Compositional and Decidable Checking for Dependent Contract Types

Kenneth Knowles Cormac Flanagan
University of California at Santa Cruz
{kknowles,cormac}@cs.ucsc.edu

Abstract

Simple type systems perform *compositional reasoning* in that the type of a term depends only on the types of its subterms, and not on their semantics. Contracts offer more expressive abstractions, but static contract checking systems typically violate those abstractions and base their reasoning directly upon the semantics of terms. Pragmatically, this noncompositionality makes the decidability of static checking unpredictable.

We first show how compositional reasoning may be restored using standard type-theoretic techniques, namely existential types and subtyping. Despite its compositional nature, our type system is *exact*, in that the type of a term can completely capture its semantics, hence demonstrating that precision and compositionality are compatible. We then address predictability of static checking for contract types by giving a type-checking algorithm for an important class of programs with contract predicates drawn from a decidable theory. Our algorithm relies crucially on the fact that the type of a term depends only the types of its subterms (which fall into the decidable theory) and not their semantics (which will not, in general).

Categories and Subject Descriptors F.3.1 [*LOGICS AND MEANINGS OF PROGRAMS*]: Specifying and Verifying and Reasoning about Programs—Specification techniques

General Terms Languages, Algorithms, Verification.

Keywords Dependent types, refinement types, abstraction, compositional reasoning

1. Compositional Reasoning for Contract Types

The construction, analysis, and verification of large programs relies on *compositional reasoning*. During program development, compositional reasoning separates a program into cognitively manageable pieces. During analysis and verification, compositional reasoning limits the amount of information that must be stored and processed.

Traditional type systems provide effective lightweight verification in part because they reason compositionally. That is, in a well-typed program such as

$$\texttt{let}\ x : T = e_1\ \texttt{in}\ e_2$$

the type T provides an abstract specification of e_1 that is concise and yet sufficiently precise to verify that e_1 and e_2 interact properly, in that the combined program does not go wrong. This verification process is compositional in that the verification of e_2 cannot rely on any properties of e_1 other than those exposed via its type T. Consequently, replacing e_1 with any equivalently-typed term does not affect the typeability of the overall program.

Contract Types The goal of this paper is to extend the benefits of compositional reasoning to *contract types*. Contract types are refinements types of the form $\{x : B \,|\, e\}$, where the contract predicate e may be an arbitrary computable boolean expression over the variable x of type B.

Contract types[1] provide a natural means for expressing a wide variety of specifications, such as the type `Pos` of positive integers $\{x : \texttt{Int} \,|\, x > 0\}$. Indeed, the type of a term can abstract its behavior to a greater or lesser degree (or, in the extreme, may not abstract its behavior at all). For example, the constant 1 has an infinite number of types, including `Int`, `Pos` and the exact type $\{y : \texttt{Int} \,|\, y = 1\}$, and these types are related via subtyping in the expected manner:

$$\{y{:}\texttt{Int} \,|\, y = 1\} \quad <: \quad \texttt{Pos} \quad <: \quad \texttt{Int}$$

Contract types co-operate in a clean and expressive manner with dependent function types. We denote dependent function types using the syntax $x : S \to T$, where argument variable x may occur free in the range type T.[2] This combination of type constructs supports a wide variety of function specifications, ranging from traditional simple types to more precise types or even to exact types. For example, the function `add1` has many types, ranging from a simple type to an exact type, with an intermediate specification describing only that `add1` is increasing. Again, subtyping appropriately relates these various specifications for `add1`:

$$\begin{array}{ll} & x{:}\texttt{Int} \to \{y{:}\texttt{Int} \,|\, y = x + 1\} \\ <: & x{:}\texttt{Int} \to \{y{:}\texttt{Int} \,|\, y > x\} \\ <: & \texttt{Int} \to \texttt{Int} \end{array}$$

Non-Compositional Dependent Types Contract types are inspired by much prior work on fully-dependent type systems, including Cayenne [Augustsson 1998], Epigram [McBride and McKinna 2004], Agda [Norell 2007] and Coq [The Coq development team 2004]. Such systems can express very precise specification, including full functional correctness in some cases. But considering the benefits of compositional reasoning, it is unfortunate that depen-

PLPV'09, January 20, 2009, Savannah, Georgia, USA.

[1] Contract types are also known as subset types [Rushby et al. 1998], refinement types [Denney 1998; Ou et al. 2004; Flanagan 2006], or Σ-types [The Coq development team 2004]. Following Xu et al. [2007], we use the term contract types to reflect their close relationship to dynamically-checked contracts [Findler and Felleisen 2002].

[2] $x : S \to T$ is equivalent to the more traditional syntax $\Pi x{:}S.\ T$.

dent type systems are based, in part, on *non-compositional* reasoning. That is, in a dependently typed program $\mathcal{C}[e]$, the type system can reason about e's interaction with its context $\mathcal{C}$ using both:

- compositional reasoning based on the *type* of e; and
- non-compositional reasoning based on the *behavior* of e.

This non-compositional nature of dependent type systems originates from the following standard type rule for dependent function application. (For simplicity, we assume an empty typing environment).

$$\frac{\vdash f \,:\, (x{:}S \to T) \qquad \vdash e \,:\, S}{\vdash f(e) \,:\, [x \mapsto e]\,T}$$

The inferred type $[x \mapsto e]\,T$ (denoting T with x replaced by e) of $f(e)$ includes the term e itself, and not just its type S.

To highlight the non-compositional nature of this rule, suppose that e is an arbitrary term of type `Pos`, and that f is the exactly-typed identity function on integers:

$$\begin{array}{lcl} e & : & \texttt{Pos} \\ f & : & (x{:}\texttt{Int} \to \{y{:}\texttt{Int} \mid y = x\}) \end{array}$$

Using compositional reasoning, we should only be able to infer that f returns its argument, which is a `Pos`, and so $f(e)$ has type `Pos`.

Under the above type rule, however, the application $f(e)$ has the more precise type (indeed, exact) type stating that $f(e)$ returns exactly the value of e.

$$f(e) \;:\; \{y{:}\texttt{Int} \mid y = e\}$$

Thus, the abstraction `Pos` of e has now been circumvented!

This non-compositional reasoning causes multiple difficulties. First, we now have an arbitrary program term (e) that has leaked into a contract predicate. Since e could be a large term, the type system now needs to deal with extremely large types, that may be much larger than any of the types present in the source program. Since the types in source programs are abstract specifications carefully chosen by the programmer, it is not clear that these new, larger types generated during type checking are at all the right abstractions for performing lightweight verification.

Second, even if the contract predicates in source programs are carefully chosen from a decidable theory, the fact that program terms leak into contract predicates during type checking means that static type checking still requires deciding implications between arbitrary program terms, which is of course undecidable.

Compositional Dependent Contract Types Given the beautiful and precise abstractions provided by dependent and contract types, and the benefits of compositional reasoning, this paper explores an alternative strategy for static checking of dependent contract types. Our guiding principle is:

> *Compositional reasoning*: Only use the abstractions provided by types when verifying that a term interacts correctly with its context.

The key idea underlying our approach is illustrated by the following rule for function application:

$$\frac{f : (x{:}S \to T) \qquad e : S' \qquad S' <: S}{f(e) : (\exists x{:}S'.\, T)}$$

The application $f(e)$ must return a value of type T, where x is bound appropriately, but what is the appropriate binding for x?

Compositionality dictates that all we are permitted to know about e (and hence about x) is that it has type S'. Thus, the inferred type of the application $f(e)$ is the existential type

$$\exists x{:}S'.\, T$$

Roughly speaking, this type denotes the union of all types of the form $[x \mapsto e']\,T$, where e' ranges over terms of type S'. (Existential types are closely related to dependent pairs $\Sigma x : S.T$, as discussed in Section 6 below.)

For the application $f(e)$ considered earlier, this rule infers the existential type

$$f(e) \;:\; (\exists x : \texttt{Pos}.\, \{y{:}\texttt{Int} \mid y = x\})$$

that is (informally) equivalent to the type `Pos`, and indeed is a subtype of `Pos`. Thus, the type rule achieves the desired goal of compositional reasoning: $f(e)$ has type `Pos` simply because the argument expression has type `Pos`. Furthermore, note that the expression e itself no longer leaks into the above contract type, which provides benefits in terms of smaller inferred types and facilitates decidable type checking.

This paper develops these ideas in the context of an idealized λ-calculus with contract types, existential types, and various kinds of primitive data types with pattern matching. As shown above, existential types are used pervasively to maintain compositionality, and inferred types often include an outer existential "wrapper" (like $\exists x : \texttt{Pos}.\, \cdots$). Where necessary, our system uses subtyping to relate existential and non-existential types, and to "hide" the existential wrappers, all without the need for explicit coercions. For example, as mentioned above, we have that:

$$(\exists x : \texttt{Pos}.\, \{y{:}\texttt{Int} \mid y = x\}) \;<:\; \texttt{Pos}$$

Expressiveness and Exactness Given that our type system is strictly limited to compositional reasoning, and cannot, for example, include the traditional rule for dependent function application, a key question that arises is:

> To what degree does the requirement for compositional reasoning limit the expressiveness of the type system?

Certainly, the contract type language itself is sufficiently expressive to describe exact types, such as the type

$$\{x{:}\texttt{Int} \mid x = 1\}$$

for the constant 1. More interestingly, we show that, if every constant in the language is assigned an exact type, then for any well-typed program e, our type system will infer an exact type that exactly captures the run-time semantics of e. This result holds even if e itself includes coarse type specifications such as `Int`.

Assigning exact types to primitives is straightforward, via *self-ification*. For example, the fixpoint primitive $\texttt{fix}_T$ is typically given the inexact type $(T \to T) \to T$. For the case where $T = x{:}T' \to B$, an exact type for $\texttt{fix}_T$ is:

$$f{:}(T \to T) \to x{:}T' \to \{y{:}B \mid y = \texttt{fix}_T\ f\ x\}$$

In practice, of course, primitives are typically assigned somewhat coarser types. Nevertheless, this completeness result indicates that the precision of the type system is entirely parameterized by the types of these primitives; the type system itself neither requires nor performs any additional abstraction.

Put differently, the precision of any compositional analysis is naturally driven by the precision of the abstractions that it composes, and careful choice of abstractions is crucial for achieving precise, scalable analyses. Our type system is entirely configurable in this regard; it does not perform any abstraction itself, and instead simply propagates the abstractions inherent in the types of constants and recursive functions, with the result that the precision of the analysis is largely under the control of the programmer.

Decidable Type Checking A key benefit of compositional type checking is that program terms never leak into contract predicates, and so the language of contract predicates can be cleanly separated from that of program terms. In particular, if contract predicates in source programs are drawn from the decidable theory of linear inequalities, then all proof obligations generated during type checking also fall within a decidable theory, and so type checking

Figure 1: Syntax

$e ::=$		*Expressions:*
	x	variable
	c	constructor
	f	primitive function
	$\lambda x{:}T.\, e$	abstraction
	$e\ e$	application
	$\texttt{case}\ e\ \texttt{of}\ c\ \bar{x}\ \triangleright\ e\ \texttt{else}\ e$	case
$v ::=$		*Values:*
	$c\ \bar{v}$	constructor application
	f	primitive function
	$\lambda x{:}T.\, e$	abstraction
$T ::=$		*Types:*
	$\{x{:}B \mid e\}$	refinement type
	$x{:}T \to T$	dependent function type
	$\exists x{:}T.\, T$	existential type
$B ::=$		*Base types:*
	$\texttt{Int}$	base type of integers
	$\texttt{Bool}$	base type of booleans
	$\texttt{IntList}$	base type of lists of integers
$\Gamma ::=$		*Typing Environments:*
	$\emptyset$	empty environment
	$\Gamma, x : T$	environment extension

is decidable. We use this result to develop a decidable type checking algorithm for an important and easily-identifiable subset of our target language. This development carefully exploits that inferred types, which may contain existential bindings, only occur on the left-hand-side in subtype checks, and so fortunately we never need to "guess" a witness for these existential types.

Outline In summary, the key contributions of this paper are:

- clarifying the non-compositionality of dependent type systems and its consequences for dependent contract types (Section 1)
- developing a compositional dependent type system for dependent contract types (Section 2)
- demonstrating through the exactness of this type system that precision and compositionality are both achievable (Section 3)
- providing a type checking algorithm for the important special case of programs whose contract predicates fall in a decidable theory (Section 4)

Section 6 discusses work with related goals and surveys other applications of similar type-theoretic techniques. Section 7 then concludes with suggestions for future work.

2. The System $\lambda^{\exists}$

We present our ideas in terms of the language $\lambda^{\exists}$, which extends the simply-typed λ-calculus with primitive functions, constructors, pattern matching, base types refined by boolean contracts, dependent function types, and existential types. The syntax of $\lambda^{\exists}$ is presented in Figure 1, and its small-step operational semantics in Figure 2. Redex evaluation $\to$ is closed under arbitrary contexts (in expression or types) to yield the reduction relation $\rightsquigarrow$. We write $\rightsquigarrow^*$ and and $\leftrightsquigarrow^*$ for the reflexive-transitive and equivalence closures of $\rightsquigarrow$, respectively.

Figure 2: Operational Semantics

Evaluation $\boxed{e_1 \to e_2}$

$$
\begin{array}{rcll}
(\lambda x{:}T.\, e_1)\ e_2 & \to & [x \mapsto e_2]\, e_1 & \text{[E-}\beta\text{]} \\
f\ e & \to & \delta(f, e) & \text{[E-PRIM]} \\
\texttt{case}\ c\ \bar{e}\ \texttt{of}\ c'\ \bar{x}\ \triangleright\ e_1\ \texttt{else}\ e_2 & \to & e_2 \quad (c \neq c') & \text{[E-FAIL]} \\
\texttt{case}\ c\ \bar{e}\ \texttt{of}\ c\ \bar{x}\ \triangleright\ e_1\ \texttt{else}\ e_2 & \to & [\bar{x} \mapsto \bar{e}]\, e_1 & \text{[E-MATCH]}
\end{array}
$$

Contextual Evaluation $\boxed{s \rightsquigarrow t}$

$$
\begin{array}{rcll}
\mathcal{C}[e] & \rightsquigarrow & \mathcal{C}[e'] & \text{[E-COMPAT]} \\
\text{whenever } e & \to & e'
\end{array}
$$

Contexts $\boxed{\mathcal{C}}$

$$
\begin{array}{rcl}
\mathcal{C} & ::= & \bullet \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid \lambda x{:}S.\, \mathcal{C} \mid \lambda x{:}\mathcal{D}.\, e \\
& & \mid \texttt{case}\ \mathcal{C}\ \texttt{of}\ c\ \bar{x}\ \triangleright\ e\ \texttt{else}\ e \\
& & \mid \texttt{case}\ e\ \texttt{of}\ c\ \bar{x}\ \triangleright\ \mathcal{C}\ \texttt{else}\ e \\
& & \mid \texttt{case}\ e\ \texttt{of}\ c\ \bar{x}\ \triangleright\ e\ \texttt{else}\ \mathcal{C} \\
\\
\mathcal{D} & ::= & x{:}\mathcal{D} \to T \mid x{:}T \to \mathcal{D} \mid \{x{:}B \mid \mathcal{C}\} \\
& & \mid \exists x{:}\mathcal{D}.\, T \mid \exists x{:}T.\, \mathcal{D}
\end{array}
$$

Primitive functions f are presumed to include basic operations of the language, such as boolean and arithmetic operations, in particular addition "+", conjunction "$\wedge$" the `length` function for lists, and a fixpoint operators $\texttt{fix}_T$ for each type T. The rule [E-PRIM] evaluates a primitive application $(f\ e)$ according to predefined δ-reduction rules. For example,

$$
\begin{array}{rcl}
\delta(\texttt{not}, \texttt{true}) & = & \texttt{false} \\
\delta(\wedge, \texttt{false}) & = & \lambda x{:}\texttt{Bool}.\, \texttt{false} \\
\delta(\wedge, \texttt{true}) & = & \lambda x{:}\texttt{Bool}.\, x \\
\delta(\texttt{fix}_T, e) & = & e\ (\texttt{fix}_T\ e).
\end{array}
$$

We define the infinite loop

$$\Omega_T = \texttt{fix}_T\ (\lambda x{:}T.\, x)$$

for each type T. We sometimes omit type subscripts when they can be inferred from the context.

Data constructors, denoted by c, include the integer and boolean constants of the language and list constructors `nil` and `cons`. The semantics of the pattern matching construct

$$\texttt{case}\ e_1\ \texttt{of}\ c\ \bar{x}\ \triangleright\ e_2\ \texttt{else}\ e_3$$

are given by [E-MATCH] and [E-FAIL]. If the value produced by e_1 matches the pattern $c\ \bar{x}$ (the constructor c applied in a curried fashion to the sequence of variables $\bar{x}$), then e_2 is evaluated with appropriate bindings for $\bar{x}$; otherwise e_3 is evaluated. Full pattern matching may be encoded as nested `case` expressions.

The $\lambda^{\exists}$ type language includes dependent function types, $x{:}T_1 \to T_2$, where T_1 is the domain type and the x may occur in the range type T_2. We omit x if it does not occur free in T_2. The language also includes existential types of the form $\exists x : T_1.\, T_2$, which classify terms of type T_2 where x represents some unknown term of type T_1.

In the *contract type* $\{x : B \mid e\}$, the contract e is a predicate (a term of type `Bool`) over the variable x of type B. For example, $\{x{:}\texttt{Int} \mid x \geq 0\}$ represents positive integers, and we abbreviate $\{x : B \mid \texttt{true}\}$ to simply B. Since $\lambda^{\exists}$ lacks polymorphism, in addition to `Bool` and `Int` we also assume a base type of lists of inte-

gers `IntList` to develop our examples; adding other monomorphic data types is straightforward.

2.1 The $\lambda^{\exists}$ Type System

The $\lambda^{\exists}$ type system is summarized in Figure 3. The core typing judgement $\Gamma \vdash e : T$ assigns type T to expression e in typing environment Γ. It is complemented by a well-formedness judgement for types, $\Gamma \vdash T$. As is customary, we apply implicit α-renaming to assure that variables are bound at most once in Γ.

Rules [T-CONST] and [T-PRIM] assign built-in types to constructors c and primitive functions f according to ty. Variables are assigned *self* types by rule [T-VAR], a key feature that we discuss in Section 2.3. Rule [T-FUN], the standard rule for λ-abstractions, assigns to $\lambda x : T_1 . e$ the dependent function type $x : T_1 \rightarrow T_2$, where the body e has type T_2.

Rule [T-APP] for typing a function application $(e_1\ e_2)$ is interesting, since it appears not to support dependent function types at all. We discuss this in detail in Section 2.2 below.

Rule [T-CASE] for $\texttt{case}\ e\ \texttt{of}\ c\ \bar{x}\ \triangleright\ e_1\ \texttt{else}\ e_2$ is rather complex, because it uses only the information contained in the type of c to provide a measure of path-sensitivity while type-checking e_1. The rule first checks that the inspected term e has the appropriate type to be constructed by c. Since e may have an existential type, the rule checks that the type of e is a refinement of the same base type B as the return type of c, possibly with existentially quantified variables $\bar{z} : \bar{T}_z$. Then, under the assumption that the pattern matching succeeds, bound variables $\bar{x}$ are added to the typing environment with their declared types $\bar{T}_x$, according to the type of c. Finally, the contract predicate e' from the type of e, and the predicate e_c from the return type of c are conjoined into the contract type $\exists \bar{z}{:}\bar{T}_z . \{y{:}B \mid e_c \wedge e'\}$ and a fresh variable y with this type is added to the environment. Though y does not occur in e_1, $\bar{x}$ and $\bar{z}$ may occur in e_c and e', respectively, and the new binding captures known relationships between these variables. In this way, the precision of [T-CASE] is determined by the types of constructors. For example, `cons` can be assigned a range of types, including:

$$
\begin{array}{lcl}
\texttt{cons} & : & \texttt{Int} \rightarrow \texttt{IntList} \rightarrow \texttt{IntList} \\
\texttt{cons} & : & \texttt{Int} \rightarrow x{:}\texttt{IntList} \rightarrow \\
 & & \quad \{y{:}\texttt{IntList} \mid \texttt{length}(y) = \texttt{length}(x) + 1\} \\
\texttt{cons} & : & n{:}\texttt{Int} \rightarrow x{:}\texttt{IntList} \rightarrow \\
 & & \quad \{y{:}\texttt{IntList} \mid y = \texttt{cons}\ n\ x\}
\end{array}
$$

In particular, if the constructor c has the precise return type $\{y{:}B \mid y = c\ \bar{x}\}$, then the new binding implies $[y \mapsto c\ \bar{x}]\, e'$. However, if c has a more coarse type, then the naïve approach of adding an assumption such as $e = c\ \bar{x}$ is reasoning noncompositionally using the exact semantics of c, ignoring its type.

Subtyping between well-formed types T_1 and T_2 is defined by the judgement

$$\Gamma \vdash T_1 <: T_2$$

shown in Figure 4. Rule [S-ARROW] is the expected rule for subtyping between function types, with the addition that the parameter x is bound in the environment when determining subtyping between the return types.

Rule [S-BASE] for subtyping between base types invokes a theorem proving oracle for implication between contracts, represented by the judgement $\Gamma \vdash e_1 \Rightarrow e_2$. For reasons of space and modularity, we leave the theorem prover abstract and axiomatize our requirements upon any prover in Section 3.1.

Subtyping between existential types is derived from the logical interpretation of subtyping as "implication" between propositions Our rule to introduce an existential on the left corresponds to the tautology of first-order logic $(\forall x \in A . P(x) \Rightarrow Q) \Rightarrow (\exists x \in A . P(x)) \Rightarrow Q$ where $x \notin fv(C)$. To introduce an existential on the right, [S-WITNESS] requires a witness e of type T_1 for the variable x. If T is a subtype of $[x \mapsto e]T_2$, then we can disregard the identity of the witness and retain only its specification to conclude that T is a subtype of $\exists x{:}T_1 . T_2$.

Corresponding to our intuition, a combination of [S-WITNESS] and [S-BIND] shows that the following reassuringly covariant rule

Figure 3: Type Rules

Typing $\boxed{\Gamma \vdash e : T}$

$$\frac{}{\Gamma \vdash c : ty(c)} \quad \text{[T-CONST]}$$

$$\frac{}{\Gamma \vdash f : ty(f)} \quad \text{[T-PRIM]}$$

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : \mathbf{self}(T, x)} \quad \text{[T-VAR]}$$

$$\frac{\Gamma \vdash T_1 \qquad \Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash (\lambda x{:}T_1 . e) : (x{:}T_1 \rightarrow T_2)} \quad \text{[T-FUN]}$$

$$\frac{\Gamma \vdash e_2 : T_1 \qquad \Gamma \vdash e_1 : T_1 \rightarrow T_2}{\Gamma \vdash e_1\ e_2 : T_2} \quad \text{[T-APP]}$$

$$\frac{\begin{array}{c} \bar{z}, y \notin fv(e_1) \\ ty(c) = \bar{x}{:}\bar{T}_x \rightarrow \{y{:}B \mid e_c\} \\ \Gamma \vdash e : \exists \bar{z}{:}\bar{T}_z . \{y{:}B \mid e'\} \\ \Gamma, \bar{x} : \bar{T}_x, y : \exists \bar{z}{:}\bar{T}_z . \{y{:}B \mid e_c \wedge e'\} \vdash e_1 : T \\ \Gamma \vdash e_2 : T \end{array}}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ c\ \bar{x}\ \triangleright\ e_1\ \texttt{else}\ e_2 : T} \quad \text{[T-CASE]}$$

$$\frac{\Gamma \vdash e : T_1 \qquad \Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2}{\Gamma \vdash e : T_2} \quad \text{[T-SUB]}$$

Well-formed Types $\boxed{\Gamma \vdash T}$

$$\frac{\Gamma \vdash T_1 \qquad \Gamma, x{:}T_1 \vdash T_2}{\Gamma \vdash x{:}T_1 \rightarrow T_2} \quad \text{[WT-ARROW]}$$

$$\frac{\Gamma, x{:}B \vdash e : \texttt{Bool}}{\Gamma \vdash \{x{:}B \mid e\}} \quad \text{[WT-BASE]}$$

$$\frac{\Gamma \vdash T_1 \qquad \Gamma, x{:}T_1 \vdash T_2}{\Gamma \vdash \exists x{:}T_1 . T_2} \quad \text{[WT-EXISTS]}$$

$$
\begin{array}{rcl}
\mathbf{self}(\{x{:}B \mid e'\}, e) & = & \{x{:}B \mid e' \wedge (x =_B e)\} \\
\mathbf{self}(x{:}T_1 \rightarrow T_2, e) & = & x{:}T_1 \rightarrow \mathbf{self}(T_2, e\ x) \\
\mathbf{self}(\exists x{:}T_1 . T_2, e) & = & \exists x{:}T_1 . \mathbf{self}(T_2, e)
\end{array}
$$

$$
\begin{array}{rcl}
\bar{x} & = & x_1, \cdots, x_n \\
\bar{T} & = & T_1, \cdots, T_n \\
\bar{x}{:}\bar{T} \rightarrow T' & = & x_1{:}T_1 \rightarrow \cdots x_n{:}T_n \rightarrow T' \\
\exists \bar{x}{:}\bar{T} . T' & = & \exists x_1{:}T_1 . \cdots \exists x_n{:}T_n . T'
\end{array}
$$

Figure 4: Subtyping

Subtyping $\boxed{\Gamma \vdash T_1 <: T_2}$

$$\frac{\Gamma \vdash T_3 <: T_1 \qquad \Gamma, x{:}T_3 \vdash T_2 <: T_4}{\Gamma \vdash (x{:}T_1 \to T_2) <: (x{:}T_3 \to T_4)} \quad \text{[S-ARROW]}$$

$$\frac{\Gamma, x : B \vdash e_1 \Rightarrow e_2}{\Gamma \vdash \{x{:}B \mid e_1\} <: \{x{:}B \mid e_2\}} \quad \text{[S-BASE]}$$

$$\frac{\Gamma \vdash e : T_1 \qquad \Gamma \vdash T <: [x \mapsto e]\, T_2}{\Gamma \vdash T <: \exists x{:}T_1.\, T_2} \quad \text{[S-WITNESS]}$$

$$\frac{\Gamma, x{:}T_1 \vdash T_2 <: T \qquad x \notin FV(T)}{\Gamma \vdash \exists x : T_1.\, T_2 <: T} \quad \text{[S-BIND]}$$

is admissible:

$$\frac{\Gamma \vdash T_1 <: T_3 \qquad \Gamma, x{:}T_1 \vdash T_2 <: T_4}{\Gamma \vdash \exists x{:}T_1.\, T_2 <: \exists x{:}T_3.\, T_4}$$

2.2 Non-dependent Function Application

As mentioned above, the typing rule for a function application $(e_1\ e_2)$ is extremely simple, and does not refer to dependent types at all. Instead, the combination of existential types and subtyping provides enough power in other areas of the type system that this straightforward rule is sufficiently expressive. In more detail, suppose e_1 has a dependent function type $x : T_1 \to T_2$ and the argument e_2 has type T_1' where $T_1' <: T_1$. Then, we use subtyping to specialize the function type as follows:

$$\begin{array}{lll} (x{:}T_1 \to T_2) & <: & (x{:}T_1' \to T_2) \\ & <: & (x{:}T_1' \to \exists x{:}T_1'.\, T_2) \\ & \equiv & (T_1' \to \exists x{:}T_1'.\, T_2) \quad \text{since } x \notin fv(T_1') \end{array}$$

The new function type $T_1' \to \exists x : T_1'.\, T_2$ is only applicable to terms of type T_1'. More interestingly, its existential return type now internalizes the assumption that x will be of type T_1', so we have a non-dependent function type that precisely characterizes the behavior of the function e_1 on arguments of type T_1', and which does not refer to the exact semantics of the argument e_2.[3]

Based on the above discussion, the following rule (mentioned in the introduction) is admissible:

$$\frac{\Gamma \vdash e_1 : x{:}T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1' \qquad \Gamma \vdash T_1' <: T_1}{\Gamma \vdash e_1\ e_2 : (\exists x{:}T_1'.\, T_2)}$$

2.3 Self Types

To motivate our non-standard rule for variable references, consider the expression $x - x$, where x has type `Int`. We wish to assign this expression the exact type $\{z{:}\texttt{Int} \mid z = 0\}$, based on the following exact type for subtraction:

$$- \ : \ (w{:}\texttt{Int} \to y{:}\texttt{Int} \to \{z{:}\texttt{Int} \mid z = w - y\})$$

[3] Non-dependent type rules for function application in a dependent type system are also used by Harper and Lillibridge [1994] and Dreyer et al. [2003] in ML module systems where computational effects make the standard substitution-based rule unsound. In the latter, a specialization of the technique we present allows the power of dependent function application when the argument is effect-free.

Under the standard rule, whereby a reference to x simply has type `Int`, the type of $x - x$ is given by the judgement

$$x{:}\texttt{Int} \vdash (x - x) \ : \ (\exists w{:}\texttt{Int}.\, \exists y{:}\texttt{Int}.\, \{z{:}\texttt{Int} \mid z = w - y\})$$

which is much too coarse. The type system has lost a key notion of *identity*, that the two variable references in $x - x$ refer to the same variable.

To solve this problem, *selfification* is used to assign to the variable reference x the exact *self* type $\{y : \texttt{Int} \mid y = x\}$, which captures the identity of x, and allows us to derive:

$$\begin{array}{rl} x{:}\texttt{Int} \vdash (x - x) \ : & \exists w{:}\{w{:}\texttt{Int} \mid w = x\}. \\ & \exists y{:}\{y{:}\texttt{Int} \mid y = x\}. \\ & \quad \{z{:}\texttt{Int} \mid z = w - y\} \end{array}$$

which assigns to $x - x$ a subtype of $\{z{:}\texttt{Int} \mid z = 0\}$, as desired. In addition to this example, *self* types are also crucial for achieving path-sensitivity in pattern matching, which requires a notion of identity for the matched expression.

Selfification (or singleton types/kinds) is a powerful technique but interacts in delicate ways with our goal of compositional reasoning. For example, Ou et al. [2004], in their declarative system, give a general selfification rule

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash e : \mathbf{self}(T, e)} \quad \text{[T-SELF]}$$

But this rule is non-compositional as we cannot replace the subderivation $\Gamma \vdash e : T$ with another $\Gamma \vdash e' : T$. In their algorithmic presentation, they give *self* types only to constants and variables. A key point not highlighted, however, is that giving *self* types to variables *is* compositional: Since there are no subderivations, compositionality is vacuous. This insight is instrumental to reconciling compositional reasoning with the precision of selfification. The restriction of selfification to variables enables us to prove the following key compositionality theorem for our system:

THEOREM 1 (COMPOSITIONALITY). *Suppose* $\Gamma \vdash \mathcal{C}[e] : T$ *based on a subderivation* $\Gamma' \vdash e : T'$ *that corresponds to the hole in* $\mathcal{C}$*, and suppose that* $\Gamma' \vdash e' : T'$ *for some* e'*. Then* $\Gamma \vdash \mathcal{C}[e'] : T$.

Proof sketch: By induction on the derivation of $\Gamma \vdash \mathcal{C}[e] : T$. □

3. Exactness

The requirement for compositional reasoning restricts the kinds of type rules our system can include. For example, we have seen that it forbids both the standard rule for dependent function application and the selfification of arbitrary terms. The type system described above carefully circumvents these restrictions to achieve compositional reasoning. We now address the important question of how much precision or expressive power our type system has lost because of its restriction to compositional type rules.

We show that, in fact, our type system is *exact*: it can assign to each program term a type that completely captures the semantics of that term. This property does require that each constant k (that is, primitive functions f and constructors c) has an exact type that completely captures its semantics, that is, $ty(k) <: \mathbf{self}(ty(k), k)$. For the duration of this section, we assume that ty has this property.

Exactness also requires that `case` expressions are "consistent" in a way that can be expected in a full language – the failure branch of each `case` expression is another `case` expression on the same expression, or Ω, and all the branches are disjoint. This condition ensures that sufficient path information is recorded in the environment. Under these assumptions, the exact information provided by types of constants and the *self* types of variables is propagated losslessly by our type system.

THEOREM 2 (EXACTNESS). *If constants have exact types and pattern matching is consistent, then* $\Gamma \vdash e : T$ *implies* $\Gamma \vdash e : \mathbf{self}(T, e)$.

Proof sketch. By induction on the typing derivation, using Lemma 1 from Section 3.1. □

Theorem 2 seems very strong: for an arbitrary term e of type T (where e's subterms may include coarse types), we can assign e the type $\mathbf{self}(T, e)$ that *exactly* captures the semantics of e, all via compositional reasoning. In exploring the conflict upon which this paper is premised, we have developed a compositional type system as powerful as one based on substitution, but without the violation of abstraction. Rather, our type system insists that whatever precision is desired must be explicitly expressed via types.

As an illustration of Theorem 2, consider the type of the fixpoint primitive $\texttt{fix}_T$, which is typically given the inexact type $(T \to T) \to T$. We can make this type exact via selffication. For example, if $T = x{:}T' \to B$, then an exact type for $\texttt{fix}_T$ is $\mathbf{self}(\ (T \to T) \to T,\ \texttt{fix}_T) =$

$$f{:}(T \to T) \to x{:}T' \to \{y{:}B \mid y = \texttt{fix}_T\ f\ x\}$$

For a more interesting example, consider Euclid's algorithm for computing greatest common divisors, where we use `if` expressions to abbreviate pattern matching:

$$\begin{array}{rcl} \texttt{gcd} & = & \texttt{fix}_T\ (\lambda g{:}T.\ \lambda a{:}\texttt{Int}.\ \lambda b{:}\texttt{Int}.\ e) \\ T & = & \texttt{Int} \to \texttt{Int} \to \texttt{Int} \\ e & = & \texttt{if}\ b = 0\ \texttt{then}\ a \\ & & \texttt{else if}\ a > b\ \texttt{then}\ g\ (a - b)\ b \\ & & \texttt{else}\ g\ a\ (b - a) \end{array}$$

If $\texttt{fix}_T$ has the exact type $\mathbf{self}(\ (T \to T) \to T,\ \texttt{fix}_T)$, then the type systems infers the following exact type for `gcd`:

$$\begin{array}{l} \exists h{:}(g{:}T \to a{:}\texttt{Int} \to b{:}\texttt{Int} \to \{r{:}\texttt{Int} \mid r = e\}). \\ \quad a{:}\texttt{Int} \to b{:}\texttt{Int} \to \{r{:}\texttt{Int} \mid r = \texttt{fix}_T\ h\ a\ b\} \end{array}$$

Note that this type includes within contract predicates both fixpoint computations and also the body e of `gcd`. Thus, the type system is essentially *translating* the entire computation into the predicate language.

While this translation ability illustrates the expressiveness of our type system, reasoning about fixpoint computations is notoriously difficult for automated theorem provers.

In a more typical setting where constants are assigned approximate types, the theorem indicates that the type system neither requires nor performs any abstraction itself; instead, the degree of abstraction in its reasoning can be entirely configured by choosing appropriate types for the constants of the language.

A more practical approach is to use the simple type $(T \to T) \to T$ for $\texttt{fix}_T$, and for the programmer to provide an appropriately-precise specification T for the recursive function `gcd`. The specification $T = \texttt{Int} \to \texttt{Int} \to \texttt{Int}$ shown above is rather coarse; a more precise (but not exact) specification is:

$$T = a{:}\texttt{Int} \to b{:}\texttt{Int} \to \{r{:}\texttt{Int} \mid a\ \texttt{mod}\ r = 0\ \wedge b\ \texttt{mod}\ r = 0\}$$

The type system can then verify this specification for `gcd`, with simple proof obligations such as "if $b\ \texttt{mod}\ r = 0$ and $(a - b)\ \texttt{mod}\ r = 0$ then $a\ \texttt{mod}\ r = 0$."

3.1 Type Safety

We now prove type safety for the exact variant of $\lambda^{\exists}$, which we will leverage to prove type safety in general. For flexibility, we parameterize our type system with respect to constructors, primitive functions, and the theorem proving oracle, making only minimal assumptions necessary to ensure soundness (most instances will have many more interesting properties).

ASSUMPTION 1. *We require of the implication relation:*

1. *(Weakening) If* $\Gamma_1, \Gamma_2 \vdash e \Rightarrow e'$ *then* $\Gamma_1, \Gamma', \Gamma_2 \vdash e \Rightarrow e'$ *where* $dom(\Gamma')$ *is disjoint from* $fv(e)$, $fv(e')$, $dom(\Gamma_1)$, *and* $dom(\Gamma_2)$.
2. *(Narrowing) If* $\Gamma_1, x : T, \Gamma_2 \vdash e \Rightarrow e'$ *and* $\Gamma_1 \vdash T' <: T$ *then* $\Gamma_1, x : T', \Gamma_2 \vdash e \Rightarrow e'$
3. *(Transitivity) If* $\Gamma \vdash e_1 \Rightarrow e_2$ *and* $\Gamma \vdash e_2 \Rightarrow e_3$ *then* $\Gamma \vdash e_1 \Rightarrow e_3$
4. *(Reflexivity)* $\Gamma \vdash e \Rightarrow e$
5. *(Faithfulness)* $\Gamma \vdash e \Rightarrow \texttt{true}$
6. *(Consistency)* $\Gamma \nvdash \texttt{true} \Rightarrow \texttt{false}$
7. *(Conjunction 1)* $\Gamma \vdash e_1 \wedge e_2 \Rightarrow e_1$
8. *(Conjunction 2)* $\Gamma \vdash e_1 \wedge e_2 \Rightarrow e_2$
9. *(Exact Quantification) If* $x : \mathbf{self}(T, e) \in \Gamma$ *then* $\Gamma \vdash p \Rightarrow [x \mapsto e]p$
10. *(Evaluation) If* $e \rightsquigarrow^* e'$ *then* $\Gamma \vdash e \Rightarrow e'$
11. *(Substitution) If* $\Gamma_1, x : T, \Gamma_2 \vdash e_1 \Rightarrow e_2$ *and* $\Gamma_1 \vdash e_3 : T$ *then* $\Gamma_1, \theta\Gamma_2 \vdash \theta e_1 \Rightarrow \theta e_2$ *and* $\Gamma, x : S, F \vdash e_1 \Rightarrow \theta e_1$ *where* $\theta = [x \mapsto e_3]$

ASSUMPTION 2. *We require of each primitive function* f*:*

1. f *has a well-formed type:* $\emptyset \vdash ty(f)$
2. f *cannot get stuck: if* $\emptyset \vdash f\ v : T$ *then* $\delta(f, v)$ *is defined.*
3. f *satisfies preservation: if* $\emptyset \vdash f\ e : T$ *and* $\delta(f, e)$ *is defined, then* $\emptyset \vdash \delta(f, e) : T$

Given exactness, an existentially quantified variable with a singleton type is equivalent to having simply performed a substitution (it is essentially an explicit substitution [Abadi et al. 1990]). This intuition is formalized in the following lemma.

LEMMA 1. *For any expression* e *and type* T*, if* $\Gamma \vdash e : T_e$ *and* $\Gamma, x{:}T_e \vdash T$ *then*

1. $\Gamma \vdash \mathbf{self}(T_e, e) <: T_e$
2. $\Gamma \vdash [x \mapsto e]\,T <: \exists x{:}\mathbf{self}(T_e, e).\,T$
3. $\Gamma \vdash \exists x{:}\mathbf{self}(T_e, e).\,T <: [x \mapsto e]\,T$

Proof Sketch. By induction on the size of T, using Assumption 1 (Exact Quantification). □

The standard substitution lemma follows from Theorem 2 and Lemma 1.

LEMMA 2 (SUBSTITUTION). *Suppose* $\Gamma_1 \vdash e_1 : T_1$ *and* $\Gamma = \Gamma_1, x{:}T_1, \Gamma_2$ *and* $\Gamma' = \Gamma_1, \theta\Gamma_2$ *where* $\theta = [x \mapsto e]$*. Then*

1. *If* $\Gamma \vdash e : T$ *then* $\Gamma' \vdash \theta e : \theta T$
2. *If* $\Gamma \vdash T$ *then* $\Gamma' \vdash \theta T$
3. *If* $\Gamma \vdash T <: T'$ *then* $\Gamma' \vdash \theta T <: \theta T'$

Proof By mutual induction on the derivations. Since variables are assigned self types, it is crucial that the substituted term also have a self type. □

Progress and preservation can now be proved for the exact variant; the proofs follow the same structure as in Flanagan [2006].

THEOREM 3 (PROGRESS). *If* $\emptyset \vdash e : T$ *then either there exists some* e' *such that* $e \rightsquigarrow e'$*, or* e *is a value.*

THEOREM 4 (PRESERVATION). *If* $\Gamma \vdash e : T$ *and* $e \rightsquigarrow e'$ *then* $\Gamma \vdash e' : T$

If a term is well-typed in any variant of our system, then it is certainly well-typed in the exact variant. Hence, a well-typed term cannot "get stuck" even in those imprecise variants where the substitution lemma cannot be proved.

Figure 5: Syntax for Algorithmic Typing

$l \subseteq e$		*Linear Inequalities*
$p ::=$		*Predicate Terms*
	l	atomic inequality
	$p \wedge p$	conjunction
	$\texttt{case}\ w\ \texttt{of}\ c\,\bar{x}\ \triangleright\ p\ \texttt{else}\ p$	case
$S ::=$		*Restricted Types:*
	$\{x\!:\!B \mid p\}$	refinement type
	$x\!:\!S \to S$	restricted function type
$A ::=$		*Augmented Types:*
	$\{x\!:\!B \mid p\}$	refinement type
	$x\!:\!S \to A$	augmented function type
	$\exists x\!:\!A.\ A$	augmented existential type
$\Delta ::=$		*Algorithmic Typing Environments:*
	$\emptyset$	empty environment
	$\Delta, x : A$	environment extension

THEOREM 5. *For any definition of ty satisfying Assumption 2, if* $\emptyset \vdash e : T$*, then* e *either reduces to a value or is nonterminating.*

Proof sketch: Map the derivation of $\Gamma \vdash e : T$ into the exact variant by induction over the derivation, then apply progress and preservation. □

There may be another syntactic approach to type soundness that applies in these systems, but it is more conceptually informative to think of those systems as coarsenings of the exact variant.

4. Algorithmic Typing

We now investigate how to provide decidable compositional type checking for an interesting subset of $\lambda^{\exists}$. Type checking for $\lambda^{\exists}$ is undecidable, despite the recovery of compositional reasoning, because implication remains undecidable. However, by restricting contract predicates to a decidable theory, we can now provide a decidable, compositional, dependent type system. Note that this decidability result crucially relies on compositional reasoning: in a traditional dependent type system, even when all annotations fall within a decidable theory, substitutions made during type checking may result in proof obligations outside of that theory.

Figure 5 defines the type language for which we provide a type-checking algorithm. For concreteness, we suppose contracts are linear inequalities, but the approach is applicable to other decidable predicate languages. During type-checking, they will be combined with conjunction and case-splitting on variables, so those are also included in the predicate sublanguage of terms, denoted by metavariables p or q.

We distinguish two sublanguages of types. The first, ranged over by metavariable S, does not include existential quantifiers and may appear in source programs. Augmented types A generated during type checking may contain existential quantification, but only in positive positions. Thus $S \subseteq A \subseteq T$. Only augmented types A are bound in the environment during type checking, so we use a new metavariable Δ to range over these A-environments.

Our algorithmic type checking judgement $\Delta \models\!\Rightarrow e : A$ is shown in Figure 6. As usual for syntax directed type checking, we remove the subsumption rule and instead inline subtyping where needed. For example, in the case for applications, the existential quantifiers are introspected to discover the underlying function type $x\!:\!S_1 \to A_1$, and then the type A_2 of the argument is checked for compatibility with S_1.

In typing a pattern matching expression, we lift the `case` syntax to types to concisely express the resulting type; intuitively, $\texttt{case}\ e\ \texttt{of}\ c\,\bar{x}\ \triangleright\ A_1\ \texttt{else}\ A_2$ is equivalent to A_1 if the pattern match succeeds, otherwise is equivalent to A_2. Figure 6 includes the exact definition. To avoid arbitrary expressions appearing in contract predicates, we require that the inspected subexpression in a `case` construct be a variable. This requirement can be satisfied by rewriting the more general form $(\texttt{case}\ e\ \texttt{of}\ c\,\bar{x}\ \triangleright\ e_1\ \texttt{else}\ e_2)$ into the semantically equivalent $((\lambda y\!:\!S.\ \texttt{case}\ y\ \texttt{of}\ c\,\bar{x}\ \triangleright\ e_1\ \texttt{else}\ e_2)\ e)$. This rewriting step introduces a type S for the variable y, and so ensures that any expression used in pattern matching will have a specification that falls in the restricted type language S.

Rule [A-VAR-BASE] uses the function $\mathbf{refine}$ to selfify variables. The function $\mathbf{refine}$ is like $\mathbf{self}$ except it avoids introducing function applications, which are no longer permitted in our contract language.

Like typing, algorithmic type checking relies on subtyping, but interestingly only for the asymmetric form $\Delta \models\!\Rightarrow A <: S$, which checks subtyping between an inferred, augmented type A and a programmer-specified, restricted type S. Since existential types never appear on the right hand side, we (fortunately!) never need to "guess" witnesses for existentials. Existentials on the left are unproblematic, because the existential quantification in a negative context transforms into a universal quantification in a positive context, which is handled algorithmically. We could add the covariant rule from the end of section 2.2 to allow some existential quantification on the right side of subtyping, but our syntactic conditions make that unnecessary.

Algorithmic typeability is not closed under evaluation; instead soundness is provided by mapping algorithmic derivations onto derivations in the full type system.

THEOREM 6 (SOUNDNESS OF ALGORITHMIC TYPING).
If $\Delta \models\!\Rightarrow e : A$ *and pattern matching is consistent, then* $\Delta \vdash e : A$

Proof sketch: By induction on the derivation of $\Delta \models\!\Rightarrow e : A$. □

While the algorithm is not complete with respect to the full type system, it is complete for the class of sublanguages we have described.

THEOREM 7 (RELATIVE COMPLETENESS).
If $\Delta \vdash e : T$ *and* e *is annotated with restricted types, and pattern matching is consistent and inspects only variables, then there exists an* A *such that* $\Delta \models\!\Rightarrow e : A$ *and* $\Delta \vdash A <: T$

Proof sketch: By induction on the derivation of $\Delta \vdash e : T$. □

These algorithmic rules characterize a decidable class of type systems for the core language: If all program annotations are restricted types whose contracts are expressions in some decidable theory (where this means that queries of the form $\Delta \models\!\Rightarrow q \Rightarrow p$ are decidable, such as linear inequalities with conjunction and case splits), then all implication queries also fall in this theory, and are decidable. Interestingly, because existentials only appear as assumptions in proof obligations, the theory need only support a restricted form of existential quantification that is "productive" in that each existential corresponds to positive existence of some term, and there is never any existential proof obligation.

5. Binary Search Trees

As an illustration of our decidable type system, we now discuss a somewhat more involved example of binary search trees. We assume an additional base type `BST` to represents binary search

Figure 6: Algorithmic Rules

$\boxed{\Delta \models e : A}$

[A-CONST] $\dfrac{}{\Delta \models c : ty(c)}$ [A-PRIM] $\dfrac{}{\Delta \models f : ty(f)}$ [A-VAR-BASE] $\dfrac{(x : A) \in \Delta}{\Delta \models x : \mathbf{refine}(A, x)}$ [A-FUN] $\dfrac{\Delta \models S \qquad \Delta, x{:}S \models e : A}{\Delta \models (\lambda x{:}S.\, e) : (x{:}S \to A)}$

[A-APP] $$\frac{\Delta \models e_1 : \exists \bar{z}{:}\bar{T}_z.\, x{:}S_1 \to A_1 \qquad \Delta \models e_2 : A_2 \qquad \Delta, \bar{z}{:}\bar{T}_z \models A_2 <: S_1}{\Delta \models e_1\ e_2 : \exists \bar{z}{:}\bar{T}_z.\, \exists x{:}A_2.\, A_1}$$

[A-CASE] $$\frac{\begin{array}{c}\bar{z}, y \notin fv(e_1) \qquad ty(c) = \bar{x}{:}\bar{S} \to \{y{:}B \mid p_c\} \\ \Delta \models w : \exists \bar{z}{:}\bar{T}_z.\, \{y{:}B \mid p_y\} \qquad \Delta, \bar{x}{:}\bar{S}, y{:}\exists \bar{z}{:}\bar{T}_z.\, \{y{:}B \mid p_c \wedge p_y\} \models e_1 : A_1 \qquad \Delta \models e_2 : A_2\end{array}}{\Delta \models (\mathtt{case}\ w\ \mathtt{of}\ c\ \bar{x} \triangleright e_1\ \mathtt{else}\ e_2) : (\mathtt{case}\ w\ \mathtt{of}\ c\ \bar{x} \triangleright A_1\ \mathtt{else}\ A_2)}$$

$\boxed{\Delta \models S}$

[AT-ARROW] $\dfrac{\Delta \models S_1 \qquad \Delta, x{:}S_1 \models S_2}{\Delta \models x{:}S_1 \to S_2}$ [AT-BASE] $\dfrac{\Delta, x : B \models p : \mathtt{Bool}}{\Delta \models \{x{:}B \mid p\}}$

$\boxed{\Delta \models A <: S}$

[AS-ARROW] $\dfrac{\Delta \models S_2 <: S_1 \qquad \Delta, x : S_2 \models A_1 <: S_3}{\Delta \models (x{:}S_1 \to A_1) <: (x{:}S_2 \to S_3)}$ [AS-BASE] $\dfrac{\Delta, x : B \vdash q \Rightarrow p}{\Delta \models \{x{:}B \mid q\} <: \{x{:}B \mid p\}}$ [AS-BIND] $\dfrac{\Delta, x{:}A_1 \models A_2 <: S_3}{\Delta \models \exists x{:}A_1.\, A_2 <: S_3}$

$$\begin{array}{rcl}\mathbf{refine}(\{x{:}B \mid e'\}, e) & = & \{x{:}B \mid e' \wedge (x =_B e)\} \\ \mathbf{refine}(x{:}T_1 \to T_2, e) & = & x{:}T_1 \to T_2 \\ \mathbf{refine}(\exists x{:}T_1.\, T_2, e) & = & \exists x{:}T_1.\, \mathbf{refine}(T_2, e)\end{array}$$

$$\begin{array}{rcl}\mathtt{case}\ w\ \mathtt{of}\ c\ \bar{x} \triangleright \exists y{:}A_1.\, A_2\ \mathtt{else}\ A_3 & = & \exists y{:}A_1.\, (\mathtt{case}\ w\ \mathtt{of}\ c\ \bar{x} \triangleright A_2\ \mathtt{else}\ A_3) \\ \mathtt{case}\ w\ \mathtt{of}\ c\ \bar{x} \triangleright A_1\ \mathtt{else}\ \exists y{:}A_2.\, A_3 & = & \exists y{:}A_2.\, (\mathtt{case}\ w\ \mathtt{of}\ c\ \bar{x} \triangleright A_1\ \mathtt{else}\ A_3) \\ \mathtt{case}\ w\ \mathtt{of}\ c\ \bar{x} \triangleright \{y{:}B \mid p_1\}\ \mathtt{else}\ \{y{:}B \mid p_2\} & = & \{y{:}B \mid \mathtt{case}\ w\ \mathtt{of}\ c\ \bar{x} \triangleright p_1\ \mathtt{else}\ p_2\} \\ \mathtt{case}\ w\ \mathtt{of}\ c\ \bar{x} \triangleright y{:}S_1 \to A_1\ \mathtt{else}\ y{:}S_2 \to A_2 & = & y{:}(\mathtt{case}\ w\ \mathtt{of}\ c\ \bar{x} \triangleright S_1\ \mathtt{else}\ S_2) \to \mathtt{case}\ w\ \mathtt{of}\ c\ \bar{x} \triangleright A_1\ \mathtt{else}\ A_2\end{array}$$

trees, with auxiliary functions:

$$\begin{array}{rcl}\mathtt{lower} & : & \mathtt{BST} \to \mathtt{Int} \\ \mathtt{upper} & : & \mathtt{BST} \to \mathtt{Int}\end{array}$$

that return the lower and upper bounds of integers in a BST, and return `maxInt`and `minInt`, respectively, on empty BSTs. We also assume some additional constants and primitive functions:

$$\begin{array}{rcl}\mathtt{minInt} & : & \mathtt{Int} \\ \mathtt{maxInt} & : & \mathtt{Int} \\ \mathtt{min} & : & x{:}\mathtt{Int} \to y{:}\mathtt{Int} \to \{z{:}\mathtt{Int} \mid z \le x \wedge z \le y\} \\ \mathtt{max} & : & x{:}\mathtt{Int} \to y{:}\mathtt{Int} \to \{z{:}\mathtt{Int} \mid z \ge x \wedge z \ge y\}\end{array}$$

The type of a binary search tree with integers in the range $[lo, hi)$ is defined by the contract type:

$$\mathtt{BST}_{lo,hi} \ = \ \{x{:}\mathtt{BST} \mid \ lo \le \mathtt{lower}(x) \wedge \mathtt{upper}(x) < hi\ \}$$

Binary search tree are created using the constructors `empty` and `node`, which are assigned the following precise types:

$$\begin{array}{rcl}\mathtt{empty} & : & lo{:}\mathtt{Int} \to hi{:}\mathtt{Int} \to \mathtt{BST}_{lo,hi} \\ \mathtt{node} & : & lo{:}\mathtt{Int} \to hi{:}\mathtt{Int} \to \\ & & v{:}\{v{:}\mathtt{Int} \mid lo \le v < hi\} \to \\ & & x{:}\mathtt{BST}_{lo,v} \to y{:}\mathtt{BST}_{v,hi} \to \mathtt{BST}_{lo,hi}\end{array}$$

Here, these constructors take additional "index" arguments lo and hi, and so we are using an index-type-like implementation of binary search trees. Using these constructors, we can define the `insert` operation on `BST`s:

$$\begin{array}{l}\mathtt{fix}_T(\lambda f{:}T. \\ \quad \lambda lo{:}\mathtt{Int}.\, \lambda hi{:}\mathtt{Int}. \\ \quad \lambda v{:}\{y{:}\mathtt{Int} \mid lo \le y < hi\}. \\ \quad \lambda x{:}\mathtt{BST}_{lo,hi}. \\ \quad\quad \mathtt{case}\ x\ \mathtt{of}\ \mathtt{empty}\ lo\ hi \triangleright \\ \quad\quad\quad (\mathtt{node}\ lo\ hi\ v\ (\mathtt{empty}\ lo\ v)\ (\mathtt{empty}\ v\ hi))\ \mathtt{else} \\ \quad\quad \mathtt{case}\ x\ \mathtt{of}\ (\mathtt{node}\ lo\ hi\ n\ l\ r) \triangleright \\ \quad\quad\quad (\mathtt{case}\ v < n\ \mathtt{of} \\ \quad\quad\quad\quad \mathtt{true} \triangleright (\mathtt{node}\ lo\ hi\ n\ (\mathtt{insert}\ lo\ n\ x\ l)\ r) \\ \quad\quad\quad\ \mathtt{else}\ \mathtt{case}\ v < n\ \mathtt{of} \\ \quad\quad\quad\quad \mathtt{false} \triangleright (\mathtt{node}\ lo\ hi\ n\ l\ (\mathtt{insert}\ n\ hi\ x\ r)) \\ \quad\quad\quad\ \mathtt{else}\ \Omega) \\ \quad\quad \mathtt{else}\ \Omega)\end{array}$$

where `insert` has type

$$\begin{array}{rcl}T & = & lo{:}\mathtt{Int} \to hi{:}\mathtt{Int} \to \\ & & v{:}\{v{:}\mathtt{Int} \mid lo \le v < hi\} \to \\ & & \mathtt{BST}_{lo,hi} \to \mathtt{BST}_{lo,hi}\end{array}$$

All the proof obligations generated for this program are formulae over linear integer inequalities, and hence decidable.

In this example, the variables lo and hi are somewhat awkward, and provide an indirect specification of the behavior of `insert`. We can express this program more naturally by removing these

parameters instead expressing the key data invariants directly in terms of the underlying data structure. In this formulation, the BST constructors have the more natural types:

$$\begin{array}{rcl} \texttt{empty} & : & \texttt{BST}_{\texttt{maxInt},\texttt{minInt}} \\ \texttt{node} & : & v{:}\texttt{Int} \rightarrow \\ & & x{:}\{x{:}\texttt{BST} \mid \texttt{upper}(x) < v\} \rightarrow \\ & & y{:}\{y{:}\texttt{BST} \mid v \leq \texttt{lower}(y)\} \rightarrow \\ & & \texttt{BST}_{\texttt{lower}(x),\texttt{upper}(y)} \end{array}$$

The revised `insert` implementation is identical to the one shown above but elides index variables:

$$\begin{array}{l} \texttt{fix}_{T'}(\lambda f{:}T'.\ \ \lambda v{:}\texttt{Int}.\ \ \lambda x{:}\texttt{BST}. \\ \quad \texttt{case}\ x\ \texttt{of}\ \texttt{empty} \triangleright (\texttt{node}\ v\ \texttt{empty}\ \texttt{empty})\ \texttt{else} \\ \quad \texttt{case}\ x\ \texttt{of}\ (\texttt{node}\ n\ l\ r) \triangleright \\ \qquad (\texttt{case}\ v < n\ \texttt{of} \\ \qquad\quad \texttt{true} \triangleright (\texttt{node}\ n\ (\texttt{insert}\ x\ l)\ r) \\ \qquad \texttt{else case}\ v < n\ \texttt{of} \\ \qquad\quad \texttt{false} \triangleright (\texttt{node}\ n\ l\ (\texttt{insert}\ x\ r)) \\ \qquad \texttt{else}\ \Omega) \\ \quad \texttt{else}\ \Omega) \end{array}$$

and has the following type T':

$$\begin{array}{rcl} T' & = & v{:}\texttt{Int} \rightarrow x{:}\texttt{BST} \rightarrow \\ & & \texttt{BST}_{\texttt{min}(\texttt{lower}(x),v),\texttt{max}(\texttt{upper}(x),v)} \end{array}$$

6. Related Work

First, we present a high-level comparison of contract types with indexed types, an alternative approach towards similar goals. We next survey other applications of existential quantification in type systems. Finally, we briefly outline the development of refinement and contract types which this work directly builds upon.

6.1 Indexed Types

One solution, adopted by Dependent ML [Xi and Pfenning 1999], ATS [Cui et al. 2005], and Ωmega [Sheard 2005], is to distinguish compile-time from run-time data, and allow types to depend only on compile-time data. This approach makes compositionality a vacuous proposition, and indeed can sometimes be encoded in existing polymorphic type systems without use of dependent types [Zenger 1997; McBride 2002].

A concrete technique that is common in all of the above systems is the use of *indexed types*. An illustrative example is the family of types $\texttt{IntList}_n$, where n indicates the length of lists inhabiting the type. The types of list constructors and the `append` function are as follows. We use the type $\mathbb{N}$ to distinguish the compile-time type of natural numbers.

$$\begin{array}{rcl} \texttt{nil} & : & \texttt{IntList}_0 \\ \texttt{cons} & : & n{:}\mathbb{N} \rightarrow \texttt{Int} \rightarrow \texttt{IntList}_n \rightarrow \texttt{IntList}_{n+1} \\ \texttt{append} & : & m{:}\mathbb{N} \rightarrow n{:}\mathbb{N} \rightarrow \\ & & \quad \texttt{IntList}_m \rightarrow \texttt{IntList}_n \rightarrow \texttt{IntList}_{m+n} \end{array}$$

The connection between run-time invariants and compile-time data is based essentially on injecting an abstraction of run-time data into compile-time data; for example lists indexed by their length use a compile-time copy of the natural numbers. The expressions that are reflected into indices can be carefully controlled to ensure that type compatibility remains decidable.

We can naturally express the indexed type $\texttt{IntList}_n$ as the contract type $\{x{:}\texttt{IntList} \mid \texttt{length}(x) = n\}$, and the above types and all proof obligations remain equivalent. In this way, by reifying the abstraction function from a value to its associated index, arbitrary indexed types can be embedded into general contract types. The abstraction function may be treated as an uninterpreted symbol – its definition cannot be necessary in proof obligation since it is not even available to indexed types.

The above type for `append` is somewhat awkward, though, as it requires the additional index parameters n and m.[4] In contrast, general contract types allow a more natural expression of the same specification for `append`, that eliminates these index parameters:

$$\begin{array}{l} x{:}\texttt{IntList} \rightarrow y{:}\texttt{IntList} \rightarrow \\ \quad \{z{:}\texttt{IntList} \mid \texttt{length}(z) = \texttt{length}(x) + \texttt{length}(y)\} \end{array}$$

More critical than the aesthetic issue of index parameters, the index of a data structure, decided by its implementor, determines which properties may be reasoned about, inhibiting reuse and composition of data structures. For example, if one is interested in verifying the ordering of a list, rather that its length, then a different index is required, specifically the minimum element (at the head of the list) to specify `cons`, and also the maximum element (at the tail) to specify `append`. We also need integers indexed by their exact values $\texttt{Int}_i$, where i is drawn from compile-time integers $\mathbb{Z}$ (with $\pm\infty$ for corner cases). We define the type $\texttt{IntList}_{i,j}$ of lists in ascending order with minimum element i and maximum element j using the following constructors. Note that predicate subtypes are used on indices of compile-time type $\mathbb{Z}$ but not on runtime terms.

$$\begin{array}{rcl} \texttt{nil} & : & \texttt{IntList}_{\infty,-\infty} \\ \texttt{cons} & : & i{:}\mathbb{Z} \rightarrow j{:}\{j{:}\mathbb{Z} \mid j \geq i\} \rightarrow k{:}\mathbb{Z} \rightarrow \\ & & \quad \texttt{Int}_i \rightarrow \texttt{IntList}_{j,k} \rightarrow \texttt{IntList}_{i,\texttt{max}(i,k)} \\ \texttt{append} & : & i{:}\mathbb{Z} \rightarrow j{:}\mathbb{Z} \rightarrow k{:}\{k{:}\mathbb{Z} \mid k \geq j\} \rightarrow l{:}\mathbb{Z} \rightarrow \\ & & \quad \texttt{IntList}_{i,j} \rightarrow \texttt{IntList}_{k,l} \rightarrow \texttt{IntList}_{\texttt{min}(i,k),l} \end{array}$$

In general, since the index of a data type determines the properties that may be reasoned about, more complex properties require embedding of more data as indices. In the limit, giving the precise type $x{:}\texttt{IntList} \rightarrow \{y{:}\texttt{IntList} \mid y = x\}$ to the identity function on lists (or any type) requires embedding the entire type of lists (or any type) into the index language, which reduces the utility of the syntactic distinction of compile-time data.

In contrast general contract types, can naturally express a wide variety of refinements over the same `IntList` type, such as $\{x : \texttt{IntList} \mid \texttt{minElem}(x) = i \wedge \texttt{maxElem}(x) = j\}$ or the most natural $\{x : \texttt{IntList} \mid \texttt{isSorted}(x)\}$. The proof obligations for the latter may be more problematic, and the current work is progress towards characterizing those situations where they are not difficult.

6.2 Existential Types

Formal existential quantification in type-theory has found a variety of uses, including closure conversion Morrisett et al. [1999]; Grossman [2002], module systems [Mitchell and Plotkin 1988; Harper and Lillibridge 1994; Dreyer et al. 2003], and semantics of object orientation Bruce et al. [1999]. Most similar to the present work are ML module systems, which also combine it with subtyping and a different (formal) form of singleton types.

The standard substitution-based typing rule for function application is unsound in the presence of effects, so dependencies have to be "forgotten" [Harper and Lillibridge 1994]. Dreyer et al. [2003] ameliorate this restriction with a sophisticated effect system that restores the power of dependent application in the event that the argument to a function is effect-free.

Also similar is the need to express sharing constraints. Just as we give self types to variables, Stone and Harper [2000] and subsequently Dreyer et al. [2003] assign singleton kinds (a special syntactic form) to modules in order to preserve information when applying a dependently-kinded functor. In contrast to module languages, we examine the consequences and form of existentials and selfification in the context of dependent contracts with pattern matching and automatic theorem proving. Our singletons are expressed in the existing type language rather than adding a special form, and in our setting a type of a variable x may constrain other

[4] Note that these index parameters can be inferred in many cases.

bound variables so we maintain this information while adding the identity information to the known type of x.

Our different approaches and goals led us to emphasize compositional reasoning as a key aspect of our system, which we present in a more minimal calculus than the large and practical ML module type systems. Compositionality is obviously important in their work as well, since it affects separate (re-)compilation.

Our existential types are a limited expression of Σ types for dependent pairs in powerful type theories such as those underlying Coq [The Coq development team 2004], Hoare Type Theory [Nanevski et al. 2006], and Epigram [McBride and McKinna 2004]. Dependent ML, in fact, makes use of types such as $\Sigma x{:}T_1.T_2$ with explicit introduction and elimination forms (i.e. without subtyping) to allow functions operating over arguments of unknown index.

With respect to Σ types, our work can be interpreted as a way of providing automation for a simple and common case where the full power of these type theories is not necessary.

6.3 Refinement and Contract Types

Much of the work we build upon directly is discussed in Sections 1, 2, and 6.1, so we present only a brief survey here. Freeman and Pfenning introduced *datasort refinements*, which express restrictions on the recursive structure of algebraic datatypes [1991], but our refinement types are most similar to those of Denney [1998], Ou et al. [2004] and those from the Hybrid Type Checking (HTC) work of Flanagan [2006], Gronski et al. [2006], and Knowles and Flanagan [2007]. As in HTC, type checking for $\lambda^{\exists}$ is undecidable, but in HTC, run-time checks are inserted into a program when an implication condition can be neither proven nor refuted, resulting in unpredictable coverage and run-time costs. Instead, we give a predictable type checking algorithm for a clear sublanguage of $\lambda^{\exists}$. In some sense, the present work characterizes when runtime checks (or manual proof) are unnecessary in the mentioned systems.

Another system with a similar language for expressing specifications, but an entirely different formal approach, is ESC/Haskell Xu [2006]; Xu et al. [2007]. which uses symbolic evaluation to prove that a program obeys its (dependent) contract. Since ESC/Haskell inlines functions and reasons about their body directly, rather than using their specification, it does not address issues of compositionality. However, ESC/Haskell gives a much more thorough treatment to the notion of blame, which is integral to a complete contract system.

7. Conclusions and Future Work

While dependent types provide a elegant foundation for many expressive type systems, they are traditionally based on non-compositional reasoning. We have shown how to restore compositional reasoning for dependent contract types using a combination of existential types and subtyping.

The precision of any compositional analysis is naturally driven by the precision of the abstractions that it composes, and careful choice of abstractions is crucial for achieving precise, scalable analyses. Our type system is entirely configurable in this regard; it does not perform any abstraction itself, and instead simply propagates the abstractions inherent in the types of constants and recursive functions, with the result that the precision of the analysis is largely under the control of the programmer. In the extreme, our type system can infer for each term an exact type that completely captures the semantics of that term. Exploring this property of the type system using the denotational semantics of contracts [Blume and McAllester 2006] may yield more satisfying theorems in the case that constants have less precise types. More generally, we are interested in exploring deeper connections between our definition of compositionality and denotational semantics, such as abstract interpretation.

Compositionality provides important practical benefits in the form of predictability in the case of dependent contract types. For the important special case of programs whose contracts fall within a decidable theory, our type checking algorithm relies on compositional reasoning to achieve the key invariant that all proof obligations generated during type checking also lie within this theory. The analysis of programs whose specifications do not fall within a decidable theory remains an important open area not addressed by our work.

References

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitution. In *Symposium on Principles of Programming Languages*, pages 31–46, 1990.

L. Augustsson. Cayenne — a language with dependent types. In *Proceedings of the ACM International Conference on Functional Programming*, pages 239–250, 1998. ISBN 1-58113-024-4.

M. Blume and D. McAllester. Sound and complete models for contracts. *Journal of Functional Programming*, 16:375 – 414, July 2006.

K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, 1999.

S. Cui, K. Donnelly, and H. Xi. ATS: A Language That Combines Programming with Theorem Proving. In *Proceedings of the 5th International Workshop on Frontiers of Combining Systems*, pages 310–320, Vienna, Austria, September 2005.

E. Denney. Refinement types for specification. In *Proceedings of the IFIP International Conference on Programming Concepts and Methods*, volume 125, pages 148–166. Chapman & Hall, 1998. ISBN 0-412-83760-9.

D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *Symposium on Principles of Programming Languages*, pages 236 – 249, 2003.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the International Conference on Functional Programming*, pages 48–59, 2002.

C. Flanagan. Hybrid type checking. In *Symposium on Principles of Programming Languages*, pages 245 – 256, 2006.

T. Freeman and F. Pfenning. Refinement types for ML. In *Conference on Programming Language Design and Implementation*, pages 268–277, 1991.

J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Practical hybrid checking for expressive types and specifications. In *Proceedings of the Workshop on Scheme and Functional Programming*, pages 93–104, 2006.

D. Grossman. Existential types for imperative languages. In *European Symposium on Programming*, pages 85–120, 2002.

R. Harper and M. Lillibridge. A type theoretic approach to higher-order modules with sharing. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 123–137, 1994.

K. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *European Symposium on Programming*, 2007.

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

C. McBride. Faking it: Simulating dependent types in haskell. *Journal of Functional Programming*, 12(4-5):375–392, 2002.

C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *Transactions on Programming Languages*, 10(3):470 – 502, 1988.

G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *International Conference on Functional Programming*, pages 62–73, 2006.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineer-

ing, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *Proceedings of the IFIP International Conference on Theoretical Computer Science*, pages 437–450, 2004.

J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: predicate subtyping in pvs. *IEEE Transactions on Software Engineering*, 24(9): 709–720, 1998.

T. Sheard. Putting curry-howard to work. In *Proceedings of the workshop on Haskell*, pages 74–85, 2005.

C. A. Stone and R. Harper. Deciding type equivalence in a language with singleton kinds. In *Symposium on Principles of Programming Languages*, pages 214 – 227, 2000.

H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL '99: 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227. ACM Press, 1999.

D. N. Xu. Extended static checking for haskell. In *Proceedings of the workshop on Haskell*, pages 48 – 59, 2006.

D. N. Xu, S. P. Jones, and K. Claessen. Static contract checking for haskell. In *Draft Proceedings of the International Symposium on Implementation and Application of Functional Languages*, pages 382 – 399, 2007.

C. Zenger. Indexed types. *Theoretical Computer Science*, 187(1–2):147–165, 1997.