Deciding Joinability Modulo Ground Equations in Operational Type Theory

Adam Petcher[1] and Aaron Stump[2]

[1] Washington University in Saint Louis adampetcher@yahoo.com
[2] The University of Iowa astump@acm.org

Abstract. Operational Type Theory (OpTT) is a theory of joinability of untyped functional terms including general recursion, pattern-matching and inductive datatypes, and higher-order functions. An algorithm is presented that can be used to automatically generate proofs of equality in OpTT. The algorithm will equate two supplied terms if and only if there exists an OpTT proof that can equate the two terms using only the proof rules related to evaluation under the operational semantics, symmetry, transitivity, and congruence with respect a set of supplied ground equations. The algorithm is proved sound, and under some natural assumptions about the termination behavior of the functions, also complete.

1 Introduction

Operational Type Theory(OpTT)[1] is a system which combines a dependently typed, pure functional programming language with a distinct proof language that can be used to specify and verify properties related to the untyped evaluation of program terms. This system, implemented in the Guru verified programming language(available at http://www.guru-lang.org), is therefore capable of checking proofs about possibly diverging programs. When writing proofs in Guru, it quickly becomes apparent that many Guru proofs are very mechanical, and these proofs can be deduced automatically.

This paper presents a common class of Guru proof that will be referred to as "joinability modulo ground equations." Then, an algorithm is described that is capable of generating any required joinability modulo ground equations proof given some practical assumptions related to termination of the program terms involved. Though the algorithm was designed to simplify proof development in OpTT, it requires no features that are peculiar to OpTT. Therefore, the algorithm can be used to generate proofs of equality for terms in any call-by-value, pure functional programming language that includes inductive types.

2 Informal Description of Problem

This section presents an introduction to OpTT and then informally describes the joinability modulo ground equations problem.

2.1 OpTT Basics

OpTT contains a call-by-value, pure functional programming language with inductive types. The system also contains a proof language in which can be used to specify properties on program terms and proofs of those properties. In OpTT, the equality of a and b is denoted `{a = b}`. Properties which have been proven or assumed can also be stored in proof variables. The system contains a number of proof tactics which are used to derive new properties from existing properties. The tactics that are relevant to this paper are:

1. symm – when provided with `{a = b}`, will derive `{b = a}`
2. trans – when provided with `{a = b}` and `{b = c}`, will derive `{a = c}`
3. cong – when provided with `{a = b}` and a program term containing one or more "holes" (denoted E^*), will derive $\{E^*[a] = E^*[b]\}$. For example, `cong fun(x).* {a = b}` results in a proof of `{fun(x).a = fun(x).b}`.
4. join – when provided with two program terms, s and t, will derive `{s = t}` iff the normal forms of terms s and t w.r.t. evaluation under the operational semantics of OpTT are equal.

When writing proofs in Guru, it is common to write large sub-proofs that use only the trans, symm, cong, and join proof tactics. Any proof containing only these proof rules is called a joinability modulo ground equations proof. The problem of deciding whether two given terms are joinable modulo a given set of ground equations is called the joinability modulo ground equations problem. The goal of this paper is to develop an algorithm that will attempt to decide this problem and automatically generate these proofs.

Note that a "ground equation" is a property which expresses an equality and contains no quantifiers. The terms in a ground equation may contain variables, and any free program variables are treated as constants. For example the ground equation `{(plus a b) = (plus b a)}` does not indicate that the "plus" function is commutative. Because a and b are interpreted as specific program variables, it can only be assumed that the program term `(plus a b)` evaluates to the same value as `(plus b a)`, or both terms diverge.

Additional details and definitions related to OpTT can be found in Appendix A.

2.2 Example Input

Consider an example in which we wish to prove that the "greater than or equal to" relation (ge) for natural numbers is reflexive. The ge function, which defines the desired relation, is given in Figure 1, and Figure 2 contains the code for the proof, called geRefl. The example makes use of the induction tactic, which is used to prove that some property holds for all values of a variable of inductive type. The induction tactic introduces some variables that contain useful proofs that can be used in the body of the induction. In the example, the variable `n_eq` contains an equation stating that the value of n matches the case of the

induction (for example, in the Z case, `n_eq` is `{n = Z}`), and `n_IH` is the induction hypothesis which states that `{(ge v v) = True}` for any v that is strictly structurally smaller than n. The term `[n_IH n']` is the instantiation of `n_IH` for n', i.e. `{(ge n' n') = True}`. Note that `n_IH` contains quantifiers and cannot be used by hypjoin directly, but `[n_IH n']` contains no quantifiers.

$$(\texttt{ge n Z}) = \texttt{True}$$
$$(\texttt{ge Z (S n')}) = \texttt{False}$$
$$(\texttt{ge (S n')(S m')}) = (\texttt{ge n' m'})$$

Fig. 1. Definition of ge

```
Define geRefl : Forall(n:nat). { (ge n n) = True } :=
  induction(n:nat) return { (ge n n) = True} with
    Z =>
      trans
        cong (ge * *) n_eq
        join (ge Z Z) True
  | S n' =>
    trans
      trans
        cong (ge * *) n_eq
        join (ge (S n') (S n')) (ge n' n')
      [n_IH n']
  end.
```

Fig. 2. A Simple OpTT Proof

Each case of the induction contains a proof that simply makes substitutions based on a set of ground equations and then attempts to join the resulting terms. Due to the simplicity of these proofs, it is possible to automatically generate them by providing two starting terms and a set of ground equations that defines the allowed substitutions. In Guru, the desired automated deduction is requested via the hypjoin proof rule. The "hyp" in hypjoin stands for hypothesis, and the name "hypjoin" represents the fact that we are attempting to join two terms given a set of user-provided hypotheses (in the form of ground equations). Using hypjoin, the example is simplified as shown in Figure 3.

```
Define geRefl : Forall(n:nat). { (ge n n) = True } :=
  induction(n:nat) return { (ge n n) = True} with
    Z => hypjoin (ge n n) True by n_eq end
  | S n' => hypjoin (ge n n) True by n_eq [n_IH n'] end
  end.
```

Fig. 3. A Simple OpTT Proof Using hypjoin

3 Algorithm Description

This section contains an informal description of an algorithm that can be used to solve the problem described in the previous section. The description starts with a basic algorithm, and then adds additional steps to the algorithm due to the requirements of more complicated example cases.

3.1 Starting Point: Simple Algorithm

Based on the example in Figure 3, the following algorithm could be used to determine whether `hypjoin s t by U` should succeed, where s and t are terms, and U is a set of ground equations.

1. Evaluate s and t to their normal forms (s' and t') under the operational semantics.
2. Search for substitutions in U that would allow s' or t' to take a step of evaluation under the operational semantics. If found, goto step 1 using these substituted terms in place of s and t.
3. Test terms s' and t' for equivalence under substitutions in U.

The hypjoin request: `hypjoin (ge n n) True by n_eq [n_IH n']`, would be tested by this algorithm as follows.

1. `(ge n n)` and `True` are normal w.r.t. evaluation under the operational semantics.
2. Because `n_eq` is `{n = (S n')}`, The algorithm with substitute `(ge n n)` with `(ge (S n') (S n'))`, which evaluates to `(ge n' n')`.
3. In the final step of the algorithm, `(ge n' n')` is tested for equivalence with `True`. Because `[n_IH n']` is `{(ge n' n') = True}`, these two terms are equivalent and hypjoin succeeds.

3.2 Refinement 1: Normalization of Equations

Consider the following hypjoin request: `hypjoin s True by P1` where P1 is `{s = (ge Z Z)}`. This hypjoin request should succeed, since the proof of this equality would only make use of the trans, cong, and join proof tactics. However, the algorithm presented previously will not succeed for this request. Fortunately, a simple addition to the algorithm will allow this request to succeed.

At the beginning of the algorithm, the equations in U must be evaluated and inter-reduced until all terms in the equations are in a normal form. In this example, `{s = (ge Z Z)}` will be replaced with `{s = True}` in the first step of the algorithm. If there are multiple equations in U, then the evaluation of the terms in one equation can be performed modulo substitutions defined by the other equations.

3.3 Refinement 2: Checking for Consistency

Consider the following example: `hypjoin (ge a Z) True by P1 P2` where P1 is `{a = Z}` and P2 = `{a = (S a')}`. The set of equations provided to this request contains a contradiction of a sort that is fairly common in OpTT. Using the algorithm presented to this point, it is unknown whether this hypjoin request will succeed. The result depends on whether the algorithm chooses P1 or P2 when searching for a substitution for `(ge a Z)`. A set of equations is "consistent" if it allows no contradictions. In order to make the results of the algorithm predictable, the user-provided equations are checked for consistency at the beginning of the algorithm and after each time they are changed as a result of the "normalization of equations" process described previously.

3.4 Refinement 3: Evaluation of Sub-terms

In some cases, it is necessary to evaluate sub-terms of program terms in order for hypjoin to be successful. For example, the programmer may want to join two fun-terms as in: `hypjoin fun(x).(ge a Z) fun(x).True by P1` where P1 is `{a = Z}`. In this example, both of the provided terms are in normal form, but if the algorithm substitutes within and evaluates the body of `fun(x).(ge a Z)`, the result will be `fun(x).True`, and hypjoin will succeed. As a result of this requirement, the hypjoin algorithm will also attempt to substitute and evaluate sub-terms of a term recursively.

4 Problem Definition

The problem that must be solved by hypjoin can be described as follows. Given two terms, s and t, and a set of ground equations U, can s be joined with t using only evaluation under the operational semantics and substitutions from U?

The cong, symm, and trans proof rules can be expressed using a single equivalence relation $\approx_U$. This relation is defined in Figure 4. The basis of the joinability modulo ground equations relation, $\Rightarrow_U$, which uses $\approx_U$, is defined in Figure 5. These definitions reference the definitions of contexts found in Figure 12 in Appendix A.

In the definitions, Γ is a list of bound variables, and Bound() is a function that takes a context with a single hole and returns a set of variables that should be considered bound in the sub-term occupying that hole. The Bound() function will also be seen later in a form that takes a context with multiple holes and a

$$\frac{}{\Gamma \triangleright s \approx_U s} \qquad \frac{\{s = t\} \in U \quad Vars(s) \cap \Gamma = \emptyset \quad Vars(t) \cap \Gamma = \emptyset}{\Gamma \triangleright s \approx_U t}$$

$$\frac{\Gamma \triangleright t \approx_U s}{\Gamma \triangleright s \approx_U t} \qquad \frac{\Gamma \triangleright s \approx_U s' \quad \Gamma \triangleright s' \approx_U t}{\Gamma \triangleright s \approx_U t} \qquad \frac{\Gamma \cup Bound(E_1^+) \triangleright s \approx_U t}{\Gamma \triangleright E_1^+[s] \approx_U E_1^+[t]}$$

Fig. 4. Definition of Equivalence ($\approx_U$)

$$\frac{\Gamma \triangleright s \approx_U t}{\Gamma \triangleright s \Rightarrow_U t} \qquad \frac{\Gamma \triangleright s \to t}{\Gamma \triangleright s \Rightarrow_U t} \qquad \frac{\Gamma \cup Bound(E_1^+) \triangleright s \Rightarrow_U t}{\Gamma \triangleright E_1^+[s] \Rightarrow_U E_1^+[t]}$$

Fig. 5. Definition of $\Rightarrow_U$

natural number indicating the position of the hole of interest. This function will be used in conjunction with a function called Holes() that takes a context and returns the number of holes in that context.

The joinability modulo ground equations relation is $\overset{*}{\Leftrightarrow}_U$ – the symmetric, transitive closure of $\Rightarrow_U$. Any two terms, s and t, are joinable modulo U iff $s \overset{*}{\Leftrightarrow}_U t$. It should be noted that the definition of $\overset{*}{\Leftrightarrow}_U$ does not imply the existence of an algorithm that decides whether two terms are related by $\overset{*}{\Leftrightarrow}_U$. The goal of this paper is to develop such an algorithm.

4.1 Practical Considerations

While any algorithm that can decide whether two terms are related by $\overset{*}{\Leftrightarrow}_U$ would be valuable, there are some practical characteristics that would make the algorithm even more useful. The algorithm presented in this paper possesses all of the characteristics described in this section.

First, the algorithm should be efficient. Given the description of the problem above, one might envision an algorithm that searches the space of allowed substitutions while attempting to join two terms. Such an algorithm would be inefficient, and an algorithm based on computation would be much more desirable from the perspective of runtime performance.

Additionally, the algorithm should be external to OpTT and it should support the creation of an OpTT proof when it is successful. It should not be necessary to modify the meta theory of OpTT in order to support this algorithm. In fact, it should be possible to implement this algorithm outside of the Guru proof checker. Such an implementation could simply translate hypjoin requests into traditional Guru proofs. Also, if the algorithm is external to OpTT, then it is possible to use the algorithm to determine the equivalence of terms in other programming languages that are similar to the programming language in OpTT. The hypjoin algorithm supports the generation of a conventional OpTT proof of equality, but the additional operations required to generate such a proof are not discussed in this paper.

Finally, in the event that hypjoin fails, an implementation of the algorithm should be able to present some information to the programmer in order to help him determine why hypjoin failed and what must be changed in order for hypjoin to succeed. An algorithm that is based on searching would not satisfy this requirement since the only information that the algorithm could provide is that the search space was exhausted and no suitable match was found. However, a computational algorithm can produce a single result and present that result to the programmer to help determine why the algorithm failed to join two terms.

4.2 Main Contribution

The primary contribution of this paper is an algorithm which attempts to decide $\overset{*}{\Leftrightarrow}_U$ and can be used to satisfy hypjoin requests in Guru. This algorithm is sound, it is complete when it terminates and the supplied equations satisfy certain consistency requirements, and it is terminating for a well-defined, practically significant class of problems. In addition to these theoretical properties, which are described in greater detail in Section 7, the algorithm satisfies the practical requirements described in Section 4.1. The algorithm described in this paper has been successfully implemented as part of the Guru verified programming language.

5 Related Work

Congruence closure(e.g. [2]) is a decision procedure that can be used to decide the equivalence of two terms given a finite set of ground equations that defines a set of allowed substitutions. Congruence closure could be used to solve a significant part of the hypjoin algorithm, but using it to solve the entire problem is challenging because it is impossible to effectively represent the operational semantics of OpTT as a finite set of ground equations.

Combining equational reasoning and lambda calculus with $\beta\eta$-equivalence has been studied previously [4].

The Calculus of Congruent Inductive Constructions(CCIC)[5][6] is an extension to the Calculus of Inductive Constructions(CIC) that incorporates decision procedures for first-order theories into the CIC conversion rule. This system is very powerful in that the user-supplied hypotheses can be equations containing variables, and as such it can simplify many types of complex proofs. Though hypjoin is less general than CCIC, it does have some advantages when compared with CCIC. Most importantly, hypjoin is an optional extension to OpTT, whereas the modified conversion rule is a core part CCIC. Due to this structure, the theoretical properties of OpTT (e.g. cut elimination, type soundness) are not affected by hypjoin, and the hypjoin algorithm can be used to reason about programs in other pure functional languages.

6 Algorithm Definition

This section describes the algorithm that can be used to satisfy hypjoin requests. The algorithm will be referred to as the hypjoin algorithm, and will be represented by $\downarrow_U$. That is, $s \downarrow_U t$ indicates that the hypjoin algorithm relates term s to term t when provided with the set of ground equations U.

6.1 Algorithm Overview

The algorithm is built around an operation referred to as evaluation modulo U, which is represented by $\rightarrow_U$ where U is a set of ground equations. This operation will, in essence, try to find a substitution in U that allows the term to take a step of evaluation with respect to the operational semantics. In the first phase of the hypjoin algorithm, all of the terms in the equations in U are put into normal form with respect to $\rightarrow_U$ and the result is a set of equations U'. Any time $\rightarrow_U$ is used, it is important that the equations in U are consistent – that is, substitutions in U cannot allow a contradiction. So this normalization of U is structured in such a way that all $\rightarrow_U$ operations make use of normal, consistent U. In the next phase, the two supplied terms are put into normal form with respect to $\rightarrow_{U'}$. Finally, the algorithm results in a value of "true" if the normal forms of the supplied terms are related in $\approx_{U'}$.

6.2 Definition of $\rightarrow_U$

The equivalence relation used in $\rightarrow_U$ is $=_U$, which is more restrictive than $\approx_U$ as it only allows substitutions in evaluation holes(defined by the E_1 context). Other than this restriction, $=_U$ is the same as $\approx_U$. The definition of $\rightarrow_U$ is provided in Figure 6. In this definition, the elements of a list of items $\bar{a}$ of length n are represented like $a_{1\ldots n}$.

$$\frac{\forall m < n\ \Gamma \cup Bound(E_1^*, m) \rhd \neg(a_m \rightarrow_U) \quad \Gamma \cup Bound(E_1^*, n) \rhd a_n \rightarrow_U a'_n}{\Gamma \rhd E_1^*[\bar{a}] \rightarrow_U E_1^*[a_{1\ldots n-1}, a'_n, a_{n+1\ldots Holes(E_1^*)}]}$$

$$\frac{\forall n\ \Gamma \cup Bound(E_1^*, n) \rhd \neg(a_n \rightarrow_U) \quad E_1^*[\bar{a}] =_U t' \quad \Gamma \rhd t' \rightarrow t}{\Gamma \rhd E_1^*[\bar{a}] \rightarrow_U t}$$

Fig. 6. Definition of $\rightarrow_U$

Note that $\rightarrow_U$ first tries to evaluate the sub-terms of t using $\rightarrow_U$ before looking for a substitution in U that allows t to evaluate. This sub-term evaluation allows hypjoin to conclude that $E_1^+[s] \downarrow_U E_1^+[t]$ if $s \downarrow_U t$, which is necessary for completeness. By evaluating these sub-terms in a well-defined order, $\rightarrow_U$ is

deterministic if the notion of equality used is $\approx_U$. The deterministic nature of this relation makes it easier to reason about $\downarrow_U$.

In order to avoid issues related to impredicativity[7], $\rightarrow_U$ is viewed as a family of unary relations (sets) indexed by the starting term. So $t \rightarrow_U$ is a set of terms that can be reached in one step of $\rightarrow_U$ starting at t. The algorithm includes a test that some sub-term does not take a step in $\rightarrow_U$, denoted $\neg(t \rightarrow_U)$ for some term t. In terms of sets, $\neg(t \rightarrow_U)$ is defined as $t \rightarrow_U = \emptyset$. Because $\rightarrow_U$ is a family of unary relations, and because the definition of $t \rightarrow_U$ only appeals to the $\rightarrow_U$ of strict sub-terms of t, $t \rightarrow_U$ is predicative for all t.

6.3 Consistency

The correctness of the algorithm depends on the consistency of U. That is, U may not allow a contradiction. Figure 7 lists the properties that can be derived from the assumption that a given U is consistent, and the abbreviation C(U) is used to describe the property that U is consistent. In order to conclude that U is consistent, the algorithm must be able to determine that the consistency properties hold for all possible terms. An algorithm which tests U for consistency is given in Appendix C.

$$\forall U, \forall f, \forall f' \quad \frac{C(U) \quad f = fun(x).t \quad f' = fun(y).t' \quad \emptyset \triangleright f \approx_U f'}{\emptyset \triangleright t \approx_U t'[x/y]}$$

$$\forall U, \forall f, \forall c \quad \frac{C(U) \quad f = fun(x).t}{\neg(\emptyset \triangleright c \approx_U f)} \qquad \forall U, \forall c, \forall c', \forall \bar{I}, \forall \bar{I'} \quad \frac{C(U) \quad \emptyset \triangleright (c\ \bar{I}) \approx_U (c'\ \bar{I'})}{c = c'}$$

Fig. 7. Consistency Rules

6.4 Normalization of U

The first phase of the algorithm is the attempted normalization of the terms in the equations in U with respect to $\rightarrow_U$. So it is necessary to define $\rightarrow_U$ for sets of equations. This relation is defined in Figure 8. Note that the $\rightarrow$ symbol is overloaded. When applied to terms, it refers to evaluation under the operational semantics. When applied to sets of ground equations, it refers to the "U evaluation" relation defined in this section. The ! symbol denotes normalization w.r.t. some relation.

6.5 Definition of $\downarrow_U$

The hypjoin algorithm is defined in Figure 9.

$$\frac{U \xrightarrow{!} U' \quad C(U') \quad \emptyset \rhd u1 \rightarrow^{!}_{U'} u1'}{U \cup \{\{u1 = u2\}\} \rightarrow U' \cup \{\{u1' = u2\}\}} \qquad \frac{U \xrightarrow{!} U' \quad \neg C(U')}{U \cup \{\{u1 = u2\}\} \rightarrow U' \cup \{\{u1 = u2\}\}}$$

Fig. 8. Evaluation of U

$$\frac{U \xrightarrow{!} U' \quad C(U') \quad \Gamma \rhd s \rightarrow^{!}_{U'} s' \quad \Gamma \rhd t \rightarrow^{!}_{U'} t' \quad \Gamma \rhd s' \approx_{U'} t'}{\Gamma \rhd s \downarrow_U t}$$

Fig. 9. Definition of $\downarrow_U$ (hypjoin)

6.6 Determining Equivalence

In several places in the hypjoin algorithm, it is necessary to determine whether two terms, s and t, are equivalent given substitutions in U. This problem is, essentially, the word problem for U where U is a finite set of ground equations. This problem is known to be decidable by congruence closure[3][2]. However, there are additional sub-problems in the hypjoin algorithm that cannot be solved directly by congruence closure. For example, given a term t and a set of equations U, locate a redex t' that is equivalent to t. A description of the equivalence problems in the hypjoin algorithm, and an algorithm which can be used to solve them, is given in Appendix C.

7 Algorithm Properties

The joinability modulo ground equations problem is not decidable in general because the problem reduces to the problem of determining whether an arbitrary program terminates. Therefore, the hypjoin algorithm can only be partially correct, but the algorithm is useful nonetheless if it possesses the following properties:

1. The reasoning of the algorithm is sound.
2. The reasoning of the algorithm is complete when the algorithm terminates and the set of ground equations is consistent.
3. The algorithm terminates in well-defined, practical cases.

Each of these properties is further described in the remainder of this section.

7.1 Soundness

When two terms are related by $\downarrow_U$, it is necessary that the two terms are joinable given substitutions in U. This property is referred to as soundness and is defined in the statement of Theorem 1.

Theorem 1.

$$\forall s, \forall t, \forall U, \forall \Gamma \quad \frac{\Gamma \triangleright s \downarrow_U t}{\Gamma \triangleright s \overset{*}{\Leftrightarrow}_U t}$$

The proof of this property is available in [9].

7.2 Completeness

It is very desirable that the reasoning of hypjoin is complete. That is, if two terms, s and t, are joinable given U, then hypjoin of s and t by U will succeed. Because an inconsistent U will always cause hypjoin to terminate and fail, the completeness property must have an assumption that U is consistent. The property must also assume termination of hypjoin, for reasons described previously. The desired completeness property can be defined as shown in Theorem 2.

Theorem 2.

$$\begin{array}{c} \forall s, \forall t, \forall s', \forall t' \\ \forall U, \forall U', \forall \Gamma \end{array} \quad \frac{\Gamma \triangleright s \overset{*}{\Leftrightarrow}_U t \quad U \overset{!}{\rightarrow} U' \quad C(U') \quad \Gamma \triangleright s \overset{!}{\rightarrow}_U s' \quad \Gamma \triangleright t \overset{!}{\rightarrow}_U t'}{\Gamma \triangleright s \downarrow_U t}$$

The proof of this property is available in [9].

7.3 Termination

The hypjoin algorithm will not terminate in all cases, but it will be useful if it terminates in most practical cases. For example, the algorithm will not terminate when supplied with a program term that does not terminate. The goal of this section is to define the cases in which hypjoin is expected to terminate.

Termination Conditions. It should be safe to assume that the programmer will not include a term that is nonterminating w.r.t. $\rightarrow$. This assumption alone, however, is not sufficient to guarantee termination of hypjoin.

Because $\rightarrow_U$ evaluates sub-terms first, all of the sub-terms of all supplied terms must also be terminating w.r.t. $\rightarrow$. The symbol $\twoheadrightarrow$ will be used to represent the evaluation relation that evaluates sub-terms first in the manner of $\rightarrow_U$, only without making any substitutions.

Additionally, a substitution could turn a "stuck" term into a non-terminating term. To account for these substitutions, the assumptions will consider ground instances of input terms. The following definitions are used. Let σ be a mapping from variables to values. If t is a term, then $\sigma(t)$ is the ground instance obtained by replacing all variables in t with the appropriate values as defined by σ.

In order to guarantee termination of $s \downarrow_U t$, it is sufficient to assume that there is some σ such that:

1. $\forall \{u1 = u2\} \in U \quad \exists r : \quad \emptyset \triangleright \sigma(u1) \overset{*}{\twoheadrightarrow} r \quad \emptyset \triangleright \sigma(u2) \overset{*}{\twoheadrightarrow} r$
2. $\forall \{u1 = u2\} \in U \quad \exists u1', u2' : \quad \emptyset \triangleright \sigma(u1) \overset{!}{\twoheadrightarrow} u1' \quad \emptyset \triangleright \sigma(u2) \overset{!}{\twoheadrightarrow} u2'$
3. $\exists s', t' : \quad \emptyset \triangleright \sigma(s) \overset{!}{\twoheadrightarrow} s' \quad \emptyset \triangleright \sigma(t) \overset{!}{\twoheadrightarrow} t'$

These are reasonable assumptions that the programmer can manually verify with little effort, and they tend to hold in typical hypjoin requests. Unfortunately, the problem of testing these assumptions for a given hypjoin input is undecidable, because the problem reduces to testing for termination of an arbitrary program. Appendix B contains an argument for the termination of hypjoin given these assumptions.

8 Conclusion

This paper presented an algorithm that can be used to deduce the equality of terms in many common cases in OpTT and other pure functional programming languages. This algorithm reduces the amount of effort required to develop most formal proofs in OpTT. In fact, hypjoin is already a significant part of the Guru library code base. At present, there are 256 hypjoin requests in the 8050 lines of Guru library code, and many additional proofs in the library could be simplified by using hypjoin. The theoretical and practical properties of the algorithm make it easy for a programmer to use the algorithm and understand the results. Due to these properties, the algorithm will likely continue to be an important element of proof development in Guru.

References

1. Stump, A., Deters, M., Schiller, T., Simpson, T., Petcher, A.: Verified Programming in Guru. Proceedings of the 3rd workshop on Programming Languages Meets Program Verification, pp. 49-58 (2009)
2. Bachmair, L., Tiwari, A., Vigneron, L.: Abstract Congruence Closure. Journal of Automated Reasoning. Volume 31, Number 2. pp. 129-168 (2003)
3. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
4. Nipkow, T.: Higher-Order Critical Pairs. Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science. pp. 342-349 (1991)
5. Blanqui, F., Jouannaud, J., Strub, P.: From formal proofs to mathematical proofs: a safe, incremental way for building in first-order decision procedures. 5th IFIP International Conference on Theoretical Computer Science (2008)
6. Blanqui, F., Jouannaud, J., Strub, P.: Building Decision Procedures in the Calculus of Inductive Constructions. Lecture Notes in Computer Science, Volume 4646, pp. 328-342 (2007)
7. Feferman, S.: Predicativity. The Oxford Handbook of Philosophy of Mathematics and Logic, pages 590-624, Oxford University Press, 2005.
8. Knuth, D., Bendix, P.: Simple Word Problems in Universal Algebras. Computational Problems in Abstract Algebra, pp. 263-297. Pergamon Press, 1970
9. Petcher, A.: Deciding Joinability Modulo Ground Equations in Operational Type Theory. Master's thesis, Washington University in Saint Louis, May 2008. http://cse.seas.wustl.edu/Research/FileDownload.asp?797

A OpTT Details

OpTT terms which are relevant to this problem have the syntax described in Figure 10. In this definition, x is a variable, c is a term constructor, and I is an inactive term(also called a value). A bar over an item (e.g. $\bar{a}$) indicates a list of items. Evaluation of program terms under the operational semantics ($\rightarrow$) is defined as shown in Figure 11. Also used throughout this paper is the concept of contexts, which are defined in Figure 12. A context is a term with one or more holes (indicated by * in the figure) which accept arbitrary sub-terms. $E_1[t]$ describes an E_1 context with sub-term t in the hole of E_1, and $E_1^*[\bar{t}]$ describes an E_1^* context with the terms in $\bar{t}$ in the respective holes of E_1^*.

$$
\begin{aligned}
I &::= x \;||\; c \;||\; (c\ I_1 \cdots I_n) \;||\; \texttt{fun}\ f(\bar{x}).\ t \\
t &::= I \;||\; (t\ t') \;||\; \texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ t' \\
&\quad\ \texttt{match}\ t\ \texttt{with}\ \ c_1\ \bar{x}_1\ \texttt{=>}\ t_1 | \cdots | c_n\ \bar{x}_n\ \texttt{=>}\ t_n\ \texttt{end}
\end{aligned}
$$

Fig. 10. Syntax of Relevant OpTT Terms(t)

$$
\frac{}{\texttt{match}(c_i\ \bar{I})\ \texttt{with}\ c_1\bar{x}_1 \texttt{=>}\ s_1|...|c_n\bar{x}_n \texttt{=>}\ s_n\ \texttt{end} \rightarrow s_i[\bar{I}/\bar{x}_i]}
$$

$$
\frac{f = \texttt{fun}\ r(x).b}{(f\ I) \rightarrow b[I/x][f/r]} \qquad \frac{}{\texttt{let}\ x\ = I\ \texttt{in}\ t \rightarrow t[I/x]} \qquad \frac{t \rightarrow t'}{E_1[t] \rightarrow E_1[t']}
$$

Fig. 11. Operational Semantics of OpTT

The classification rules for the join, cong, symm, and trans proof tactics are shown in Figure 13. These classification rules define the equations that can be derived by these proof tactics.

B Termination Argument

This section explains why the $\downarrow_U$ algorithm will terminate given the termination arguments described in Section 7.3. The following approach is taken to demonstrate that $\downarrow_U$ terminates:

1. Given the termination assumptions, show that $\rightarrow_U$ terminates for normal, consistent U.
2. Show that the properties required for termination of $\rightarrow_U$ are preserved by $\rightarrow_U$ and thus preserved throughout the normalization of U.

$$
\begin{aligned}
E_1 ::= {} & (* \ t) \mid\mid (I \ *) \mid\mid \texttt{let}\ x\ \texttt{=}\ *\ \texttt{in}\ t \mid\mid \\
& \texttt{match}\ *\ \texttt{with}\ c_1\ \bar{s}_1\ \bar{x}_1\ \texttt{=>}\ t_1 \mid \cdots \mid c_n\ \bar{s}_n\ \bar{x}_n\ \texttt{=>}\ t_n\ \texttt{end} \\
E_1^+ ::= {} & (* \ t) \mid\mid (t \ *) \mid\mid \texttt{fun}\ f(\bar{x}).\ *\ \mid\mid \\
& \texttt{let}\ x\ \texttt{=}\ *\ \texttt{in}\ t \mid\mid \texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ *\ \mid\mid \\
& \texttt{match}\ *\ \texttt{with}\ c_1\ \bar{x}_1\ \texttt{=>}\ t_1 \mid \cdots \mid c_n\ \bar{x}_n\ \texttt{=>}\ t_n\ \texttt{end} \mid\mid \\
& \texttt{match}\ t\ \texttt{with}\ c_1\ \bar{x}_1\ \texttt{=>}\ * \mid \cdots \mid c_n\ \bar{x}_n\ \texttt{=>}\ t_n\ \texttt{end} \mid\mid \\
& \texttt{match}\ t\ \texttt{with}\ c_1\ \bar{x}_1\ \texttt{=>}\ t_1 \mid \cdots \mid c_m\ \bar{x}_m\ \texttt{=>}\ *\ \mid \cdots \mid c_n\ \bar{x}_n\ \texttt{=>}\ t_n\ \texttt{end} \mid\mid \\
& \texttt{match}\ t\ \texttt{with}\ c_1\ \bar{x}_1\ \texttt{=>}\ t_1 \mid \cdots \mid c_n\ \bar{x}_n\ \texttt{=>}\ *\ \texttt{end} \\
E_1^* ::= {} & (* \ *) \mid\mid \texttt{fun}\ f(\bar{x}).\ *\ \mid\mid \texttt{let}\ x\ \texttt{=}\ *\ \texttt{in}\ *\ \mid\mid \\
& \texttt{match}\ *\ \texttt{with}\ c_1\ \bar{x}_1\ \texttt{=>}\ * \mid \cdots \mid c_n\ \bar{x}_n\ \texttt{=>}\ *\ \texttt{end}
\end{aligned}
$$

Fig. 12. Contexts

$$
\frac{\exists t'' \quad t \xrightarrow{!} t'' \quad t' \xrightarrow{!} t''}{\Gamma \vdash \texttt{join}\ t\ t' : \{t\ \texttt{=}\ t'\}} \qquad \frac{\Gamma \vdash P : \{t\ \texttt{=}\ t'\}}{\Gamma \vdash \texttt{cong}\ E_1^*\ P : \{E_1^*[t]\ \texttt{=}\ E_1^*[t']\}}
$$

$$
\frac{\Gamma \vdash P : \{t\ \texttt{=}\ t'\}}{\Gamma \vdash \texttt{symm}\ P : \{t'\ \texttt{=}\ t\}} \qquad \frac{\Gamma \vdash P : \{t\ \texttt{=}\ t'\} \quad \Gamma \vdash P' : \{t'\ \texttt{=}\ t''\}}{\Gamma \vdash \texttt{trans}\ P\ P' : \{t\ \texttt{=}\ t''\}}
$$

Fig. 13. Classification Rules for join, cong, symm, and trans Tactics

3. Given that $\rightarrow_U$ terminates for normal, consistent U, show that the normalization of U terminates.

It is possible to show that $\rightarrow_U$ is terminating by considering the relationship between $s \rightarrow_U$ and $\sigma(s) \twoheadrightarrow$. If U represents the complete mapping σ, that is, U contains an equation v = I for each variable v in U and s, then $s \rightarrow_U$ and $\sigma(s) \twoheadrightarrow$ normalize to the same term. As $s \rightarrow_U$ proceeds, it will substitute values for variables in order to evaluate terms, and thus will proceed along the same path as $\sigma(s) \twoheadrightarrow$. In the case of any other U, as $s \rightarrow_U$ proceeds, substitutions will be made to allow terms to evaluate only when a suitable equation exists in U. Any such substitution and evaluation will by mirrored by a similar, multi-step evaluation in $\sigma(s) \twoheadrightarrow$. Because $s \rightarrow_U$ will follow a path that is similar to $\sigma(s) \twoheadrightarrow$, it must terminate somewhere before (or at) the end of that path.

Next, it is necessary to show that the termination properties are preserved by $\rightarrow_U$. From Theorem 3, $\sigma(u1) \overset{*}{\twoheadrightarrow} \overset{*}{\twoheadleftarrow} \sigma(u2)$ (property 1) is preserved by $\rightarrow_U$ because if $\{u1 = u2\} \in U$ and $u1 \rightarrow_U u1'$, then $\sigma(u1) \overset{*}{\twoheadrightarrow} \overset{*}{\twoheadleftarrow} \sigma(u1')$. Because $\twoheadrightarrow$ is deterministic, joinability by $\twoheadrightarrow$ is transitive, so $\sigma(u1') \overset{*}{\twoheadrightarrow} \overset{*}{\twoheadleftarrow} \sigma(u2)$. Theorem 3 can also be used to show that termination of the σ-instances of terms w.r.t. $\twoheadrightarrow$ is preserved by $\rightarrow_U$ (properties 2 and 3). If $s \rightarrow_U s'$ and $\sigma(s)$ is terminating, then $\sigma(s')$ is also terminating because $\sigma(s) \overset{*}{\twoheadrightarrow} \overset{*}{\twoheadleftarrow} \sigma(s')$ and $\twoheadrightarrow$ is deterministic.

Theorem 3.

$$\forall \Gamma \quad \frac{\Gamma \rhd s \rightarrow_U s'}{\exists s'' : \Gamma \rhd \sigma(s) \overset{*}{\twoheadrightarrow} \sigma(s'') \wedge \Gamma \rhd \sigma(s') \overset{*}{\twoheadrightarrow} \sigma(s'')}$$

Finally, it is necessary to show that the normalization of U will terminate as long as $\rightarrow_U$ terminates. Because it is assumed that $\rightarrow_U$ terminates, the only case in which $U \rightarrow$ could possibly be non-terminating is if there exists some infinite sequence in which changes to some equations allow other, stuck terms to be evaluated. In such a scenario, the evaluation of some term t in U could only be "triggered" by one of the following changes to the other equations. Because each type of event can only occur a finite number of times, such an infinite sequence cannot exist, so $U \rightarrow$ must terminate.

1. There exists an equation $\{u1 = u2\} \in U$ where $t = E_1[u1]$ and $\neg E_1[u2] \rightarrow$. $\{u1 = u2\}$ is replaced with $\{u1 = u2'\}$ where $E_1[u2'] \rightarrow$. Due to the way $\rightarrow_U$ is defined, u2 must be a non-value, and u2' must be a value. When a top-level term in U is a value, that term will never be replaced with a non-value, so the replacement of a non-value with a value is irreversible. Because there is a finite number of terms in U, this event can only occur a finite number of times.
2. There exists an equation $\{u1 = u2\} \in U$ where $\neg(t = E_1[u1])$. $\{u1 = u2\}$ is replaced with $\{u1' = u2\}$ where $t = E_1[u1']$ and $E_1[u2] \rightarrow$. Because $t = E_1[u1']$, u1' is strictly smaller than t. In order for this event to occur an

infinite number of times, there must exist an infinite sequence of terms in which each term is strictly smaller than the term which precedes it. Because there are a finite number of terms and each term is finite in size, this event can only occur a finite number of times.

C Determing Equivalence and Consistency

The hypjoin algorithm contains three sub-problems for which no algorithm has been given:

1. In $s \rightarrow_U$, it is necessary to locate a term s' such that $s =_U s'$ and $s' \rightarrow$.
2. When testing for consistency the algorithm must be able to find pairs of terms that are both equivalent and contradicting (as defined by the consistency rules in Figure 7).
3. In the last step of the algorithm, given terms s and t, and equations U, it is necessary to determine whether $s \approx_U t$.

As mentioned previously, problem 3 can be solved using congruence closure. It would not be reasonable to solve the remaining problems by searching the (possibly infinite) space of equivalent terms. Instead, all of the problems related to equivalence can be solved by completing[8] the equations and then rewriting with the resulting rules. In order to solve all of the problems described previously, the resulting rewriting system R must have all of the following properties:

1. R must rewrite non-values to values, if possible.
2. R must rewrite variables to constructor terms and fun terms, if possible.

These properties can be obtained by using a suitable termination order in the completion procedure. The order used in the implementation of the algorithm is a lexicographic path order(e.g. [3]) in which the top-level terms are ordered as follows: $let > match > term\ application > variable > function > constructor\ term$.

The term rewriting system resulting from this completion procedure can then be used to solve the equivalence problems introduced earlier (respectively):

1. Because R will rewrite any term to a value, if possible, R will also rewrite any non-redex to an equivalent redex, if possible (by rewriting sub-terms in evaluation holes to values). So given a term t, if there exists a term t' such that $t =_U t'$ and t' is a redex, then the normal form of t w.r.t. R will be a redex that is equivalent to t.
2. If U is inconsistent, then R must contain a rewrite rule in which the left-hand side contradicts with the right-hand side. This property can be explained by noting that all consistency rules involve pairs of values, and values can only rewrite to values. So any inconsistency will require at least one rule that rewrites a value to a conflicting value, and it is possible to detect inconsistency by performing the completion procedure and then examining the resulting rewrite rules individually.

3. In the last step of the hypjoin algorithm, it is necessary to determine whether two terms, s and t, are equivalent. This equivalence can be determined by computing s' and t', the normal forms of s and t w.r.t. R, and then testing s' and t' for equality.