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

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

Aaron Stump
University of Iowa
astump@acm.org

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

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

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

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

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

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

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

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

ABSTRACT

Dependently typed programming languages provide a mechanism for integrating verification and programming by encoding invariants as types. Traditionally, dependently typed languages have been based on constructive type theories, where the connection between proofs and programs is based on the Curry-Howard correspondence. This connection comes at a price, however, as it is necessary for the languages to be normalizing to preserve logical soundness. Trellys is a call-by-value dependently typed programming language currently in development that is designed to integrate a type theory with unsound programming features, such as general recursion, `Type:Type`, and 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

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 [20, 4, 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 logics, 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[1], 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

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

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 Sep^3, one of several core language designs that we are developing in the Trellys project. The Sep^3 core language is a work-in-progress, and still under development. As a result it has not been subject to metatheoretic study and proofs of standard type system results (such as type soundness) are beyond the scope of this paper. In this regard, it is similar to works like that on ΠΣ, which explore a novel language design without conducting any substantial metatheoretic investigations [1]. 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 Sep^3 language definition has been mildly extended. More significant changes include:

- Section 6.1 describes a number of lightweight proof automation techniques we developed in the course of experimenting with Sep^3. Sep^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^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.[3]. Though the Sep^3 solution was not submitted to the competition, in this section we contrast the Sep^3 solution with published solutions of systems that did compete.

$\Gamma \vdash \mathtt{p} : \mathtt{P}$	Proof `p` shows proposition `P`
$\Gamma \vdash \mathtt{P} : \mathtt{Formula}$	`P` is a well-formed formula
$\Gamma \vdash \mathtt{t} : \mathtt{T}$	Term `t` has type `T`
$\Gamma \vdash \mathrm{val}\ \mathtt{t}$	Term `t` has a value
$\mathtt{t}_1 \rightsquigarrow \mathtt{t}_2$	Term $\mathtt{t}_1$ reduces to $\mathtt{t}_2$ in one step
$\mathtt{t}_1 \rightsquigarrow^{<m} \mathtt{t}_2$	Term $\mathtt{t}_1$ reduces to $\mathtt{t}_2$ within m steps

?!2 Basic judgment forms

The remainder of this paper is structured as follows. Section 2 provides a brief overview of a Sep^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^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^3 language is one core language that we are developing to explore the design space of dependently typed languages. This name is short for

[2] The bug in Coq, despite being 4.5 years old, still persists in the most recent version of Coq, Coq 8.4.

[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	$\hat{\texttt{P}}$	::=	`... P ...` | `~p`	
program conversion context	$\hat{\texttt{T}}$	::=	`... T ...` | `~p`	
typing contexts	Γ	::=	$\emptyset$ | Γ, `x:A` | Γ, $\texttt{x:A}^{\text{val}}$	
substitution	σ	::=	$[\texttt{e}_1/\texttt{x}]\texttt{e}_2$ | $[\texttt{e}_1/\texttt{x}_1, \ldots, \texttt{e}_i/\texttt{x}_i]\texttt{e}_t$	

proofs	`p`	::=	
	`x` | `\(x:A) => p` | `p e`		*abstraction and application*
|	`join n`		*equality axiom*
|	`valax t`		*value axiom*
|	`termcase t {y} of abort` $\rightarrow$ `p1` | `!` $\rightarrow$ `p2`		*termination case*
|	`ord p` | `ordtrans p1 p2`		*ordering axiom, transitivity*
|	`ind f(x:T) [q]. p`		*induction*
|	`case t {y} p of C1` $\rightarrow$ `p1 ... Cn` $\rightarrow$ `pn`		*case analysis*
|	`pack x,p`		*exists intro*
|	`unpack p as (x,p) in p`		*exists elim*
|	`conv p at` $\hat{\texttt{P}}$		*conversion*
|	`let x = p1 in p2`		*let-proof*
|	`let x {y} = t in p`		*let-prog*
|	`contra p1` | `contraabort p1 p2`		*contradiction*

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

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

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

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

?!3 Sep^3 unannotated syntax

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

?!4 Erasing annotated programs to unannotated programs.

$$\frac{\mathtt{x} : \mathtt{A} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{A}}\ \text{TVar} \qquad \frac{\Gamma \vdash \mathbf{val}\ \mathtt{t}}{\Gamma \vdash \mathtt{valax}\ \mathtt{t} : \mathtt{t}!}\ \text{TValAx} \qquad \frac{\Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{p} : \mathtt{P}}{\Gamma \vdash \backslash(\mathtt{x} : \mathtt{T}) \Rightarrow \mathtt{p} : \mathtt{forall}(\mathtt{x} : \mathtt{T}).\mathtt{P}}\ \text{TLamProof}$$

$$\frac{\Gamma \vdash \mathtt{p} : \mathtt{forall}(\mathtt{x} : \mathtt{A}).\mathtt{P} \quad \Gamma \vdash \mathtt{e} : \mathtt{A}}{\Gamma \vdash \mathtt{p}\ \mathtt{e} : [\mathtt{e}/\mathtt{x}]\mathtt{P}}\ \text{TAppProof}$$

$$\frac{\Gamma \vdash |\mathtt{t}_1| \rightsquigarrow^{<\mathtt{m}} \mathtt{t}_1' \quad \Gamma \vdash |\mathtt{t}_2| \rightsquigarrow^{<\mathtt{m}} \mathtt{t}_2' \quad \mathtt{t}_1' \equiv \mathtt{t}_2' \quad \Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \quad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{join}\ \mathtt{m} : \mathtt{t}_1\ =\ \mathtt{t}_2}\ \text{TJoin}$$

$$\frac{\Gamma, \mathtt{y} : \mathtt{t} = \mathtt{abort}\ \mathtt{t}' \vdash \mathtt{p_a} : \mathtt{P} \quad \Gamma, \mathtt{y} : \mathtt{t}! \vdash \mathtt{p_!} : \mathtt{P} \quad \Gamma \vdash \mathtt{t} : \mathtt{t}'}{\Gamma \vdash \mathtt{termcase}\ \mathtt{t}\ \{\mathtt{y}\}\ \mathtt{of}\ \mathtt{abort}\ \rightarrow\ \mathtt{p_a} \mid !\ \rightarrow \mathtt{p_!} : \mathtt{P}}\ \text{TTermCase} \qquad \frac{\Gamma \vdash p : (\Gamma \downarrow \hat{P})}{\Gamma \vdash \mathtt{conv}\ \mathtt{p}\ \mathtt{at}\ \hat{\mathtt{P}} : (\Gamma \downarrow \hat{\mathtt{P}})}\ \text{TConvProof}$$

$$\frac{\Gamma \vdash \mathbf{val}\ \mathtt{t}_1 \quad \Gamma \vdash \mathbf{val}\ \mathtt{t}_2 \quad \Gamma \vdash \mathtt{p} : \mathtt{t}_2\ =\mathtt{C_i}\ \mathtt{u}_0 \ldots \mathtt{t}_1 \ldots \mathtt{u}_n}{\Gamma \vdash \mathtt{ord}\ \mathtt{p} : \mathtt{t}_1 < \mathtt{t}_2}\ \text{TOrd}$$

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

$$\frac{\Gamma \vdash \mathtt{x} : \mathtt{T} \quad \Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{p} : \mathtt{P}}{\Gamma \vdash \mathtt{pack}\ \mathtt{x}, \mathtt{p} : \mathtt{exists}\ (\mathtt{x} : \mathtt{T}).\mathtt{P}}\ \text{TPack} \qquad \frac{\Gamma \vdash \mathtt{p}_1 : \mathtt{exists}\ (\mathtt{y} : \mathtt{T}).\mathtt{P}_2 \quad \Gamma, \mathtt{x} : \mathtt{T}, \mathtt{p}_2 : \mathtt{P}_2 \vdash \mathtt{p}_3 : \mathtt{P}_3}{\Gamma \vdash \mathtt{unpack}\ \mathtt{p}_1\ \mathtt{as}\ (\mathtt{x}, \mathtt{p}_2)\ \mathtt{in}\ \mathtt{p}_3 : \mathtt{P}_3}\ \text{TUnpack}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{t}' : \mathtt{C}'\,\overline{\mathtt{a}} \\ \Gamma \vdash \mathtt{p} : \mathtt{t}'! \\ \Gamma \vdash \mathtt{C}' : \overline{(\mathtt{y} : \mathtt{A}_1)} \rightarrow \mathtt{Type} \\ \mathtt{For\ each\ branch}\ \overline{\mathtt{C}_i\,\overline{\mathtt{w}} \rightarrow \mathtt{p}_i} \\ \quad \Gamma \vdash \mathtt{C}_i : \overline{(\mathtt{y} : \mathtt{A}_1)} \rightarrow \overline{(\mathtt{x} : \mathtt{A}_2)} \rightarrow \mathtt{C}'\,\overline{\mathtt{y}} \\ \quad \Gamma, \overline{\mathtt{w} : [\overline{\mathtt{a}}/\overline{\mathtt{y}}][\overline{\mathtt{w}}/\overline{\mathtt{x}}]\mathtt{A}_2}^{\mathbf{val}}, \mathtt{z} : \mathtt{t}' = \mathtt{C}\,\overline{\mathtt{a}}\,\overline{\mathtt{w}} \vdash \mathtt{p}_i : \mathtt{P}\end{array}}{\Gamma \vdash \mathtt{case}\ \mathtt{t}'\ \{\mathtt{z}\}\ \mathtt{p}\ \mathtt{of}\ \overline{\mathtt{C}_i\,\overline{\mathtt{w}} \rightarrow \mathtt{p}_i} : \mathtt{P}}\ \text{TCaseProof}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{P} : \mathtt{Formula} \\ \Gamma \vdash \mathtt{p_a} : \mathtt{t} = \mathtt{abort}\ \mathtt{T} \qquad \Gamma \vdash \mathtt{p_t} : \mathtt{t}!\end{array}}{\Gamma \vdash \mathtt{contrabort}\ \mathtt{p_a}\ \mathtt{p_t} : \mathtt{P}}\ \text{TContraAbort} \qquad \frac{\begin{array}{l}\Gamma \vdash \mathtt{P} : \mathtt{Formula} \\ \Gamma \vdash \mathtt{p_a} : \mathtt{t} = \mathtt{abort}\ \mathtt{T} \qquad \Gamma \vdash \mathtt{p_t} : \mathtt{t}!\end{array}}{\Gamma \vdash \mathtt{contrabort}\ \mathtt{p_a}\ \mathtt{p_t} : \mathtt{P}}\ \text{TContraAbort}$$

$$\frac{\Gamma, \mathtt{x} : \mathtt{P}_1 \vdash \mathtt{p}_2 : \mathtt{P}_2 \quad \Gamma \vdash \mathtt{p}_1 : \mathtt{P}_1}{\Gamma \vdash \mathtt{let}\ \mathtt{x}\ =\mathtt{p}_1\ \mathtt{in}\ \mathtt{p}_2 : \mathtt{P}_2}\ \text{TLetProofProof}$$

$$\frac{\Gamma, \mathtt{x} : \mathtt{T}, \mathtt{y} : \mathtt{x} = \mathtt{t} \vdash \mathtt{p} : \mathtt{P} \quad \Gamma \vdash \mathtt{t} : \mathtt{T} \quad \Gamma \vdash \mathtt{T} : \mathtt{Type}}{\Gamma \vdash \mathtt{let}\ \mathtt{x}\ \{\mathtt{y}\}\ =\mathtt{t}\ \mathtt{in}\ \mathtt{p} : \mathtt{P}}\ \text{TLetProofProg}$$

?!5 Proof typing rules

$$\frac{\mathtt{x} : \mathtt{A} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{A}}\ \text{TVar} \qquad \frac{\Gamma, \mathtt{x} : \mathtt{T}^{\mathbf{val}} \vdash \mathtt{t} : \mathtt{T}'}{\Gamma \vdash \backslash(\mathtt{x} : \mathtt{T}) \to \mathtt{t} : (\mathtt{x} : \mathtt{T}) \to \mathtt{T}'}\ \text{TLam} \qquad \frac{\Gamma \vdash \mathtt{t} : (\mathtt{x} : \mathtt{A}) \to \mathtt{T} \quad \Gamma \vdash \mathtt{e} : \mathtt{A}}{\Gamma \vdash \mathtt{t}\ \mathtt{e} : [\mathtt{e}/\mathtt{x}]\mathtt{T}}\ \text{TAppProg}$$

$$\frac{\mathtt{x} \notin |\mathtt{FV(t)}| \quad \Gamma, \mathtt{x} : \mathtt{A}^{\mathbf{val}} \vdash \mathtt{t} : \mathtt{T}'}{\Gamma \vdash \backslash[\mathtt{x} : \mathtt{A}] \to \mathtt{t} : [\mathtt{x} : \mathtt{A}] \to \mathtt{T}'}\ \text{TLamImp} \qquad \frac{\Gamma \vdash \mathtt{t} : [\mathtt{x} : \mathtt{A}] \to \mathtt{T} \quad \Gamma \vdash \mathtt{e} : \mathtt{A}}{\Gamma \vdash \mathtt{t}\ [\mathtt{e}] : [\mathtt{e}/\mathtt{x}]\mathtt{T}}\ \text{TAppImp}$$

$$\frac{\Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{t} : \mathtt{Type}}{\Gamma \vdash (\mathtt{x} : \mathtt{A}) \to \mathtt{t} : \mathtt{Type}}\ \text{TPi} \qquad \frac{}{\Gamma \vdash \mathtt{Type} : \mathtt{Type}}\ \text{TType} \qquad \frac{\Gamma, \mathtt{x} : \mathtt{T}_1, \mathtt{f} : (\mathtt{x} : \mathtt{T}_1) \to \mathtt{T}_2 \vdash \mathtt{t} : \mathtt{T}_2}{\Gamma \vdash \mathtt{rec\ f}\ (\mathtt{x} : \mathtt{T}_1).\mathtt{t} : (\mathtt{x} : \mathtt{T}_1) \to \mathtt{T}_2}\ \text{TRec}$$

$$\frac{\Gamma \vdash \mathtt{t} : (\Gamma \lfloor \hat{\mathtt{T}})}{\Gamma \vdash \mathtt{conv\ t\ at}\ \hat{\mathtt{T}} : (\Gamma \lfloor \hat{\mathtt{T}})}\ \text{TConvProg} \qquad \frac{\Gamma \vdash \mathtt{t}\ :\ \mathtt{Type}}{\Gamma \vdash \mathtt{abort\ t}\ :\ \mathtt{t}}\ \text{TAbort} \qquad \frac{\Gamma \vdash \mathtt{t} : \mathtt{T} \quad \Gamma \vdash \mathtt{p} : \mathtt{t}\ !}{\Gamma \vdash \mathtt{tcast\ t\ by\ p} : \mathtt{T}}\ \text{TTCast}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{t}' : \mathtt{C}'\,\overline{\mathtt{a}} \\ \Gamma \vdash \mathtt{C}' : \overline{(\mathtt{y} : \mathtt{A}_1)} \to \mathtt{Type} \\ \mathtt{For\ each\ branch}\ \overline{\mathtt{C}_i\,\overline{\mathtt{w}} \to \mathtt{t}_i} \\ \quad \Gamma \vdash \mathtt{C}_i : \overline{(\mathtt{y} : \mathtt{A}_1)} \to \overline{(\mathtt{x} : \mathtt{A}_2)} \to \mathtt{C}'\,\overline{\mathtt{y}} \\ \quad \Gamma, \overline{\mathtt{w} : [\overline{\mathtt{a}}/\overline{\mathtt{y}}][\overline{\mathtt{w}}/\overline{\mathtt{x}}]\mathtt{A}_2}^{\mathbf{val}}, \mathtt{z} : \mathtt{t}' = \mathtt{C}\,\overline{\mathtt{a}}\,\overline{\mathtt{w}} \vdash \mathtt{t}_i : \mathtt{t}\end{array}}{\Gamma \vdash \mathtt{case\ t}'\ \{\mathtt{z}\}\ \mathtt{of}\ \overline{\mathtt{C_i}\,\overline{\mathtt{w}} \to \mathtt{t_i}} : \mathtt{t}}\ \text{TCaseProg} \qquad \frac{\Gamma, \mathtt{x} : \mathtt{P} \vdash \mathtt{t} : \mathtt{T} \quad \Gamma \vdash \mathtt{p} : \mathtt{P}}{\Gamma \vdash \mathtt{let\ x\ =\ p\ in\ t} : \mathtt{T}}\ \text{TLetProgProof}$$

$$\frac{\Gamma, \mathtt{x} : \mathtt{T}_1, \mathtt{y} : \mathtt{x} = \mathtt{t}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2 \quad \Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \quad \Gamma \vdash \mathtt{T} : \mathtt{Type}}{\Gamma \vdash \mathtt{let\ x}\ \{\mathtt{y}\}\ =\ \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2}\ \text{TLetProgProg}$$

?!6 Program typing rules

evaluation contexts

$E ::= \square \mid E\ \mathtt{u} \mid \mathtt{v}\ E \mid \mathtt{case}\ E\ \mathtt{of}\ \overline{\mathtt{C_i x_0 \ldots x_n}\ \to\ \mathtt{u_i}} \mid \mathtt{let\ x\ =}\ E\ \mathtt{in\ u}$

$$\frac{u \rightsquigarrow u'}{E[\mathtt{u}] \rightsquigarrow E[\mathtt{u'}]}\ \text{CtxStep} \qquad \frac{}{E[\mathtt{abort}] \rightsquigarrow \mathtt{abort}}\ \text{EAbort} \qquad \frac{}{E[(\backslash\mathtt{x\ ->\ t})\,\mathtt{v}] \rightsquigarrow E[[\mathtt{v}/\mathtt{x}]\mathtt{t}]}\ \text{EBeta}$$

$$\frac{}{E[\mathtt{let\,x = v\,in\,u}] \rightsquigarrow E[[\mathtt{v}/\mathtt{x}]\mathtt{u}]}\ \text{Let} \qquad \frac{}{E[(\mathtt{rec\,f(x).u})\,\mathtt{v}] \rightsquigarrow E[[\mathtt{rec\,f(x).u/f, v/x}]\mathtt{u}]}\ \text{Rec}$$

$$\frac{}{E[\mathtt{tcast\ v}] \rightsquigarrow E[\mathtt{v}]}\ \text{TCast} \qquad \frac{}{E[\mathtt{case\ C_j\ v_0 \ldots v_n\ of\ \overline{C_i x_0 \ldots x_n \to u_i}}] \rightsquigarrow E[[\mathtt{v_0}/\mathtt{x_0}, \ldots, \mathtt{v_n}/\mathtt{x_n}]\mathtt{u_j}]}\ \text{CaseTerm}$$

?!7 CBV Operational Semantics for Programs

"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 `p` refers to proofs, and `P` to propositions. Propositions classify proofs (i.e. `p : 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 `t` and `T` for expressions when we wish informally to emphasize the roles of subject (`t`) and classifier (`T`), respectively. We will use the metavariable `e` to represent either proofs (`p`) or programs (`t`) and the metavariable `A` to represent either propositions (`P`) or program types (`T`). The metavariables `x`, `y`, `f`, and `q` range over both program and proof variables.

The syntax of the features of Sep^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:A] -> t`. In the rules, we assume all typing contexts are well-formed; that is, for any $\mathtt{x} : \mathtt{A}$ in a context Γ the type `A` is kindable. In the proof fragment this requires the kinding judgment $\Gamma \vdash \mathtt{P} : \mathtt{Formula}$ over propositions. We elide the definition.

As noted above, Sep^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^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 (`conv`) 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^3 are purely specificational, and do not have any run-time behavior.

Sep^3 provides a mechanism for a programmer to specify which *programs* should be preserved across the eraser, by allowing abstractions and applications to be annotated as compile-time or run-time. For example, type arguments to polymorphic functions generally do not contain computational content and are annotated as compile-time. In the concrete syntax of the language presented in this paper, compile-time abstractions are marked by wrapping the abstraction variable and type annotation with square brackets, as shown in the polymorphic identity function:

```
\[a:Type] -> \(x:a) -> x
```

Similarly, applications to compile-time arguments are marked with square brackets, as in the `Nat` type argument to the identity function:

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

Compile-time applications and abstractions are erased before executing a program and when proving programs equal with `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 `VNil` (defined in the datatype section below) that differ

only in the representations of the compile-time length parameter (e.g. `Z` vs. `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 [33, 17, 21].

Equality Formulas The Sep^3 proof language includes a primitive formula representing the propositional equality of two programs, written $\mathtt{t}_1 = \mathtt{t}_2$. A proof of an equality is given by the expression `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 `join` is shown as TJoin in Figure 5. In this rule and throughout the sequel we denote α-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 α-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 $\mathtt{conv\,t\,at\,\hat{T}}$ (for programs) or $\mathtt{conv\,p\,at\,\hat{P}}$ (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 (`T` or `P`), extended with a special *escape* form. The escape form `~p` identifies locations where the type should be changed using a proof of an equality `t1 = t2`. We define functions $\lfloor$ and $\rfloor$ that replace an escape occurring in a conversion context with the left (respectively, right) hand side of the equation proved by the escaped proof.

$$\Gamma \lfloor \mathtt{\sim p} = \mathtt{t1} \quad \text{when } \Gamma \vdash \mathtt{p} : \mathtt{t1} = \mathtt{t2}$$
$$\Gamma \rfloor \mathtt{\sim p} = \mathtt{t2} \quad \text{when } \Gamma \vdash \mathtt{p} : \mathtt{t1} = \mathtt{t2}$$

The $\lfloor$ and $\rfloor$ functions are defined recursively on `P` and `T`. For any term that is not an escape, the function defined is just applied to the subterms, and the original term is returned. For example:

$$\Gamma \lfloor \mathtt{t\ e} = (\Gamma \lfloor \mathtt{t}) \ (\Gamma \lfloor \mathtt{e})$$

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

```
conv v at (Vector a ~p)
```

Conversion in Sep^3 is not automatically inferred, so must be supplied by the programmer. This is because conversion is based on equality between programs, which is undecidable in Sep^3, since the (programming) language is not normalizing.

The `let` form for introducing local program bindings includes an additional variable representing a proof that equates the bound name with its definition. If the locally-bound name is captured in the type of the body of the let term (due to a dependent type), then it is necessary to cast the type of the body to remove 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, `t!`, indicating 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 `valax` (pronounced "value axiom") form. This includes programs which are syntactic values. Furthermore, termination proofs can be introduced using a termination case proof construct, `termcase`, that case-splits non-constructively on the termination behavior of a program (see Section 5). Finally, with `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 `rec` form for defining general recursive functions. This construct does not constrain the arguments to recursive calls, potentially allowing diverging computation. The proof language, in contrast, provides an `ind` form for induction over

programs. The 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 `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 `ordtrans` expresses transitivity of the structural ordering, thereby allowing `ind` to be used for course-of-values induction. The typing rules for `ind`, `ord`, and `ordtrans` are shown in Figure 5, and the TRec typing rule for `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 example, is terminating using induction with our structural ordering over natural numbers. The proof is simply a nested natural-number induction, where the outer induction proves a quantified statement which is itself proved by an inner induction, in the cases of the outer induction. Similarly, mutual induction of multiple theorems can be encoded as a single induction yielding a conjunction of proofs of the constituent theorems.

Data Types Sep^3 includes support for algebraic data types and indexed type families. These types are purely programmatic data, and may include diverging programs. We use the term "inductive datatype" to cover both.

An example non-indexed family is that of natural numbers, shown below. This definition is elaborated to the core language by introducing constants `Nat : Type` , `Z : Nat`, and `S : (x:Nat) -> Nat`.

```
data Nat : Type where
  Z : Nat
  S : (x:Nat) -> Nat
```

When case splitting on a program that yields an element of an algebraic data type, in each branch we get a proof that equates the scrutinee with the pattern of the branch alternative. This proof is given a name taht is supplied, in the `case` syntax, in braces following the scrutinee program. Moreover, when case-splitting in a proof, we are required to supply a proof that the scrutinee terminates. This is because, as just noted, (programmatic) datatypes may be inhabited by diverging programs, and hence we must know that the scrutinee does not diverge, in order to case split safely on its form. To illustrate the case typing rules, Figure 8 shows an example instantiation of TCaseProof for `Nat`.

The TContra rule allows us to reason using contradictions arising from equations between dissimilar constructor expressions. For example, given an proof of the `S Z = Z`, we can use `contra` to prove any proposition.

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 data type.

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 constructor parameter. The type of `VNil` requires a proof that the length `n` is equal to `Z`. Similarly, the type of `VCons` constructor takes an argument `m` representing the length of the tail of the vector, as well as a proof that `n` is the successor of `m`.

```
data Vec : [a:Type] [n:Nat] -> Type where
    VNil : [q:n = Z] -> Vec a n
  | VCons : [m:Nat] -> [q:n = S m] ->
      (x:a)->(xs:Vec a m) ->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 `Vec : (a :Type)-> (n:Nat)-> Type`. The compile-time arguments `[a:Type]` and `[n: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 `Vec [bool] [n] = Vec [nat] [n]` for unsoundly casting a vector of booleans to a vector of

$$\frac{\begin{array}{l}\Gamma \vdash \texttt{t} : \texttt{Nat} \\ \Gamma \vdash \texttt{p} : \texttt{t!} \\ \Gamma \vdash \texttt{Nat} : \texttt{Type} \\ \Gamma \vdash \texttt{Z} : \texttt{Nat} \\ \Gamma, \texttt{z} : (\texttt{t} = \texttt{Z}) \vdash \texttt{pz} : \texttt{P} \\ \Gamma \vdash \texttt{S} : (\texttt{x} : \texttt{Nat}) \to \texttt{Nat} \\ \Gamma, \texttt{w} : \texttt{Nat}^{\mathbf{val}}, \texttt{z} : (\texttt{t} = \texttt{S}\,\texttt{w}) \vdash \texttt{ps} : \texttt{P}\end{array}}{\Gamma \vdash \texttt{case t \{z\} p of Z} \to \texttt{pz}; \texttt{S w} \to \texttt{ps} : \texttt{P}}$$

?!8 Case instantiation for natural numbers

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 `VNil` constructor elaborates to a constant with type `[a:Type] -> [n:Nat] -> [q:n = Z] -> Vec a n`. The length index `n` is marked as compile-time, and will be erased in applications of `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^3 design, we allow a programmer to define functions directly with general recursion, and then later prove termination separately. These positions occupy two different points in the design space; we believe that the Sep^3 design offers the advantage of incrementality.

3 A Value Judgment

In Sep^3, we syntactically classify some programs as values, as is usual when defining a language. Typically, in a call-by-value language like Sep^3 variables are identified as values, since the operational semantics of the language dictates that when reducing an application the argument must be reduced to a value to get call-by-value β-redex 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^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^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 re-

$$\frac{\texttt{x} : \texttt{A}^{\textbf{val}} \in \Gamma}{\Gamma \vdash \textbf{val}\ \texttt{x}}\ \text{ValVar}$$

$$\frac{}{\Gamma \vdash \textbf{val}\backslash(\texttt{x} : \texttt{A}) \to \texttt{t}}\ \text{ValLam}$$

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

$$\frac{}{\Gamma \vdash \textbf{val}\ \texttt{tcast t by p}}\ \text{ValTCast}$$

?!9 Selected Value Judgment Rules

duce the programs those theorems range over to values. For example, `join m : plus x Z = x` regardless of whether `x` terminates. If `x` diverges then both sides of the equality diverge, and are hence still equal.

For Sep^3, we modify the definition of the set of 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` : `Nat`$^{\textbf{val}}$ 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 normal form of the scrutinee will be a constructor applied to arguments that are values. Thus, when performing reasoning within a case branch, the pattern variables for the case branch will necessarily be instantiated with values. Consequently, when adding those variable to the context when type checking a case branch, we mark those variables as values, as shown in the TCaseProof rule in Figure 5.

4 Evaluation with Termination Cast

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

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

Consider a theorem that expresses that the application of a polymorphic identity function to *any* argument (including diverging arguments) can always be substituted with the argument itself.

The identity function and proof definitions are de-

fined in the syntax of the Sep[3] tool, which allows top-level program declarations to be defined by combining a type annotation and a program definition.[4] For example, the `Program` declaration below defines a name, `id`, and term `\[a:Type] -> \(y:a) -> y`, with type `[a:Type] -> \(y:a) -> a`. Similarly, `id_is_id` defines a proof with an associated quantified proposition.

```
Program id : [a:Type] (x:a) -> a := x
Theorem id_is_id :
  forall (a:Type)(x:a).id [a] x = x :=
    join 100
```

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

`id [a] x`	{by erasure}
`id x`	{by def. of `id`}
`(\y.y) x`	$\not\rightarrow$

The β-redex `(\y.y) x` is blocked because the Sep[3] programming language has a call-by-value semantics, and the variable `x` is not a value, as it was introduced by a proof. 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 β-redex `(\y.y) x` would step, because it is not necessary to reduce the argument to an application to a value prior to β-reduction. Making such a change would be unsafe, however, because it could allow us to observe different termination behaviors of the program, depending on whether the program is being reduced inside a proof (using call-by-name) or during actual execution (using call-by-value).

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

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

Sep[3] provides a *termination cast* construct, `tcast`, that allows a programmer to mark expressions as known to be terminating. The `tcast` construct takes a program and a proof that the program has a value. The 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 hold for terminating arguments, by adding an additional parameter to the theorem that proves `x` is terminating. The proof is shown below.

```
Theorem id_is_id_term :
  forall (a:Type)(x:a)(x_term:x!).
      (id [a] x = x) :=
    join 100 :
      (id [a] (tcast x by x_term) = x)
```

In Section 3 we introduced a value judgment to differentiate between variables introduced by proof abstractions from those introduced by program abstractions. Using the value judgment, it is possible to redefine the β-rule to use the value judgment.

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

This rule allows reduction to reuse the value judgment, but it comes at the cost of complicating the reduction relation. No longer is reduction defined syntactically, but now it is a contextual relation. The alternative approach, and the one we take in Sep[3], is to use the syntactic view of evaluation. We 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 production `v` in Figure 3. Also included in this syntactic category are programs wrapped with termination casts. Variables are not classified as values, as before. If a term can be judged a value, then

[4] The type annotation is necessary because the typechecker implementation uses a bidirectional algorithm that combines type checking with type synthesis.

it is possible to extract a term in this class of syntactic values using a termination cast. For example, if a variable `x` is judged a value (but would not be syntactically classified a value), then `tcast x by valax x` is the associated syntactic value. In this way, we simplify the reduction relation to use standard call-by-value β-reduction over a class of syntactic values including `tcast` constructs, while still using the value judgment for typing.

When constructing a proof of equality using `join`, the programs being equated are erased, as described previously. However, the `tcast` constructs are preserved (the termination proof is dropped, as it has no computational content). Additionally, reduction allows a `tcast` to be dropped when the expression being cast is a syntactic value. This prevents `tcast` from blocking a redex, as would be the case with `(tcast \x.x by p) v`.

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

Preserving termination casts when reasoning about programs using `join` may cause a term to take more reduction steps inside proofs, because the EBeta reduction step may duplicate `tcast` 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 scrutinized program terminates or diverges, with `y` bound as a proof of the corresponding termination assumption in each branch. In the terminates (`!`) branch, `y` is a proof that the scrutinee terminates. In the diverges (`abort`) branch, `y` is a proof that the scrutinee is equal to `abort`, signifying divergence. The typing rule TAbort, for `abort` in the annotated language, requires an annotation `t` providing the type of the `abort` term. Reduction for `abort` is defined by the rule EAbort. If `abort` appears in evaluation position, then the term immediately steps to `abort`. This means that all provably diverging programs (identified by equivalence to `abort`) are contextually equivalent.

The `termcase` construct is non-constructive, as it axiomatizes excluded middle for termination, an undecidable property. This relies on an oracle that can determine whether any given term normalizes to a value. The construct can be viewed as an internalization of the theorem of type soundness for the program fragment, relaxed to partial correctness: if $\vdash$ `t : T`, then either `t` $\rightsquigarrow^*$ `v` (where `v` is a syntactic value) and $\vdash$ `v : T` , or else `t` diverges. This has already been observed in Wright and Felleisen's classic paper [36].

Using `termcase` we can strengthen proofs of algebraic theorems, which are subject to termination preconditions, to stronger theorems that do not require termination preconditions. As a simple example, we can return to the proof of `id_is_id` from Section 4. To prove the theorem above, it is necessary to have a proof of termination available to `tcast` the variable `x`, so that the `join` proof can succeed. Nevertheless, the theorem is valid for all inputs, irrespective of termination 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
   abort ->
    let u1 = join 100
          : id [a] (abort a) = (abort a)
    in conv u1
        at (id [a] ~x_term = ~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 termi-

nation of the functions we are reasoning over.

6 Implementation and Experience

To gauge the feasibility of the Sep^3 language design, we have implemented a prototype type checker and evaluator. This implementation has been useful in guiding the language design, and the design of Sep^3 and the implementation have continued to evolve in tandem. The implementation is available online.[5] The examples in this paper can be found in `Tests/unittests/PiI.sep`.

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

6.1 Proof Automation

Sep^3 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.

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,

[5] `http://trellys.googlecode.com/svn/tags/pii-release/lib/sepp`

```
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
```

?!10 Proofs that `=` is an equivalence.

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 Sep^3 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 Sep^3, simply supplying those type arguments leads to very verbose programs. Because reasoning in Sep^3 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 Sep^3, 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 Sep^3 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 $\texttt{S}|_p$ to represent the subterm of `S` occuring at position p, for some subterm position p of `S` (and similarly for `T` and `C`). If $\texttt{S}|_p$ and $\texttt{T}|_p$ differ, we check to see if there is an equation in the context `h :` $\texttt{S}|_p = \texttt{T}|_p$. If so, then we set $\texttt{C}_p$ to be `~h` (the escape of `h`). If $\texttt{S}|_p$ and $\texttt{T}|_p$ are the same, then we set $\texttt{C}|_p$ to be $\texttt{S}|_p$. As with the `equiv` tactic, we do not need to trust the implementation of `autoconv` as the resuting 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 Sep^3 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

```
Program (&&) : (x:Bool)(y:Bool) -> Bool :=
   case x {x_eq} of
     True -> y
    | False -> False


Theorem and_commutes :
  forall(x:Bool)(y:Bool)(x_term:x!)(y_term:y!).
     (x && y) = (y && 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:Bool)(y:Bool)(x_term:x!)(y_term:y!).
   (x && y) = (y && 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 ->
    (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})
```

?!11 Conjuction is commutative

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 supplied to `morejoin` includes a proof of `p:t2 !`, then a termination cast is inserted around `t2`, and `v1 (tcast t2)` is a β-redex. Furthermore, if the term to be reduced is a case expression `case t1 of { ...}`, and the list of proofs supplied to `morejoin` includes `p : t1 = t2`, then the term is rewritten to `case t2 of {...}`, and reduction proceeds.

Combining inserting termination casts and rewriting is quite useful, as oftentimes we will want to prove some formula `f t1 = t2` where `f` does an immediate case-split on its argument, and we have proofs of `t1!` and `t1 = v1`. Intuitively, these proofs imply that the value of `t1` is `v1`, so we would like to reduce `f t1` and then reduce the case-split on `v1`. But in general `t1` need not actually reduce to `v1` (or indeed any value), and so we need to insert a `tcast` around `t1` to enable the first reduction (of `f t1`). Inserting this `tcast` around `t1` prevents further reduction within `t1` (since `tcast E` is not included as a form of evaluation context in Figure 7). This means that `morejoin` will reduce `f t1` to a case-split on `tcast t1`, and then substitute `v1` for `t1`. The TCast rule (of Figure 7) will then reduce this term to a case-split just on `v1`, which can finally reduce using the CaseTerm rule.

Inserting termination casts and rewriting case scrutinees can be justified quite directly.

If `t` $\rightsquigarrow^m$`v1 t2`, and `p:t2!`, then `join m : t = v1 t2`. If `join 0 : v1 t2 = v1 (tcast t2 by p)` using transitivity we can prove `t = v1 (tcast t2 by p)`. Similarly, for case expressions, if `t` $\rightsquigarrow^m$ `case t1 of {...}` and `p : t1 = t2`, then `conv (join m )at t = case ~p of {...}`. If `case t2 of {...}` reduces further, we can produce a proof equating it with its contractum using `join`, and compose the proofs using `trans`.

The primitive `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 `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 `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 `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 liberally use `morejoin` to automatically insert conversions and termination casts when constructing equalities, rather than using the more verbose `join`.

These examples use additional notation from the Sep[3] implementation to introduce top-level definitions. The `Recursive` form used by `append` elaborates to a `rec` construct directly. The body of the `rec` includes nested lambda-abstractions for each additional parameter to the `Recursive` definition.

The `Inductive` notation used with `append_term` is somewhat more complicated. In the example, we are performing on induction on the argument `xs`. 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 preceeding 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 `ind` form. All following arguments will be bound by lambda-abstractions in the body. In the `append_term` example, the proof term will desugar to the following.

```
\(a:Type) =>
  ind (xs:List a) [xs_term] .
    \(ys:List) => \(ys_term:ys!) =>  ...
```

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 ar-

[6]In principle, the evaluator should also keep a trace of rewrites used, so that a proper proof term with the required `conv`s can be reconstructed, although the current implementation simply trusts the instrumented evaluator.

```
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)
```

?!12 List Append

```
Inductive append_assoc_term :
  forall (a:Type) (xs:List a){xs_term}
      (ys:List a)(ys_term:ys!)(zs:List a)(zs_term:zs!) .
    append [a] xs (append [a] ys zs) =
     append [a] (append [a] xs ys) zs :=
   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, xs_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)
```

?!13 Associativity of List Append

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

?!14 Proof that append terminates on terminating inputs

```
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);
          u1 =  trans aleft aright
      in conv u1 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);
              u1 = trans aleft aright
          in conv u1 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)));
                 u1 = trans aleft aright
              in conv u1 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
```

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

guments 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_term` indicates that the preceding argument `xs` is the induction variable. When applying `append_assoc_term` from outside the proof, the argument in `xs_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 `append_assoc_term`, we supply a proof that `xs' < xs`, as shown in the `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 `morejoin` using the supplied termination proofs, to reduce programs involving these variables.

The proof uses an additional lemma (Figure 14) that proves `append` total on terminating inputs. This is a convenience, to simplify presentation, and could be avoided with additional reasoning using `termcase`. For more complex functions, where the proof of totality is not so straightforward, using `termcase` may be preferable.

Because the programs `xs`, `ys`, and `zs` are all used in strict positions on both sides of the equality, the formula can be strengthened to an equality over all 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 `termcase` to consider the termination behavior of each argument in turn. In each `abort` branch, the EAbort rule allows us to join an application of append to the diverging argument with abort, demonstrating that both sides of the associativity formula join with abort. In the final terminates branch, the context contains proofs `xs_term`, `ys_term`, and `zs_term` that prove the associated arguments are terminating. With these proofs available, the weaker `append_assoc_term` lemma can be invoked.

7 Example: Combinators

The `append_assoc` example of the previous section provides a small flavor of proving system properties within Sep^3. 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 Sep^3 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 Sep^3 solution below, we use these solutions as points of comparison to the Sep^3 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 website[7]. In the interests of making the description of the Sep^3 solution self-contained, we reproduce parts of the problem as necessary.

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

```
t := S | K | t t,
```

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:

```
v := K | S | K v | S v | S v v
```

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

```
C = □ | C t | v C.
```

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

[7] `https://sites.google.com/site/vstte2012/compet`

$$
\begin{array}{rcl}
\Box[\texttt{t}] & = & \texttt{t} \\
\texttt{(v C)}[\texttt{t}] & = & \texttt{v C}[\texttt{t}] \\
\texttt{(C t)}[\texttt{t'}] & = & \texttt{C}[\texttt{t'}]\ \texttt{t}
\end{array}
$$

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

$$
\begin{array}{rcl}
\texttt{C[K t1 t2]} & \rightarrow & \texttt{C[t1]} \\
\texttt{C[S t1 t2 t3]} & \rightarrow & \texttt{C[(t1 t2) (t1 t3)]}
\end{array}
.
$$

We call the transitive-reflexive closure $\rightarrow^*$ of $\rightarrow$ the reduction relation. Finally, for a term `t`, if there does not exist a term `t'` such that $\texttt{t} \rightarrow \texttt{t'}$, then we write $\texttt{t} \not\rightarrow$.

The programming task is to define a function `reduction` that takes a term and returns a term `t'` such that $\texttt{t} \rightarrow^* \texttt{t'}$ and $\texttt{t'} \not\rightarrow$. If there does not exist such a `t'`, then `reduction` diverges. The verification problem consists of three parts. In the first, we are required to prove `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 Sep^3.

7.2 Sep^3 solution

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

```
data Term : Type where
    S : Term
  | K : Term
  | App : Term -> Term -> Term
```

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 Sep^3, 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 Sep^3, 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.

```
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}.
    isRedex t ! := <omitted>
```

?!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 terminating inputs, although we elide the proof.

The formulation of the small-step reduction relation in the problem description is expressed in terms of evaluation contexts. Rather than implement this directly, we instead define a function `step` only on redexes instead of all reducible terms. We then define a function `decompose` for identifying a redex and evaluation context of a term, as well as a function `plug` performing the substitution of a term for $\Box$ in an evaluation 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
```

```
-- Recursive value predicate
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 !
```

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 effectively 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 designated `Term`. Whatever the choice, it may not be immediately 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 programmatic type `RedexProp t` representing the inductive proposition that a term is a redex. Moreover, the data constructors for `RedexProp` carry exactly the information about the shape of the index term `t` necessary for defining the `step` function.

```
data RedexProp : (t:Term)  -> Type where
 RedexK : (t1:Term) -> (t2:Term) ->
    [p:t = App (App K t1) t2] ->
    [p1:isValue t1 = True] ->
    [p2:isValue t2 = True] ->
    RedexProp t
|RedexS :
  (t1:Term) -> (t2:Term) -> (t3:Term) ->
  [p:t = App (App (App S t1) t2) t3] ->
  [p1:isValue t1 = True] ->
  [p2:isValue t2 = True] ->
  [p3:isValue t3 = True] ->
  RedexProp t
```

Given a proof `p:isRedex t = True`, we can define a recursive program `redexProp` that constructs 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 application 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 contradiction apparent, using the `contra` primitive.

The definition of `step`, in turn, never directly decomposes the input term `t`, but rather first constructs a `RedexProp` proposition and then case splits on it. The associated `step_terminates` theorem shows that `step` is total, directly appealing to the `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
```

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 $\rightarrow$. 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' ->
```

```
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)]
                         [and_left (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
   | ! ->
       case t {t_eq} t_term of
         K -> let u1 = morejoin {t_term,sym t_eq} : False = isRedex t
              in contra (equiv 3 : False = True)
       | S -> ...
       | App f1 t1 -> ...
```

?!18 Redex Inductive Proposition

```
data Decomp : 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)
```



```
     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 $\rightarrow^*$ 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 Sep[3] 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 Sep[3] `_ < _` 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, `iter_reduce` – a *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, `reduction` is defined by appealing to an oracle supplying an appropriate number of reduction steps.

```
iter_reduce(n)(R): RECURSIVE term =
 (IF n = 0
   THEN R
   ELSE
    (LET Q = reduce(R)
     IN IF R = Q
          THEN R
```

```
        ELSE iter_reduce(n - 1)(Q)
       ENDIF)
  ENDIF)
 MEASURE n

reducible?(R): bool =
  (EXISTS n: value?(iter_reduce(n)(R)))

reduction(R: (reducible?)): (value?)
 = iter_reduce(choose! n:
     value?(iter_reduce(n)(R)))(R)
```

This solution uses a predicate subtype `reducible?` to ensure that `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 `reduction` is not defined on all combinator terms.

Sep[3] provides a similar oracle by virtue of the `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 `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 `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.

```
(defchoose steps-required (n) (x)
  (b* (((mv terminates &)
        (reduce-n n x)))
      terminates))
```

Next, the `reduction` function is defined. In contrast to the PVS solution, this definition is separated into two parts – a logical specification that uses the `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 Sep[3].

The comparison between the 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 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 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 `s_free` predicate is defined recursively on the structure of the input term `t`.

```
Recursive s_free : (t:Term) -> Bool :=
  case t {t_eq} of
      K -> True
    | S -> False
    | App t1 t2 -> s_free t1 && 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 `(reduction t)!` directly.

```
Inductive s_free_term :
   forall (t:Term){t_term}
    (p:s_free t = True).(reduction t) !
```

The proof again follows the structure of the `reduction` function. We had to prove two main lemmas that the `decomp` and `plug` functions preserve `s_free` of their results.

```
Inductive decomp_preserves_s_free :
   forall(t:Term){t_term}
     (t':Term)(t'_term:t'!)
     (c:Ctx)(c_term:c!)
     (p:s_free t = True)
     (q:decompose t = Dec c t').
    (s_free t' = True) :=
     <omitted>

Inductive plug_preserves_s_free :
 forall (t:Term)(t_term:t!)
        (t':Term)(t'_term:t'!)
        (c:Ctx){c_term}
        (t'':Term)(t''_term:t''!)
        (p:decompose t = Dec c t')
       (q1:s_free t = True)
       (q2:s_free t'' = True).
      (s_free (plug c t'') = True) :=
        <omitted>
```

A `s_free` redex will always produce a smaller term – taking `K t1 t2` to `t1`. However, the small-step reduction relation is phrased in terms of evaluation contexts. Using `plug` the contractum is plugged into the evaluation context, which obscures the decrease in size. Consequently, we are required to prove a

theorem `plug_preserves_ord` that expresses the decrease in size of the term using the `_ < _` formula.

```
Inductive plug_preserves_ord :
   forall(t1:Term)(t1_term:t1!)
         (t2:Term)(t2_term:t2!)
         (p:t1 < t2)(c:Ctx){c_term}.
     plug c t1 < plug c t2 := <omitted>
```

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 [15, 16, 5, 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, Haswegawa 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 [23, 22]. Thus, Ynot provides an internal verification solution to the problem of interest in this paper, and indeed to the further difficult matter of reasoning about state. But, to our knowledge, Ynot is not intended for external reasoning about programs; rather, it uses a monad indexed by pre- and post-conditions on the imperative state in order to perform internal verification of programs. Previous work of Stump and co-authors on Guru has similar goals as Sep[3], with a similar language design separating proofs and programs, and using termination casts with a judgmental notion of value [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[8].

Somewhat less closely related are works based on the Edinburgh Logical Framework (LF) (also known as the λ^{Π} type theory) [10]. Systems like Beluga and Delphin add the ability to write functional programs operating over datatypes described in LF [29, 30]. Like the type theories mentioned above, these systems support terminating recursion over indexed

[8] This issue with the Guru implementation, discovered in the course of the current research on Sep[3], remains to be repaired.

datatypes. Since these datatypes are expressed in LF, however, they may use higher-order abstract syntax to encode object-level binders using λ-abstraction in LF. On the other hand, the main application of these systems so far is to formalized meta-theory of languages with binders. They are not concerned with proving properties about general recursive functions under call-by-value reduction semantics. The Twelf tool does allow one to define terminating as well as general-recursive functions over datatypes defined in LF, in this sense providing features similar to those we are aiming at in Trellys [28]. Unlike Beluga and Delphin (and Trellys), Twelf uses the logic-programming paradigm for expressing such functions, thus imposing an additional conceptual burden on programmers unfamiliar with logic programming. Furthermore, to date no theory has been worked out for extending Twelf with a number of constructs standardly found in Type Theory (e.g., polymorphism and large eliminations).

9 Conclusion

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

Sep^3 uses a syntactic distinction between the proof and programming languages to isolate non-termination in the programming language from the proof language. Despite the syntactic distinction proofs can *mention* 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, depending on context, over values or expressions, so Sep^3 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 Sep^3 represents one attempt at defining a core language to support the desired goal of combining dependent types and a call-by-value language including general recursion. While the principal language design includes the concepts presented here as well as many other features, much work remains, most importantly the analysis of the meta-theoretical properties of the language design. Sep^3 depends on a syntactic separation between the proof and programming fragments of the language. The Trellys team continues investigation into methods to remove this syntactic distinction, including an internalized type representing the proof/program classification of a term. This allows terms to safely be migrated from the proof language to the programming language, and vice-versa.

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

References

[1] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. PiSigma: Dependent Types without the Sugar. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, pages 40–55, 2010.

[2] Ana Bove and Venanzio Capretta. Modelling General Recursion in Type Theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

[3] Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda - A Functional Language with Dependent Types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009.

[4] Ana Bove, Peter Dybjer, and Andrés Sicard-Ramírez. Embedding a Logical Theory of Constructions in Agda. In Thorsten Altenkirch and Todd D. Millstein, editors, *PLPV*, pages 59–66. ACM, 2009.

[5] Venanzio Capretta. General Recursion via Coinductive Types. *Logical Methods in Computer Science*, 1(2), 2005.

[6] Arthur Charguéraud. Program Verification through Characteristic Formulae. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional programming*, ICFP '10, pages

321–332, New York, NY, USA, 2010. ACM.

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

[8] Jared Davis, Matt Kaufmann, J Strother Moore, and Sol Swords. ACL2 Solution to VSTTE 2012 Problem 2. Website. `http://www.cs.utexas.edu/users/moore/acl2/vstte-2012/acl2-dkms/problem2/`.

[9] Solomon Feferman. *In the Light of Logic*. Oxford University Press, 1998.

[10] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[11] Masahito Hasegawa and Yoshihiko Kakutani. Axioms for Recursion in Call-by-Value. *Higher Order Symbol. Comput.*, 15(2-3):235–264, September 2002.

[12] Chung Kil Hur, 2010. Message on the Agda mailing list, `https://lists.chalmers.se/pipermail/agda/2010/001526.html`.

[13] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-aided Reasoning: An Approach*. Springer Netherlands, 2000.

[14] Garrin Kimmell, Aaron Stump, Harley D. Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins, and Ki Yung Ahn. Equational Reasoning about Programs with General Recursion and Call-by-value Semantics. In Koen Claessen and Nikhil Swamy, editors, *PLPV*, pages 15–26. ACM, 2012.

[15] Alexander Krauss. Partial and Nested Recursive Function Definitions in Higher-Order Logic. *Journal of Automated Reasoning*, 44(4):303–336, 2010.

[16] Alexander Krauss. Recursive Definitions of Monadic Functions. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *Workshop on Partiality and Recursion in Interactive Theorem Provers (PAR)*, volume 43 of *EPTCS*, pages 1–13, 2010.

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

[18] John Longley and Randy Pollack. Reasoning About CBV Functional Programs in Isabelle/HOL. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *TPHOLs*, volume 3223 of *Lecture Notes in Computer Science*, pages 201–216. Springer, 2004.

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

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

[21] Nathan Mishra-Linger and Tim Sheard. Erasure and Polymorphism in Pure Type Systems. In Roberto M. Amadio, editor, *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2008.

[22] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and Separation in Hoare Type Theory. In *Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 62–73. ACM, 2006.

[23] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent Types for Imperative Programs. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming (ICFP)*, pages 229–240, 2008.

[24] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[25] Nicolas Oury, 2008. Message on the coq-club mailing list, `https://sympa.inria.fr/sympa/arc/coq-club/2008-06/msg00022.html`.

[26] Sam Owre, John Rushby, and Natrajan Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[27] Sam Owre and Natrajan Shankar. PVS Solution to VSTTE 2012 Problem 2. Website. `http://www.csl.sri.com/users/shankar/pvs-examples/sri-vstte12-competition.tgz`.

[28] Frank Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.

[29] Brigitte Pientka and Joshua Dunfield. Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description). In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, pages 15–21, 2010.

[30] Adam Poswolsky and Carsten Schürmann. System Description: Delphin - A Functional Programming Language for Deductive Systems. *Electr. Notes Theor. Comput. Sci.*, 228:113–120, 2009.

[31] Alex Simpson and Gordon Plotkin. Complete Axioms for Categorical Fixed-Point Operators. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 30–41. IEEE Computer Society, 2000.

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

[33] Aaron Stump, Morgan Deters, Adam Petcher, Todd Schiller, and Timothy Simpson. Verified Programming in Guru. In T. Altenkirch and T. Millstein, editors, *Programming Languges meets Program Verification (PLPV)*, 2009.

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

[35] Christoph Walther and Stephan Schweitzer. Automated Termination Analysis for Incompletely Defined Programs. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference (LPAR), Montevideo, Uruguay*, pages 332–346, 2005.

[36] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and computation*, 115(1):38–94, 1994.