightly, and the Sep³ language definition has been mildly extended. More significant changes include:

1. Section 6.1 describes a number of lightweight proof automation techniques we developed in the

²¹ We write “program” for terms in the non-logical fragment of the language, and write “proof” for terms in the logical fragment.
²² The bug in Coq, despite being 4.5 years old, still persists in the most recent version of Coq 8.4.
course of experimenting with Sep\textsuperscript{3}. Sep\textsuperscript{3} is designed as a core language design, and as a result is rather verbose. The techniques described in this section demonstrate that – with relatively simple methods – we can eliminate some of the drudgery from programming with the core language.

- Section 7 describes a solution in Sep\textsuperscript{3} to a verification problem for an interpreter for SK combinators. This problem, taken from the 2012 VSTTE Verified Software competition is interesting because the central program in question (i.e., the interpreter) is non-terminating.\textsuperscript{3)} Though the Sep\textsuperscript{3} solution was not submitted to the competition, in this section we contrast the Sep\textsuperscript{3} solution with published solutions of systems that did compete.

The remainder of this paper is structured as follows. Section 2 provides a brief overview of a Sep\textsuperscript{3}, and outlines the key design principles we followed. In Sections 3 through 5 we detail the problems identified above and present their associated solutions. Section 6 gives a brief overview of our experiences using a prototype implementation of Sep\textsuperscript{3}, identifies opportunities for proof automation, and discusses an example proof. Section 7 presents the larger case study of verification of properties of the SK-combinator interpreter. Section 8 compares Trellys with related work. We conclude and identify directions for future work in Section 9.

2 Language overview

The Sep\textsuperscript{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. A central feature design decision is that all proofs are computationally irrelevant: they are erased prior to reduction. We also allow programmers to mark certain parts of programs as compile-time. They will also be erased prior to reduction.

In the exposition and judgments that follow, we use a few conventions to make the syntactic distinction clear. The metavariable \( \mathfrak{p} \) refers to proofs, and \( \mathfrak{p} \) to propositions. Propositions classify proofs (i.e. \( \mathfrak{p} : \mathfrak{P} \)), in the sense that types classify programs in the programming language. The programming language, in contrast, has a collapsed syntax where programs and types are taken from a single syntactic category. We will use metavariables \( \tau \) and \( \mathfrak{P} \) for expressions when we wish informally to emphasize the roles of subject (\( \tau \)) and classifier (\( \mathfrak{P} \)), respectively. We will use the metavariable \( \mathfrak{e} \) to represent either proofs (\( \mathfrak{p} \)) or programs (\( \tau \)) and the metavariable \( \mathfrak{A} \) to represent either propositions (\( \mathfrak{P} \)) or program types (\( \mathfrak{T} \)). The metavariables \( \mathfrak{x}, \mathfrak{y}, \mathfrak{f}, \mathfrak{q} \) range over both program and proof variables.

The syntax of the features of Sep\textsuperscript{3} that we discuss in this paper is summarized in Figure 1. Various judgments used in the paper are summarized in Figure 2. Typing rules for proofs are given in Figure 5, and for programs in Figure 6. Typing is defined for annotated programs (and proofs). We also define the syntax of unannotated programs in Figure 3. The language’s call-by-value (CBV) operational semantics is defined on unannotated programs in Figure 7. Annotated programs erase to unannotated ones (with similar names), using an eraser function defined in Figure 4. This function is also invoked in the TLamImp rule, to ensure that a compile-time input is not used in the erasure of the body of the compile-time abstraction \( \{ x : \mathfrak{A} \} \mapsto \tau \). In the rules, we assume all typing contexts are well-formed; that is, for any \( x : \mathfrak{A} \) in a context \( \Gamma \) the type \( \mathfrak{A} \) is kindable. In the proof fragment this requires the kinding judgment \( \Gamma \vdash \mathfrak{p} : \text{Formula} \) over propositions. We elide the definition.

As noted above, Sep\textsuperscript{3} is intended as a core language, so we are willing to require significant annotations in order to avoid complicating the design with complex machinery for type inference or term reconstruction. As the examples later in this paper make apparent, writing proofs and programs directly in the core language can be 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 Erasure The use of annotations in Sep\textsuperscript{3} ensures typing is algorithmic. These annotations do not have computational content, as they are erased prior to reduction. For example, changing the type of a program requires a programmer to explicitly insert the cast (\( \text{cast} \)) into the program. But such casts are dropped by erasure. Erasure also drops type annotations on abstractions, and all proofs. This is because proofs in Sep\textsuperscript{3} are purely specification, and do not have any run-time behavior.

Sep\textsuperscript{3} provides a mechanism for a programmer to specify which programs should be preserved across the eraser, by allowing abstractions and applications to be

\textsuperscript{3)} See https://sites.google.com/site/vstte2012/compet Author Stump was a co-organizer of the competition, and submitted the problem description upon which this section is based.
variables \ w, x, y, z, f, q
natural numbers \ m, n
expressions \ e ::= p | t
classifiers \ A ::= P | T
proof conversion context \ \hat{P} ::= \ldots P \ldots | \neg p
program conversion context \ \hat{T} ::= \ldots T \ldots | \neg p
typing contexts \ \Gamma ::= \emptyset | \Gamma, x:A | \Gamma, x:A^\text{val}
substitution \ \sigma ::= [e_1/x]e_2 | [e_1/x_1, \ldots, e_i/x_i]e_i

proofs \ p ::= x | \{x:A\} => p | p e
| join n
| valax t
| termcase t \{y\} of abort \rightarrow p_1 | ! \rightarrow p_2
| ord p | ordttrans p_1 p_2
| ind f(x:T) \{q\}. p
| case t \{y\} p of C_1 \rightarrow p_1 \ldots C_n \rightarrow p_n
| pack x,p
| unpack p as (x,p) in p
| conv p at \hat{P}
| let x = p in p_2
| let x \{y\} = t in p
| contra p_1 | contraabort p_1 p_2

properties \ P ::= \forall (x:A). P | \exists (x:A). P
| t_1 = t_2 | t_1 < t_2
| t!

programs and types \ t, T ::= x | \{x:A\} => t | t e
| (x:A) \rightarrow T
| Type
| rec f(x:T_1). t_2
| conv t at \hat{T}
| \{x:A\} => t | t [e]
| [x:A] \rightarrow T
| case t \{y\} of C_1 \rightarrow t_1 \ldots C_n \rightarrow t_n
| T t_1 \ldots t_n | C
| abort T
| tcast t by p
| let x = p in t
| let x \{y\} = t_1 in t_2

Fig. 1 Sep^1 basic syntax.
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 polymorphic identity function:
\[
\{ a : \text{Type} \} \rightarrow \{ (x : a) \rightarrow x \} \text{Nat} \ Z
\]

Similarly, applications to compile-time arguments are marked with square brackets, as in the \text{Nat} type argument to the identity function:
\[
\Gamma \vdash p : P
\]

Proof \( p \) shows proposition \( P \)
\[
\Gamma \vdash t : T
\]

Term \( t \) has type \( T \)
\[
\Gamma \vdash \text{val} t
\]

Term \( t \) has a value
\[
t_1 \leadsto t_2
\]

Term \( t_1 \) reduces to \( t_2 \) in one step
\[
t_1 \leadsto^m t_2
\]

Term \( t_1 \) reduces to \( t_2 \) within \( m \) steps

Fig. 2 Basic judgment forms.

unannotated programs \( u, U \) ::=  
\[
\begin{align*}
x \mid \{ x \rightarrow u \mid u_1 u_2 \\
(x : A) \rightarrow U \\
\text{Type} \\
\text{rec } f(x). \ u \\
T u_1 \ldots u_n | C \\
\text{case } u \{ y \} \rightarrow C_1 \rightarrow u_1 \ldots C_n \rightarrow u_n \\
\text{abort} \\
\text{tcast } u \\
\text{let } x = u_1 \text{ in } u_2
\end{align*}
\]

abstraction and application  
dependent function type  
type of types  
recursion  
datatype, data constructor  
case analysis  
failure  
termination cast  
local binding

unannotated program values \( v \) ::=  
\[
\begin{align*}
\{ x \rightarrow u \\
(x : A) \rightarrow U \\
\text{Type} \\
\text{rec } f(x). \ u \\
C v_1 \ldots v_n \\
T v_1 \ldots v_n \\
\text{tcast } u
\end{align*}
\]

abstraction  
dependent function type  
type of types  
recursion  
constructions  
datatype  
termination cast

Fig. 3 Sep³ unannotated syntax.

Compile-time applications and abstractions are erased before executing a program and when proving programs equal with \text{join}. Erasing compile-time elements allow proofs of equality to be constructed without reasoning about specificational data. For example, two constructions of the empty vector \text{VNil} (defined in the datatype section below) that differ only in the representations of the compile-time length parameter (e.g. \text{Z} vs. \text{plus Z Z}) can be proved equal without reasoning about equalities of addition since the length index is marked compile-time. This approach adapts ideas on erasure from several previous works [17], [21], [33].

Equality Formulas The Sep³ proof language includes a primitive formula representing the propositional equality of two programs, written \( t_1 = t_2 \). A proof of an equality is given by the expression \text{join } n where \( n \) is a meta-level natural number serving as an upper bound on the number of reductions steps the type checker will use when attempting to decide joinability of the respective programs. The typing rule for \text{join} is shown as TJoin in Figure 5. In this rule and throughout the sequel we denote \( \alpha \)-equivalence between programs \( t_1 \) and \( t_2 \) as \( t_1 \equiv t_2 \). TJoin makes use of the eraser function (Figure 4). If a program in an equality proved by
join reduces to a normal form in fewer steps than the bound given as an argument to join, then we compare that normal form (modulo \( a \)-equivalence) with the program resulting from reducing the other side of the equality in a similar manner. Equalities can be proved between programs that are non-terminating, provided that those programs are joinable in a number of steps less than or equal to the given bounds.

Conversion Sep\(^3\) programs and proofs can utilize equalities to change the type of a given proof or program. The typing rule for conversion is shown as rules TConvProof and TConvProg in Figures 5 and 6, respectively.

Syntactically, a conversion has the form \( \text{conv } t \at \hat{T} \) (for programs) or \( \text{conv } p \at P \) (for proofs). This form casts the type (or proposition) of the program (or proposition) to the left of the \( \at \) to the type indicated by the conversion context to the right of the \( \at \) keyword. A conversion context has the syntactic form of the underlying syntactic category (\( \Gamma \) or \( P \)), extended with a special escape form. The escape form \( \hat{\ldots} \) identifies locations where the type should be changed using a proof of an equality \( t_1 = t_2 \). We define functions \( \rightarrow \) and \( \hat{\rightarrow} \) that replace an escape occurring in a conversion context with the left (respectively, right) hand side of the equation proved by the escaped proof.

\[
\begin{align*}
|x| &= x \\
|C| &= c \\
|\lambda (x:T) \rightarrow t| &= \lambda x \rightarrow |t| \\
|\lambda [x:A] \rightarrow t| &= |t| \\
|\lambda [x:P] \rightarrow t| &= |t| \\
|\text{rec } f(x:t). t| &= \text{rec } f(x). |t| \\
|t_1 \vdash t_2| &= |t_1| \rightarrow |t_2| \\
|t_1 \rightarrow t_2| &= |t_1| \\
|t_1 \; p| &= |t_1| \\
|\text{conv } t \at T| &= |t| \\
|\text{case } t \{ y \} \rightarrow C_1 \rightarrow t_1 \ldots C_n \rightarrow t_n| &= \text{case } |t| \{ y \} \rightarrow |t_1| \ldots |t_n| \\
|\text{tcast } t \by p| &= \text{tcast } |t| \\
|\text{abort } t| &= \text{abort} \\
|\lambda (x:A) \rightarrow T| &= \lambda [x:A] \rightarrow |T| \\
|\lambda [x:A] \rightarrow T| &= \lambda [x:A] \rightarrow |T| \\
|\text{Type}| &= \text{Type} \\
|\text{let } x \{ y \} = t_1 \in t_2| &= \text{let } x = |t_1| \in |t_2| \\
|\text{let } x = p \in t| &= |t|
\end{align*}
\]

Fig. 4 Erasing annotated programs to unannotated programs.


The bound variable. The additional variable in the let binding provides the equation necessary to perform this cast.

**Termination Reasoning** Termination of programs is expressed with the termination formula, \( \text{val} \). A program \( t \) is total. Termination proofs can be constructed in two ways. First, programs which can be judged (see Section 3) to be values, can trivially be proved terminating using the \text{valax} (pronounced "value axiom") form. This includes programs which are syntactic values. Furthermore, termination proofs can be introduced using a termination case proof construct, \text{termcase}, that case-splits non-constructively on the termination behavior of a program (see Section 5). Finally, with \text{contraabort} we can combine clashing proofs of termination and divergence for the same program to reason using contradiction.

**Recursion and Induction** The Sep\(^3\) programming
language includes a \texttt{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 \texttt{ind} form for induction over programs. The \texttt{Tind} typing rule for 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 \texttt{ord} form. A structural ordering statement of this form can only be proved between programs \( t_1 \) and \( t_2 \) when both are terminating and \( t_1 \) is an application of a constructor to arguments including \( t_2 \). This ensures that the value of \( t_1 \) is structurally larger than the value of \( t_2 \), so the induction is well founded. An \texttt{ordtrans} expresses transitivity of the structural ordering, thereby allowing \texttt{ind} to be used for course-of-values induction. The typing rules for \texttt{ind}, \texttt{ord}, and \texttt{ordtrans} are shown in Figure 5, and the \texttt{TRec} typing rule for \texttt{rec} is shown in Figure 6.

One could certainly imagine strengthening this ordering, but it is worth recalling that in the presence of higher types, it is already quite powerful. For example, terminating recursion based on a lexicographic combination of the natural number ordering with itself is subsumed by natural-number recursion at higher-type. So one can easily prove that Ackermann’s function, for ex-
Dependently typed languages typically include the ability to define indexed type constructors, where the index may vary in the result type of each constructor. Sep\(^3\) data types only support parameters to types, where the range of every constructor must be the same. Indexed types can be simulated, however, by having constructors for the datatype accept proofs (as additional arguments) of equations that constrain the type constructor parameters appropriately. These proofs are then used to refine the type of the data constructor when case-splitting on values of the datatype.

The encoding of the archetypal indexed data type of length-indexed vectors is shown below. This defines a polymorphic data type that carries its length as a type parameter. The type of \(\text{VNil}\) requires a proof that the length \(n\) is equal to \(Z\). Similarly, the type of \(\text{VCons}\) constructor takes an argument \(m\) representing the length of the tail of the vector, as well as a proof that \(m\) is the successor of \(n\).

\[
\text{data } \text{Vec} : [a : \text{Type}] \to [n : \text{Nat}] \to \text{Type} \\
\text{VNil} : [q : n = Z] \to \text{Vec } a \ n \\
\text{VCons} : [m : \text{Nat}] \to [q : n = S \ m] \to \\
(a : a) \to (xs : \text{Vec } a \ m) \to \text{Vec } a \ n
\]

The elaboration of type constructors to the core language has a subtle interaction with erasure. The elaboration of Vec results in a constant \(\text{Vec} : (a : \text{Type}) \to (n : \text{Nat}) \to \text{Type}\). The compile-time arguments \([a : \text{Type}]\) and \([n : \text{Nat}]\) become run-time arguments for the type constructor Vec. Without this restriction, one can construct examples violating type soundness, where we use a (provable) equation like \(\text{Vec } [\text{bool}] [n] = \text{Vec } [\text{nat}] [n]\) for unsoundly casting a vector of booleans to a vector of naturals. One can then extract the head of the vector at the wrong type. Making these parameters run-time arguments Vec avoids this problem. On the other hand, we can respect the stage (compile-time or run-time) of these parameters when we add them as inputs to the constructors for the datatype. So the \(\text{VNil}\) constructor elaborates to a constant with type \([a : \text{Type}] \to [n : \text{Nat}] \to [q : n = Z] \to \text{Vec } a \ n\). The length index \(n\) is marked as compile-time, and will be erased in applications of \(\text{VNil}\).

**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 are interested in allowing non-termination not necessarily because we want to define partial computations, but rather because...
general recursion is often the most straightforward way to define functions of interest, regardless of whether they terminate. Languages that require termination of all functions in contrast take the approach of encoding non-termination indirectly. In the Sep\textsuperscript{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\textsuperscript{3} design offers the advantage of incrementality.

3 A value judgment

In Sep\textsuperscript{3}, we syntactically classify some programs as values, as is usual when defining a language. Typically, in a call-by-value language like Sep\textsuperscript{3} variables are identified as values, since the operational semantics of the language dictates that when reducing an application the argument must be reduced to a value to get call-by-value β-reduction that can be reduced. Hence, inside the body of the abstraction being applied, the bound variable can be assumed to range over values, since it will only be instantiated by values.

However, in Sep\textsuperscript{3} we must distinguish between abstraction in the program fragment and in the proof fragment: variables introduced by λ-abstraction in programs will only be instantiated with values, while λ-abstraction (that is, quantification) of programs in the proof language 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. 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\textsuperscript{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 the programs those theorems range over to values. For example, join m : plus x Z = x regardless of whether x terminates.

\[
\frac{x : A \text{val} \in \Gamma}{\Gamma \vdash \text{val } x} \quad \text{ValVar}
\]

\[
\frac{\Gamma \vdash \text{val} \, \lambda \, x : A \to t}{\Gamma \vdash \text{val} \, \lambda \, x : A \to t} \quad \text{ValLam}
\]

\[
\frac{\forall \, i. \, \Gamma \vdash \text{val } t_i}{\Gamma \vdash \text{val } C \, t_0 \ldots t_n} \quad \text{ValCons}
\]

\[
\frac{\Gamma \vdash \text{val } \text{tcast } t \text{ by } p}{\Gamma \vdash \text{val } \text{tcast } t \text{ by } p} \quad \text{ValTCast}
\]

Fig. 9 Selected value judgment rules.

nates. If \( x \) diverges then both sides of the equality diverge, and are hence still equal.

For Sep\textsuperscript{3}, we modify the definition of the set of programs which are values. In addition to a simple syntactic definition, we utilize a judgmental definition of value, allowing the context to be used when determining whether a term is a value. Later, we will identify a class of syntactic values to be used only by reduction. In the case of abstraction in the proof fragment (Figure 5, rule TLamProof), the variable is added to the context without a value annotation. Conversely, in the program fragment (Figure 6, rule TLamProg) the variable is added with an additional val annotation on the typing assumption.

The value judgment for programs (Figure 9) 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 (Figure 5). For example, we can prove S Z terminates (represented by the proposition \( p : S \ Z \)!) because S Z is judgmentally a value. More interestingly, if we have \( x : \text{Nat}^{\text{val}} \) in the typing context 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 nor-
The type annotation is necessary because the typechecker implementation uses a bidirectional algorithm that combines type checking with type synthesis.

Theorem id_is_id :
\[ \text{Program id : } [a: Type] (x:a) \rightarrow a := x \]

join 100

This proof fails because the two programs are not joinable. The reduction sequence below shows the problem.

\[
\begin{align*}
\text{id } [a] x & \quad \text{(by erasure)} \\
\text{id } x & \quad \text{(by def. of id)} \\
(\forall y.y) x & \quad \not\rightarrow \\
\end{align*}
\]

The \(\beta\)-redex \((\forall y.y) x\) 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. The equality proved by join is too fine, because it can only equate programs 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 \(\beta\)-redex \((\forall y.y) x\) would step, because it is not necessary to reduce the argument to an application to a value prior to \(\beta\)-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 \((\forall x:Nat.Z)\) loop, where loop stands for any diverging computation. Inside a proof, using a call-by-name semantics we can prove \((\forall x:Nat.Z)\) 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 \(\beta\)-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 TTCast typing rule for tcast is shown in Figure 6.

To allow reasoning over expressions including tcast constructs, 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
Theorem id_is_id \_term :
forall (a:Type)(x:a)(x_term:x!).
(id [a] x = x) :=
join 100 :
(id [a] (tcast x by x\_term) = x)

In Section 3 we introduced a value judgment to dif-
ferentiate between variables introduced by proof ab-
stractions from those introduced by program abstrac-
tions. Using the value judgment, it is possible to rede-
fine the \(\beta\)-rule to use the value judgment.

\[
\Gamma \vdash \text{val} u'
\]
\[
\Gamma \vdash E((\lambda x.u)\ u') \leadsto E[u'/x]u
\]

This rule allows reduction to reuse the value judg-
ment, but it comes at the cost of complicating the re-
duction relation. No longer is reduction defined syntac-
tically, but now it is a contextual relation. The alternate-
tive approach, and the one we take in Sep\(^3\), is to use the syn-
tactic view of evaluation. We identify a syntactic class of values for reduction purposes that contains all of the syntactic forms which are trivially judged to be values by the value judgment, represented by the pro-
duction \(\nu\) 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 termi-
nation cast. For example, if a variable \(x\) is judged a value (but would not be syntactically classified a value), then \(\text{tcast} x\) by \(\text{valax x}\) is the associated syntactic value. In this way, we simplify the reduction relation to use standard call-by-value \(\beta\)-reduction over a class of syntactic values including \(\text{tcast}\) constructs, while still using the value judgment for typing.

When constructing a proof of equality using \(\text{join}\), the programs being equated are erased, as described previously. However, the \(\text{tcast}\) constructs are pre-
served (the termination proof is dropped, as it has no computational content). Additionally, reduction al-
 lows a \(\text{tcast}\) to be dropped when the expression be-
ing cast is a syntactic value. This prevents \(\text{tcast}\) from blocking a redex, as would be the case with \(\lambda\ x.\ \text{tcast} \ \lambda\ x.\ p\ \nu\).

Preserving termination casts during \(\text{join}\) reduction is necessary to allow \(\text{join}\) to construct equalities be-
tween programs involving expressions that do not nor-
malize to values. After programs are reduced with \(\text{join}\), they are compared modulo termination casts and renaming of bound variables.

Preserving termination casts when reasoning about programs using \(\text{join}\) may cause a term to take more reduction steps inside proofs, because the \(\text{EBeta}\) re-
duction step may duplicate \(\text{tcast}\) constructs 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.

5 Termination case

Sep\(^3\) includes a termcase (Figure 5) construct that allows a proof to case-split on whether a scruti-
nized program terminates or diverges, with \(y\) bound as a proof of the corresponding termination assump-
tion in each branch. In the terminates \(()\) branch, \(y\) is a proof that the scrutinee terminates. In the di-
 verges \(\text{(abort)}\) branch, \(y\) is a proof that the scrutinee is equal to abort, signifying divergence. The typing rule TAabort, 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 EAabort. If abort appears in evaluation position, then the term immediately steps to abort. This means that all prov-
ably diverging programs (identified by equivalence to abort) are contextually equivalent.

The termcase construct is non-constructive, as it axiomatizes excluded middle for termination, an unde-
cidable property. This relies on an oracle that can de-
termine 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 frag-
ment, relaxed to partial correctness: if \(\vdash t : T\), then ei-
 ther \(\nu \leadsto t\nu\) (where \(\nu\) is a syntactic value) and \(\nu : T\), or else \(\nu\) diverges. This has already been observed in Wright and Felleisen’s classic paper[36].

Using termcase we can strengthen proofs of alge-
bric theorems, which are subject to termination pre-
conditions, to stronger theorems that do not require ter-
pinuation preconditions. As a simple example, we can use the proof of id\_is\_id from Section 4. To prove the theorem above, it is necessary to have a proof of termination available to \(\text{tcast}\) the variable \(x\), so that the \(\text{join}\) proof can succeed. Nevertheless, the theorem is valid for all inputs, irrespective of termina-
tion behavior. Using termcase, this theorem can be strengthened.

Theorem id\_is\_id :
forall (a:Type)(x:a).id [a] x = x :=
termcase x \{x\_term\} of
\text{abort} \rightarrow
let u1 = join 100
: id [a] (\text{abort} a) = (\text{abort} a)
in conv u1
at (id [a] \text{\_x\_term} = \text{\_x\_term})
| ! -> id_is_id_term a x x_term |

In the abort branch for the id_is_id example above, we do a conversion where we change a proof of (id [a] (abort a)) = (abort a) to a proof of (id [a] \(x\)) = \(x\) using a conversion context (id [a] \(~x\)_term) = \(~x\)_term.

This shows how we can do multiple conversion steps at one time, by including more than one splice, although in this simple example both splices refer to the proof \(x\)_term : (abort a) = x.

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.

6 Implementation and experience

To gauge the feasibility of the Sep3 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 Sep3 and the implementation have continued to evolve in tandem. The implementation is available online.51 The examples in this paper can be found in Tests/unittests/PiI.sep.

Sep3 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.

6.1 Proof automation

Sep3 programs rely heavily on the use of join to construct equations between programs, and the use of conv to change the type of a term or a proof. These tools are often irritatingly precise, as they provide a very low-level interface for proof. Quite often, constructing a proof in this language consists of the following.

Theorem refl :
forall (a:Type)(t:a).t = t :=
join 0

Theorem sym :
forall (a:Type)(b:Type)(t1:a)(t2:b)
(p:t1 = t2).(t2 = t1) :=
conv (refl a t1) at \(~p\) = t1

Theorem trans :
forall (a:Type)(b:Type)(c:Type)
(t1:a)(t2:b)(t3:c)
(p:t1 = t2)(q:t2 = t3).(t1 = t3) :=
conv p at t1 = \(~q\)

Fig. 10 Proofs that = is an equivalence.

1. Use join to prove a number of equations between programs.

2. Finesse the equations into just the right form.

3. Use conv, in concert with the generated equations, to cast the type of a term or the formula of a proof.

In a number of instances, we can provide simple automated support for common proof tasks. Below, we describe a collection of such automated tactics, starting with steps 2 and 3 from above, which are quite simple, and finally addressing the first step, which is considerably more complicated. The constructs for automated proof support will be used extensively in the examples in the remainder of this section and the next.

First, the primitive propositional formula judging two programs t1 and t2 to be joinable under call-by-value reduction, t1 = t2, is an equivalence relation. Moreover we can prove this within the language, as shown in Figure 10.

Undoubtedly, these theorems are simple enough to prove, but when it comes to using these lemmas, the story is quite different. The equality of Sep3 is heterogeneous, as the types of the two sides of an equation can be different. Consequently, in the proofs above we are required to take, as type parameters, the types of the equated programs. In the absence of general type inference support in Sep3, simply supplying those type arguments leads to very verbose programs. Because reasoning in Sep3 depends so heavily on manipulation of equality proofs, the prototype implementation provides primitive constructs trans and sym for transitivity and symmetry respectively that infer the necessary type arguments.

Even with primitives for reasoning about symmetry and transitivity, verification often requires a large...
number of tedious steps that amount to applications of these operators. As an additional automation capability, the prototype implementation provides an `equiv` form that eliminates the need to manually perform this reasoning. Operationally, `equiv` collects all of the (finite number of) equations present in the current typing context and takes the symmetric-transitive closure of them. Moreover, we associate with each equation in the closure the proof term built from applications of `sym` and `trans`. When checking a proof term `equiv` against a particular formula `t1 = t2`, the typechecker simply looks for the expected formula in the transitive-symmetric closure and, if it is found, checks the associated proof term, or alternatively raises an error if the formula is not found.

Using `equiv` eliminates a great deal of the tedium of working with equations in proof. After proving just the right equalities, the next step is often to use those equations to cast the type of a term or a proof. And in Sep3, to do a cast requires the programmer to supply a `conv` term with a conversion context that identifies the proper places for those equations to be used.

The Sep3 implementation provides an `autoconv` tactic that constructs a conversion context syntactically. The tactic takes a term `t` and an expected type – the intended type of the result of the cast – `T`, and calculates a term `conv t` at `C`, for some conversion context `C`. The `autoconv` tactic first synthesizes the actual type `S` of `t`, and then does a comparison between the actual type `S` and the intended type `T`. We use the notation `S[p]` to represent the subterm of `S` occurring at position `p`, for some subterm position `p` of `S` (and similarly for `T` and `C`). If `S[p]` and `T[p]` differ, we check to see if there is an equation in the context `C` such that `C[p] = T[p]`. If so, then we set `C[p]` to be `True` (the escape of `h`). If `S[p]` and `T[p]` are the same, then we set `C[p]` to be `S[p]`. As with the `equiv` tactic, we do not need to trust the implementation of `autoconv` as the resulting conversion proof `conv t` at `C` is subsequently checked using the core typing rules.

The `equiv` and `autoconv` tactics compact the second and third bookkeeping steps of proofs identified above, but they do nothing to address the issue of constructing initial set of equations using `join`. To illustrate the usability challenges with reasoning directly with `join`, consider the proof of the property `and_commutes`, capturing that the boolean `&&` function commutes. This is proved in Figure 11.

In each case split on `x` and `y`, we get proofs `x_eq` and `y_eq` that respectively prove formulas `p = x` and `p = y` in each case branch, where `p` is the pattern (either `true` or `false` in this example) associated with the branch. Since the only tool we have to generate equalities between programs is normalization of open programs using `join`, the programmer is forced to prove a simpler equation using the patterns, and then insert a conversion using the proofs of equalities introduced by the case split. To simplify matters slightly, though, the `false` branch for `x` uses `autoconv` to generate the conversion context automatically.

In practice, using `join` to prove equations is even more troubling, as the desired reduction may depend on some intermediate term that we have an equation for, yet the intermediate term is not exposed in the top level formula. For example, suppose the formula to be proven is `t1 = t2`, and we have a proof `p : e = C` in the context, where `e` is an arbitrary expression and `C` a nullary constructor. For the sake of the example, assume that `t1` reduces to some intermediate term `case e {e_eq} of C -> t2`. Using `join`, we would first prove that `t1` equals `case e {e_eq} of C -> t2`, and then conv with the equation to get that this equals `case C {e_eq} of C -> t2`. Finally, applying `join` again, we can get that the latter term equals `t2`. The issue here is that since `e` is some arbitrary (and perhaps reducible) expression, we are required to fiddle with the bound on reduction steps passed to `join` to ensure that we don’t reduce `e` in the process of reducing `t1`. For if we reduce `e` to some `e'`, our equation `e = C` can no longer be applied.

The difficulty with using `join` in this way is that it is not aware of equations that are available in the typing context, so it forces the programmer to manually perform the task of rewriting intermediate programs using `conv`. A partial solution to this problem is to make the implementation of the reduction semantics aware of equations, and to perform the rewriting automatically. The Sep3 implementation provides a tactic, called `morejoin` to do precisely this. In our experience, this tactic has proved invaluable for constructing equations feasible for programs of any reasonable size. For example, in the proof `morejoin_and_commutes` in Figure 11, all of the cases are proven directly with the same invocation of `morejoin`.

The interface for `morejoin` is quite simple – it simply takes a list of proofs and the expected equation to be proved, and invokes the evaluator in the same way as `join`. On the other hand, the implementation is quite more involved, as it requires instrumenting the evaluator to perform rewrites[6]. The proofs supplied must either prove equations, which are treated as left-to-right oriented rewrite rules, or else termination proofs, which are used to automatically insert termination casts.

When reducing a term, if an application `v1 t2` is encountered where `v1` is a value, and the list of proofs

---

[6] In principle, the evaluator should also keep a trace of rewrites used, so that a proper proof term with the required `conv` can be reconstructed, although the current implementation simply trusts the instrumented evaluator.
Program ($\land\land$) : $(x{:}\text{Bool})(y{:}\text{Bool})$ -> $\text{Bool}$ :=

```plaintext
case x {x_eq} of
  True -> y
| False -> False
```

Theorem and_commutes :
forall(x{:}\text{Bool})(y{:}\text{Bool})(x_term{:}x!)(y_term{:}y!).
($x \land\land y$) = ($y \land\land x$) :=
case x {x_eq} x_term of
  True ->
    (case y {y_eq} y_term of
      True ->
        let u1 = join 100 : (True \& True) = (True \& True)
in conv u1 at (~x_eq \& ~y_eq) = (~y_eq \& ~x_eq)
      | False ->
        let u1 = join 100 : (True \& False) = (False \& True)
in conv u1 at (~x_eq \& ~y_eq) = (~y_eq \& ~x_eq)
    | False ->
      (case y {y_eq} y_term of
        True ->
          let u1 = join 100 : (False \& True) = (True \& False)
in autoconv u1 -- conv u1 at (~x_eq \& ~y_eq) = (~y_eq \& ~x_eq)
        | False ->
          let u1 = join 100 : (False \& False) = (False \& False)
in autoconv u1) -- conv u1 at (~x_eq \& ~y_eq) = (~y_eq \& ~x_eq)

Theorem morejoin_and_commutes :
forall(x{:}\text{Bool})(y{:}\text{Bool})(x_term{:}x!)(y_term{:}y!).
($x \land\land y$) = ($y \land\land x$) :=
case x {x_eq} x_term of
  True ->
    (case y {y_eq} y_term of
      True -> morejoin {sym x_eq,sym y_eq,x_term,y_term}
      | False -> morejoin {sym x_eq,sym y_eq,x_term,y_term}
    | False -> morejoin {sym x_eq,sym y_eq,x_term,y_term}

Fig. 11  Conjunction is commutative.

supplied to morejoin includes a proof of $p:t_2!$, then a termination cast is inserted around $t_2$, and $v_1$ (tcast $t_2$) is a $\beta$-redex. Furthermore, if the term to be reduced is a case expression $\text{case } t_1 \text{ of } \{\ldots\}$, and the list of proofs supplied to morejoin includes $p : t_1 = t_2$, then the term is rewritten to $\text{case } t_2 \text{ of } \{\ldots\}$, and reduction proceeds.

Combining inserting termination casts and rewriting is quite useful, as oftentimes we will want to prove some formula $f$ of $t_1 = t_2$ where $f$ does an immediate case-split on its argument, and we have proofs of $t_1!$ and $t_1 = v_1$. Intuitively, these proofs imply that the value of $t_1$ is $v_1$, so we would like to reduce $f$ to $t_1$ and then reduce the case-split on $v_1$. But in general $t_1$ need not actually reduce to $v_1$ (or indeed any value), and so we need to insert a tcast around $t_1$ to enable the first reduction (of $f$ to $t_1$). Inserting this tcast around $t_1$ prevents further reduction within $t_1$ (since tcast $E$ is not included as a form of evaluation context in Figure 7). This means that morejoin will reduce $f$ to $t_1$ to a case-split on tcast $t_1$, and then substitute $v_1$ for $t_1$. The TCast rule (of Figure 7) will then reduce this term to a case-split just on $v_1$, which can finally reduce using the CaseTerm rule.

Inserting termination casts and rewriting case scrutinees can be justified quite directly.
If \( t \sim^m v_1 t_2 \), and \( p : t_2 \), then \( \text{join} \ m : t = v_1 t_2 \). If \( \text{join} \ 0 : v_1 t_2 = v_1 (\text{tcast} t_2 \ by \ p) \) using transitivity we can prove \( t = v_1 (\text{tcast} t_2 \ by \ p) \). Similarly, for case expressions, if \( t \sim^m \text{case} \ t_1 \ of \{ \ldots \} \) and \( p : t_1 = t_2 \), then \( \text{conv} \ (\text{join} \ m) \ at \ t = \text{case} \ p \ of \{ \ldots \} \). If case \( t_2 \) of \{ \ldots \} reduces further, we can produce a proof equating it with its contractum using \text{join}, and compose the proofs using \text{trans}.

The primitive \text{join} proof term provides a precise mechanism for controlling the number of steps used to reduce a term, which also serves as an upper bound on the number of reduction steps, ensuring that type checking \text{join} will terminate. On the other hand, it’s not clear how to count reduction steps when using rewriting, as a single rewrite may simulate some arbitrary (but finite) number of reduction steps. As a practical matter we simply set the upper bound of steps for \text{morejoin} to some arbitrary large constant; this has been sufficient in practice. On the other hand, the current implementation does not provide any accounting for number of rewrites. Given a poor choice of rewrite rules (for example, anything with a \text{refl} proof), rewriting may not terminate.

6.2 Example: Append is associative

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

These examples use additional notation from the Sep\(^3\) implementation to introduce top-level definitions. The Recursive form used by \text{append} elaborates to a \text{rec} construct directly. The body of the \text{rec} includes nested lambda-abstractions for each additional parameter to the Recursive definition.

The Inductive notation used with \text{append_term} is somewhat more complicated. In the example, we are performing induction on the argument \( ys \). However, the type of \( xs \) refers to the argument \( a \). In the Inductive notation, the definition will elaborate to a core language term where all of the arguments preceding the inductive argument will be introduced by lambda abstractions. The inductive argument is indicated by being followed by a name, in braces, that represents a proof that the inductive argument terminates. This inductive argument (and the termination argument) will be introduced with a core language \text{ind} form. All following arguments will be bound by lambda-abstractions in the body. In the \text{append_term} example, the proof term will desugar to the following.

\[
\begin{align*}
\forall (a : \text{Type} \Rightarrow \text{List} a) & \Rightarrow (\text{append_term}) \Rightarrow \text{ind} \ (xs : \text{List} a) \ [xs \text{ term}] . \\
\forall (ys : \text{List} \Rightarrow (ys \text{ term} : \text{ys} !) & \Rightarrow \ldots 
\end{align*}
\]

The proof of associativity is by induction on the structure of the list argument. Because 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 13. The proof proceeds by induction on \( xs \), so it requires a proof that \( xs \) is terminating.

In the syntax of Sep\(^3\), the presence of curly braces and lack of type ascription for the parameter \( xs \text{ term} \) indicates that the preceding argument \( xs \) is the induction variable. When applying \text{append_assoc_term} from outside the proof, the argument in \( xs \text{ term} \) position should be a proof of \( xs ! \). However, when we appeal to the induction hypothesis for a subterm \( xs' \) of \( xs \) within the body of \text{append_assoc_term}, we supply a proof that \( xs' < xs \), as shown in the \text{ih} binding in the proof.

Within the body of the proof, we prove equalities involving the variables \( ys \) and \( zs \). These variables are introduced by a proof abstraction, so they range over expressions. It is necessary to use termination casts, inserted by \text{morejoin} using the supplied termination proofs, to reduce programs involving these variables.

The proof uses an additional lemma (Figure 14) that proves \text{append} total on terminating inputs. This is a convenience, to simplify presentation, and could be avoided with additional reasoning using \text{termcase}. For more complex functions, where the proof of totality is not so straightforward, using \text{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 programs 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 \text{termcase} to consider the termination behavior of each argument in turn. In each \text{abort} branch, the \text{EAbort} rule allows us to join an application of \text{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 \( xs \text{ term}, ys \text{ term}, \) and \( zs \text{ term} \) that prove the associated arguments are terminating. With these proofs available, the weaker \text{append_assoc_term} lemma can be invoked.
data List : [a:Type] -> Type where
  Nil : List a
| Cons : (x:a) -> (xs:List a) -> List a

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

Fig. 12 List append.

Inductive append_assoc_term :
append [a] xs (append [a] ys zs) =
  append [a] (append [a] xs ys) zs :=
let term_xs_ys = append_term [a] xs xs_term ys ys_term;
term_ys_zs = append_term [a] ys ys_term zs zs_term
in case xs { xs_eq} xs_term of
  Nil ->
    let u1 = morejoin {sym xs_eq, ys_term, xs_term}
    : ys = append [a] xs ys;
    u2 = morejoin {sym xs_eq, xs_term}
    : append [a] xs (tcast (append [a] ys zs) by term_ys_zs)
    = append [a] ys zs;
    u3 = morejoin {sym xs_eq, ys_term, ys_term}
    : ys = append [a] xs ys
    in conv u2 at append [a] xs (append [a] ys zs) = append [a] ~u3 zs
| Cons x xs' ->
  let unroll_app = 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 = append_assoc_term [a] xs' (ord xs_eq) ys ys_term zs zs_term;
u1 = conv unroll_app at
  append [a] xs (append [a] ys zs) = Cons [a] x ~ih;
  u2 = morejoin {sym xs_eq, xs_term, ys_term}
  : append [a] xs ys = Cons [a] x (append [a] xs' ys);
  term_xs'_ys = append_term [a] xs' (value xs') ys ys_term;
u3 = 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 = morejoin {zs_term, ys_term, value 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 = trans u3 u4
  : append [a] (append [a] xs ys) zs
  = Cons [a] x (append [a] (append [a] xs' ys) zs)
in conv u1 at append [a] xs (append [a] ys zs) = ~(sym u5)

Fig. 13 Associativity of list append.

7 Example: Combinators
The append_assoc example of the previous section provides a small flavor of proving system properties within Sep3. While the proof does not rely on the append function terminating, it is easy to prove this inductively. Examples of programs that we cannot
prove total yet we wish to perform external verification upon abound.

Below, we describe the Sep³ solution to a problem posed as part of the 2012 VSTTE verified software competition. The problem involves proofs of properties of an interpreter for a SK combinator language. Because the SK calculus is Turing complete, any interpreter of terms over this language will necessarily be partial.

Following the VSTTE competition, a number of participants made their submitted solutions available. Two particular solutions [8], [27] were developed in verification environments based on normalizing programs. In the description of the Sep³ solution below, we use these solutions as points of comparison to the Sep³ approach, where we support external verification of non-terminating functions.

7.1 Problem definition

The description for the SK combinator problem is available from the VSTTE 2012 program verification competition website⁷). In the interests of making the description of the Sep³ solution below self-contained, we reproduce parts of the problem as necessary.

The SK combinator calculus consists of a simple term language defined as follows:

\[
\begin{align*}
t & := S \mid K \mid t \ t, \\
v & := K \mid S \mid K v \mid S v \mid S v v
\end{align*}
\]

where terms are made up of constants S and K, along with left-associative application. Values v are a subset of the language defined by the grammar:

\[
\begin{align*}
v & := K \mid S \mid K v \mid S v \mid S v v
\end{align*}
\]

Reduction is call-by-value. Terms can be decomposed to an evaluation context, given by the grammar:

\[
C = \Box \mid C \ t \mid v \ C.
\]

The operation \( C[t] \) produces a term by substituting \( \Box \) with \( t \) in the evaluation context \( C \). It is defined by the following set of equations:

\[
\begin{align*}
\Box[t] &= t, \\
(v \ C)[t] &= v \ C[t], \\
(C \ t)[t'] &= C[t'] \ t
\end{align*}
\]

Finally, we have a single-step reduction relation defined as follows:

\[
\begin{align*}
C[K \ t_1 \ t_2] & \to C[t_1], \\
C[S \ t_1 \ t_2 \ t_3] & \to C[(t_1 \ t_2) \ (t_1 \ t_3)]
\end{align*}
\]

We call the transitive-reflexive closure \( \to^* \) of \( \to \) the reduction relation. Finally, for a term \( t \), if there does not exist a term \( t' \) such that \( t \to t' \), then we write \( t \not\to \). The programming task is to define a function \( \text{reduction} \) that takes a term and returns a term \( t' \) such that \( t \to^* t' \). If \( t \) does not exist, then \( \text{reduction} \) diverges. The verification problem consists of three parts. In the first, we are required to prove \( \text{reduction} \) correctly implements its specification. In the second, we are to show that reduction on a term containing no S subterms always terminates. The third requires us to demonstrate a property of the reduction of the terms consisting of left-recursive nested applications of K.

In the remainder of this section, we will focus on the solution to the programming task and the verification of the second problem, as these serve to demonstrate the ability to perform external verification over potentially non-terminating programs in Sep³.

7.2 Sep³ solution

We begin by defining a datatype for terms, corresponding to the term grammar.

\[
data \text{Term} : \text{Type} \text{where}
S : \text{Term} \\
K : \text{Term}
\]

The operation \( C[t] \) produces a term by substituting \( \Box \) with \( t \) in the evaluation context \( C \). It is defined by the following set of equations:

\[
\begin{align*}
\Box[t] &= t, \\
(v \ C)[t] &= v \ C[t], \\
(C \ t)[t'] &= C[t'] \ t
\end{align*}
\]

Finally, we have a single-step reduction relation defined as follows:

\[
\begin{align*}
C[K \ t_1 \ t_2] & \to C[t_1], \\
C[S \ t_1 \ t_2 \ t_3] & \to C[(t_1 \ t_2) \ (t_1 \ t_3)]
\end{align*}
\]

We call the transitive-reflexive closure \( \to^* \) of \( \to \) the reduction relation. Finally, for a term \( t \), if there does not exist a term \( t' \) such that \( t \to t' \), then we write \( t \not\to \). The programming task is to define a function \( \text{reduction} \) that takes a term and returns a term \( t' \) such that \( t \to^* t' \). If \( t \) does not exist, then \( \text{reduction} \) diverges. The verification problem consists of three parts. In the first, we are required to prove \( \text{reduction} \) correctly implements its specification. In the second, we are to show that reduction on a term containing no S subterms always terminates. The third requires us to demonstrate a property of the reduction of the terms consisting of left-recursive nested applications of K.

In the remainder of this section, we will focus on the solution to the programming task and the verification of the second problem, as these serve to demonstrate the ability to perform external verification over potentially non-terminating programs in Sep³.
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 :=

  termcase xs {xs_term} of
   abort ->
    let aleft = join 100 :
      (append [a] (abort (List a)) (append [a] ys zs)) =
      (abort (List a));
    aright = join 100 :
      (abort (List a)) = (append [a] (append [a] (abort (List a)) ys) zs);
    ul = trans aleft aright
  in conv ul at
    append [a] ~xs_term (append [a] ys zs) =
    append [a] (append [a] ~xs_term ys) zs
  | ! ->

  termcase ys {ys_term} of
   abort ->
    let aleft = morejoin {xs_term} :
      (append [a] xs (append [a] (abort (List a)) zs)) =
      (abort (List a));
    aright = morejoin {xs_term} :
      (abort (List a)) =
      (append [a] (append [a] xs (abort (List a))) zs);
    ul = trans aleft aright
  in conv ul at
    append [a] xs (append [a] ~ys_term zs) =
    append [a] (append [a] xs ~ys_term) zs
  | ! ->

  termcase zs {zs_term} of
   abort ->
    let aleft = morejoin {xs_term,ys_term} :
      (append [a] xs (append [a] ys (abort (List a)))) =
      abort (List a);
    a_x_y_term = append_term a xs xs_term ys ys_term;
    aright = morejoin {xs_term,ys_term,a_x_y_term} :
      (abort (List a)) =
      (append [a] (append [a] xs ys) (abort (List a)));
    ul = trans aleft aright
  in conv ul at
    append [a] xs (append [a] ys ~zs_term) =
    append [a] (append [a] xs ys) ~zs_term
  | ! -> append_assoc_term [a] xs xs_term ys ys_term zs zs_term

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

Moreover, we define a program isValue, defined over terms. In normalizing dependent type theories, a common practice when defining a predicate is to define an inductive proposition representing the predicate. In Sep3, we cannot do this directly, as data types are exclusively programmatic, and may be inhabited by diverging programs. Therefore, a logical interpretation of an element of an inductive type is not valid. In Sep3, we define the isValue function directly as a recursive function over terms, returning a Bool.

The definition of isValue, along with a proof of termination (on terminating input) for isValue, is given in Figure 16. The proof of isValue_term closely mirrors the structure of the program isValue. This is to be expected, as the proof largely follows from equations constructed with join. A more sophisticated surface language may be able to derive such termination proofs for a subclass of recursive functions using syntactic methods such as those employed in tools like Coq and Agda.
Recursive isValue predicate

```coq
Recursive isValue : (t:Term) -> Bool :=
case t {t_eq} of
  K -> True
| S -> True
| App l r -> (case l {l_eq} of
  K -> isValue r
| S -> isValue r
| App l' r' -> (case l' {l'_eq} of
    S -> isValue r' && isValue r
    | K -> False
    | App a b -> False))
```

isValue terminates on terminating terms.
Note that this pretty much duplicates the code from isValue.

Inductive isValue_term : forall (t:Term){ t_term}.isValue t ! :=
case t {t_eq} t_term of
  K -> let u1 = morejoin { t_term , sym t_eq}
      : True = isValue t
      in conv valax True at "u1 !
  | S -> let u1 = morejoin { t_term , sym t_eq}
      : True = isValue t
      in conv valax True at "u1 !
  | App l r ->
case l {l_eq} (valax l) of
  K ->
    let u1 = isValue_term r (ord t_eq)
    : isValue r !;
    u2 = morejoin {sym t_eq , sym l_eq, t_term, valax r}
    : isValue r = isValue t
    in conv u1 at "u2 !
  | S -> let u1 = isValue_term r (ord t_eq)
      : isValue r !;
    u2 = morejoin {sym t_eq , sym l_eq, t_term, valax r}
    : isValue r = isValue t
    in conv u1 at "u2 !
  | App l' r' ->
case l' {l'_eq} valax l' of
  S ->
    let ih_r = isValue_term r (ord t_eq : r < t);
    ih_r' = isValue_term r'
      : (ordtrans (ord l_eq : r' < l) (ord t_eq : l < t));
    u1 = and_term (isValue r') (isValue r) ih_r' ih_r;
    u2 = morejoin {sym t_eq,sym l_eq,sym l'_eq, t_term, valax l,valax l'}
    : ((isValue r' && isValue r) = isValue t)
    in conv u1 at "u2 !
  | K ->
    let u1 = morejoin {sym t_eq,sym l_eq,sym l'_eq, t_term, valax l,valax l'}
    : False = isValue t
    in conv valax False at "u1 !
  | App a b ->
    let u1 = morejoin {sym t_eq,sym l_eq,sym l'_eq, t_term, valax l,valax l'}
    : False = isValue t
    in conv valax False at "u1 !
```

Fig. 16 isValue.
Program isRedex : (t:Term) -> Bool :=
case t {t_eq} of
  K -> False
  S -> False
  App f1 t1 ->
    case f1 {f2_eq} of
      K -> False
      S -> False
      App f2 t2 ->
        case f2 {f2_eq} of
          K -> isValue t1 && isValue t2
          S -> False
          App f2 t3 ->
            case f2 {f2_eq} of
              K -> False
              S -> isValue t1 &&
                    isValue t2 &&
                    isValue t3
            | App f3 t4 -> False

Inductive isRedex_terminates :
  forall (t:Term){ t_term}. The signature
  data RedexK : (t1:Term) -> (t2:Term) ->
    [p1:isValue t1 = True] ->
    [p2:isValue t2 = True] ->
    RedexProp t
| RedexS :
  [p1:isValue t1 = True] ->
  [p2:isValue t2 = True] ->
  [p3:isValue t3 = True] ->
  RedexProp t

Fig. 17 isRedex.

Similarly to the definition of isValue, we define a
recursive function isRedex (Figure 17) which returns
a Bool. This function is used to determine if a term
is a reducible expression. As with isValue, we can
prove by induction that isRedex terminates on all termi-

nating inputs, although we elide the proof.

The formulation of the small-step reduction relation
in the problem description is expressed in terms of eval-

uation contexts. Rather than implement this directly,
we instead define a function step only on redexes in-
stead of all reducible terms. We then define a func-
tion decompose for identifying a redex and evalua-
tion context of a term, as well as a function plug per-
forming the substitution of a term for □ in an evalua-
tion context.

The step function is defined only on redexes, so it
takes a proof p:isRedex t = True. The signature for
step is

Program step :
  [t:Term][p:isRedex t = True] -> Term

Although the step function is parameterized over all
terms, the precondition isRedex t = True identifies a subset of those terms which actually are
redexes - that is, they have the forms
App (App K t1) t2 or
App (App (App S t1) t2) t3 for some
terms t1, t2, and t3 for which isValue returns
True.

Case-splitting on any term that is not of this form will
yield a contradiction, as we will be able to prove that
isRedex t = False. To prove this contradiction
directly in the definition of step requires us to effec-
tively inline the definition of isRedex in the body of
step. Branches where isRedex t = False can be
proved are unreachable, yet we would still have to
return some result, for example abort or some design-
nated Term. Whatever the choice, it may not be imme-
diately obvious to a programmer reading the code that
the case branch is unreachable.

Our solution to unreachable case branches draws
from experience with normalizing type theories that
include inductive propositions. We define a program-

matic type RedexProp t representing the inductive
proposition that a term is a redex. Moreover, the data
constructors for RedexProp carry exactly the infor-
mation about the shape of the index term t necessary
defining the step function.

Given a proof p:isRedex t = True, we can
define a recursive program redexProp that con-
structs a term of type RedexProp t. Moreover,
we can prove that redexProp is total on all inputs.
In effect, the combination of the redexProp and
redexPropTerm allow us to migrate programmatic
data to the proof language. We can soundly case-split
on a RedexProp resulting from a terminating applica-
tion of redexProp in the proof language if we know
that the RedexProp is a value.

A portion of the definitions of redexProp and
redexPropTerm are shown in Figure 18. Note that
in the definition of redexProp, we return abort
when a pattern match leads to a contradiction. The
associated termination proof makes this contradic-
tion apparent, using the contra primitive.

The definition of step, in turn, never directly
decomposes the input term t, but rather first con-
structs a RedexProp proposition and then case splits
on it. The associated step_terminates theorem shows
that step is total, directly appealing to the
Program redexProp : (t:Term)[p:isRedex t = True] -> RedexProp t :=
case t {t_eq} of
  K -> abort (RedexProp t) -- Contradiction, since isRedex t = True
| S -> abort (RedexProp t) -- Contradiction, since isRedex t = True
| App f1 t1 ->
case f1 {f1_eq} of
  K -> abort (RedexProp t) -- Contradiction, since isRedex t = True
| S -> abort (RedexProp t) -- Contradiction, since isRedex t = True
| App f2 t2 ->
case f2 {f2_eq} of
  K -> let [u1] = (conv sym t_eq at t = App ~(sym f1_eq) t1)
    : t = App (App f2 t2) t1;
  [u2] = (conv u1 at t = App (App ~(sym f2_eq) t2) t1)
    : t = App (App K t2) t1;
  [u3] = morejoin { u2, valax t2, valax t1}
    : isRedex t = (isValue t1 && isValue t2);
  [u4] = trans (sym p) u3
    : True = (isValue t1 && isValue t2)
in RedexK t t2 t1 [u2]
  [and_right (isValue t1) (isValue t2) (sym u4)]
| S -> abort (RedexProp t) -- Contradiction, as isRedex t = True
| App f3 t3 -> ...

Theorem redexPropTerm :
forall(t:Term)(p:isRedex t = True). redexProp t [p] ! :=
termcase t {t_term} of
  abort ->
  let isredex_t_aborts = aborts (isRedex ~t_term)
    : ((abort Bool) = (isRedex t));
  isredex_t_terminates = (conv valax True at ~(sym p) !)
    : (isRedex t) !
in contraabort isredex_t_aborts isredex_t_terminates
  [and_right (isValue t1) (isValue t2) (sym u4)]
| S -> ...
| App f1 t1 -> ...

Fig. 18 Redex inductive proposition.

redexPropTerm lemma rather than by inductively reasoning on the input term t.

Program step : (t:Term)[p:isRedex t = True] -> Term :=
case redexProp t [p] {redex_eq} of
  RedexK t1 t2 [isapp] [isval1] [isval2] -> t1
| RedexS t1 t2 t3 [isapp] [isval1] [isval2] [isval3]
  -> App (App t1 t2) (App t1 t3)

Theorem step_terminates :
forall (t:Term)(p:isRedex t = True).
  step t [p] !

The step function operates on redexes, but the small-step reduction relation is defined over reducible terms, including those containing a redex in a subterm. The Ctx type captures evaluation contexts and is defined by the following inductive data type.

data Ctx : Type where
  Box : Ctx
| C1 : Ctx -> Term -> Ctx
| C2 : (t:Term) -> [p:isValue t = True] -> Ctx -> Ctx
Equational reasoning about programs with general recursion and call-by-value semantics

```
data Dec : Type where
  Dec : (c:Ctx) -> (t:Term) -> Decomp

Recursive decompose :
  (t:Term) -> Decomp :=
  case isRedex t {redex_t} of
  True -> Dec Box t
  | False ->
    case t {e_eq} of
    K -> Dec Box t
    | S -> Dec Box t
    | App x y ->
      case isValue x {x_val} of
      True ->
        (case decompose y {y_eq} of
        Dec c' t' ->
        Dec (C2 x [sym x_val] c') t')
      | False ->
        (case decompose x {x_eq} of
        Dec c' t' ->
        Dec (C1 c' y) t')

Recursive plug :
  (c:Ctx)(t:Term) -> Term :=
  case c {c_eq} of
  Box -> t
  | C1 c' t' -> App (plug c' t) t'
  | C2 v [pf] c' -> App v (plug c' t)
```

Fig. 19 Decomposition and context plugging.

We relate terms to evaluation contexts by way of the decompose and plug functions (Figure 19). Since decompose must return a pair of Ctx and Term, we define a type Decomp to represent this pair.

The machinery for performing reduction is now in place – step relates a redex with its contractum. Using decompose separates a term into an evaluation context and possible redex. Composing step with decompose, followed by plug with the evaluation context and the contractum, gives us the small-step reduction relation →. Finally, recursively reducing the resulting term yields the transitive-reflexive closure of the small-step reduction relation. The reduction function, shown below, captures this process.

```
Recursive reduction :
  (t:Term) -> Term :=
  case decompose t {dec_t} of
  Dec c t' ->
    case isRedex t' {red_t'} of
    True ->
      reduction (plug c (step t' [sym red_t']))
    |False -> plug c t'
```

The definition of the reduction function follows the definition of the reduction relation →∗ faithfully. The primary deviation is the use of the decompose function to identify an evaluation context and possible redex; a relationship between terms and evaluation contexts that is left implicit in the specification of the plug function. Indeed, as part of the development of reduction, we proved the connection between decompose and plug:

```
Inductive plug_decompose_inv :
  forall (t:Term)(t_term:t!)
  (t':Term)(t'_term:t'!)
  (c:Ctx){c_term}
  (p:decompose t = Dec c t').
  plug c t' = t
```

Discussion The reduction function is intentionally defined to match the specification of the reduction relation. However, this particular implementation decision has its drawbacks, particularly in efficiency. In each small-step reduction, a term is decomposed to an evaluation context and redex, the redex is contracted, and then the contractum plugged back into the evaluation context. In the transitive closure of this implementation of the small-step reduction relation, the reduction will repeatedly plug a contractum into an evaluation context and then immediately in the recursive call decompose the result, undoing the work of the plug function. A more efficient implementation may eliminate this overhead by continuing reduction immediately upon producing a new redex when plugging the contractum into the context.

Unfortunately, the resulting reduction function differs substantially from the specification of the reduction relation, because it interleaves the plug function with reduction, effectively defining a large-step reduction semantics. This in turn requires additional verification effort to prove that such a large-step interpreter simulates the small-step interpreter.

It is illustrative to compare the Sep3 definition to solutions in other verification environments, graciously posted online following the competition[8],[27]. We examine two top-scoring solutions implemented using ACL2[13] and PVS[26]. Both ACL2 and PVS are language-based verification tools based on a logic of total functions. In particular, we see that the handling of non-termination is central to the definition of the reduction function.

The PVS solution defines a terminating reduce function implementing large-step reduction, which is later proven to simulate the small-step reduction relation. This function can be proved terminating in PVS by a measure function defined on the structure of the input term, much in the same way as the Sep3 _< _
formula is used in inductive proofs.

More interesting is the definition of the reduction function implementing the transitive reflexive closure of small-step reduction. First, \( \text{iter\_reduce} \) – a \textit{bounded} definition of reduction – is defined to take, in addition to a term to reduce, a maximum bound on the number of reduction steps required to yield a value. Next, \( \text{reduction} \) is defined by appealing to an oracle supplying an appropriate number of reduction steps.

\[
\text{iter\_reduce}(n)(R) : \text{RECURSIVE term} = \\
\begin{cases} 
\text{IF} \ n = 0 \\
\text{THEN} \ R \\
\text{ELSE} \\
\ (\text{LET} \ Q = \text{reduce}(R) \\
\text{IN} \ \text{IF} \ R = Q \\
\text{THEN} \ R \\
\text{ELSE} \ \text{iter\_reduce}(n - 1)(Q) \\
\text{ENDIF}) \\
\text{ENDIF}
\end{cases}
\]

\[
\text{MEASURE} \ n
\]

\[
\text{reducible?}(R) : \text{bool} = \\
(\text{EXISTS} \ n: \text{value?}(\text{iter\_reduce}(n)(R)))
\]

\[
\text{reduction}(R: (\text{reducible?})) : (\text{value?}) = \text{iter\_reduce}(\text{choose!} \ n: (\text{value?}(\text{iter\_reduce}(n)(R))))(R)
\]

This solution uses a predicate subtype \text{reducible?} to ensure that \text{iter\_reduce} is only defined on normalizing terms – those for which there exists an appropriate finite number of reduction steps to yield a value, so \text{reduction} is not defined on all combinator terms.

\( \text{Sep}^3 \) provides a similar oracle by virtue of the \texttt{termcase} construct, yet this is purely a proof language construct. Because proofs are erased from programs prior to execution, there is no need to implement such an oracle.

The ACL2 solution uses a similar approach, utilizing a \textit{steps-required} oracle to generate a bound \( n \) on the number of steps required to produce a value term. As commentary on the solution describes, this introduces an axiom that \textit{steps-required} always returns some value, but doesn’t indicate how to calculate that value. That is, some entity from ACL2’s untyped universe is returned, but it is not possible to reason about this entity in the ACL2 universe.

\[
\text{defchoose \ steps\_required}(\text{n})(x) = \\
\text{(b} * (((\text{mv terminates} \ & \ \\
\text{reduce\_n n x}))) \\
\text{terminates}))
\]

Next, the \text{reduction} function is defined. In contrast to the PVS solution, this definition is separated into two parts – a logical specification that uses the \textit{steps-required} oracle and an executable definition that simply recursively invokes the terminating small-step reduction function. The definition furthermore asserts that the specification and the implementation are equal. The separation between logical specification and executable implementation is similar to the proof/program separation in \( \text{Sep}^3 \).

The comparison between the \( \text{Sep}^3 \) implementation to the PVS and ACL2 solutions is intended to highlight the differences between language-based verification in a logic of total functions from \( \text{Sep}^3 \), which internalizes the notion of termination within the logic. This allows the language to relax the constraint that all programs terminate. ACL2 and PVS are both mature logics and tools, and the above is not intended to be an exhaustive comparison with \( \text{Sep}^3 \).

7.3 Reasoning about reduction

One of the verification exercises for the combinator problem involves proving termination of combinator terms which do not contain any \( S \) subterms. Intuitively, this is because any redexes in such a term will always produce a smaller term.

The \text{s\_free} predicate is defined recursively on the structure of the input term \( t \).

\[
\text{Recursive \ s\_free} : \ (t: \text{Term}) \rightarrow \text{Bool} = \\
\text{case} \ t \ {\text{[t\_eq]} \text{ of} \\
\text{K} \rightarrow \text{True} \\
\text{S} \rightarrow \text{False} \\
\text{App} \ t1 \ t2 \rightarrow \text{s\_free} \ t1 \ &\ & \text{s\_free} \ t2
\]

The main theorem is proved by induction on the structure of the term \( t \). We outline a sketch of the proof below. The internalized termination formula allows us to express the theorem \( \text{reduction} \ t \)! directly.

\[
\text{Inductive \ s\_free\_term} : \\
\text{forall} \ (t: \text{Term}):(t\_term) \ (p:s\_free \ t = \text{True}). (\text{reduction} \ t) \\
\]

The proof again follows the structure of the \text{reduction} function. We had to prove two main lemmas that the \texttt{decomp} and \texttt{plug} functions preserve \text{s\_free} of their results.

\[
\text{Inductive \ decomp\_preserves\_s\_free} : \\
\text{forall} \ (t: \text{Term}):(t\_term) \ (t' : \text{Term}):(t'_\ term:t') \\
(c: \text{Ctx}):(c\_term:c!) \\
(p:s\_free \ t = \text{True}) \\
(q: \text{decompose} \ t = \text{Dec} \ c \ t'). \\
(s\_free \ t' = \text{True}) := \\
\text{<omitted>}
\]

\[
\text{Inductive \ plug\_preserves\_s\_free} :
\]
The remainder of the proof is directly reasoning using computation and application of the inductive hypothesis.

8 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[7]. 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 [24], [34]. ACL2 also requires totality of functions [19]. 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 [5], [15], [16], [18]. 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 [4]). 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 undefined functions [35]. 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 [9 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 [6].

On the semantic side, several recent works are concerned with axiomatizing fixed-point operators (including call-by-value ones) that arise in various categorical structures [11], [31]. These works are focused on foundations, and propose general axioms applicable to a wide range of specific structures, including models of call-by-value computation with effects. In contrast, our interest is in applied reasoning in a specific pure call-by-value functional language. A major technical difference is that the notion of equality we have adopted here is not extensional, and hence would not (it seems) be able to validate the axioms proposed by, say, Hassegawa and Kakutani, which are expressed as equalities between denotationally equivalent higher-order terms [11].

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 [22], [23]. 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 Sep3, with a similar language design separating proofs and programs, and using termination casts with a judgmental notion of value [32], [33]. 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.

Somewhat less closely related are works based on the Edinburgh Logical Framework (LF) (also known as the ℒD₁ type theory) [10]. Systems like Beluga and Delphin add the ability to write functional programs oper-

---

8 This issue with the Guru implementation, discovered in the course of the current research on Sep3, remains to be repaired.
a dependently typed language requires a number of convergence, a capability dubbed “Freedom of Speech”. Sep3 includes a value judgment to differentiate the two. Depending on context, over values or expressions, so maintaining expressiveness. Variables can range, de- modifications to the logic to ensure soundness while possibly diverging programs without inheriting their di- notation

despite the syntactic distinction proofs can in the programming language from the proof language. Sep 3 supports internal and external verification, while not requiring a programmer to resort to indirect encodings to implement general recursive functions.

Sep3 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 possibly diverging programs without inheriting their divergence, 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, de- pending on context, over values or expressions, so Sep3 includes a value judgment to differentiate the two. Equality proofs are constructed by reducing open programs, 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 programs 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 Sep3 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. Sep3 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.

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

References


Equational reasoning about programs with general recursion and call-by-value semantics


Garrin KIMMELL
Garrin Kimmell is a researcher at the Kestrel Institute. Prior to joining Kestrel, he received his PhD from the University of Kansas and was a Postdoctoral Researcher at the University of Iowa.

Aaron STUMP
Aaron Stump is an Associate Professor of Computer Science at The University of Iowa. His primary research interests are in programming languages, verification, and term rewriting.

Harley EADES
Harley Eades is a PhD student at the University of Iowa. His research interests include programming languages, type theory, term rewriting, proof theory, and logic in computer science.

Peng FU
Peng Fu is a PhD student at the University of Iowa working on metatheory for type systems.

Tim SHEARD
Tim Sheard is a Professor at Portland State University. His research interests include functional programming, dependently typed languages, and meta-programming systems.

Stephanie WEIRICH
Stephanie Weirich is an Associate Professor at the University of Pennsylvania. She joined Penn after receiving her PhD from Cornell University in 2002. Her research concerns the type systems of functional programming languages, generic programming, dependent type systems and type inference.

Chris CASINGHINO
Chris Casinghino is a PhD student at the University of Pennsylvania. He studies programming languages, with an emphasis on type theory and logic. Currently, his research focuses on demonstrating that dependent types make new programming techniques possible and are a practical tool for every-day verification tasks.

Vilhelm SJÖBERG
Vilhelm Sjöberg PhD student at the Department of Computer and Information Science, University of Pennsylvania. His research interests are in programming languages, type systems and logic in particular dependent types.

Nathan COLLINS
Nathan Collins is a PhD student at Portland State University studying programming languages.

Ki YUNG AHN
Ki Yung Ahn is a Ph.D. student in Computer Science Department at Portland State University studying Programming Languages.