Automated Termination Proofs Using Walther Recursion

by

Alexander Wu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering in Computer Science and Engineering

and

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

Author .....
Department of Electrical Engineering and Computer Science
May 23, 1994

Certified by .....
David A. McAllester
Associate Professor
Thesis Supervisor

Accepted by .....
Leonard A. Gould
Chairman, Departmental Committee on Graduate Students

Automated Termination Proofs Using Walther Recursion

by

Alexander Wu

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1994, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Computer Science and Engineering
and
Bachelor of Science in Computer Science and Engineering

Abstract

In this thesis, I present an improved method for automatic proofs of termination through a syntactic measurement of sizing that was first developed by Walther. We discuss formal requirements for proving termination using this method and how it can be generalized to other languages. We present a simple set-theoretic language as an example which can be used to write basic Lisp-like programs. Using this language, we define strict syntactic requirements on how to prove termination and then produce inference rules that indicate how these requirements can be established. Several sample algorithms whose termination proofs are sketched using the inference rules are presented in order to show the power of the approach. Last, we discuss the issue of soundness of these rules, as well as issues of the decidability and efficiency of implementing such a system.

Thesis Supervisor: David A. McAllester
Title: Associate Professor

Acknowledgments

I would especially like to thank David McAllester, my thesis advisor, for all his help and more importantly, patience and understanding. None of this work would have been possible without his guidance. Special thanks also go to Carl Witty, Robert Givan, and Luis Rodriguez, whose showed me some of the more unusual aspects of graduate life.

This work is dedicated to my sister, Cynara, who is undergoing trying times right now. I hope that everything turns out for the better. I may not be the best friend that you need right now but I still love you very much. I am also forever indebted to my parents for everything they have done for me all these years. Thanks, Mom and Dad. This is for you.

Finally, a thousand thanks go to all my friends here who never gave up on me despite all the time I spent doing thesis instead of with you guys. James, maybe I still have time to pick up ultimate. Henry, once this damned thing is finished, we can play tennis. Howard, thanks for trying to make me a "Superstar". It didn't quite work but you did the best with what you had. Yong, I'm sorry I didn't have quite what it takes – you are the "Chosen One" in more ways than one. To everyone that can't be named, or this thesis would double in size, you all have made these five years the best of my life. I'll always remember the great times there were.

Contents

1 Introduction **7**

1.1 A Termination Proof 8

1.2 Description of the Approach 10

1.3 Comparison with Other Approaches 12

2 Walther Recursion **14**

2.1 Fixed Point Semantics 14

2.2 Proving Termination 17

3 First Order Logic **24**

3.1 Language 25

3.1.1 Description 26

3.1.2 Grammar 28

3.1.3 Operational Semantics 28

3.1.4 Application 29

3.2 Meta-Formulas 33

3.2.1 Description 34

3.2.2 Grammar 37

3.2.3 Semantics 37

3.3 Inference Rules 37

3.3.1 Description 39

3.3.2 Examples 43

3.3.3 Soundness 45

3.3.4 Decidability 46

4 Conclusion **48**

A Soundness **50**

A.1 Boolean Rules for Formulas 50

A.2 Argument Bounded Terms 50

A.3 Argument Bounded Functions 52

A.4 Formulas 52

A.5 Difference Literals 53

A.6 Well-Founded Functionals 55

List of Figures

3-1 Grammar of the Language 28
3-2 Semantics of the Language 30
3-3 Grammar of the Meta-Formulas 37
3-4 Semantics of the Meta-Formulas 38

Chapter 1

Introduction

In formal reasoning with mechanical verification systems, such as those used for hardware and software verification, it is often necessary to show that recursive functions are well-defined. When defining languages for computers, an imperative model is often used, and therefore the problem can be treated as constructing a proof of termination. Since there are recursive definitions whose termination properties are bound to be intractable (instances of the halting problem, for example), it is necessary to resort to heuristic algorithms to determine the well-formedness of recursive definitions.

One common approach for proving termination was invented by Floyd[6]. Floyd's approach relies on the ability to find some measure of the arguments that is well-ordered according to some relation, $<$. If a measure exists which is guaranteed to decrease according to this relation on every recursive call, the function must terminate. The key innovation required in such proofs, however, is the ability to find a measure with the necessary properties.

Walther[12] expanded on Floyd's work by using a specific class of measures which applies to a large class of natural recursive procedures and for which a decidable algorithm could be constructed which automatically determines the applicability of the measures to a given procedure. The measure is based on a syntactically-defined notion of the size of an argument. Based on this size order, Walther describes a sound as well as decidable method for proving automatically that a large class of recursive procedures terminate.

This research generalizes the work of Walther in several ways. First, we apply Walther recursion to a language with a more general idea of primitive data structures than the shell principle used in Walther. Second, we extend Walther recursion to allow certain transfinite recursive definitions to be accepted. Third, we describe a rule-based implementation of our algorithm which is strictly based on the syntactics of our language. Finally, the theoretical framework used by Walther in his paper did not give clear semantical and syntactical descriptions of the language being described, making it very difficult to formally generalize the approach. With a grammar defining the syntax and a meaning function determining the semantics of our approach, extensions to other languages are more readily apparent.

1.1 A Termination Proof

In order to illustrate how we prove termination, we begin with an example proof that Euclid's algorithm for finding the greatest common denominator terminates for the whole numbers. Specifically, using a syntax resembling that of Lisp[11], our algorithm is defined to be:

```
(define (gcd a b)
  (if (= a 0)
     b
     (gcd (remainder b a) a)))
```

where the **remainder** function is defined as:

```
(define (remainder x y)
  (if (= y 0)
      y
      (if (< x y)
          x
          (remainder (- x y) y))))
```

In order to prove the **gcd** function terminates, we must show there is a well-founded order, $<$, on its arguments $\langle a,b \rangle$, such that for any recursive call, **gcd**(x,y) made within the function, $\langle x,y \rangle < \langle a,b \rangle$. In this paper, we restrict the well-founded

orders to being dependent on a single argument only. By observation, this can be possible only if for all ⟨a,b⟩ such that $a \neq 0$, a <(**remainder** b a). Upon inspecting our **remainder** function, since $y \neq 0$, **remainder** returns x if x <y or (**remainder** (- x y) y) otherwise. If x <y, then (**remainder** b a) = x ⇒ (**remainder** b a) <a.

Otherwise, (**remainder** x y) = (**remainder** (- x y) y). Another recursive call to **remainder** is made where $y \neq 0$. Since the minus operator defines a well-founded order on the whole numbers, **remainder** will eventually stop and since $y \neq 0$ always holds, it must terminate with a value x where x <a. Thus, for all ⟨a,b⟩ such that $a \neq 0$, a <(**remainder** b a). We therefore have established a well-founded ordering on ⟨a,b⟩ which the desired properties. Therefore, the **gcd** function must terminate.

The above sketch of a proof roughly describes the approach taken by Walther[12]. Although it is a reasonable sketch of a termination proof for **gcd**, using this approach in an automated system has several weaknesses. First, the chain of reasoning which lead to conclusion that the **gcd** function is Walther is a bit unclear. Although there is a formal algorithm for determining whether a function is Walther recursive, the semantics of the language as well as clear inferences leading to the proof had been abstracted out of the discussion. Second, the automated system discussed by Walther produces *generation hypotheses* and relies on an induction theorem prover to verify these hypotheses. In our approach, we have developed a formal, set-theoretic definition of Walther recursion, proving the necessary definability properties, in order to generate a sound set of inference rules which essentially lead to termination proofs, as opposed to termination hypotheses. The soundness of the rules should be indeed verified by the induction theorem prover but this can be done offline and thus efficiency can be increased significantly.

For example, in our approach the above termination proof would be proved with the following chain of inferences:

- Function − (minus) terminates
- Given arguments x,y $y \neq 0, x \neq 0$, **minus**(x,y) is less than x
- ⊢ In (remainder (− x y) y), (- x y) is less than x

- $\vdash$ **remainder** teminates
- **remainder**(x, y) returns a value less than or equal to y
- $\vdash$ Given arguments x,y $y \neq 0$, **remainder**(x, y) returns a value less than y
- $\vdash$ In (gcd (remainder b a) a), (remainder b a) is less than b
- $\vdash$ **gcd** terminates

1.2 Description of the Approach

As opposed to the generate-and-test paradigm combined with an induction theorem prover that is used by Walther, we use an inference rule system for automated termination proofs. However, we have adapted the generate-and-test paradigm taken by Walther to generate inference rules that are essentially equivalent to Walther's approach. Since we do not use an induction theorem prover in our approach, we have established a set of formal requirements for recursive definitions to be Walther recursive and then proved these requirements guarantee a well-defined function. We then defined a set of meta-formulas which within our language of discussion, indicate how terms in our language satisfy the necessary formal requirements.

The meta-formulas, also termed properties by Walther, which are required must determine sufficient conditions for proving a function terminates. Since our approach is based on a well-founded size order which can establish that the size of an argument to a recursive function decreases on every recursive call, meta-formulas for determining when a recursive function satisfies this property are needed. In order to establish this property is true for a given recursive function, we must show that the size of the argument decreases. Unfortunately, the size of a term has a semantic meaning, which determining in the general case is intractable. We resolve this problem by constructing a meta-formula which is a *syntactic* estimate of the relative size of two terms. The basis for recognizing that terms are relatively smaller or larger is through the idea of *argument-bounded functions.* Argument-bounded functions are

functions which return results that are strictly less than or equal to a given argument to the function. For example, the **remainder** function in the above example was argument-bounded on its second argument. Thus, meta-formulas which indicate what functions are argument-bounded on which arguments are required. For a base set of argument-bounded functions, we use a set of primitive selectors defined on a given set of datatypes.

This, however, is still insufficient. Argument-bounded functions guarantee that the result returned is less than or equal to the value of an argument, whereas for proving termination, we require that in the call to a recursive function the argument is *strictly decreasing.* Therefore, in addition to recognizing functions are argument-bounded, we must also recognize *when* such functions return a result that is strictly less than an argument. "When" translates into establishing conditions on the arguments to a function, or so-called *difference literals* that ensure that the result will be strictly less than an argument.

Given these meta-formulas, we constructed a set of inference rules which indicate how the properties can be provably established within the language. Once our meta-formulas and inference rules are established and proved to be correct, we can incorporate them into rule-based system. Because generating inferences is the only dynamic portion of the system and because the rules can be implemented as a recursive descent parsing algorithm, an efficient implementation exists for such a rule-based system.

Chapter 2 describes the Walther recursion more fully and extends Walther's work to include a set-theoretical framework for Walther recursive functions with proofs of the fixed-point properties of these functions leading to semantically well-defined meanings. We use this framework in chapter 3 to discuss the inference rules necessary to determine whether definitions are Walther recursive in a simple language. The decidability of the inference rules is established and their soundness is discussed. A series of example algorithms are also given which are provably Walther through the inference rules. These illustrate the types of recursive definitions which can be expressed by Walther recursion. We conclude in chapter 4 with a brief discussion of

the strengths and weaknesses of our approach.

1.3 Comparison with Other Approaches

The works presented here are all discussed in Walther's paper[12]. Much of the work in this area not only deals with termination proofs but correctness proofs as well. Since the primary motivation for this paper is for verifying termination of recursive functions, correctness proofs are not of major concern, and therefore are not discussed here. However, it should be noted that algorithms incorporating correctness with termination proofs often requires a distinct different approach.

Floyd[6] first suggested the use of a termination function and the properties of well-founded sets for proving termination of flow chart programs. These ideas were adapted by Cooper[5] and used to implement a mechanical verification system. This system was a semi-automatic facility for generating and verifying so called *convergence conditions* which were sufficient for proving termination.

In [8], four different methods of proving termination are compared. For Floyd's technique[6] as discussed above and for the *loop approach*, which is based on establishing an upper bound for each increasing counter in a loop, a termination function is used. The *exit approach* uses *exit conditions*, conditions upon which a loop is guaranteed to terminate, and attempts to show how these conditions are satisfied at some stage in the loop. Proofs of non-termination can also be done using this approach. Finally, the paper describes *Burstall's[4]* method, which uses structural parallel induction to prove termination and correctness simultaneously.

Burstall's method is useful in unusual recursions which are difficult to understand and verify such as *McCarthy's* 91-function and *Ashcroft's* algorithm for list reversal[10][9] where termination and correctness must be shown together. Since such algorithms are not often of practical use, we consider it a minor drawback that an approach based strictly on termination such as Walther recursion cannot prove the termination of such algorithms. The key problem to approaches based on this method is that determining the correct well-founded order often requires the outside aid and

therefore is not fully automated.

The approach taken by the author is based on Walther recursion[12]. This is an adaptation of Floyd's approach with termination functions through a syntactic measure of the size of arguments to a recursive function. A similar approach was used in the system developed Boyer and Moore[3]. However, that system required outside intervention by the user. Walther's approach fully automates the construction of termination proof.

Chapter 2

Walther Recursion

In order to prove the consistency of recursive functions, it is first necessary to formally define Walther recursion. Our treatment will be from a set-theoretic standpoint, which allows sufficient expressibility to be applicable to most areas within computer science as well as a large area of mathematics. Specifically, we assume that sets are axiomatically defined within the realm ZFC set theory.

2.1 Fixed Point Semantics

In order to establish our formal framework, we must remember that it should satisfy our intuitive notion of Walther recursion as set forth in this paper. Thus, let us return to our definition of a function for computing the greatest common denominator:

```
(define (gcd a b)
  (if (= a 0)
     b
     (gcd (remainder b a) a)))
```

Assuming a function **remainder** is already defined with provably the desired properties we need, there is still a question of how we can mathematically express this function in such a way that it yields the results we intuitively expect it to give.

For example,

$$gcd(\langle a, b\rangle) = \begin{cases} b \text{ if } a = 0 \\ gcd(remainder(b, a), a) \text{ otherwise} \end{cases}$$

has no definable meaning since the function is defined in terms of itself. Thus, our first goal is to provide a definition for a recursive function. We take the standard approach as described by Manna[10], through using a fixed point theory of recursion functions. We define a *partial function* to be a function which maps elements in the domain to elements whose value is within the given range or some pre-defined element, $\emptyset$, which may or may not be in the range, where $\emptyset$ is considered the undefined value. Partial functions intuitively allow us to define the value of a function on a subset of the domain, allowing us to provide an approximation of a recursive function. For example, the first approximation of **gcd** could be:

$$gcd(\langle a, b\rangle) = \begin{cases} b \text{ if } a = 0 \\ \emptyset \text{ otherwise} \end{cases}$$

The next step is to provide some method of generating better approximations of the recursive function we seek to define. In order to do so, we define a *functional*. Functionals are functions which, given a partial function, will return a partial function. Therefore, they can be used to model functions which given an approximation to a recursive function, return a better approximation. For **gcd**:

$$GCD(gcd') = \{(\langle a, b\rangle, y) : y = \begin{cases} b \text{ if } a = 0 \\ gcd'(\langle a, b\rangle) \text{ otherwise} \end{cases}$$

In this case, the **GCD** functional will return a new approximation of the **gcd** function, given a prior approximation, **gcd'**. Although we can take better and better approximations, mathematically we want the limiting case where we have converged

on desired recursive function, i.e. a function, **gcd**, such that

$$GCD(gcd) = gcd$$

We call **gcd** a fixed point of the functional **GCD** and take it to be the natural meaning of the recursive definition of the greatest common denominator. Thus, in order to describe a recursive function mathematically, we define a functional and determine its fixed point. We proceed to provide formal definitions for the above concepts.

Definition 1 A *partial function* from a set X to a set Y, denoted as $f : X \rightarrow Y$, is a set of ordered pairs, $\{(x, y) : x \in X, y \in Y\}$. The *value, f(x),* $x \in X$, is considered *undefined, or* $f(x) = \emptyset$ if there does not exist y such that $(x, y) \in f$. Otherwise, the value *f(x)* is considered $y, (x, y) \in f$. More formally,

$$f(x) = \begin{cases} \emptyset \text{ if } \neg\exists y, (x, y) \in f \\ y \text{ s.t. } (x, y) \in f, \text{ otherwise} \end{cases} \tag{2.1}$$

We call X, the *domain* of f and Y, the *range* of f.

Definition 2 Let $\tau, \tau_1, \tau_2, ..., \tau_n$ be sets and let P be the set of all partially defined multi-valued functions $g : \tau_1 \times \tau_2 ... \times \tau_n \rightarrow \tau$ over a given set of n variables from $\tau_1, ..., \tau_n$ to τ. A *functional* in P is defined as a function $F : P \rightarrow P$. By abuse of notation, we define $F^\alpha(x)$, for any any ordinal α as

$$F^\alpha(x) = \begin{cases} F(x) & \text{if } \alpha = 1 \\ F(F^\beta(x)) & \text{if } \exists\beta, \alpha = \beta + 1 \\ F^\alpha(x) = \{(a, b) : \exists\beta < \alpha \text{ s.t. } \forall\gamma > \beta, (a, b) \in F^\gamma(x)\} & \text{otherwise} \end{cases} \tag{2.2}$$

Furthermore, we use the following notation for a partial function whose value is $\emptyset$ on the entire domain:

$$F^\alpha(f_\emptyset) \equiv F^\alpha(\{\}) \tag{2.3}$$

2.2 Proving Termination

Walther recursion is based on the idea of a termination function as described by Manna. Specifically, a *size order* which is a measurement of the size of the arguments to the function is used to show that if the size order is well-founded and if the arguments to the function are smaller in relation to the order, the function will terminate.

Since set theory is often defined by an axiomatic construction of sets, stage by stage using the ordinals, this leads to a natural method of defining a well-founded size ordering on the sets using the rank of the set. Specifically,

Definition 3 For any ordinal α we define V_α as

- $V_0 = 0$
- $V_{\alpha+1} = P(V_\alpha)$
- $V_\beta = \bigcup_{\alpha<\beta} V_\alpha$

Then, for any set x, let $|x|$ be the least α such that $x \subseteq V_\alpha$. We call $|x|$ the *rank* of set x.

It is a common knowledge that such an ordering using the ordinals imposed a well-founded size ordering on sets. Therefore, we assume, without proof, that the above ordering is well-founded and we can perform induction using this size ordering.

Given a formal method of defining recursive functions and a size ordering imposed on sets, we can now proceed to define the notion of Walther recursion. First, we need some method of comparing different partial functions as approximations of recursive functions.

Definition 4 Let *f,g* be partially defined multi-valued functions. For any i, α, we define $f =_{i,\alpha} g$ to mean that for all $v = \langle x_1, ..., x_n \rangle, |x_i| < \alpha \Rightarrow f(v) = g(v)$. Thus,

$$f =_{i,\alpha} g \equiv \forall v = \langle x_1, ..., x_n \rangle, |x_i| < \alpha \Rightarrow f(v) = g(v) \tag{2.4}$$

Example 1 Let *f,g* be functions such that

$$f = \{(\langle 1,2,3\rangle, 9), (\langle 4,1,5\rangle, -5), (\langle 7,8,4\rangle, 5), (\langle 5,6,7\rangle, 8)\}$$

$$g = \{(\langle 1,2,3\rangle, 9), (\langle 4,1,5\rangle, -5), (\langle 7,8,4\rangle, 5), (\langle 6,5,8\rangle, 1)\}$$

Then $f =_{1,5} g$, $f =_{2,5} g$, and $f =_{3,7} g$.

Next, recall that a Walther recursive function is a recursive function that guarantees that on every recursive call, some measure of the function is guaranteed to decrease according to some well-ordering. This leads to our set-theoretic definition of a Walther recursive function.

Definition 5 Given a set of functions $P = (\tau_1 \times \tau_2 ... \times \tau_n) \rightarrow \tau$, a functional $F : P \rightarrow P$ is defined to be *well-founded* if and only if $\exists m, 1 \leq m \leq n$ where the following condition holds:

- For any functions $f, g \in P$, $\forall \alpha, (f =_{m,\alpha} g \Rightarrow F(f) =_{m,\alpha+1} F(g))$.

We say F is well-founded on argument m and call m the *measure argument* of functional F.

Claim 1 Let N = set of non-negative integers and $P = N \rightarrow N$ be a function space over which the following functional is defined:

$$F(h) = \{(1,0)\} \cup \{(x,y) : x > 0 \wedge x \in N, y = x * h(x-1)\}$$

Then F is well-founded on its first and only argument x. Note that this functional describes the factorial function.

Proof: Let *f,g* be any functions such that $f, g \in P$. Assume to the contrary that

$$\neg \forall \alpha, (f =_{1,\alpha} g \Rightarrow F(f) =_{1,\alpha+1} F(g))$$

Which means that

$$\exists \alpha = \alpha' \text{ s.t. } \neg (f =_{1,\alpha} g \Rightarrow F(f) =_{1,\alpha+1} F(g))$$

Then, for α',

$$f =_{1,\alpha'} g \wedge F(f) \neq_{1,\alpha'+1} F(g)$$

Since $F(f) \neq_{1,\alpha'+1} F(g)$,

$$\exists v = v' \text{ s.t. } |v'| < \alpha' + 1 \wedge [F(f)](v') \neq [F(g)](v')$$

Using our definition of the functional F, we get

$$[F(f)](v') = \begin{cases} 1 & \text{if } v' = 0 \\ v' * f(v'-1) & \text{otherwise} \end{cases}$$

and

$$[F(g)](v') = \begin{cases} 1 & \text{if } v' = 0 \\ v' * g(v'-1) & \text{otherwise} \end{cases}$$

However, since $f_{1,\alpha'}g$ and $|v'| < \alpha'+1 \Rightarrow |v'-1| < \alpha'$, $f(v'-1) = g(v'-1)$. Therefore, $[F(f)](v') = [F(g)](v')$ which is a contradiction. Thus,

$$\forall \alpha, (f =_{1,\alpha} g \Rightarrow F(f) =_{1,\alpha+1} F(g))$$

which means that F is well-founded on its first argument. ■

Claim 2 Well-founded functionals do not have to be monotonic. For example, let N = set of non-negative integers and $P = N \rightarrow N$ be a function space over which the following functional is defined:

$$F(h) = \{(1,0)\} \cup \{(x,y) : x > 0 \wedge x \in N, y = \begin{cases} 0 & \text{if } \neg\exists y' \text{ s.t. } y' \in h(x-1) \\ h(x-1) & \text{otherwise} \end{cases} \}$$

Then F is well-founded on its first and only argument x.

Proof: Let *f,g* be any functions such that $f, g \in P$. Assume to the contrary that

$$\neg\forall \alpha, (f =_{1,\alpha} g \Rightarrow F(f) =_{1,\alpha+1} F(g))$$

Which means that

$$\exists \alpha = \alpha' \text{ s.t. } \neg(f =_{1,\alpha} g \Rightarrow F(f) =_{1,\alpha+1} F(g))$$

Then, for α',

$$f =_{1,\alpha'} g \wedge F(f) \neq_{1,\alpha'+1} F(g)$$

Since $F(f) \neq_{1,\alpha'+1} F(g)$,

$$\exists v = v' \text{ s.t. } |v'| < \alpha' + 1 \wedge [F(f)](v') \neq [F(g)](v')$$

Using our definition of the functional F, we get

$$[F(f)](v') = \begin{cases} 1 & \text{if } v' = 0 \\ 0 & \text{if } \neg\exists y' \text{ s.t. } y' \in f(v'-1) \\ f(v'-1) & \text{otherwise} \end{cases}$$

and

$$[F(g)](v') = \begin{cases} 1 & \text{if } v' = 0 \\ 0 & \text{if } \neg\exists y' \text{ s.t. } y' \in g(v'-1) \\ g(v'-1) & \text{otherwise} \end{cases}$$

However, since $f_{1,\alpha'}g$ and $|v'| < \alpha'+1 \Rightarrow |v'-1| < \alpha'$, $f(v'-1) = g(v'-1)$. Therefore, $[F(f)](v') = [F(g)](v')$ which is a contradiction. Thus,

$$\forall \alpha, (f =_{1,\alpha} g \Rightarrow F(f) =_{1,\alpha+1} F(g))$$

which means that F is well-founded on its first argument. ■

Theorem 1 Given any well-founded functional F on measure argument at m and any ordinal α, $\forall \beta > \alpha, F^{\alpha}(f_{\emptyset}) =_{m,\alpha} F^{\beta}(f_{\emptyset})$.

Before proving this theorem, we state the following lemma:

Lemma 1 Given any well-founded functional F on measure argument m and any

ordinal α such that $F^{\alpha}(f_{\emptyset}) =_{m,\alpha} F^{\alpha+1}(f_{\emptyset})$, we have that for all $\beta > \alpha, F^{\alpha}(f_{\emptyset}) =_{m,\alpha} F^{\beta}(f_{\emptyset})$.

Proof: Let $\Phi[\beta]$ represent the statement $F^{\alpha}(f_{\emptyset}) =_{m,\alpha} F^{\beta}(f_{\emptyset})$. We will prove the lemma by transfinite induction on the ordinal β. For our base case, $\beta = \alpha + 1$, the proof is automatic by our assumption that $F^{\alpha}(f_{\emptyset}) =_{m,\alpha} F^{\alpha+1}(f_{\emptyset})$.

In our induction step for non-limit ordinals, we wish to prove $\Phi[\beta] \rightarrow \Phi[\beta + 1]$. Since F is a well-founded functional,

$$\Phi[\beta] \rightarrow F(F^{\alpha}(f_{\emptyset})) =_{m,\alpha+1} F(F^{\beta}(f_{\emptyset}))$$

Relaxing the restriction from $=_{m,\alpha+1}$ to $=_{m,\alpha}$, we get

$$\Phi[\beta] \rightarrow F^{\alpha+1}(f_{\emptyset}) =_{m,\alpha} F^{\beta+1}(f_{\emptyset})$$

Given that $F^{\alpha+1}(f_{\emptyset}) =_{m,\alpha} F^{\alpha}(f_{\emptyset})$, by transitivity of $=_{m,\alpha}$,

$$F^{\beta+1}(f_{\emptyset}) =_{m,\alpha} F^{\alpha}(f_{\emptyset})$$

and therefore $\Phi[\beta] \rightarrow \Phi[\beta + 1]$.

For limit ordinals, β, we need to prove $\forall\gamma$ s.t. $\alpha < \gamma < \beta, \Phi[\gamma] \rightarrow \Phi[\beta]$. Let $v = \langle x_1, ..., x_n \rangle, |x_m| < \alpha$, be an arbitrary n-tuple. By our initial assumption,

$$\forall\alpha < \gamma < \beta, \Phi[\gamma] \rightarrow F^{\alpha}(f_{\emptyset}) =_{m,\alpha} F^{\gamma}(f_{\emptyset})$$

which means for arbitrary y,

$$(v, y) \in F^{\alpha}(f_{\emptyset}) \Rightarrow \forall\alpha < \gamma < \beta, (v, y) \in F^{\gamma}(f_{\emptyset})$$

Thus, by the definition of $F^{\beta}(f_{\emptyset})$ for transfinite β

$$(v, y) \in F^{\alpha}(f_{\emptyset}) \Rightarrow (v, y) \in F^{\beta}(f_{\emptyset})$$

Similarly, in order to show that $(v,y) \in F^{\beta}(f_{\emptyset}) \Rightarrow (v,y) \in F^{\alpha}(f_{\emptyset})$

$$(v,y) \in F^{\beta}(f_{\emptyset}) \Rightarrow \exists \beta' < \beta \text{ s.t. } \forall \beta > \gamma > \beta', (v,y) \in F^{\gamma}(f_{\emptyset})$$

Thus, for any such γ such that $\beta > \gamma > \alpha$,

$$(v,y) \in F^{\beta}(f_{\emptyset}) \Rightarrow (v,y) \in F^{\gamma}(f_{\emptyset}) \Rightarrow (v,y) \in F^{\alpha}(f_{\emptyset})$$

Since v is arbitrary,

$$\forall v = \langle x_1, ..., x_n \rangle \text{ s.t. } |x_m| < \alpha, [F^{\alpha}(f_{\emptyset})](v) = [F^{\beta}(f_{\emptyset})](v)$$

meaning we have $F^{\alpha}(f_{\emptyset}) =_{m,\alpha} F^{\beta}(f_{\emptyset})$ which implies that $\forall \alpha < \gamma < \beta, \Phi[\gamma] \rightarrow \Phi[\beta]$ ■

Theorem 2 Given any well-founded functional F on measure argument at m and any ordinal α, $\forall \beta > \alpha, F^{\alpha}(f_{\emptyset}) =_{m,\alpha} F^{\beta}(f_{\emptyset})$.

Proof: Let $\Psi[\alpha]$ represent the statement $\forall \beta > \alpha, F^{\alpha}(f_{\emptyset}) =_{m,\alpha} F^{\beta}(f_{\emptyset})$. We will prove the theorem by transfinite induction:

- $\Psi[\alpha = 0]$: Since there does not exist $v = \langle x_1, ..., x_n \rangle$ such that $|x_m| < 0$, $F^0(f_{\emptyset}) =_{m,0} F^1(f_{\emptyset})$. Therefore, by the lemma, $\forall \beta > 0, F^0(f_{\emptyset}) =_{m,0} F^{\beta}(f_{\emptyset})$.

- $\Psi[\alpha] \rightarrow \Psi[\alpha + 1]$: Given $\Psi[\alpha]$, $F^{\alpha}(f_{\emptyset}) =_{m,\alpha} F^{\alpha+1}(f_{\emptyset})$. By the definition of a well-founded functional, $F^{\alpha}(f_{\emptyset}) =_{m,\alpha} F^{\alpha+1}(f_{\emptyset}) \Rightarrow F(F^{\alpha}(f_{\emptyset})) =_{m,\alpha+1} F(F^{\alpha+1}(f_{\emptyset}))$. Therefore, $F^{\alpha+1}(f_{\emptyset}) =_{m,\alpha+1} F^{\alpha+2}(f_{\emptyset})$, which implies by the lemma that $\forall \beta > \alpha + 1$,
  $F^{\alpha+1}(f_{\emptyset}) =_{m,\alpha+1} F^{\beta}(f_{\emptyset})$. Thus, $\Psi[\alpha] \rightarrow \Psi[\alpha + 1]$.

- For any limit ordinal α ($\forall \gamma < \alpha, \Psi[\gamma] \rightarrow \Psi[\alpha]$): Let $v = \langle x_1, ..., x_n \rangle, |x_m| < \alpha$, be an arbitrary n-tuple. Since $|x_m| < \alpha$ and α is a limit ordinal, $|x_m| + 1 < \alpha and therefore \Psi[|x_m| + 1]$ is true, if we let γ be $|x_m| + 1$. Therefore, $\forall \beta > \gamma, F^{\gamma}(f_{\emptyset}) =_{m,\gamma} F^{\beta}(f_{\emptyset})$. Knowing that $\gamma < \alpha < \alpha + 1$, we obtain

$$F^{\alpha}(f_{\emptyset}) =_{m,\gamma} F^{\gamma}(f_{\emptyset}) =_{m,\gamma} F^{\alpha+1}(f_{\emptyset})$$

Thus, since v is arbitrary,

$$\forall v = \langle x_1, ..., x_n \rangle \text{ s.t. } |x_m| < \alpha, [F^\alpha(f_\emptyset)](v) = [F^{\alpha+1}(f_\emptyset)](v)$$

which is by definition

$$F^\alpha(f_\emptyset) =_{m,\alpha} F^{\alpha+1}(f_\emptyset)$$

By the lemma, this implies that $\Psi[\alpha]$ is true.

■

Corollary 1 Let F be a well-founded functional on argument m. There exists α such that $F^\alpha(f_\emptyset)$ is a fixed point.

Proof: Since F is a well-founded functional, for any ordinal α $F^\alpha(f_\emptyset) =_{m,a} F(F^\alpha)(f_\emptyset)$. Let $\alpha = |\tau_m|$. Since $\forall v = \langle x_1, ..., x_n \rangle \in \tau_1 \times ... \times \tau_n$, $|x_m| \in \tau_m \Rightarrow |x_m| < \alpha$,

$$\forall v, F^\alpha(f_\emptyset)(v) = [F(F^\alpha)(f_\emptyset)](v)$$

Therefore, $F^\alpha(f_\emptyset) = F^{\alpha+1}(f_\emptyset)$, which means that $F^\alpha(f_\emptyset)$ is a fixed point. ■

Definition 6 Let F be a well-founded functional with measure argument m and let $\alpha = |\tau_m|$. We define the function $F^\alpha(f_\emptyset)$, the fixed point of F, to be the *recursive function definition* of F. We use the notation $Y(F)$, for any well-founded functional, to denote the recursive function definition of F.

Chapter 3

First Order Logic

One issue that wasn't addressed in our termination proof for **gcd** was how to prove that **remainder** has the desired properties. Since we are only concerned with termination, it suffices therefore to show that the **remainder** function terminates and returns a result that guarantees that the measure argument of **gcd** decreases on every recursive call. The termination properties of **remainder** can be shown by the same method that we used for **gcd**; specifically, verifying that **remainder** is Walther recursive. However, proving that the result returned by the **remainder** function is less than the measure argument in the general case requires computing the value of the function itself, which can require a large amount of computation time, or in the worse case, be an undecidable problem. Therefore, we are forced to result to heuristic algorithms, and perhaps more importantly, efficient methods of estimating the size of a value.

Thus, the main idea behind Walther recursion is the ability to use a *calculus of estimation*, a syntactic and therefore decidable method of giving bounds on the size of terms. The calculus of estimation must be sound but cannot, as discussed above, be complete. It is based on the idea of *argument estimation rules.* These rules indicate when the results of functions are guaranteed to be smaller than or equal to the value of one of the arguments. These functions are known as *argument-bounded function symbols.* Proving termination requires only that we determine the relative, not absolute, size of a term when compared to the measure argument, allowing these estimation

rules to be sufficient. However, since proving termination requires a strict inequality, we associate a *difference literal* with each argument-bounded function symbol which indicates when the inequality is strict.

We recognize argument-bounded function symbols either axiomatically, as primitive operators (selectors) on primitive data structures, or algorithmically, through a case analysis proof that shows for all possible situations, the result of the function is smaller than or equal to the value of the argument in question. These comparative judgments can be made through the calculus of estimation. Furthermore, for recursive functions we can perform inductive proofs that the function is argument-bounded provided the argument be sent into the recursive call is smaller than the argument in question. It is also sometimes necessary to manipulate algorithms in other manners in order to make them argument-bounded.

Therefore, our discussion now turns from a mathematical standpoint to a practical standpoint where decidability and efficiency are the primary concerns, making the semantic value of a term not nearly as important as a syntactic approximation of it. It is therefore necessary to first define a "practical" language which is the basis for our discussion of the syntactic methods used in Walther recursion to prove termination. Thus, we first turn our attention towards this task. Once that is complete, we describe the properties, or meta-formulas, which we need to infer for a given term or definition in the language in order to prove argument-boundedness or well-foundedness. We also provide a set of inference rules in order to give a sound, but not complete, method of inferring the properties/meta-formulas under consideration. Finally, we briefly discuss the efficient implementation of these rules in order to show that such a system is usable practically.

3.1 Language

We choose the language of first order logic extended to include set theory because of its simple semantics yet ability to express the concepts in many advanced mathematical disciplines. Furthermore, it's functional paradigm is better suited to Walther

recursion than the non-functional approach many languages take. Although there is a distinct disadvantage using Walther recursion in non-functional languages, we note that automated termination proofs based on well-orderings in such languages have been successful[7]. Therefore, it is not unreasonable to believe that Walther is extendable to such languages.

Nevertheless, in situations where a functional approach is taken, Walther recursion is much more natural and concise. Therefore, our treatment of its applicability to a language in which a high degree of expressibility and flexibility is possible hopefully illustrates that the concepts behind Walther recursion are essentially language independent.

3.1.1 Description

A language in first-order logic as defined by Barwise[1] of a set L of constant symbols, function symbols, and relation symbols. Since we are defined a language using first-order logic in the context of set theory, we define our constant symbols to be sets. Sets can be constructed using the standard notion of braces, $\{\}$, with the elements of the set enclosed within the braces, or through pre-defined 0-ary function symbols. Each function symbol f has a non-negative integer, n, assigned to it, where f is considered an n-ary function symbol. Let M = the collection V of all sets. Then if $f \in L$ is an n-ary function symbol, then $f : M^n \rightarrow M$. Likewise, each relation symbol R has a positive integer, n assigned to it, where R is considered an n-ary relation symbol. We deviate here from Barwise's definition of relation symbols by defining symbols, **True** and **False** as constant symbols such that if $R \in L$ is an n-ary relation symbol, then $R : M^n \rightarrow \{\mathbf{True}, \mathbf{False}\}$. This is equivalent to Barwise's definition of relation symbols, R' through the mappings $R' = \{x : R(x) = \mathbf{True}\}$ and $R = \{(x, y) : x \in M^n, y = \mathbf{True}$ if $x \in R', y = \mathbf{False}$ otherwise$\}$.

By doing so, we have made *formulas* as defined in Barwise, to be a subset of *terms*. Terms, which are always sets, are thus one of the following:

- a constant symbol

- the result of a function which maps terms to terms
- a formula
- a set construct using {}
- the result of an **if** operator
- the result of an $\bigcup$ operator

Formulas are the result of mapping terms to a boolean value through a relation symbol, the result of the standard equality, $=$, operator, the result of the set-theoretic membership $\in$ operator applied to two terms, or those formulas created by combining the standard logic operators $\neg, \vee$ and quantifier $\exists$.

Functions in our language are either function symbols, or the result of the Y-operator (defined below) applied to a functional. Functionals are essentially equivalent to those defined in the previous chapter and their use is restricted to generating recursive functions.

In order to have some sort of control structure, we also define an **if** operator which takes on its intuitive meaning. Specifically, it is a 3-argument operator which takes a formula and two terms and returns the first term is the formula is not **False**, and returns the second term if the formula is **False**. Note that as a consequence since formulas need not return a value in {**True**,**False**}, any value not in this set is considered a **True** value.

Although we have the ability to construct sets currently using our language, we still lack method of extracting elements from these sets. Unfortunately, there is no general way of doing so without using the Axiom of Choice. However, the doing so would require using of an arbitrary choice function which, for defining a language, it not a very appealing choice. Therefore, we avoid the notion of a choice function altogether and define a $\bigcup$ operator which takes in a variable, v and two terms, t_1 and t_2 and returns the union of all possible values of t_2, with the variable v bound to some element of t_1. Mathematically, this can be represented as $\bigcup_{v \in t_1} t_2$ where v has its specified meaning in t_2. Although at first glance this operator does not provide

$$
\begin{array}{lcl}
t & := & x \mid (F\ t_1 \ldots t_n) \mid (\text{if } \Phi\ t_1\ t_2) \mid \{t_1, \ldots, t_n\} \mid (\bigcup\ x\ t_1\ t_2) \\
\Phi & := & (=\ t_1\ t_2) \mid (\in\ t_1\ t_2) \mid (\neg\ \Phi) \mid (\wedge\ \Phi_1\ \Phi_2) \mid (\exists\ x\ t) \\
F & := & f \mid (\lambda\ (x_1 \ldots x_n)\ t) \mid (Y\ f\ F) \\
x & := & \text{first-order variable} \\
f & := & \text{function variable}
\end{array}
$$

Figure 3-1: Grammar of the Language

any clear method for extracting elements from a set, it is sufficient for most purposes as will be demonstrated below.

We proceed with a more formal grammar and semantics for the language.

3.1.2 Grammar

Figure 3-1 gives a complete grammar for our language as described in section 3.1.1. Note that t denote terms, F denote functions, Φ denote formulas, and f denote function symbols. Also note that syntactically, formulas and terms are equivalent. Semantically, formulas differ from terms in that formulas are used as the first argument to the **if** operator, whereas terms are used in all other situations. This distinction is made in order to clarify the presentation of the inference rules in section 3.3.

3.1.3 Operational Semantics

We define a meaning function associated with each grammar rule. The meaning of each of our terms corresponds directly to its equivalent meaning in first order logic as described in section 3.1.1. Refer to Figure 3-2.

We now state two key properties of our language that will be important in understanding our language and establishing the inference rules. First, although we stated in the previous chapter that sets are well-founded given a size order based on the ordinals, we still do not have a method of establishing the relative size of two sets. The below theorem, stated without proof, is from [2]:

Theorem 3 Let x be any set. Then $\forall y \in x, |y| < |x|$.

The next theorem is for recursive functions defined within our language. Semantically, our definition of a recursive function corresponds to that of Corollary 1. We must therefore show that the definition has the fixed point property that we need.

Theorem 4 Let $(\mathrm{Y}\ f\ (\lambda\ (x_1\ ...\ x_n)\ t))$ be a recursive function definition which has semantic meaning $f_m = M[(\mathrm{Y}\ f\ (\lambda\ (x_1\ ...\ x_n)\ t)), \rho]$ under some interpretation, ρ. Given the functional $F(f') = M[(\lambda\ (x_1\ ...\ x_n)\ t), \rho[f := f']]$, if F is well-founded, then $F(f_m) = f_m$.

Proof: Since the functional $F(f') = M[(\lambda\ (x_1\ ...\ x_n)\ t), \rho[f := f']]$ is well-founded, by Corollary 1, a fixed point $F^{\alpha}(f_{\emptyset})$ exists for the functional. Because

$$\forall \gamma > \alpha, (a, b) \in F^{\gamma}(f_{\emptyset})$$

$F^{\alpha}(f_{\emptyset}) \subseteq M[(\mathrm{Y}\ f\ (\lambda\ (x_1\ ...\ x_n)\ t)), \rho]$. Furthermore, since

$$\forall \beta > \alpha, F^{\beta}(f_{\emptyset}) = F^{\alpha}(f_{\emptyset})$$

$M[(\mathrm{Y}\ f\ (\lambda\ (x_1\ ...\ x_n)\ t)), \rho] \subseteq F^{\alpha}(f_{\emptyset})$. Therefore, $(\mathrm{Y}\ f\ (\lambda\ (x_1\ ...\ x_n)\ t))$ is a fixed point of F. ■

3.1.4 Application

With the grammar and semantics of our language determined, we once again return to our definition of **gcd** and **remainder** to illustrate how functions are defined in the language. In order to keep the syntax of our language as similar as possible to the syntax used in our examples in preceeding chapters, we require several more conventions, though not within the language itself, enhance readability and allow for easier understanding. First, we assume that for any function variable, f, arguments, $x_1, ..., x_n$, and term, t, the following expression:

(define (f x_1 ... x_n) t)

$$
\begin{aligned}
M[(F\ x_1\ ...\ x_n), \rho] &= \begin{cases} f'(x'_1, ..., x'_n) \\ \quad \text{where } f' = M[F, \rho] \\ \qquad x'_1 = M[x_1, \rho], ..., x'_n = M[x_n, \rho] \end{cases} \\
M[(\bigcup\ x t_1 t_2), \rho] &= \begin{cases} \bigcup_{x' \in t'_1} t'_2 \\ \quad \text{where } t'_1 = M[t_1, \rho], t'_2 = M[t_2, \rho[x = \\ \qquad x']] \end{cases} \\
M[(\text{if } \Phi\ t_1\ t_2), \rho] &= \begin{cases} M[t_2, \rho] \text{ if } M[\Phi, \rho] = \textbf{False} \\ M[t_1, \rho] \text{ otherwise} \end{cases} \\
M[\{t_1, ..., t_n\}, \rho] &= \begin{cases} \{t'_1, ..., t'_n\} \\ \quad \text{where } t'_1 = M[t_1, \rho], ..., t'_n = M[t_n, \rho] \end{cases} \\
M[(\mathrm{Y}\ f\ l), \rho] &= \begin{cases} \{(a, b) : \exists \beta \text{ s.t. } \forall \gamma > \beta, (a, b) \in F^\gamma(f_\emptyset)\} \\ \quad \text{where } F = \{(f_x, f_y) : f_y = M[l, \rho[f := f_x]]\} \end{cases} \\
M[(\lambda\ (x_1\ ...\ x_n)\ t), \rho] &= \{(\langle x'_1, ..., x'_n \rangle, y') : y' = M[t, \rho[x_1 = x'_1, ..., x_n = x'_n]]\} \\
M[x, \rho] &= \rho(x) \\
M[f, \rho] &= \rho(f) \\
M[(=\ t_1\ t_2), \rho] &= \begin{cases} \textbf{True} \text{ if } M[t_1, \rho] = M[t_2, \rho] \\ \textbf{False} \text{ otherwise} \end{cases} \\
M[(\in\ t_1\ t_2), \rho] &= \begin{cases} \textbf{True} \text{ if } M[t_1, \rho] \in M[t_2, \rho] \\ \textbf{False} \text{ otherwise} \end{cases} \\
M[(\neg\ \Phi), \rho] &= \begin{cases} \textbf{False} \text{ if } M[\Phi, \rho] = \textbf{True} \\ \textbf{True} \text{ otherwise} \end{cases} \\
M[(\wedge\ \Phi\ \Psi), \rho] &= \begin{cases} \textbf{True} \text{ if } M[\Phi, \rho] = \textbf{True} \text{ and } M[\Psi, \rho] = \textbf{True} \\ \textbf{False} \text{ otherwise} \end{cases} \\
M[(\exists\ x\ \Phi), \rho] &= \begin{cases} \textbf{True} \text{ if } \exists z, \text{ s.t. } M[\Phi, \rho[x := z]] = \textbf{True} \\ \textbf{False} \text{ otherwise} \end{cases}
\end{aligned}
$$

Figure 3-2: Semantics of the Language

to mean the function symbol, f, is a pre-defined symbol with meaning equivalent to

$$(\mathrm{Y}\ f\ (\lambda\ (x_1\ \ldots\ x_n)\ t))$$

Likewise, for constant symbols, we assume for any constant symbol, c, and term, t, the following expression:

```
(define c t)
```

indicates the constant symbol, c, is pre-defined to have the meaning of t. Now, we have the ability to define our functions with the exact same syntax as before:

```
(define (gcd a b)
  (if (= a 0)
      b
      (gcd (remainder b a) a)))

(define (remainder x y)
  (if (= y 0)
      y
      (if (< x y)
          x
          (remainder (- x y) y))))
```

Furthermore, since functions cannot return formulas because terms and functions are distinct categories in the grammar, we shall adopt another more notational convenience. Whenever a term, t is used as a formula, we assume it's meaning is the formula $(=\ t\ \mathbf{True})$. On the other hand, whenever a formula, Φ is used as a term, we assume it's meaning is the term (if Φ **True False**).

Although the syntax of the language is now the same, we still require a semantic notion for the constant symbol, 0, as well as the function symbols, $<$ and $-$. Doing so requires some formal structure for defining the set of non-negative integers. We do so using the standard definition of the set through the ordinals:

```
(define 0 {})
```

```
(define (succ x)
  (⋃ y x {x,y}))

(define (pred x)
  (⋃ y x
    (if (= (succ y) x)
        {y}
        {})))
```

Careful examination will show these definitions are equivalent to their set-theoretic equivalents.

In order to create a more generalized notion of a data structure, we show how to use our primitive set structures to define an ordered pair, which is equivalent to the Lisp cons cell.

Since sets do not have any inherent ordering, to creating an ordered pair, $\langle x, y \rangle$, it is necessary to impose this ordering on the elements. One common method of doing so is by defining $\langle x, y \rangle$ to be $\{x, \{x, y\}\}$. Therefore, we obtain the following definition for a function which constructs ordered pairs, **cons**:

```
(define (cons x y)
  {x,{x,y}})
```

We also need some method of extract the respective elements, x and y, from the pair. The functions, **car** and **cdr** extract the first and second elements, respectively, from an ordered pair. We define them as follows:

```
(define (car x)
  (if (= x nil)
      nil
      (⋃ a x
        (⋃ b x
          (if (= a b)
              {}
              (⋃ c b
                (if (= c a)
                    c
                    {})))))))
```

```
(define (cdr x)
  (if (= x nil)
      nil
      (⋃ a x
        (⋃ b x
          (if (= a b)
              {}
              (⋃ c b
                (if (= c a)
                    {}
                    c)))))))
```

Close examination will show that they return the correct results. Specifically, given the pair $\{x, \{x, y\}\}$, **car** will extract the $x \in \{x, y\}$ and **cdr** will extract the $y \in \{x, y\}$.

In languages such as Lisp which use ordered pairs, pairs are then chained together through their second element to form lists, or ordered sets. A convention for an empty list, **nil** must therefore be decided. The obvious choice is:

```
(define nil {})
```

Although these functions do not return any sensical value if given improper arguments, it is simple with the above conventions to create a well-typed set of data structures which could verify that its arguments are of the proper type. These conventions could also be also be defined within the language but because they provide no additional functionality, there is no reason for doing so.

3.2 Meta-Formulas

The meta-formulas for our language are designed to prove that terms within our language have the necessary formal characteristics to prove well-foundedness of functionals. These meta-formulas must be provably sufficient conditions under which sound inference rules can be established that functionals within our language are indeed well-founded.

In order for well-foundedness to be established, we must establish a well-ordering on one of the arguments of the functional and then show a measure argument for the functional exists. Proving that an argument is a measure argument requires showing that in every recursive call made to the functional, the argument is smaller as established by the well-ordering.

As discussed, these well-orderings must be based on some sort of size order. Since our language uses sets as its elements, our size order is defined to be the ordinal size of the set because of the simple semantics it provides, thereby allowing a correspondingly simple syntactic meaning. Although such an ordering may not be possible in other languages, it is easy to show that similar orderings may be established in more practical languages. For example, in Walther's paper, data structures based on Boyer Moore's shell principle were used. Furthermore, as discussed above, we have the ability to define the data structures present in most other languages through our use of sets. If these structures are represented in the "proper" manner, as is the case with the ordered pair and non-integer integer sets defined above, our well-ordering for sets will be applicable as well-ordering for other structures.

3.2.1 Description

We need proceed to define the necessary meta-formulas needed in order to show well-foundedness of a functional. Specifically, we provide formulas to establish sets are smaller than other sets based on the well-ordering we have imposed and we provide formulas to prove that a measure argument exists for a functional.

Our first meta-formula establishes the syntactical method of determining when a set is relatively smaller than another, based on the semantic well-ordering of sets using the ordinals. Being syntactically based, it is an approximation of the relative size; specifically, we establish meta-formulas for a lower bound on the degree which one set is smaller than another as well for an upper bound on the degree which one set is larger than another.

Definition 7 Let t_1 and t_2 be terms. For any non-negative integer a, we define

$t_1 \leq_a t_2$ to represent the fact that $|t_1| + a \leq |t_2|$.

Definition 8 Let t_1 and t_2 be terms. For any negative integer a, let $b = -a$. We define $t_1 \leq_a t_2$ to represent the fact that $|t_1| \leq b + |t_2|$.

Although the above definitions provide the basic means necessary to determine the relative size of sets, we still need some method of determining when the relative size of the result of a function compared to its arguments. The solution to this problem Walther presents is establish a property which determines when the result of a function is guaranteed to be less than or equal to a given argument. Functions with this property are called *argument-bounded* functions. Likewise, we establish similar meta-formulas which describes functions which are argument-bounded.

Definition 9 Let f be an n-ary function and $1 \leq i \leq n$ be a number representing the ith argument of f. For any non-negative integer a, we define $f \leq_a i$ to represent the fact that for all $x_1, ..., x_n$, $|f(x_1, ..., x_n)| + a \leq |x_i|$.

Definition 10 Let f be an n-ary function and $1 \leq i \leq n$ be a number representing the ith argument of f. For any negative integer a, let $b = -a$. We define $f \leq_a i$ to represent the fact that for all $x_1, ..., x_n$, $|f(x_1, ..., x_n)| \leq b + |x_i|$.

Proving a recursive function terminates requires that in every recursive call, one of the arguments is *strictly* decreases. In Walther's approach to determining which argument satisfies such a condition, using argument-bounded functions is sufficient since they guarantee only that the result is less than or equal to the given argument. Therefore, some method of determining when an argument-bounded function returned a result strictly less than a given argument was required. This required the use of *difference literals* which indicate conditions on the arguments to a function which guarantee that the result returned is strictly less than a given argument.

Although our definition of argument-bounded functions technically allows for functions to guarantee than the result is strictly less than an argument, such functions cannot exist since any function can accept the empty set as an argument. Thus, the

use of difference literals is still necessary. For example, for a simple operation like **cdr**, the result returned is argument-bounded, but unless we guarantee that the argument sent to **cdr** is not **nil**, we fail to recognize that **cdr** returns a value which is strictly less than its argument for all other cases. Thus, we regard argument-bounded functions to yield an *initial approximation* to the size of a function's result relative to its arguments, and use difference literals to attempt to infer further possible information about the size of the result if some set of conditions are satisfied.

First, we define a meta-formula which allows us to recognize functions which return results which will be used as formulas. Since formulas use the boolean constants, **True** and **False**, we just need to recognize their occurrence in functions.

Definition 11 Let f be an n-ary function and $1 \leq i \leq n$ be a number representing the ith argument of f. For any non-negative integer a and any predicate function, Φ, also of arity n, we define value (f, Φ, t) to represent the fact that for all $x_1, ..., x_n$, $\Phi(x_1, ..., x_n) = \textbf{True} \Rightarrow f(x_1, ..., x_n) = t$.

Next we define a difference literal. A difference literal indicates that if a set of conditions on the arguments to a function are true, then a tighter bound on the size of the return value for the function can be achieved.

Definition 12 Let f be an n-ary function and $1 \leq i \leq n$ be a number representing the ith argument of f. For any non-negative integer a and any predicate function, Φ, also of arity n, we define $\Delta(f, \Phi, \leq_a, i)$ to represent the fact that for all $x_1, ..., x_n$, $\Phi(x_1, ..., x_n) = \textbf{True} \Rightarrow |f(x_1, ..., x_n)| + a \leq |x_i|$.

Definition 13 Let f be an n-ary function and $1 \leq i \leq n$ be a number representing the ith argument of f. For any negative integer a, where $b = -a$, and any predicate function, Φ, also of arity n, we define $\Delta(f, \Phi, \leq_a, i)$ to represent the fact that for all $x_1, ..., x_n$, $\Phi(x_1, ..., x_n) = \textbf{True} \Rightarrow |f(x_1, ..., x_n)| \leq b + |x_i|$.

Now that we have a complete set of meta-formulas for establishing the relative size of

$$
\begin{array}{lll}
T & := & t \leq_a t \mid F \leq_a n \mid \text{value}\,(F, \Phi, t) \mid \\
 & & \Delta(F, \Phi, \leq_a, n) \mid \text{wf}\,((\lambda\,(f)\,t), x, n) \\
a & := & \text{an integer} \\
n & := & \text{a positive integer} \\
t, \Phi, F, x & := & \text{non-terminals from the language}
\end{array}
$$

Figure 3-3: Grammar of the Meta-Formulas

sets, we define a meta-formula to indicates when an argument to a recursive function is strictly decreasing on every recursive call.

Definition 14 Let t be a term occurring in the body of some n-ary function f and x be the ith formal parameter of f. We say that t makes *strictly decreasing calls* with respect to the function f by the ith argument, x, written as $\text{wf}\,((\lambda\,(f)\,t), x, i)$ if and only if the function $F = (\lambda\,(f)\,t)$ which maps functions to terms, given any functions f,g and any interpretation of the variables, ρ, $f =_{i,\alpha} g \Rightarrow F'(f) = F'(g)$.

3.2.2 Grammar

Figure 3-3 provides a complete grammar for our meta-formulas as defined in section 3.2.1.

3.2.3 Semantics

We define a meaning function associated with each grammar rule. The meaning of each of our meta-formulas corresponds directly to our equivalent definitions as described in section 3.2.1. This table summarizes those definitions. Refer to Figure 3-4.

3.3 Inference Rules

Our inference rules provide a method of automating our termination proofs by establishing a syntactic method of recognizing when a meta-formula is satisfied by a term

$$M[t_1 \leq_a t_2, \rho] = \begin{cases} \textbf{True} \text{ if } a \geq 0 \text{ and } |M[t_1,\rho]| + a \leq |M[t_2,\rho]| \\ \textbf{True} \text{ if } a < 0 \text{ and } |M[t_1,\rho]| \leq b + |M[t_2,\rho]|, b = -a \\ \textbf{False} \text{ otherwise} \end{cases}$$

$$M[F \leq_a n, \rho] = \begin{cases} \textbf{True} \text{ if } a \geq 0 \text{ and } \forall x_1, ..., x_n, |f(x_1, ..., x_n)| + a \leq |x_i| \\ \textbf{True} \text{ if } a < 0 \text{ and } \forall x_1, ..., x_n, |f(x_1, ..., x_n)| \leq b + |x_i| \\ \quad \text{where } \begin{array}{l} f = M[F,\rho], \\ b = -a \end{array} \\ \textbf{False} \text{ otherwise} \end{cases}$$

$$M[\text{value } (F, \Phi, t), \rho] = \begin{cases} \textbf{True} \text{ if } \forall u_1, ..., u_n, \Phi'(u_1, ..., u_n) \Rightarrow f'(u_1, ..., u_n) = t' \\ \quad \text{where } \begin{array}{l} t' = M[t,\rho], \\ f' = M[F,\rho], \\ \Phi' = M[(\lambda\ (x_1\ ...\ x_n)\ \Phi), \rho], n = \text{arity of } f' \end{array} \\ \textbf{False} \text{ otherwise} \end{cases}$$

$$M[\Delta(F, \Phi, \leq_\alpha, i), \rho] = \begin{cases} \textbf{True} \text{ if } \forall u_1, ..., u_n, \Phi'(u_1, ..., u_n) \Rightarrow f'(u_1, ..., u_n) \leq_\alpha u_i \\ \quad \text{where } \begin{array}{l} f' = M[F,\rho], \\ \Phi' = M[(\lambda\ (x_1\ ...\ x_n)\ \Phi), \rho], n = \text{arity of } f' \end{array} \\ \textbf{False} \text{ otherwise} \end{cases}$$

$$M[\text{wf } ((\lambda\ (f)\ t), x, i), \rho] = \begin{cases} \textbf{True} \text{ if } \forall f, g, \forall \alpha f =_{i,\alpha} g \Rightarrow F(f) = F(g) \\ \quad \text{where } F(f') = M[t, \rho[f := f']] \\ \textbf{False} \text{ otherwise} \end{cases}$$

Figure 3-4: Semantics of the Meta-Formulas

through the semantic meanings ascribed to the meta-formulas in section 3.2.1. This is equivalent to Walther's notion of the calculus of estimation in addition to his method of generating termination hypotheses. We first describe the entire set of inference used for our language in the following section. A discussion of the soundness of the rules is given in Appendix A.

3.3.1 Description

Boolean Rules for Formulas

These rules define standard inferences in boolean logic. They are used when attempting to match formulas for difference literals.

- $$\frac{\Sigma \vdash \Phi \quad \Sigma \vdash \Psi}{\Sigma \vdash (\wedge\ \Phi\ \Psi)}$$

- $$\frac{\Sigma \vdash \Phi \quad \Sigma \vdash (\neg\ (\wedge\ \Phi\ \Psi))}{\Sigma \vdash (\neg\ \Psi)}$$

Argument Bounded Terms

These rules establish when comparison rules between two terms with respect to the well-founded order already established. In addition to rules for showing that one term is less than another, the reflexive and transitive properties of the well-orderings is used.

- $(=\ x\ x)$

- $$\frac{\Sigma \vdash (=\ x\ y)}{\Sigma \vdash x \leq_0 y}$$

- $$\frac{\Sigma \vdash (=\ x\ y)}{\Sigma \vdash y \leq_0 x}$$

- $\Sigma \vdash (\in\ x\ y)$

  $\overline{\Sigma \vdash x \leq_1 y}$

- $\Sigma \vdash (\in\ x_i\ \{x_1, ..., x_n\})$

- $\Sigma, (\in\ x\ t_1) \vdash t_2 \leq_a t_1$

  $\overline{\Sigma \vdash (\bigcup\ x\ t_1\ t_2) \leq_{a+1} t_1}$

- $\Sigma, \Phi \vdash t_1 \leq_a t$

  $\Sigma, (\neg\ \Phi) \vdash t_2 \leq_b t$

  $\overline{\Sigma \vdash (\text{if}\ \Phi\ t_1\ t_2) \leq_{min(a,b)} t}$

- $\Sigma \vdash x \leq_a y$

  $\Sigma \vdash y \leq_b z$

  $\overline{\Sigma \vdash x \leq_{a+b} z}$

Argument Bounded Functions

These rules establish when a function is argument-bounded with respect to one of its arguments. In addition to rules for pre-defined functions, rules for recursively defined functions, i.e. an induction rule for argument-bounded functions, are also used. This is equivalent to Walther's use of *recursion elimination* to optimize difference algorithms.

- $\Sigma \vdash f \leq_a i$

  $\Sigma \vdash s_i \leq_b t$

  $\overline{\Sigma \vdash f(u_1, ..., u_{i-1}, s_i, u_{i+1}, ..., u_n) \leq_{a+b} t}$

- $\Sigma, f \leq_0 i \vdash t \leq_a x_i$

  $\overline{\Sigma \vdash (\text{Y}\ f\ (\lambda\ (x_1\ ...\ x_n)\ t)) \leq_a i}$

Formulas

These rules establish a method of determining the primitive formulas which are true when a non-primitive function is used as a formula. Establishing the formulas that

hold for variables in any portion of a definition is essential for successful use of difference literals.

- $\Sigma \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ \mathbf{True}), \mathbf{True}, \mathbf{True})$
- $\Sigma \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ \mathbf{False}), \mathbf{True}, \mathbf{False})$
- $\dfrac{\Sigma, \Phi \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ w), \Psi, t)}{\Sigma \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ (\text{if}\ \Phi\ w\ u)), (\wedge\ \Phi\ \Psi), t)}$
- $\dfrac{\Sigma, (\neg\ \Phi) \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ u), \Psi, t)}{\Sigma \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ (\text{if}\ \Phi\ w\ u)), (\wedge\ (\neg\ \Phi)\ \Psi), t)}$
- $\dfrac{\Sigma \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ t_1), \Psi, \mathbf{True}) \quad \Sigma \vdash (=\ f(u_1, ..., u_n)\ \mathbf{False})}{\Sigma \vdash (\neg\ \Psi[u_1/x_1\ ...\ u_n/x_n])}$
- $\dfrac{\Sigma \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ t_1), \Psi, \mathbf{False}) \quad \Sigma \vdash (=\ f(u_1, ..., u_n)\ \mathbf{True})}{\Sigma \vdash (\neg\ \Psi[u_1/x_1\ ...\ u_n/x_n])}$

Difference Literals

These rules establish the difference literals of a function through a case by case analysis. Induction rules are established for recursive functions through the use of argument-bounded functions.

- $\dfrac{\Sigma \vdash t \leq_a x_i}{\Sigma \vdash \Delta((\lambda\ (x_1\ ...\ x_n)\ t), \mathbf{True}, \leq_a, i)}$
- $\dfrac{\Sigma, \Phi \vdash \Delta((\lambda\ (x_1\ ...\ x_n)\ w), \Psi, \leq_a, i)}{\Sigma \vdash \Delta((\lambda\ (x_1\ ...\ x_n)\ (\text{if}\ \Phi\ w\ u)), (\wedge\ \Phi\ \Psi), \leq_a, i)}$
- $\dfrac{\Sigma, (\neg\ \Phi) \vdash \Delta((\lambda\ (x_1\ ...\ x_n)\ u), \Psi, \leq_a, i)}{\Sigma \vdash \Delta((\lambda\ (x_1\ ...\ x_n)\ (\text{if}\ \Phi\ w\ u)), (\wedge\ (\neg\ \Phi)\ \Psi), \leq_a, i)}$

- $\Sigma \vdash \Delta((\lambda\ (x_1\ ...\ x_n)\ t), \Psi, \leq_a, i)$

  $\Sigma \vdash \Psi[u_1/x_1\ ...\ u_n/x_n]$

  $\overline{\Sigma \vdash f(u_1, ..., u_n) \leq_a u_i}$

Well-Founded Functionals

These rules determine when a functional is well-founded. Aside from the basic definition of well-foundedness, rules are needed in order to establish well-foundedness in expressions which allows means of combining terms. Thus, since our means of combination within our language allow an arbitrary number of arguments in a few cases, several of these rules are, in some sense, meta-rules, because they are actually a set of rules rather than a single rule.

- $\Sigma \vdash \text{wf}\ ((\lambda\ (f)\ w_1), x, i)$

  $\vdots$

  $\Sigma \vdash \text{wf}\ ((\lambda\ (f)\ w_n), x,)$

  $\Sigma \vdash w_i \leq_{a>0} x$

  $\overline{\Sigma \vdash \text{wf}\ ((\lambda\ (f)\ f(w_1, ..., w_n)), x, i)}$

- $\Sigma \vdash \text{wf}\ ((\lambda\ (f)\ w_1), x, i)$

  $\vdots$

  $\Sigma \vdash \text{wf}\ ((\lambda\ (f)\ w_n), x, i)$

  $g \neq f$

  $\overline{\Sigma \vdash \text{wf}\ ((\lambda\ (f)\ g(w_1, ..., w_n)), x, i)}$

- $\Sigma \vdash \text{wf}\ ((\lambda\ (f)\ \Phi), x, i)$

  $\Sigma, (\neg\ \Phi) \vdash \text{wf}\ ((\lambda\ (f)\ u), x, i)$

  $\Sigma, \Phi \vdash \text{wf}\ ((\lambda\ (f)\ w), x, i)$

  $\overline{\Sigma \vdash \text{wf}\ ((\lambda\ (f)\ (\text{if}\ \Phi\ w\ u)), x, i)}$

- $\Sigma \vdash \text{wf}\ ((\lambda\ (f)\ \Phi), x, i)$

  $\Sigma \vdash \text{wf}\ ((\lambda\ (f)\ \Psi), x, i)$

  $\overline{\Sigma \vdash \text{wf}\ ((\lambda\ (f)\ (\wedge\ \Phi\ \Psi)), x, i)}$

- $\Sigma \vdash \mathrm{wf}\,((\lambda\,(f)\,t_1), x, i)$

  $\Sigma \vdash \mathrm{wf}\,((\lambda\,(f)\,t_2), x, i)$

  $\overline{\Sigma \vdash \mathrm{wf}\,((\lambda\,(f)\,(\in\ t_1\ t_2)), x, i)}$

- $\Sigma \vdash \mathrm{wf}\,((\lambda\,(f)\,\Phi), x, i)$

  $\overline{\Sigma \vdash \mathrm{wf}\,((\lambda\,(f)\,(\neg\,\Phi)), x, i)}$

- $\Sigma \vdash \mathrm{wf}\,((\lambda\,(f)\,\Phi), x, i)$

  $\overline{\Sigma \vdash \mathrm{wf}\,((\lambda\,(f)\,\exists x \Phi), x, i)}$

- $\Sigma \vdash \mathrm{wf}\,((\lambda\,(f)\,w_1), x, i)$

  $\vdots$

  $\Sigma \vdash \mathrm{wf}\,((\lambda\,(f)\,w_n), x, i)$

  $\overline{\Sigma \vdash \mathrm{wf}\,((\lambda\,(f)\,\{w_1, ..., w_n\}), x, i)}$

- $\mathrm{wf}\,((\lambda\,(f)\,x), x', i)$

3.3.2 Examples

We first illustrate the use of our inference rules using the example that has been discussed throughout this paper:

```
(define (gcd a b)
  (if (= a 0)
     b
     (gcd (remainder b a) a)))
```

where the **remainder** function is defined as:

```
(define (remainder x y)
  (if (= y 0)
      y
      (if (< x y)
          x
          (remainder (- x y) y))))
```

In addition, we require the $<$ and $-$ operators to be defined. Standard definitions might be:

```
(define (< x y)
  (if (= y 0)
      False
      (if (= x 0)
          True
          (< (pred x) (pred y)))))

(define (- a b)
  (if (= a 0)
      0
      (if (= b 0)
          a
          (- (pred a) (pred b)))))
```

Ideally, such definitions of $<$ and $-$ would generate inferences that lead to proofs that **remainder** and **gcd** terminate. Although $-$ will trigger rules which allow the proper inferences to be made, $<$ will not. The reason is that $<$ is a *primitive comparator* for the set of non-negative integers. As a result, it must establish the well-ordering on the set, which is cannot do given our current representation of integers combined with the absence of type-checking. Therefore, we must define set-theoretically $<$ as:

```
(define (< a b)
  (∈ a b))
```

We summarize the inference made to prove that **gcd** terminates below:

- $\vdash (\text{pred } a) \leq_1 a$
- $\vdash \text{wf}\,(-, a, 1)$
- $\vdash - \leq_0 1$
- $\vdash \Delta(-, (\wedge\ (\neg\ (=\ y\ 0))\ (\neg\ (=\ x\ 0))), \leq_1, 1)$
- $\vdash (\text{-}\ x\ y) \leq_1 x$
- $\vdash \text{wf}\,(\text{remainder}, x, 1)$
- $\vdash \text{remainder} \leq_0 1$

- $\vdash \Delta(\text{remainder}, (\neg\ (=\ y\ 0)), \leq_1, 1)$
- $\vdash (\text{remainder}\ a\ b) \leq_1 b$
- $\vdash \text{wf}\,(\text{gcd}, a, 1)$

We now illustrate the ability of our language to define transfinite recursions and recognize that they terminate. The following function determines if a set is an ordinal:

```
(define (ordinal? alpha)
  (if (= alpha {})
      True
      (if (∃ x (∧ (= (succ x) alpha) (ordinal? x)))
          True
          (if (∈ False (⋃ y alpha (ordinal? y)))
              False
              (= alpha (⋃ x alpha (⋃ y x {y})))))))
```

The rules for proving termination are briefly as follows:

- $(=\ (\text{succ}\ x)\ alpha) \vdash (\in\ x\ alpha)$
- $\vdash x \leq_1 alpha$
- $\vdash \text{wf}\,((\lambda\ (\text{ordinal?})\ (\text{ordinal?}\ x)), alpha, 1)$
- $(\bigcup\ y\ alpha\ (\text{ordinal?}\ y)) \vdash (\in\ y\ alpha)$
- $\vdash y \leq_1 alpha$
- $\vdash \text{wf}\,((\lambda\ (\text{ordinal?})\ (\text{ordinal?}\ y)), alpha, 1)$
- $\vdash \text{wf}\,(\text{ordinal?}, alpha, 1)$

3.3.3 Soundness

The soundness of the inference rules is proved is Appendix A. From these, we wish to establish the following:

Theorem 5 Let (Y f (λ (x_1 ... x_n) t)) be the recursive function of n arguments, $x_1, ..., x_n$ and body t defined by the functional $F(f') = M[(\lambda\ (x_1\ ...\ x_n)\ t), \rho[f := f']]$. If wf $((\lambda\ (f)\ t), x, i)$ is true, then F is well-founded on argument i.

Proof: Let $F(f') = M[(\lambda\ (x_1\ ...\ x_n)\ t), \rho[f := f']]$ and $F_2(f') = M[t, \rho[f := f']]$. By the definition of wf $((\lambda\ (f)\ t), x, i)$, for any functions g,h such that $f =_{i,\alpha} g$, $F_2(f) = F_2(g)$, which means that

$$M[t, \rho[f := g]] = M[t, \rho[f := h]]$$

Thus, it immediately follows that

$$M[(\lambda\ (x_1\ ...\ x_n)\ t), \rho[f := g]] = M[(\lambda\ (x_1\ ...\ x_n)\ t), \rho[f := h]]$$

Therefore, $f =_{i,\alpha} g \Rightarrow F(f) =_{i,\alpha+1} F(g)$, so F is well-founded on argument i. ■

Corollary 2 Let (λ (f) (λ ($x_1, ..., n_n$), t)) be a functional on functions of n arguments, $x_1, ..., x_n$ and body t. (Y f (λ (x_1 ... x_n) t)) is a fixed-point of the functional.

Proof: Immediate from Theorem 5 and Corollary 4. ■

3.3.4 Decidability

Theorem 6 Given an finite set of pre-defined functions and meta-formulas derived for those functions, for any expression in the language, the closure of all meta-formulas which are derivable from the inference rules about that expression is decidable.

Proof: Given the inference rules stated above, with the exception of one rule, every rule in the system derives meta-formulas about expressions larger than the antecedent expressions. Thus, if we wish to derive all the meta-formulas for any given expression, we are only required to examine meta-formulas on expressions of smaller or equal size. Since the initial expression is finite, the number of possible meta-formulas and rules which need to be examined is finite.

The only rule which is an exception is:

- $\Sigma \vdash \Delta((\mathrm{Y}\ f\ (\lambda\ (x_1\ ...\ x_n)\ t)), \Psi, \leq_a, i)$
  $\Sigma \vdash \Sigma \Rightarrow ((\mathrm{Y}\ f\ (\lambda\ (x_1\ ...\ x_n)\ \Psi))\ u_1\ ...\ u_n)$
  $\overline{\Sigma \vdash f(u_1, ..., u_n) \leq_a u_i}$

However, since this rule is only appropriate for a pre-defined set of functions, there are only a finite number of inferences which can be with this rule.

Thus, since a finite number of inferences can be generated with the above rule, and all other rules are restricted to a finite number of inferences about a given expression, there exists a decidable algorithm to generate all meta-formulas which are derivable for a given expression. ■

Chapter 4

Conclusion

In this paper, we have presented an alternative method of viewing Walther recursion. The primary goal was to a provide as broad of a framework as possible in which to view termination proofs using this syntactic notion of a size order. Forgoing predefined primitive data structures such as numbers and lists, already with the necessary argument-bounded primitive selectors, we chose sets with the membership operator as our single data structure. Using sets allowed us to establish a theoretical framework for defining Walther recursion under fixed point semantics.

Once we established a formal notion of Walther recursion, we proceeded to define a set-theoretic language. Given the proper operators, it is possible through sets to define structures such as numbers, lists, and more complex types, while preserving well-orderings for these types. Unfortunately, because our approach lacks any type restrictions, it is we occasionally necessary to artificially restrict how we define, using sets, our types as well as primitve operators for these types since intuitively natural properties of certain functions on their parameters do not always provably have the necessary properties since arguments which do not match the "expected" type specifications can be given to the function. This restriction could severely restrict the usefulness of this approach unless a bit of caution is taken.

However, despite this limitation, using a set-theoretic framework provides much more flexibility than the framework Walther selected since data structures do not have to be defined in a set manner. In addition, our method can prove termination of a

class of transfinite computations. Because the system is rule-based, deductions being performed are immediately apparent, also making it simple to create a system which not only automates termination proofs, but which can under certain circumstances determine why an algorithm does not provably terminate under these rules. Since the approach also remains essentially the same as Walther's, based a syntactic size order, we believe that termination proofs based on such a method can be established for a large class of naturally recursive algorithms.

Appendix A

Soundness

What follows is a series of informal arguments of the soundness of each of the individual inference rules through the semantics given by the language and by the meta-formulas.

A.1 Boolean Rules for Formulas

The following are inference rules that are derivable from the axioms of boolean algebra:

- $$\frac{\Sigma \vdash \Phi \quad \Sigma \vdash \Psi}{\Sigma \vdash (\wedge\ \Phi\ \Psi)}$$

- $$\frac{\Sigma \vdash \Phi}{\Sigma \vdash (\neg\ (\wedge\ \Phi\ \Psi)) \quad \Sigma \vdash (\neg\ \Psi)}$$

A.2 Argument Bounded Terms

The following rules are based on the axioms of set theory along with Theorem 3 which determines the relative size of sets.

- $(= \ x \ x)$

  For any set x, by the reflexive property of equality, $x = x$.

- $\Sigma \vdash (= \ x \ y)$

  $\overline{\Sigma \vdash x \leq_0 y}$

  By the substitution principle of equality, the rule immediately is true.

- $\Sigma \vdash (= \ x \ y)$

  $\overline{\Sigma \vdash x \leq_0 y}$

  By the substitution principle of equality, the rule immediately is true.

- $\Sigma \vdash (\in \ x \ y)$

  $\overline{\Sigma \vdash x \leq_1 y}$

  By Theorem 3, this rule immediately is true.

- $\Sigma \vdash x_i \in \{x_1, ..., x_n\}$

  The $\{\}$ operator in the language corresponds to construction of a set with members $x_1, ..., x_n$, the rule immediately follows from the axioms of set theory.

- $\Sigma, x \in t_1 \vdash t_2 \leq_a t_1$

  $\overline{\Sigma \vdash (\bigcup \ x \ t_1 \ t_2) \leq_a t_1}$

  By the semantics of the $\bigcup$ operator, $x \in t_1$ and $t_2 \subseteq (\bigcup \ x \ t_1 \ t_2)$. The second statement leads to the fact that $(\bigcup \ x \ t_1 \ t_2) \leq_0 t_2$ since the rule must hold for all interpretations of t_2. By the transitive property of inequality for ordinals the rule follows.

- $\Sigma, \Phi \vdash t_1 \leq_a t$

  $\Sigma, (\neg \ \Phi) \vdash t_2 \leq_b t$

  $\overline{\Sigma \vdash (\text{if } \Phi \ t_1 \ t_2)} \leq_{min(a,b)} t$

  By the semantics of the **if** operator, the value must be either t_1 or t_2. Therefore, the lower bound must be true.

- $\Sigma \vdash x \leq_a y$

  $\Sigma \vdash y \leq_b z$

$\overline{\Sigma \vdash x \leq_{a+b} z}$

If $x \leq_a y$ and $y \leq_b z$, by the transitive property of inequality for ordinals, $x \leq_{a+b} z$.

A.3 Argument Bounded Functions

- $\Sigma \vdash f \leq_a i$

  $\Sigma \vdash s_i \leq_b t$

  $\overline{\Sigma \vdash f(u_1, ..., u_{i-1}, s_i, u_{i+1}, ..., u_n) \leq_{a+b} t}$

  By the definition of the meta-formula for $f \leq_a i$, $|f(u_1, ..., u_{i-1}, s_i, u_{i+1}, ..., u_n)| + a \leq |s_i|$, where $f = M[F, \rho]$. Then, since $s_i \leq_b t \Rightarrow |s_i| + b \leq |t|$, by the additive property of inequality, $|f(u_1, ..., u_{i-1}, s_i, u_{i+1}, ..., u_n)| + a + b \leq |t| \Rightarrow F(u_1, ..., u_{i-1}, s_i, u_{i+1}, ..., u_n) \leq_{a+b} t$.

- $\Sigma, f \leq_0 i \vdash t \leq_a x_i$

  $\Sigma \vdash \text{wf}\ ((\lambda\ (f)\ (\text{Y}\ f\ (\lambda\ (x_1\ ...\ x_n)\ t))), x, n)$

  $\overline{\Sigma \vdash (\text{Y}\ f\ (\lambda\ (x_1\ ...\ x_n)\ t)) \leq_a i}$

  Proof by induction can be established using the fact that a base case must exist since the function is well-founded.

A.4 Formulas

- $\Sigma \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ \textbf{True}), \textbf{True}, \textbf{True})$

  If $f = M[(\lambda\ (x_1\ ...\ x_n)\ \textbf{True}), \rho]$, $\forall u_1, ..., u_n, f(u_1, ...u_n) = M[\textbf{True}, \rho]$. The inference rule immediately follows.

- $\Sigma \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ \textbf{False}), \textbf{True}, \textbf{False})$

  If $f = M[(\lambda\ (x_1\ ...\ x_n)\ \textbf{False}), \rho]$, $\forall u_1, ..., u_n, f(u_1, ...u_n) = M[\textbf{False}, \rho]$. The inference rule immediately follows.

- $\Sigma, \Phi \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ w), \Psi, t)$

  $\overline{\Sigma \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ (\text{if}\ \Phi\ w\ u)), (\wedge\ \Phi\ \Psi), t)}$

If $f = M[(\lambda\ (x_1\ ...\ x_n)\ (\text{if}\ \Phi\ w\ u)), \rho]$, if Φ is true, $f(u_1, ...u_n) = M[w, \rho]$. Furthermore, if Ψ is true, $f(u_1, ...u_n) = M[w, \rho] = M[t, \rho]$. Therefore, the rule follows.

- $\Sigma, (\neg\ \Phi) \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ u), \Psi, t)$

  $\overline{\Sigma \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ (\text{if}\ \Phi\ w\ u)), (\wedge\ (\neg\ \Phi)\ \Psi), t)}$

  If $f = M[(\lambda\ (x_1\ ...\ x_n)\ (\text{if}\ \Phi\ w\ u)), \rho]$, if $\neg\Phi$ is true, $f(u_1, ...u_n) = M[u, \rho]$. Furthermore, if Ψ is true, $f(u_1, ...u_n) = M[u, \rho] = M[t, \rho]$. Therefore, the rule follows.

- $\Sigma \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ t_1), \Psi, \textbf{True})$

  $\Sigma \vdash (=\ f(u_1, ..., u_n)\ \textbf{False})$

  $\overline{\Sigma \vdash (\neg\ \Psi[u_1/x_1\ ...\ u_n/x_n])}$

  Since $(=\ f(u_1, ..., u_n)\ \textbf{False})$, and value $((\lambda\ (x_1\ ...\ x_n)\ t_1), \Psi, \textbf{True})$, Ψ under the current interpretation must be false. Through β-reduction, we obtain the desired result.

- $\Sigma \vdash \text{value}\ ((\lambda\ (x_1\ ...\ x_n)\ t_1), \Psi, \textbf{False})$

  $\Sigma \vdash (=\ f(u_1, ..., u_n)\ \textbf{True})$

  $\overline{\Sigma \vdash (\neg\ \Psi[u_1/x_1\ ...\ u_n/x_n])}$

  Since $(=\ f(u_1, ..., u_n)\ \textbf{True})$, and value $((\lambda\ (x_1\ ...\ x_n)\ t_1), \Psi, \textbf{False})$, Ψ under the current interpretation must be false. Through β-reduction, we obtain the desired result.

A.5 Difference Literals

- $\Sigma \vdash t \leq_a x_i$

  $\overline{\Sigma \vdash \Delta((\lambda\ (x_1\ ...\ x_n)\ t), \textbf{True}, \leq_a, i)}$

  By the semantics of the λ operator,

$$\forall u_1, ..., u_n, f(u_1, ..., u_n) = M[t, \rho]$$

where $f = M[(\lambda\ (x_1\ ...\ x_n)\ t), \rho]$. Since $\Sigma \vdash t \leq_a x_i$, it immediately follows that

$$f(u_1, ..., u_n) \leq_a u_i$$

and therefore the rule is sound.

- $$\frac{\Sigma, \Phi \vdash \Delta((\lambda\ (x_1\ ...\ x_n)\ w), \Psi, \leq_a, i)}{\Sigma \vdash \Delta((\lambda\ (x_1\ ...\ x_n)\ (\text{if}\ \Phi\ w\ u)), (\wedge\ \Phi\ \Psi), \leq_a, i)}$$

  By the semantics of the λ operator,

$$\forall u_1, ..., u_n, f(u_1, ..., u_n) = M[(\text{if}\ \Phi\ w\ u), \rho]$$

where $f = M[(\lambda\ (x_1\ ...\ x_n)\ (\text{if}\ \Phi\ w\ u)), \rho]$. Let

$$g = M[(\lambda\ (x_1\ ...\ x_n)\ (\wedge\ \Phi\ \Psi)), \rho]$$

then for all $u_1, ..., u_n$,

$$g(u_1, ..., u_n) \Rightarrow \Phi'(u_1, ..., u_n) \wedge \Psi'(u_1, ..., u_n)$$

where $\Phi' = M[(\lambda\ (x_1\ ...\ x_n)\ \Phi), \rho]$ and $\Psi' = M[(\lambda\ (x_1\ ...\ x_n)\ \Psi), \rho]$. Therefore,

$$g(u_1, ..., u_n) \Rightarrow \Phi \Rightarrow f(u_1, ..., u_n) = M[w, \rho]$$

Given that $\Sigma, \Phi \vdash \Delta((\lambda\ (x_1\ ...\ x_n)\ w), \Psi, \leq_a, i)$ and $g(u_1, ..., u_n) \Rightarrow \Phi \wedge \Psi$,

$$g(u_1, ..., u_n) \Rightarrow w \leq_a u_i$$

Thus, the rule is sound.

- $$\frac{\Sigma, (\neg\ \Phi) \vdash \Delta((\lambda\ (x_1\ ...\ x_n)\ u), \Psi, \leq_a, i)}{\Sigma \vdash \Delta((\lambda\ (x_1\ ...\ x_n)\ (\text{if}\ \Phi\ w\ u)), (\wedge\ (\neg\ \Phi)\ \Psi), \leq_a, i)}$$

By the semantics of the λ operator,

$$\forall u_1, ..., u_n, f(u_1, ..., u_n) = M[(\text{if } \Phi\ w\ u), \rho]$$

where $f = M[(\lambda\ (x_1\ ...\ x_n)\ (\text{if } \Phi\ w\ u)), \rho]$. Let

$$g = M[(\lambda\ (x_1\ ...\ x_n)\ (\wedge\ (\neg\ \Phi)\ \Psi)), \rho]$$

then for all $u_1, ..., u_n$,

$$g(u_1, ..., u_n) \Rightarrow \Phi'(u_1, ..., u_n) \wedge \Psi'(u_1, ..., u_n)$$

where $\Phi' = M[(\lambda\ (x_1\ ...\ x_n)\ (\neg\ \Phi)), \rho]$ and $\Psi' = M[(\lambda\ (x_1\ ...\ x_n)\ \Psi), \rho]$. Therefore,

$$g(u_1, ..., u_n) \Rightarrow (\neg\ \Phi) \Rightarrow f(u_1, ..., u_n) = M[u, \rho]$$

Given that $\Sigma, (\neg\ \Phi) \vdash \Delta((\lambda\ (x_1\ ...\ x_n)\ u), \Psi, \leq_a, i)$ and $g(u_1, ..., u_n) \Rightarrow \Phi \wedge \Psi$,

$$g(u_1, ..., u_n) \Rightarrow u \leq_a u_i$$

Thus, the rule is sound.

- $\Sigma \vdash \Delta((\lambda\ (x_1\ ...\ x_n)\ t), \Psi, \leq_a, i)$

  $\Sigma \vdash \Psi[u_1/x_1\ ...\ u_n/x_n]$

  $\overline{\Sigma \vdash f(u_1, ..., u_n) \leq_a u_i}$

  The rule immediately follows from β-reduction and the definition of the δ meta-formula.

A.6 Well-Founded Functionals

- $\Sigma \vdash \text{wf}\ ((\lambda\ (f)\ w_1), x, i)$

  $\vdots$

  $\Sigma \vdash \text{wf}\ ((\lambda\ (f)\ w_n), x,)$

$$\frac{\Sigma \vdash w_i < x}{\Sigma \vdash \text{wf}\,((\lambda\,(f)\; f(w_1, ..., w_n)), x, i)}$$

Let j,k be any functions such that $\forall \alpha, j =_{i,\alpha} k$. By the semantics of our language,

$$F(j) = M[f(w_1, ..., w_n), \rho[f := j]]$$

which means that $F(j) = j(w'_1, ..., w'_n)$ where

$$w'_1 = M[w_1, \rho[f := j]], ..., w'_n = M[w_n, \rho[f := j]]$$

By similar reasoning, we also obtain $F(k) = k(w''_1, ..., w''_n)$ where

$$w''_1 = M[w_1, \rho[f := k]], ..., w''_n = M[w_n, \rho[f := k]]$$

Given our initial assumptions that $\text{wf}\,((\lambda\,(f)\; w_1), x, .)..\text{wf}\,((\lambda\,(f)\; w_n), x, i)$,

$$G_1(j) = G_1(k) \Rightarrow M[w'_1, \rho[f := j]] = M[w''_1, \rho[f := k]]$$

$$\vdots$$

$$G_n(j) = G_n(k) \Rightarrow M[w'_n, \rho[f := j]] = M[w''_n, \rho[f := k]]$$

Since our initial assumption is that $w_i < x$, and because $g =_{i,|x|} h$, $F(g) = g(w'_1, ..., w'_n) = h(w''_1, ..., w''_n) = F(h)$.

- $$\begin{array}{l} \Sigma \vdash \text{wf}\,((\lambda\,(f)\; w_1), x, i) \\ \vdots \\ \Sigma \vdash \text{wf}\,((\lambda\,(f)\; w_n), x, i) \\ g \neq f \\ \hline \Sigma \vdash \text{wf}\,((\lambda\,(f)\; g(w_1, ..., w_n), x, i) \end{array}$$

  Let j,k be any functions such that $\forall \alpha, j =_{i,\alpha} k$ By the semantics of our language,

$$F(j) = M[g(w_1, ..., w_n)), \rho[f := j]]$$

which means that $F(j) = g(w'_1, ..., w'_n)$ where

$$w'_1 = M[w_1, \rho[f := j]], ..., w'_n = M[w_n, \rho[f := j]]$$

By similar reasoning, we also obtain $F(k) = g(w''_1, ..., w''_n)$ where

$$w''_1 = M[w_1, \rho[f := k]], ..., w''_n = M[w_n, \rho[f := k]]$$

Given our initial assumptions that wf $((\lambda\ (f)\ w_1), x, i)$...wf $((\lambda\ (f)\ w_n), x, i)$,

$$G_1(j) = G_1(k) \Rightarrow M[w'_1, \rho[f := j]] = M[w''_1, \rho[f := k]]$$

$$\vdots$$

$$G_n(j) = G_n(k) \Rightarrow M[w'_n, \rho[f := j]] = M[w''_n, \rho[f := k]]$$

By similar reasoning, we also obtain $F(k) = g(w_1, ..., w_n)$. Therefore, $F(j) = g(w'_1, ..., w'_n) = g(w''_1, ..., w''_n) = F(k)$.

- $\Sigma \vdash$ wf $((\lambda\ (f)\ \Phi), x, i)$
  $\Sigma, \neg\Phi \vdash$ wf $((\lambda\ (f)\ u), x, i)$
  $\Sigma, \Phi \vdash$ wf $((\lambda\ (f)\ w), x, i)$
  $\overline{\Sigma \vdash \text{wf}\ ((\lambda\ (f)\ (\text{if}\ \Phi\ w\ u)), x, i)}$

  Let g,h be any functions such that $\forall\alpha, g =_{i,\alpha} h$. By the semantics of our language,

  $$F(g) = M[(\text{if}\ \Phi\ w\ u), \rho[f := g]]$$

  and

  $$F(h) = M[(\text{if}\ \Phi\ w\ u), \rho[f := h]]$$

  By the semantics of the if, the above values are dependent on Φ. Furthermore, since $\Sigma \vdash$ wf $((\lambda\ (f)\ \Phi), x, i)$, if we let $G = M[(\lambda\ (f)\ \Phi), \rho]$, $M[\Phi, \rho[f := g]] = G(g) = G(h) = M[\Phi, \rho[f := h]]$. Thus, we can examine the value of the if

statement case by case on the value of $\Phi = M[\Phi, \rho[f := g]] = M[\Phi, \rho[f := h]]$.

- $\Phi =$ **True**: We then have

$$F(g) = M[w, \rho[f := g]]$$

and

$$F(h) = M[w, \rho[f := h]]$$

Since $\Sigma, \Phi \vdash \text{wf}\ ((\lambda\ (f)\ w), x, i)$, by the definition of well-foundedness, given that $G = M[(\lambda\ (f)\ w), \rho]$, $G(f) = G(g) \Rightarrow M[w, \rho[f := g]] = M[w, \rho[f := h]]$. Therefore, $F(f) = F(g)$.

- $\Phi =$ **False**: We then have

$$F(g) = M[u, \rho[f := g]]$$

and

$$F(h) = M[u, \rho[f := h]]$$

Since $\Sigma, \neg\Phi \vdash \text{wf}\ ((\lambda\ (f)\ u), x, i)$, by the definition of well-foundedness, given that $G = M[(\lambda\ (f)\ u), \rho]$, $G(f) = G(g) \Rightarrow M[u, \rho[f := g]] = M[u, \rho[f := h]]$. Therefore, $F(f) = F(g)$.

- $\Phi = \bot$: We then have

$$F(g) = M[\bot, \rho[f := g]]$$

and

$$F(h) = M[\bot, \rho[f := h]]$$

Therefore, $F(f) = F(g)$.

Since in all possible cases, $F(f) = F(g)$, $\text{wf}\ ((\lambda\ (f)\ (\text{if}\ \Phi\ w\ u)), x, i)$.

- $\Sigma \vdash \text{wf}\,((\lambda\,(f)\,\Phi), x, i)$
  $\Sigma \vdash \text{wf}\,((\lambda\,(f)\,\Psi), x, i)$
  $\overline{\Sigma \vdash \text{wf}\,((\lambda\,(f)\,\Phi \vee \Psi), x, i)}$

  Let j,k be any functions such that $\forall \alpha, j =_{i,\alpha} k$ By the semantics of our language,

$$F(j) = M[\Phi \vee \Psi, \rho[f := j]]$$

  and

$$F(k) = M[\Phi \vee \Psi, \rho[f := g]]$$

  By our initial assumptions,

$$G(j) = G(k) \Rightarrow M[\Phi, \rho[f := j]] = M[\Phi, \rho[f := k]]$$

  and

$$H(j) = H(k) \Rightarrow M[\Psi, \rho[f := j]] = M[\Psi, \rho[f := k]]$$

  Since $M[\Phi \vee Psi, \rho']$ is dependent only upon $M[\Phi, \rho']$ and $M[\Psi, \rho']$,

$$M[\Phi \vee \Psi, \rho[f := j]] = M[\Phi \vee \Psi, \rho[f := k]] \Rightarrow F(j) = F(k)$$

- $\Sigma \vdash \text{wf}\,((\lambda\,(f)\,t_1), x, i)$
  $\Sigma \vdash \text{wf}\,((\lambda\,(f)\,t_2), x, i)$
  $\overline{\Sigma \vdash \text{wf}\,((\lambda\,(f)\,t_1 \in t_2), x, i)}$

  Let j,k be any functions such that $\forall \alpha, j =_{i,\alpha} k$ By the semantics of our language,

$$F(j) = M[t_1 \in t_2, \rho[f := j]]$$

  and

$$F(k) = M[t_1 \in t_2, \rho[f := g]]$$

By our initial assumptions,

$$G(j) = G(k) \Rightarrow M[t_1, \rho[f := j]] = M[t_1, \rho[f := k]]$$

and

$$H(j) = H(k) \Rightarrow M[t_2, \rho[f := j]] = M[t_2, \rho[f := k]]$$

Since $M[t_1 \in t_2, \rho']$ is dependent only upon $M[t_1, \rho']$ and $M[t_2, \rho']$,

$$M[t_1 \in t_2, \rho[f := j]] = M[t_1 \in t_2, \rho[f := k]] \Rightarrow F(j) = F(k)$$

- $\dfrac{\Sigma \vdash \text{wf}\,((\lambda\ (f)\ \Phi), x, i)}{\Sigma \vdash \text{wf}\,((\lambda\ (f)\ \neg\Phi), x, i)}$

Let j,k be any functions such that $\forall\alpha, j =_{i,\alpha} k$ By the semantics of our language,

$$F(j) = M[\neg\Phi, \rho[f := j]]$$

and

$$F(k) = M[\neg\Phi, \rho[f := g]]$$

By our initial assumption,

$$G(j) = G(k) \Rightarrow M[\Phi, \rho[f := j]] = M[\Phi, \rho[f := k]]$$

Since $M[\neg\Phi, \rho']$ is dependent only upon $M[\Phi, \rho']$,

$$M[\neg\Phi, \rho[f := j]] = M[\neg\Phi, \rho[f := k]] \Rightarrow F(j) = F(k)$$

- $\dfrac{\Sigma \vdash \text{wf}\,((\lambda\ (f)\ \Phi), x, i)}{\Sigma \vdash \text{wf}\,((\lambda\ (f)\ \exists x\Phi), x, i)}$

Let j,k be any functions such that $\forall \alpha, j =_{i,\alpha} k$ By the semantics of our language,

$$F(j) = M[\exists x \Phi, \rho[f := j]]$$

and

$$F(k) = M[\exists x \Phi, \rho[f := g]]$$

By our initial assumption,

$$G(j) = G(k) \Rightarrow M[\Phi, \rho[f := j]] = M[\Phi, \rho[f := k]]$$

Since $M[\neg \Phi, \rho']$ is dependent only upon $M[\Phi, \rho']$,

$$M[\exists x \Phi, \rho[f := j]] = M[\exists x \Phi, \rho[f := k]] \Rightarrow F(j) = F(k)$$

- $\Sigma \vdash \text{wf}\,((\lambda\,(f)\,w_1), x, i)$

  $\vdots$

  $\Sigma \vdash \text{wf}\,((\lambda\,(f)\,w_n), x, i)$

  $\overline{\Sigma \vdash \text{wf}\,((\lambda\,(f)\,\{w_1, ..., w_n\}), x, i)}$

  Let j,k be any functions such that $\forall \alpha, j =_{i,\alpha} k$ By the semantics of our language,

  $$F(j) = M[\{w_1, ..., w_n\}, \rho[f := j]]$$

  which means that $F(j) = \{w_1', ..., w_n'\}$ where

  $$w_1' = M[w_1, \rho[f := j]], ..., w_n' = M[w_n, \rho[f := j]]$$

  By similar reasoning, we also obtain $F(k) = \{w_1'', ..., w_n''\}$ where

  $$w_1'' = M[w_1, \rho[f := k]], ..., w_n'' = M[w_n, \rho[f := k]]$$

Given our initial assumptions that $\text{wf}((\lambda\ (f)\ w_1), x, i)$...$\text{wf}((\lambda\ (f)\ w_n), x, i)$,

$$G_1(j) = G_1(k) \Rightarrow M[w_1', \rho[f := j]] = M[w_1'', \rho[f := k]]$$

$$\vdots$$

$$G_n(j) = G_n(k) \Rightarrow M[w_n', \rho[f := j]] = M[w_n'', \rho[f := k]]$$

By similar reasoning, we also obtain $F(k) = \{w_1, ..., w_n\}$. Therefore, $F(j) = \{w_1', ..., w_n'\} = \{w_1'', ..., w_n''\} = F(k)$.

- $\text{wf}((\lambda\ (f)\ x), x', i)$

- $\dfrac{\Sigma \vdash \text{wf}((\lambda\ (f)\ t), x, i)}{\Sigma \vdash \mathit{Walther}((\text{Y}\ f\ (\lambda\ (x_1\ ...\ x_n)\ t)), f, x, i)}$

  Immediate from definitions

Bibliography

[1] Jon Barwise, editor. *Handbook of Mathematical Logic.* North-Holland Publishing Company, New York, 1977.

[2] J.L. Bell and M. Machover. *A Course in Mathematical Logic.* North-Holland Publishing Company, New York, 1977.

[3] Robert S. Boyer and J. Strother Moore. *A Computational Logic.* Academic Press, 1979.

[4] R.M. Burstall. Program proving as hand simulation with a little induction. In *Proceedings IFIP Congress 1974 Stockholm*, pages 308–312, 1974. As cited in [12].

[5] D.C. Cooper. Programs for mechanical program verification. *Machine Intelligence*, 6:43–59, 1971.

[6] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, volume 19, 1967.

[7] C.A.R. Hoare. Find. *Communications of the ACM*, 12:321, 1961.

[8] S. Katz and Z. Manna. A closer look at termination. *Acta Informatica*, 5:333–352, 1975. As cited in [12].

[9] Manna, Ness, and Vuillemin. Inductive methods for proving properties of programs. *Communications of the ACM*, 16(8):491–502, 1973.

[10] Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

[11] Guy L. Steele. *Common Lisp - The Language.* Digital Press, 1984.

[12] Christoph Walther. *Automated Termination Proofs.* Universitat Karlsruhe, Institut fur Logik, Komplexitat und Deduktionssysteme, 1989.